Towards feasible, machine-assisted verification of object-oriented programs

Francis Hin-Lun Tang

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2002

Abstract

This thesis provides an account of a development of tools towards making verification of object-oriented programs more feasible. We note that proofs in program verification logics are typically long, yet, mathematically, not very deep; these observations suggest the thesis that computers can significantly ease the burden of program verification. We give evidence supporting this by applying computers to (1) automatically check and (2) automatically infer large parts of proofs.

Taking the logic (AL) of Abadi and Leino as our starting point, we initially show how the logic can be embedded into a higher-order logic theorem prover, by way of introducing axioms, using a mix of both higher-order abstract syntax (HOAS) and a direct embedding of the assertion logic. The tenacity and exactness of the theorem prover ensures that no proof obligation is inadvertently lost during construction of a proof; we inherit any automatic facilities such as tactics which take us part way towards goal (2); and moreover, we achieve goal (1), since we inherit *machine proofs* which can be checked automatically. We present some extended examples in AL that have been proved using this embedding.

Since we use a mix of HOAS and the underlying logic, justification of the axioms is no longer trivial. In particular, because we use a HOAS-style encoding of the program syntax, using a standard interpretation, the set of programs does not correspond to that defined using the first-order syntax. Thus, we construct a non-standard model of higher-order logic using categorical constructs, and show, in this interpretation (i.e. model), that the axioms are sound.

Further towards goal (2), we consider a verification condition generator (VCG) algorithm which automatically infers large parts of a proof. The VCG takes, as input, a program c—possibly with annotations—and a specification S, and outputs a logical sentence called a verification condition (VC). The intention is that to prove c satisfies S, it suffices to prove the VC; or equivalently, the VCG automatically proves c satisfies S, modulo validity of the VC.

The VC resulting from the algorithm is, in general, a first-order formula with fix-points. In particular cases, it is known that there exist brute force checking algorithms for such formulae which are much more efficient than those for checking

arbitrary second-order formulae. Furthermore, we can improve on this: we show that by inserting (suitably many) annotations, it is possible to ensure that the VC is fix-point-free, that is, we obtain a purely first-order formula. A first-order formula further improves on a fix-point formula—as far as automatic checking is concerned—and we give accounts of specific examples where the resulting VC can be checked using an automatic, first-order logic theorem prover. Such examples demonstrate: (i) the possibility to eliminate syntactic overhead leaving only the algorithmic contents of a program; and (ii) the possibility of using the VCG to provide a "push-button" solution to program verification.

As a key component of the VCG, we develop a type inference algorithm for AL. Partly because AL includes base types (such as natural numbers and booleans) and also because of the more expressive form of subtyping (resulting from covariant method return types), existing type inference algorithms for object-oriented languages could not be applied directly. Instead, we develop an original type inference solution by applying, to existing approaches, non-trivial modifications which do not compromise asymptotic performance.

We give an account of examples that have been attempted with a prototype implementation of the VCG, including experimental evidence that this approach is somewhat stable with respect to different implementations of the same algorithm.

Acknowledgements

The first person I would like to thank is Martin Hofmann, my supervisor. Not only has he been a good friend, but also a great teacher: of computer science, mathematics and writing. He has been patient, when I was demanding of his time, supportive, when I felt lost, and is simply an exceptional academic to look up to. I would also like to thank: Paul Jackson and Ian Stark who took on the supervisor's rôle for the periods when Martin was away. Thanks to (again) Paul and Martín Abadi for agreeing to be my examiners, and for their detailed suggestions for improvement.

In Edinburgh, I would like to thank the members of the LFCS for providing such an exciting, inspiring and friendly environment in which to do a PhD. I would especially like to thank my fellow students Lennart Beringer, Tom Chothia and Jan Jürjens for listening to my rants when things didn't work and for providing refreshing conversation to preserve my sanity. I would also like to thank the computing and support officers for keeping the computers working all the time—a tough job which, when done well, is hardly ever noticed. And long owed thanks to Peter and Judy Hancock who so kindly put me up in 1998—of all times, during the Festival!

Thanks go to various flatmates: Rob Mawdsley, Christine Mawdsley, Laura Ferguson, Harriet Hall, Michael Eng, Leia Ho and Sheri Tay; firstly for living with me but more for keeping me sane through good company.

In Darmstadt, I am greatly indebted to Peter Lietz and Frau Bergsträßer for so efficiently making local arrangements for my visit, especially to Peter who even attended to my social well-being. Thanks also go to Thomas Streicher and other members of the AG14 group for inspiring and enlightening discussions.

In Munich, I am immensely indebted to Ralph Matthes who, on such short notice, put me up for the last few days of my visit. Thanks go to: Max Jakob for providing first-class computing support; Jan Johannson and (again) Ralph for a thorough commentary on Bavarian cuisine and language; officemates Steffen Jost and Favio Miranda for providing stress-relieving conversation; and Andreas Abel for bringing us together for his evenings of Tyrolean and Bavarian food,

and board-games. Special thanks to David von Oheimb for providing fellowship, showing me the wonders of the Bavarian Alps, and making helpful comments on a draft of this thesis.

Also, thanks go to Christoph Weidenbach, SPASS implementor, for quickly implementing bug fixes.

I wish to acknowledge financial support from EPSRC Studentship 98318205, GKLI (Munich) visitors' grant, and DAAD grant A/01/45666; a presentation of some of this work was partly funded by EU TYPES project (number 29001, IST programme).

Thank you: Mum, Dad and Sylvia, my dear sister; thank you for your continual love, support and encouragement. And finally, special thanks to T. I.; thank you for staying with me through the highs and lows, and through the pains of separation; this thesis has been a challenge for you as much as it has been for me.

Declaration

I declare that this thesis was composed by myself, that at least 50% of the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification. Early versions of parts of this thesis have already been published elsewhere:

- Chapter 3 (except Section 3.5) as [HT00] (joint with Martin Hofmann); and
- Chapter 5 and Section 6.2 as extended abstract [TH02] (joint with Martin Hofmann), and also as extended abstract [Tan01] (with accompanying poster).

(*Francis Hin-Lun Tang*)

This thesis was only possible by the grace of our Lord;
and I dedicate it to Him.

Table of Contents

1 Introduction 1

1.1 Object-oriented programming 2
1.2 Verification 2
1.3 Feasible object-oriented verification 4
1.4 Related work 6
1.4.1 Program verification 6
1.4.2 Object-oriented verification 7
1.4.3 Computer-assisted verification 9
1.5 Overview of this thesis 10

2 The logic of objects of Abadi and Leino 11

2.1 Programming language 11
2.1.1 Examples 14
2.1.2 Types 17
2.2 Axiomatic semantics 17
2.2.1 Verification rules 19
2.3 Metatheory 22
2.3.1 (In)completeness 23
2.3.2 When syntactic sugar leaves a bitter after-taste 24
2.4 Summary 25

3 AL in higher-order logic 27

3.1 Related work 30
3.2 Metalanguage 32

3.2.1 Program logic 37
3.3 Practical considerations 44
3.4 Examples 46
3.4.1 Reversible let rule 47
3.4.2 Greatest common divisor 48
3.4.3 Dining philosophers 49
3.5 Soundness 54
3.5.1 Preliminaries 54
3.5.2 Presheaves 55
3.5.3 Category of interpretation 57
3.5.4 Interpretation 62
3.5.5 Justification 86
3.6 Conclusions 96

4 Type inference **99**
4.1 An overview 99
4.1.1 Related work 100
4.2 Type derivations 102
4.3 Pretypes and types 103
4.3.1 Constructions on pretypes 107
4.3.2 Ordering pretypes 108
4.4 Generating constraints 109
4.5 Constraint graphs 110
4.5.1 Presolutions 114
4.5.2 Presolutions of closures 115
4.5.3 From presolutions to solutions 115
4.5.4 From constraints to graphs 116
4.6 A family of automata 116
4.6.1 Well-formed constraint graphs 118
4.6.2 Completeness 119
4.6.3 Soundness 120
4.6.4 Type inference 120

4.7 Algorithms 122
4.7.1 Efficient closure 123
4.7.2 Well-formedness 128
4.8 Type inference summary 129
4.9 Conclusions and further work 129

5 **Verification Condition Generator** **133**
5.1 Related work 135
5.2 Second-order verification conditions 138
5.3 Soundness and completeness 145
5.4 First-order, fix-point verification conditions 147
5.4.1 Success 154
5.5 First-order verification conditions 160
5.5.1 Dependency graphs 161
5.5.2 Bridges 167
5.5.3 Cycles 167
5.5.4 How many annotations? 173
5.6 Conclusions and further work 174

6 **A prototype VCG implementation and examples** **185**
6.1 Prototype implementation 185
6.1.1 The back-end 186
6.2 Euclid's algorithm(s) 188
6.3 Dining philosophers revisited, (almost) push-button-style 192
6.4 Conclusions 197

7 **Conclusions and further work** **199**
7.1 Supporting more language features 200
7.1.1 Recursive object types 201
7.1.2 Further language features 203
7.2 Changing the underlying program logic 204
7.2.1 Incompleteness 204
7.2.2 Data abstraction 206

7.2.3 Support for abstract fields 207
7.3 Improving back-end support 208

A Proofs **211**
A.1 Proofs for Chapter 3 211
A.1.1 Proof of Lemma 1 on page 58 211
A.1.2 Proof of Theorem 2 on page 64 212
A.2 Proofs for Chapter 4 215
A.2.1 Theorem 5 on page 108 215
A.2.2 Theorem 6 on page 115 216
A.2.3 Theorem 7 on page 115 217
A.2.4 Theorem 9 on page 119 220
A.2.5 Theorem 10 on page 120 222
A.2.6 Lemma 10 on page 123 224
A.2.7 Theorem 11 on page 123 224
A.2.8 Lemma 12 on page 127 225
A.3 Proofs for Chapter 5 232
A.3.1 Proof of Proposition 3 232

B Input scripts for VCG examples **239**
B.1 Example SPASS input script 239
B.2 Dining philosophers example input script 242
B.3 Example SPASS input script for dining philosophers example . . . 244

Bibliography **257**

List of Figures

5.1 Dependency graph for *if* x *then* a_0 *else* a_1 168

5.2 Two possible cycles in *if* x_j *then* a_0 *else* a_1 168

5.3 Dependency graph for *let* x=a *in* b 169

5.4 Two possible cycles in *let* x=a *in* b 170

5.5 Dependency graph for $[f = x_j, m = \varsigma(y)b]$ 171

5.6 Four possible cycles in $[f{=}x_j, m{=}\varsigma(y)b]$ 172

5.7 Another possible cycle in *let* x=a *in* b 172

6.1 Concrete syntax for Euclid's algorithm 188

List of Tables

2.1 Operational semantics for AL 15
2.2 AL compared to Hoare logic 20
2.3 Predicate symbols 20

3.1 Encodings of logical connectives 34
3.2 Rules for classical higher-order logic 35
3.3 Auxiliary subsumption relation for soundness proof 82
3.4 Auxiliary derivability relation for soundness proof 83

4.1 Syntax of AL types 102
4.2 Definition of subtype 102
4.3 Typing rules with built-in subsumption 104
4.4 Constraints for type inference 111
4.5 Efficient closure rules for type inference 132

5.1 Subspecification relation 137
5.2 AL rules with built-in subsumption 178
5.3 Standard renamings 179
5.4 Verification condition at a specific switching 179
5.5 Dependency graph 180
5.6 Overview of VC simplification algorithm 181
5.7 Expression rewriting 182
5.8 Simplification of $- \lhd - \rhd -$ expressions 182
5.9 Miscellaneous simplification rules 183
5.10 Simplification by minimal set instantiation 183

5.11 Equality eliminating simplifications 184

6.1 Performance summary for verification of dining philosophers . . . 196

Chapter 1

Introduction

Computers have become ubiquitous in modern life, and evidence suggests that this is likely to continue for the foreseeable future. Since the the middle of the 20th century, the early days of the modern computer, we have seen computers become bigger[1], faster and cheaper. Bigger and faster computers have allowed them to be used in applications requiring more complex solutions. Cheaper computers have made them more accessible. Complexity and accessibility introduce further problems.

Complicated software is difficult to build, and worse still, the relationship is not linear: a program that is twice as long requires more than twice the effort. It appears also that accessibility has led to dependence, resulting in increasing importance of computers for economy and society. As a consequence, software engineers are charged with the task of building software that both is increasingly complicated and has increasingly significant consequences if incorrect.

Research and experience has resulted in, amongst other solutions, the object-oriented (OO) programming paradigm, and program verification.

[1]bigger in capacity, yet smaller in physical form

1.1 Object-oriented programming

When presented with a large problem, a reasonable, and common practice is to break it up into smaller problems which are easier to solve, in such a way that these smaller solutions can be composed to give a solution to the large problem. In practice, large problems change in an evolutionary way, and so it is desirable to break up the problem in such a way that likely changes affect as few of the smaller problems as possible, so that more of the smaller solutions can be reused.

In the case of software, this has resulted in a modular style of programming: modules solve smaller problems and we combine them to solve larger problems. Although there is still no consensus as to what is the best modular unit, nevertheless, in the past decade, objects have proven to be a popular choice.

In previous approaches to modular programming, the emphasis is on code and data come second: typically, we have code in procedures and functions, which manipulate data. In OO programming, we flip this on its head, and consider data as first-class citizens. Objects are data together with computational capabilities: given some object, we can make it perform some computation and thereby possibly modifying itself or other data. Perhaps objects and the way we formulate problems fit together quite naturally, and this explains the success of the OO paradigm.

1.2 Verification

Towards the end of the 20th century, the show of public anxiety over the so-called *Millennium Bug* perhaps highlighted the extent of social and economic dependence on software, a situation which appears unlikely to diminish. The ability to decide whether a program, when executed, will behave "correctly" is clearly of value, and has been a long-standing desire.

The most widely used approximation to this ability is *testing*: simply execute the program for various inputs and observe whether it behaves as expected. But a program that passes testing is only known to be correct for those possibilities that have been tested. The only way to be able to guarantee correctness for all

possibilities, without further analysis of the program, is to test all possibilities. Either this is impossible, infeasible (since there are too many possibilities), or if it is feasible then the question is not interesting.

An alternative way to obtain this guarantee is to analyse the program. For example, given a rigorous operational semantics, which describes how a program is executed, we can use mathematics and logical arguments to predict how a program behaves when executed for different inputs. The ability of mathematics to describe the infinite allows us to predict behaviour for *all* possible inputs.

It has long been realised that, without machine assistance, this is only a theoretical solution, in view of the size of programs and the resulting proof obligations. In pursuit of machine assistance, research has concentrated on formalising the required mathematics as logics.

Examples of such logics are type systems of languages such as ML [MTHM97], Pascal and Java [AG98]. Here specifications are *types*, and thus correctness is a very coarse notion, yet fine enough to catch some errors, such as trying to compute $1 + \mathtt{tt}$, the sum of an integer and a boolean. But the types of these languages are not expressive enough to be able to state that a program computes a certain value. The lack of expressivity, however, buys us decidability: we can automatically determine whether an annotated program is well-typed; and, in the case of ML, we can automatically find proofs too.

Putting decidability aside opens up the possibility of logics which can determine functional correctness. The prototypical example is the verification calculus of Hoare, which is now referred to as Hoare Logic. Here we determine statements of the form $\{P\}\,c\,\{Q\}$ where c is a program, and P, Q are assertions which are predicates over states. Traditionally, such a statement means that starting from a state satisfying P and executing c, provided execution terminates, we may conclude that the resulting state satisfies Q.

As in common use, for the rest of this thesis, we reserve the term *verification* for problems of the latter, undecidable case and refer to logics for verification as *program logics*.

1.3 Feasible object-oriented verification

Despite the pervasiveness of the OO paradigm, verification of OO programs is still in its infancy. Existing program logics are either non-modular or incomplete (or both), and in either case cumbersome to use, both for finding and checking proofs. This thesis contributes towards the latter: reducing the chore of program verification.

Experience has found that proofs in program logics are usually very long. This has consequences for checking and finding proofs.

We note that the purpose of a proof of some statement P is to convince someone beyond all reasonable doubt that P is true. In mathematical textbooks, famous proofs which have stood the test of time are invariably short, and for good reason too. A proof that is so long that even after checking every detail, one is still unsure if it is correct, has less value as a proof. Fortunately, long proofs are acceptable if one can delegate the task to a reliable and trustworthy friend, such as a computer.

Finding proofs is hard, and finding long proofs is harder still! Since we have given up decidability in exchange for expressivity, we must accept that, in general, human creativity is required to find proofs. Typically, in the case of program verification, creativity is required only in a very small part of a proof; the bulk can be found automatically.

This thesis proposes that feasible program verification with a program logic requires the ability to automatically infer large parts of, and to automatically check proofs. We give evidence by example; in the context of the logic AL of Abadi and Leino (a summary of which is given in Chapter 2), we present:

1. a generic technique for embedding AL into simply-typed lambda calculus with higher-order logic, allowing us to use various higher-order theorem provers; and

2. an algorithm, called a verification condition generator (VCG), reducing the proof burden of verification by distilling the essential mathematical content of a proof.

In so doing, we make the following contributions to the area of OO program verification.

- Our approach to embedding AL into a higher-order theorem prover postulates axioms using both higher-order abstract syntax (HOAS) and a direct embedding of the logical side-conditions. These two implementation decisions allow us to inherit both the variable management and theorem-proving tactics of the underlying theorem prover.

- We justify the axioms by checking their interpretations in a model of higher-order logic, which we construct. This is an application of more general techniques for reasoning about HOAS, using non standard interpretations of the metalogic.

- We present an algorithm for reducing proof burden in two stages. The first stage infers the shape of a proof and its specifications to give skeleton proof. This component amounts to inference in the type system of the underlying language of AL. It is based on the algorithms of Palsberg [Pal95, PWO97], which use constraints solving and automata-theoretic methods. Since the type system of AL is more expressive, because of covariant method subtyping, nontrivial and original modifications were required.

- The second stage fleshes out the skeleton proof by introducing predicate variables for the unknowns in the proof, giving a natural existentially quantified, second-order formula whose validity is a sufficient (and necessary) condition for the existence of a proof. We present sound and complete rules that simplify this formula, eliminating the existentially quantified predicates in favour of fix-point operators. Moreover, by allowing annotations, e.g. loop invariants, we show that it is possible, given sufficiently many annotations, to obtain a resulting formula that is purely first-order. We show that in some cases it is possible to delegate proving this formula to an automatic theorem prover, with appropriate lemmata.

1.4 Related work

Here we give a summary of the related work in the field of program verification; firstly program verification in general, and then in particular verification for OO programs and finally computer aided program verification.

1.4.1 Program verification

In Hoare's seminal article [Hoa69] of 1969, he presents an axiomatic semantics for a simple imperative (vs. functional) language based on earlier work by Floyd [Flo67]. Today, such a formalism is described as a Hoare Logic, and in a modern presentation it is defined as a collection of rules allowing one to derive correctness statements of the form $\{P\}\, c\, \{Q\}$ where P, Q are assertions on the state space and c is a program (fragment). Such a statement, informally, means that providing the state satisfies P, if executing c terminates, then the resulting state satisfies Q.

Perhaps the most significant contribution of Hoare and Floyd was the formalisation of a proof technique for showing correctness of loops via invariants. Another significant contribution was its formalisation of reasoning about assignment: $\{Q[e/x]\}\, x{:=}e\, \{Q\}$. Here $Q[e/x]$ denotes syntactic substitution of (logical) expression e for free occurrences of x in Q. Already, the early pioneers of program logics argued that this formalisation allows for easier mechanisation. Unfortunately, this rule is unsound in the presence of aliasing.

Nevertheless, Hoare's early work provided inspiration for subsequent research in program verification. Hoare's logic was extended to more elaborate programming languages[2], for example, to those that include recursive methods. We find, in Kleymann's thesis [Kle98], machine-checked proofs of soundness and completeness of logics for one such language (in the special case of one procedure with one parameter). In the same vein, von Oheimb formally checked soundness and completeness metatheorems for a Hoare logic supporting mutually recursive methods (with several parameters and local variables) [Ohe99]. Formal checking of such

[2]In this thesis, we concern ourselves only with sequential languages.

metatheorems is not our objective: this thesis concerns checking theorems about concrete programs.

Reynolds, in [Rey82], presents a verification logic for an Algol-like language, called Idealised Algol. In particular, this language allows for aliased variables caused, for example, by repeated parameters in procedure calls. To be able to cope with aliasing, Reynolds separates the notion of state into *environments* and *stores*: environments map program variables to locations, and stores map locations to values. Furthermore, he introduces extra assertions expressing interference—whether executing a program can change a logical expression—and well-behavedness of variables—for example, if m, n are variables then, in Idealised Algol, *if* $m = 1$ *then* m *else* n is a variable and thus can be assigned to, but it is not well-behaved in the sense that Hoare's assignment rule breaks down.

1.4.2 Object-oriented verification

With respect to this thesis, the most important OO verification logic is that of Abadi and Leino (AL) [AL97, AL98]. We save an overview of this logic until Chapter 2. Another logic of Leino can be viewed as a modification of AL to address the issue of recursive types [Lei98]. In this logic, we see a name-matching style of subtyping which admits recursive types without the need for fix-point operators. In a departure from AL, specification is intertwined with the underlying programming language; explicit specifications have to be provided for all methods. However, the logic does not feature covariant subtyping. The consequences of this limitation are discussed in Section 7.1.1. Ecstatic [Lei97], also by Leino, introduces an object-oriented programming language with a weakest liberal precondition semantics. Correctness of programs is defined via a VCG. Similar to the other logic of Leino in [Lei98], Ecstatic also uses a name matching-style of subtyping, also requires all methods to be specified, and also lacks covariant method return types. However, specifications are provided as pre- and post-conditions, and methods are also specified with *modifies* clauses; the author views this as the most significant difference between Ecstatic and the other, previously mentioned works of Leino.

Another logic for OO programs is that of de Boer [dB91, dB99]. This is proved to be both sound and complete with respect to its denotational semantics, but it appears that this is possible in part because of a non-standard interpretation, in assertions, of universal quantification, which requires a non-standard definition of syntactic substitution. In particular, for x a variable (ranging over objects), $P \stackrel{\text{def}}{=} \forall x.\phi(x)$ does not imply $\phi(e)$ for all objects e, but only for those objects e which exist (i.e., have already been created) in a state which satisfies P. Furthermore, the features of the underlying language appear limited: for example, methods are non-reentrant, and in particular, cannot be recursive; and also, there is no subtyping.

The logic of von Oheimb [Ohe01] allows us to reason about programs written in a fragment of Java. In an exercise reminiscent of Kleymann's, he finds machine proofs of both soundness and correctness using Isabelle/HOL. However, when reasoning about method invocation, we must be able to look up the method bodies of all candidate method implementations (since in Java we have dynamic dispatch of methods and so we do not know, a priori, what code will actually be executed). In particular, this means the logic is non-modular since by extending the program (e.g., by creating a new subclass) the proof may no longer be valid.

Müller's thesis [Mül01] presents a formalism allowing one to verify programs of Mojave, a Java-like language, in a modular fashion. Modular verification (and programming) is considered at the methodological level, which is beyond the scope of AL. A crucial goal is the ability to verify fragments of a whole program in isolation. To this end, a type system is introduced to constrain possible runtime aliasing to within program fragments.

From the point of view of function and purpose, the program logic in loc. cit., which is based on that of Poetzsch-Heffter and Müller detailed in [PHM99], is most related to AL. We see that subtyping induces further proof obligations: in particular, if S is a subtype of T and we want to show $\{P\}\, T{:}m\, \{Q\}$ for (virtual) method $T{:}m$, then we are certainly obliged to show $\{P\}\, S{:}m\, \{Q\}$ (and this is also the case in von Oheimb's logic, albeit in a different notation). (If we consider the subspecification relation of AL as subtyping, since it is behavioural—in the

sense of [LW94]—such proof obligations are automatically satisfied because of the formulation of the object construction rule.)

Now, if we prove a program correct in (Poetzsch-Heffter and) Müller's logic, and subsequently extend it by adding a new subclass, again, like in the case of von Oheimb's logic, the proof may no longer be valid. However, in loc. cit., Müller—and, in [PHM99], Poetzsch-Heffter also—make an explicit distinction between *virtual methods* $T{:}m$ and *method implementations* $T@m$, and formulate the rules in such a way that the extra proof obligations are required only in subderivations concluding in $\{P\}\,T{:}m\,\{Q\}$ (i.e. statements about virtual methods), and not those concluding in $\{P\}\,v = w.T{:}m(p)\,\{Q\}$ (i.e. statements about method invocations). In particular, this allows them to verify programs in a modular fashion by providing a relatively straightforward, algorithmic description of how to determine what proof obligations are required for the proof to remain valid. In contrast, there is no distinction between virtual methods and their implementations in von Oheimb's logic of [Ohe01], and, in particular, extra proof obligations arise in derivations for method invocations. However, in a later logic [ON02], von Oheimb and Nipkow factor the rule for method invocation by incorporating virtual methods, à la Poetzsch-Heffter and Müller. Thus, it is likely that the extra proof obligations resulting from subclassing can also be determined in this later logic, using a similar algorithm.

1.4.3 Computer-assisted verification

The works mentioned so far mainly address the theoretical foundations of program verification: formulations of logics suitable for verifying programs. However, in practice, such logics are cumbersome to use on anything other than small, illustrating examples. This thesis aims to narrow the gap between such theoretical foundations and verification in practice by using computer assistance.

Although in Kleymann's and von Oheimb's work we see implementations of program logics, in those cases, computer assistance was used primarily as an aid to check metatheoretical properties of program logics, as opposed to verifying specific programs (which is the topic of this thesis). The latter is also possible

in these implementations, but, as Kleymann and also von Oheimb state, this is beyond the scope of their work.

We see in the work of Gordon [Gor88], Homeier [Hom95, HM94], Jacobs and Huisman [Hui01, HJ00], and Leino et al. (ESC) [DLNS98] implementations of theory for verification of specific programs. We relate these works to this thesis in Sections 3.1 and 5.1.

1.5 Overview of this thesis

We start with an overview of AL, the logic of Abadi and Leino, in Chapter 2, highlighting those areas that the author believes are interesting in relation to this thesis. The first part of Chapter 3 presents a scheme allowing us to embed AL into a higher-order logic theorem prover by way of introducing axioms: this exercise allows us to automatically inherit machine-checkable versions of proofs in AL. The second part justifies the correctness of the implementation by way of constructing a model of higher-order logic in which we can check our axioms. Chapter 4 describes a type inference algorithm for AL which is a crucial component of the *verification condition generator* (VCG) algorithm presented in Chapter 5. We provide an account of a prototype implementation of the VCG and examples that have been attempted with it, in Chapter 6. Finally we describe further work and conclude the thesis in Chapter 7.

This thesis can be read in different ways. For example, though type inference is crucial to the VCG, we only use it as a black box, i.e. the precise details of the algorithm are not important, and so the reader may read Chapter 5 without reading Chapter 4. Similarly, neither the type inference nor the VCG algorithms depend on Chapter 3. Alas, since this thesis builds on AL, it is recommended that the reader is familiar with AL, or at least reads Chapter 2 before reading the rest of the chapters.

Chapter 2

The logic of objects of Abadi and Leino

Subsequent chapters will concentrate on how to use AL to verify programs in a feasible manner. So we concentrate, in this chapter, on AL itself, providing a summary of what it does and provides, and also attempt to highlight those properties that make it interesting. The reader who is interested in the details of what is detailed here is advised to look up the loci classici [AL97, AL98]. In this chapter, we do not discuss topics significantly beyond what is covered in loc. cit.

Logic AL provides a programming language, syntax-directed operational semantics, a type system and verification rules, which we also refer to as axiomatic semantics. We will introduce, discuss and summarise each of these in turn.

2.1 Programming language

The language of AL exists as a vehicle on which we can build and study its verification rules. The primary objective is to answer questions regarding how OO features affect verification. To keep the metatheory as manageable as possible, the syntax is deliberately kept small, yet keeping those OO features that we are interested in.

Logic AL provides OO features such as objects, instance variables (fields),

(recursive) methods with an implicit *self* argument; however it does not feature classes (and hence inheritance) nor method update. Objects are thus created directly and not from classes, and in this sense, the language is often referred to as a *prototype-based language* [AC96] (versus *class-based*).

We now simultaneously introduce the language syntax and informal operational semantics. The operational semantics, reproduced from locus classicus, is defined formally in Table 2.1. We introduce the following definitions to allow us to describe the abstract interpreting machine. The interpreter has a *store* and a *stack*, which are modified during program execution. We assume we have the following sets: locn of object names, which can be considered as references, or store locations; fname of field names; mname of method names; var of variables; val of values, which are constants (such as the two booleans) and object names; and prog of syntactic programs. Now we define the collection of *stacks*, *closures* and *stores* as follows.

$$\begin{aligned}\mathsf{stack} &\stackrel{\text{def}}{=} \mathsf{var} \rightharpoonup \mathsf{val}\\ \mathsf{closure} &\stackrel{\text{def}}{=} \mathsf{var} \times \mathsf{prog} \times \mathsf{stack}\\ \mathsf{store} &\stackrel{\text{def}}{=} \mathsf{locn} \rightharpoonup ((\mathsf{fname} \rightharpoonup \mathsf{val}) \times (\mathsf{mname} \rightharpoonup \mathsf{closure}))\ .\end{aligned}$$

Thus a stack is a partial function mapping variables to values. A closure is a triple consisting of the program text of a method body, its special variable denoting *self*, and a stack. A store is a partial function mapping a location to a pair of partial functions: one mapping field names to values; and the other mapping method names to closures.

The first language construct we consider is the let binding:

$$let\ x{=}a\ in\ b$$

which introduces the variable x. (The only other way to introduce variables is through the object construct, which is explained later.) The abstract interpreter executes this program, by first executing a and then executing b. However before b is executed, the stack is augmented so that x is mapped to the value resulting from evaluating a. Executing a modifies the store of the machine, and b is exe-

cuted starting with the modified store. The return result of executing the whole program is the return result of the execution of b.

At this point we note that variables are immutable in the sense that they are introduced and have a value which cannot later be changed. This is different from variables in languages such as Java and C++, and closer to variables in λ-calculi.

Another language construct is that of object creation:

$$[f_i{=}x_i{}^{i=1..k}, m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}]$$

which creates a new object. Note that, to create an object, we explicitly provide the code (b_j) for the method bodies, and initial values (x_i) for the fields. In this way, we can create objects without classes. In the definition of method m_j, we notice a stylised sigma ς. This is nothing more than a binder (like λ is a binder in lambda calculus), binding the variable y_j, which represents *self*. Through this bound variable, the method body can access its sibling fields and methods. To execute this program, the machine finds an unused location h, say, and then updates its store σ so that $\sigma(h, f_i) = S(x_i)$, where $S(x_i)$ is the value of variable x_i, as determined by the stack S. It also updates its store so that $\sigma(h, m_j) = (y_j, b_j, S)$ where S is the current stack. The return result of an object creation is the location of the newly created object. This is a general feature of the programming language: all objects are passed by reference, i.e. we pass the locations of objects, not records. As a consequence, we can also model the phenomenon of aliasing.

The last important language construct is that of method invocation:

$$x.m()$$

which invokes method m of object x. What this means, operationally, is the machine looks up, in the store σ, the closure (y, b, S') and then proceeds to execute the program b using the stack S' augmented so that y is mapped to $S(x)$. After executing b, the interpreter reverts to stack S. Of note is the fact that methods have exactly one parameter: the implicit self variable. This comes back to the desire to have a simple, small syntax. Invoking a method with several arguments

can be simulated by using extra fields, to which we assign before invoking the method.

The remaining constructs are if-then-else, field selection, field update, variables and constants which are implemented in a standard way. The reader may be alarmed that there is no while construct: this has been deliberately omitted, again, to keep the syntax small. We do not lose expressivity since we can write programs with loops using method recursion. Furthermore, verification of while programs is well understood.

2.1.1 Examples

Here we present some examples of programs written AL. As in locus classicus, we liberally use syntactic sugar. For example, $a; a'$ (sequential composition) is an abbreviation for $let\ x{=}a\ in\ a'$ where x is chosen to be a variable not occurring free in a, a'. Also, we allow constants to be used in places where, in the strict syntax, only variables are allowed (e.g. $[f{=}true]$ and $y.f{:=}false]$). This is encoded using the let construction as explained in locus classicus.

The first example demonstrates the phenomenon of aliasing. Consider

$$
\begin{array}{l}
let\ x{=}[f{=}true]\ in \\
let\ y{=}x\ in \\
\quad y.f{:=}false; \\
\quad x.f
\end{array}
$$

Note that *false*, *true* are syntax for the boolean values $\mathtt{ff}, \mathtt{tt}$. We first create an object with a field initialised to $\mathtt{tt}$, naming (the location of) this object x. We then introduce another variable y which takes the same value as x, namely the location of the previously created object. We then modify f through variable y. When we look up f through x, we observe that its value is now $\mathtt{ff}$. This is because x and y are both references to the same object.

The next example (example 3 of locus classicus), demonstrates how we can use recursive methods to write programs with looping behaviour. Euclid's algorithm

$$\frac{S(x) = v}{\sigma, S \vdash x \rightsquigarrow v, \sigma} \quad \text{(os_var)}$$

$$\frac{}{\sigma, S \vdash \mathit{false} \rightsquigarrow \mathtt{ff}, \sigma} \qquad \frac{}{\sigma, S \vdash \mathit{true} \rightsquigarrow \mathtt{tt}, \sigma} \quad \text{(os_const)}$$

$$\frac{S(x) = \mathtt{ff} \quad \sigma, S \vdash a_1 \rightsquigarrow v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \rightsquigarrow v, \sigma'} \qquad \frac{S(x) = \mathtt{tt} \quad \sigma, S \vdash a_0 \rightsquigarrow v, \sigma'}{\sigma, S \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \rightsquigarrow v, \sigma'} \quad \text{(os_cond)}$$

$$\frac{\sigma, S \vdash a \rightsquigarrow v, \sigma' \quad \sigma', S[x \mapsto v] \vdash b \rightsquigarrow v', \sigma''}{\sigma, S \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b \rightsquigarrow v', \sigma''} \quad \text{(os_let)}$$

$$\frac{\begin{array}{l} S(x_i) = v_i \ {}^{i=1..k} \quad h \notin \mathsf{dom}(\sigma) \quad h \in \mathsf{locn} \\ \sigma' = \sigma[h \mapsto (f_i \mapsto v_i \ {}^{i=1..k}, m_j \mapsto \langle\varsigma(y_j)b_j, S\rangle \ {}^{j=1..\ell})] \end{array}}{\sigma, S \vdash [f_i = x_i \ {}^{i=1..k}, m_j = \varsigma(y_j)b_j \ {}^{j=1..\ell}] \rightsquigarrow h, \sigma'} \quad \text{(os_obj)}$$

$$\frac{S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(f) = v}{\sigma, S \vdash x.f \rightsquigarrow v, \sigma} \quad \text{(os_fsel)}$$

$$\frac{\begin{array}{l} S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(m) = \langle\varsigma(y)b, S'\rangle \\ \sigma, S'[y \mapsto h] \vdash b \rightsquigarrow v, \sigma' \end{array}}{\sigma, S \vdash x.m() \rightsquigarrow v, \sigma'} \quad \text{(os_minv)}$$

$$\frac{\begin{array}{l} S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(f) \text{ is defined} \\ S(y) = v \quad \sigma' = \sigma[h \mapsto (\sigma(h)[f \mapsto v])] \end{array}}{\sigma, S \vdash x.f{:=}y \rightsquigarrow h, \sigma'} \quad \text{(os_fupd)}$$

Table 2.1: Operational semantics. Note, we write $\langle\varsigma(y)b, S\rangle$ for the triple $(y, b, S) \in$ closure.

for computing the greatest common divisor (gcd) can be written as follows.

$$
\begin{array}{ll}
\textit{let } x= [& f = 1, g = 1, \\
 & m = \varsigma(y)\ \textit{if } y.f < y.g \textit{ then} \\
 & \qquad y.g{:=}y.g - y.f; \\
 & \qquad y.m() \\
 & \quad \textit{else if } y.g < y.f \textit{ then} \\
 & \qquad y.f{:=}y.f - y.g; \\
 & \qquad y.m() \\
 & \quad \textit{else } y.f \qquad\qquad] \\
\textit{in } & x.f{:=}426; \\
 & x.g{:=}792; \\
 & x.m()
\end{array}
$$

This program computes the gcd of 426 and 792.

Finally, we give an example of a program that is quite unusual. Though the expressivity it demonstrates does not lend itself to necessarily useful programming patterns, we hope it hints at some of the difficulties that we must face when working with the metatheory of this language.

$$\textit{let } u{=}[f{=}[m{=}\varsigma(y)\textit{true}]] \textit{ in } (u.f{:=}[m{=}\varsigma(y)u.f.m()]; u.f.m())$$

Here we create an object named u with one field f initialised to an object with one method m which computes something trivial. Then we update f of u with a new object whose method m invokes m of $u.f$. Then invoking m of $u.f$ puts us into an endless loop, and we have done this without using a loop construct nor method recursion. (We do have a recursive method, but bound at runtime.)

The reason why we can write such a program is due to the fact we have a mutable, higher-order store: a store that contains higher-order entities, in our case methods. Looking at the program more closely, it appears that the field update allows us to by-pass syntactic scoping constraints: we could not have written

$$\textit{let } u{=}[f{=}[m{=}\varsigma(y)u.f.m()]] \textit{ in } u.f.m()$$

since u is not in scope in the method body. In fact, we can modify this example and write any recursive method in this style.

2.1.2 Types

In locus classicus, Abadi and Leino present a first-order type system for the programming language. Since they later show that the axiomatic semantics (verification rules) subsumes the functionality of the type system, we shall describe the verification rules immediately.

2.2 Axiomatic semantics

In AL, we have verification rules allowing us to derive judgements of the form

$$x_1{:}A_1, \dots, x_n{:}A_n \vdash a : A :: T$$

where A and A_i $(i = 1..n)$ are *specifications*, and T is a *transition relation*. Such a judgement is interpreted informally as follows: assuming variables $x_1 \cdots x_n$ satisfy specifications $A_1 \cdots A_n$, executing program a, provided it terminates, computes a value satisfying specification A, and modifies the store in a way satisfying transition relation T. Note that AL deals only with partial correctness.

Transition relations describe the dynamic behaviour of programs. In the syntactic view taken by Abadi and Leino, a transition relation is defined to be a first-order formula with free occurrences of program variables (subject to scope) and also pseudovariables r denoting the result of a computation, $\grave{\sigma}$ and $\acute{\sigma}$ denoting the *(flat) store* before and after execution. (Since they are working in first-order logic, these functions are total, and since flat stores are partial, they also introduce predicate symbols $\grave{alloc}$ and $\acute{alloc}$, which denote the domains of $\grave{\sigma}$ and $\acute{\sigma}$ respectively.) A flat store is precisely the component of a store that describes only the fields of objects: transition relations can only be expressed using the values of the fields of objects, not the method closures. We define

$$\mathsf{fstore} \stackrel{\text{def}}{=} \mathsf{locn} \rightharpoonup (\mathsf{fname} \rightharpoonup \mathsf{val}) \ .$$

Taking a more semantic view, it is also convenient to consider a transition relation as a relation over the return value and flat store before and after, and also n values, where n is the number of variables in scope. For example, in the judgement above, we can consider $T \subseteq \mathsf{val}^n \times \mathsf{val} \times \mathsf{fstore} \times \mathsf{fstore}$.

Specifications describe the static properties of a value. They are exactly object types further endowed with transition relations describing the behaviour of methods on invocation. The syntax for the specifications of values of base type is the same as that for their types, since they contain no methods. Writing U for transition relations, the syntax of an object specification is

$$[f_i{:}A_i{}^{i=1..k}, m_j{:}B_j{::}U_j{}^{j=1..\ell}] \ .$$

Here, U_j is the transition relation describing the dynamic behaviour of method m_j and B_j is the specification of its return value. Similarly, A_i is the specification of field f_i.

We take this opportunity to point out: (1) the static specification of a value also contains the dynamic specifications of its methods; and (2) pairs $x_j{:}A_j$ appearing on the left of the turnstile take the rôle of formal assumptions. These two properties, plus Rule (Let) introduced later (which can be considered to be a cut), make AL modular in the sense that it is possible to break a large program into smaller programs (that are likely to have free variables) which can be separately verified. We elaborate further on this when we introduce the verification rules for object creation, method invocation and let.

Just as subtyping allows us to type check more programs, we have an analogous notion for specifications. We define the reflexive subspecification relation as follows: we write

$$[f_i{:}A_i{}^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}U_j{}^{j=1..\ell}] <: [f_i{:}A_i{}^{i=1..k'}, m_j{:}\varsigma(y_j)B'_j{::}U'_j{}^{j=1..\ell'}]$$

precisely when $k' \leq k$, $\ell' \leq \ell$, and both $B_j <: B'_j$ and $U_j \subseteq U'_j$ for $j = 1..\ell'$. We read $A <: A'$ as A is a subspecification of A'. In words: one specification is a subspecification of a syntactically longer specification; furthermore, subspecification is covariant along methods. Observe that a subspecification is a stronger specification, in the sense that fewer programs satisfy it.

The possibility of covariant method specifications is perhaps more important than covariant method return types in subtyping. Together with the formulation of the object creation rule, it allows more programs to be verified.

We introduce the first verification rule which uses the subspecification relation to its full effect. The subsumption rule:

$$\frac{E \vdash a : A :: T}{E \vdash a : A' :: T'} \quad \text{provided} \quad A <: A' \text{ and } T \subseteq T' \ .$$

This is a weakening rule: to derive a judgement $E \vdash a : A' :: T'$, it suffices to derive a stronger judgement.

At this point, it is perhaps useful to compare AL to the more familiar Hoare logic. Table 2.2 gives the comparison in table form. Here we consider Hoare logic for a language with side-effecting expressions. We follow the convention as used by von Oheimb [Ohe01], and attributed to Kowaltowski [Kow77], of using a distinguished variable to denote the result (like r in our transition relations). Transition relations are closest to pairs of pre-/post-conditions (assertions), since they describe how the store changes during the execution of a program. There is no direct analogy to specifications in a standard presentation of Hoare Logic for a while language without procedures. Finally, the subsumption rule is analogous to the rule of consequence. And since we have static specifications also, our rule weakens those too.

2.2.1 Verification rules

In addition to the subsumption rule, we have a rule for each program construct. To streamline the presentation, we introduce predicate symbols Res, T_{obj}, T_{fsel} and T_{fupd} defined in Table 2.3. For example, we use the predicate symbol Res for transition relations that do not change the store but simply set the return value register.

For example, the rule for let can be formulated as

$$\frac{E \vdash a : A :: T \quad E, x{:}A \vdash b : B :: U}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : A' :: T'} \quad \left\{ \ T;_x U \subseteq T' \ \right\} \qquad \text{(Let)}$$

where $B <: A'$, and $T;_x U$ is defined to be the transition relation

$$T;_x U \stackrel{\text{def}}{=} \exists \check{\sigma}, \check{alloc}, x.T[x/r, \check{\sigma}/\acute{\sigma}, \check{alloc}/\acute{alloc}] \wedge U[\check{\sigma}/\grave{\sigma}, \check{alloc}/\grave{alloc}] \ .$$

	Hoare logic	AL
Judgements	$\{p\}\,c\,\{q\}$	$E \vdash a : A :: T$
Dynamic specifications	assertions $\langle p, q\rangle \in (\mathrm{Pow}(\mathsf{fstore} \times \mathsf{val}))^2$	transition relations $T \in \mathrm{Pow}(\mathsf{val} \times \mathsf{fstore}^2)$
Static specifications	—	A
Weakening rule	$\dfrac{\{p\}\,c\,\{q\}}{\{p'\}\,c\,\{q'\}}$, provided $p' \subseteq p$, $q \subseteq q'$, (rule of consequence)	subsumption

Table 2.2: AL compared to Hoare logic: judgements, dynamic specifications, static specifications, and weakening.

$$
\begin{aligned}
Res(e) &\stackrel{\mathrm{def}}{=} r = e \wedge \forall x, f.\grave{\sigma}(x,f) = \acute{\sigma}(x,f) \wedge \forall x.\grave{alloc}(x) \equiv \acute{alloc}(x) \\
T_{\mathsf{obj}}(x_1 \cdots x_n) &\stackrel{\mathrm{def}}{=} \neg\grave{alloc}(r) \wedge \acute{alloc}(r) \wedge \\
&\qquad \forall z.z \neq r \supset \grave{alloc}(z) \equiv \acute{alloc}(z) \wedge \\
&\qquad \acute{\sigma}(r, f_1) = x_1 \wedge \cdots \wedge \sigma(r, f_n) = x_n \wedge \\
&\qquad \forall z, w.z \neq w \supset \grave{\sigma}(z,w) = \acute{\sigma}(z,w) \\
T_{\mathsf{fsel}}(x, f) &\stackrel{\mathrm{def}}{=} r = \grave{\sigma}(x,f) \wedge \\
&\qquad \forall x, f.\grave{\sigma}(x,f) = \acute{\sigma}(x,f) \wedge \\
&\qquad \forall x.\grave{alloc}(x) \equiv \acute{alloc}(x) \\
T_{\mathsf{fupd}}(x, f, y) &\stackrel{\mathrm{def}}{=} r = x \wedge \acute{\sigma}(x,f) = y \wedge \\
&\qquad \forall z, w.(z,w) \neq (r,f) \supset \grave{\sigma}(z,w) = \acute{\sigma}(z,w) \wedge \\
&\qquad \forall z.\grave{alloc}(z) \equiv \acute{alloc}(z)
\end{aligned}
$$

Table 2.3: Predicate symbols streamlining presentation of verification rules.

The composition operator $;_x$ is not so different from sequential composition. Its more complicated definition is due, in part, to the fact that transition relations are asymmetric: transition relations can depend on the result after, but not before. Note how (Let) allows us to use assumptions $x{:}A$. In this sense, the rule is analogous to the cut rule in logic.

In locus classicus, transition relations are considered as first-order formulae, and $\grave{\sigma}, \acute{\sigma}$ as function symbols, jso the expression $T;_x U$ is not expressible as a transition relation. To circumvent this problem, the subsumption rule is incorporated into (Let), resulting in what appears to be a less streamlined presentation. On the other hand, by taking our more semantic view of transition relation, we have in effect given ourselves free rein to choose a language for transition relations, and so we may assume that $T;_x U$ is transition relation.

Another important rule is that for field update. The following is an informal formulation

$$\frac{E \vdash x : [\cdots f{:}A \cdots] :: \mathit{Res}(x) \quad E \vdash y : A :: \mathit{Res}(y)}{E \vdash x.f{:=}y : [\cdots f{:}A \cdots] :: T_{\mathsf{fupd}}(x, f, y)} \ . \tag{Fupd}$$

When studying the soundness proof in [AL98], we observe that its formulation is crucial, since it modifies the store in a safe way; by updating $x.f$ with y which has the same specification A, we are preserving the specification of object x. In the soundness proof, this allows us to state an invariance property of the store.

The method invocation rule again illustrates the modularity of AL. If we study

$$\frac{E \vdash x : [m{:}\varsigma(y)B{::}U] :: \mathit{Res}(x)}{E \vdash x.m() : B[x/y] :: U[x/y]} \tag{Minv}$$

we notice that we can derive judgements about method invocations without the need for the program text of the method body. Clearly this is a desirable property towards modular verification.

Of course, we never get anything for free, and the work of verifying the method bodies has to be done at some point. We find that the extra work is required in the rule for object creation:

$$\frac{E \vdash x_i : A_i :: \mathit{Res}(x_i)^{i=1..k} \quad E, y_j{:}A \vdash b_j : B_j :: U_j{}^{j=1..\ell}}{E \vdash [f_i = x_i{}^{i=1..k}, m_j = \varsigma(y_j)b_j{}^{j=1..\ell}] : A :: T_{\mathsf{obj}}(\vec{x})} \tag{Obj}$$

where $A \stackrel{\text{def}}{=} [f_i{:}A_i{}^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}U_j]$. Here we see that to prove the conclusion, we are obliged to prove, for each method m_j, its body b_j satisfies the respective specification and transition relations. In particular, note that we are allowed to use, as an assumption, the fact that y_j satisfies the specification of the whole object. This is necessary to allow us to reason about a method body which uses its sibling fields and methods, or indeed itself. However, it is only sound to assume that y_j has the same (or weaker) specification than the whole object itself. Therefore, covariant method specifications are immensely useful; in their absence, a method that correctly implements its specification may not be verifiable. (For example, to prove a statement P, one often proves a stronger statement P' which allows for an inductive argument. The possibility of strengthening P is crucial in such cases.)

Finally we return to Rule (Let). This can be considered as our cut (or modus ponens) rule, and through this we internalise, in the logic, certain uses of lemmata. Thus, in AL, we prove correctness of each method body once and once only. Alternatively, for example, in Müller's logic [Mül01], when finding a proof (i.e. derivation) we may need to prove (i.e. derive) a method annotation $\{P\}\,T{:}m\,\{Q\}$ in several places. In practice, one introduces a meta-lemma stating this statement is provable (i.e. derivable), but the proof itself has several copies of the proof (i.e. derivation) found in the (meta-level) proof of this meta-lemma.

There are further rules not detailed here. The reader can find the original presentation in locus classicus as well as an the slightly different presentation in Table 5.2 on page 178.

2.3 Metatheory

In locus classicus, we find a type system for the underlying language of AL, and then the verification rules. This gives an explicit demonstration of how the verification rules are a generalisation of the typing rules. We also find two simple propositions which relate typing with verification. Firstly, if one can derive the verification $E \vdash a : A :: T$, then one can also derive the typing $E' \vdash a : A'$

where E', A' are obtained from E, A by deleting all transition relation information. Secondly, and conversely, supposing we can derive the typing $E \vdash a : A$ then we can derive $E' \vdash a : A' :: \mathit{true}$, where now E', A' are obtained from E, A by inserting trivial transition relations, such as *true*. And in this sense, the verification rules subsume the typing rules.

However, the main contribution of AL was the soundness proof available in [AL98]. This states that if we can derive $\emptyset, \emptyset \vdash b \rightsquigarrow v, \sigma'$, that is, executing b starting from an empty stack and store terminates with result v, and we can derive $\vdash b : \mathit{Bool} :: r = \mathtt{tt}$, then in fact $v = \mathtt{tt}$. And similarly for transition relation $r = \mathtt{ff}$.

2.3.1 (In)completeness

Unfortunately, the logic is not complete. In locus classicus, we are presented with the program

$$a \stackrel{\text{def}}{=} \mathit{let}\ y{=}\mathit{true}\ \mathit{in}\ [m = \varsigma(z)y]$$
$$b_1 \stackrel{\text{def}}{=} \mathit{let}\ x{=}a\ \mathit{in}\ x.m()\ .$$

We can use the operational rules to show b_1 evaluates to $\mathtt{tt}$. Completeness would imply that we would also be able to derive $\vdash b_1 : \mathit{Bool} :: r = \mathtt{tt}$. However, a proof with the existing rules requires us to derive

$$y{:}\mathit{Bool}, z{:}[\cdots] \vdash y : \mathit{Bool} :: r = \mathtt{tt}$$

but this is not possible since our assumptions are too weak. (We may only assume that x satisfies specification *Bool*, but not that x in fact has value $\mathtt{tt}$.)

Another incompleteness artefact that the authors mention is that they can only prove that $\mathit{let}\ x{=}[f = \mathit{true}]\ \mathit{in}\ (y.m(); x.f)$ has transition relation $r = \mathtt{tt}$ by assuming an unnecessarily strong specification for y, such as "m does not change field f of any object". Technically this is not an incompleteness issue, if one considers completeness for closed programs. However, it is clearly important for modular verification.

2.3.2 When syntactic sugar leaves a bitter after-taste

Let us consider a practical consequence of the first of these incompleteness issues. The plain syntax of AL is very restrictive. For example, for the field selection construct, we cannot write $a.f$ for an arbitrary program a, we can only write $x.f$ for variable x. A simple solution to this is to introduce *syntactic sugar*. Namely, we allow programs such as $a.f$, but insist that this is merely an abbreviation for $let\ x{=}a\ in\ x.f$.

For verification, we can add further syntactic sugar and compute derived rules for the sugared syntax. For example, we can compute the rule

$$\frac{E \vdash a : [f{:}A] :: T}{E \vdash a.f : A :: T ;_x T_{\text{fsel}}(x, f)}$$

by combining the rules for field selection and let constructs. Furthermore, we can in fact derive the original rule for field selection from this rule, and so we can now dispense with the old rule for field selection altogether.

However, when one tries the same exercise for method invocation and condition constructs, i.e. derive rules for *if* a *then* a_0 *else* a_1 and $a.m()$ for arbitrary programs a, we find that we cannot dispense with the original rules. The desugaring process introduces extra let constructs and we lose information. To avoid even further incompleteness, we must keep both the original and derived rules.

For example, assuming $a.m()$ is an abbreviation for $let\ x{=}a\ in\ x.m()$ (assuming x is a fresh variable). By composing rules together, we obtain the following derived rule

$$\frac{E \vdash a : [m{:}\varsigma(y)B{::}U] :: T}{E \vdash a.m() : A' :: T ;_y U} \quad \{B <: A'\} \ .$$

Suppose

$$E \vdash z : [m{:}\varsigma(y)B{::}U] :: Res(z) \ .$$

To prove

$$E \vdash z.m() : B[z/y] :: U[z/y] \ ,$$

we require to prove $B <: B[z/y]$. In general, this is not true.

2.4 Summary

Logic AL provides a (formally proven) sound axiomatic semantics for an OO language with imperative (versus functional) semantics, featuring aliasing and recursive methods. Furthermore, it allows for modular verification: we may derive verification statements about programs with free variables by assuming their specifications. Not only does this allow us to verify a large program as several smaller components, but it also allows us to use formal verification on only the critical parts of a larger system.

We further discuss incompleteness and its other weaknesses in Chapter 7.

Chapter 3

AL in higher-order logic

In this chapter, we describe a first step towards feasible program verification using AL: namely, we demonstrate how AL can be embedded into a higher-order logic theorem prover. We use the underlying higher-order logic of the theorem prover, both as a framework logic (allowing us to formulate the axioms and to build proof trees), as well as the assertion language for AL. In this sense the underlying logic of the theorem prover plays a dual rôle. The benefits of this embedding exercise are two fold: firstly and primarily, it allows us to check *proofs* of verification judgements, and, secondly, it allows us to find proofs of verification judgements.

Rather than describing an embedding of AL into a specific theorem prover such as LEGO, we work with a general metalanguage which is a fragment of the languages of higher-order logic theorem provers such as LEGO, COQ, PVS and Isabelle/HOL. Thus the technique described here is generic.

From studying the literature on embedding of program languages and logics into theorem provers, one finds that there are broadly two approaches. On one hand, there is the shallow embedding of a language, whereby the syntactic aspects of programs are not represented as objects in the encoding, but programs are represented directly by their denotations. (See [Gor88, Hui01] for examples of such embeddings.) The advantage of this approach is that there is less overhead. However, as explained in [NvOP00], when trying to prove a metatheoretical property that is expressed syntactically, one often requires extra definitions. The shallow embedding amounts to providing a denotational semantics for the programming

language.

Suppose we want to give a shallow embedding of AL programs. Thus we must find a denotational semantics. Let us consider denotations of programs as functions that map stores to pairs of stores and (result) values, viz,

$$P \stackrel{\text{def}}{=} \mathsf{store} \rightarrow (\mathsf{store} \times \mathsf{val}) \ .$$

Thus the corresponding definition of closure is

$$\mathsf{closure} \stackrel{\text{def}}{=} \mathsf{var} \times P \times \mathsf{stack} \ .$$

Of course, the definition of store and stack can stay as they are:

$$\mathsf{store} \stackrel{\text{def}}{=} \mathsf{locn} \rightharpoonup ((\mathsf{fname} \rightharpoonup \mathsf{val}) \times (\mathsf{mname} \rightharpoonup \mathsf{closure}))$$
$$\mathsf{stack} \stackrel{\text{def}}{=} \mathsf{var} \rightharpoonup \mathsf{val} \ .$$

Now writing these as one equation, we find that we have to solve

$$\mathsf{store} = \cdots \times (\cdots \rightharpoonup \cdots \mathsf{mname} \rightharpoonup (\mathsf{var} \times (\mathsf{store} \rightarrow (\mathsf{store} \times \mathsf{val})) \times \mathsf{stack}))) \ .$$

And since store appears on the right hand side of this equation in both a negative and positive position, to be able to encode store, we need an embedding of a theory of domains, or similar. Of course different denotations may be possible, but in any case, a shallow embedding is less appealing for our purposes since we want to encode derivability of verification judgements by recursion over syntax.

On the other hand, there is the deep embedding of a language, whereby the syntax is explicitly encoded in the theorem prover. (See [Kle98, Hom95, NvOP00] for examples where programming languages are given deep embeddings.) In this case, it is possible to prove, directly, metatheoretical properties by induction over program syntax. For example, type safety of Java is mechanically proved in [NvOP00]. Also, one can work from operational semantics (which typically is defined recursively over syntax). However when working with a deep embedding of a first-order syntax with bound variables—such as that underlying AL—one must explicitly encode operations such as substitution and fresh variables, which again adds extra overhead.

However, there is a way to maintain the benefits of both of the above approaches. Our embedding uses *higher-order abstract syntax* (HOAS) which delegates the task of dealing with binding of variables to the metalanguage itself. Furthermore, since the syntax is explicitly represented in the embedding, it is possible to use syntactically defined devices, such as operational and axiomatic semantics.

As a concrete example, consider

$$a \equiv \mathit{true}$$
$$b \equiv \mathit{if}\ x\ \mathit{then}\ \mathit{false}\ \mathit{else}\ \mathit{true}$$
$$c \equiv \mathit{let}\ x{=}a\ \mathit{in}\ b\ .$$

Suppose $t, f : P$ are the constructors for the constants, and $i : V \times P \times P \to P$ is the constructor for conditional. In the HOAS-style, we introduce a constructor $l : P \times (V \to P) \to P$ for the let construct. We then define the encoding $\ulcorner b \urcorner$ of b to be $i(x, f, t)$, which we observe has free occurrences of x. We note that

$$x{:}V \rhd \ulcorner b \urcorner : P\ ,$$

that is, assuming x has type V, $\ulcorner b \urcorner$ has type P, so

$$\rhd \lambda x. \ulcorner b \urcorner : V \to P\ ,$$

thus we can define $\ulcorner c \urcorner \stackrel{\text{def}}{=} l(t, \lambda x. \ulcorner b \urcorner)$. Note that x is thus bound in $\ulcorner c \urcorner$.

In contrast, in a first-order deep embedding, one introduces a constructor $l : V \times P \times P \to P$ and define $\ulcorner c \urcorner \stackrel{\text{def}}{=} l(x, t, i(x, f, t))$. Note that the variable x occurs free in $\ulcorner c \urcorner$. Thus one requires more care, for example to then encode $\mathit{let}\ x{=}d\ \mathit{in}\ c$ for some d, since $l(x, \ulcorner d \urcorner, \ulcorner c \urcorner)$ results in variable-capture.

The same decision (deep vs. shallow) can be made again for the assertion language when embedding a program logic. In our case, we note that, though the language of transition relations is defined in locus classicus [AL98] to be first-order logic (plus some extra function symbols), the soundness of the verification rules does not depend on derivability in first-order logic. Thus, there appears to be few advantages gained from explicitly embedding the syntax of transition

relations, and so we choose a shallow, *direct embedding.* In the context of Hoare logic, we note that shallow embeddings of assertions are used in [Gor88, Kle98, Hui01, Ohe01].

A shallow embedding of transition relations allows us to directly use the features of the underlying theorem prover to discharge the side conditions of verification rules. In particular, we can immediately use the high-level proof tactics provided in provers such as PVS and Isabelle/HOL. In contrast, a deep embedding of transition relations requires us to also encode the axioms and rules for first-order (or other) logic. Also, the shallow embedding of transition relations allows us side-step the issue of completeness of the assertion logic. The resulting completeness problem is in fact that of relative correctness in the sense used by Cook [Coo78].

To allow us to justify our encoding, we give the metalanguage a non-standard interpretation using functor categories, in the style of [Hof99]. More precisely, we construct a category $\mathcal{D}$, and assign to each type τ an object $[\![\tau]\!]$ of $\mathcal{D}$, and to each constant $c : \tau$, a morphism $[\![c]\!] : \mathbf{1} \to [\![\tau]\!]$. We then show that the interpretation admits a functor $\mathbf{Pred} : \mathcal{D}^{\mathrm{op}} \to \mathbf{Poset}$ which satisfies specific properties (namely those of a tripos [HJP80, Pit02]) and thus the interpretation models the usual axioms and rules for higher-order logic.

In particular, in this interpretation, programs correspond to first-order programs allowing us to interpret the verification judgement as an auxiliary derivability relation which can be defined, as usual, by induction over the first-order syntax. We then relate the auxiliary relation to the operational semantics by giving a proof in the spirit of the soundness proof as found in locus classicus.

3.1 Related work

On the subject of embedding program logics in theorem provers, we believe that one of the first presentations was that of Gordon [Gor88]. In loc. cit., programs are given a shallow embedding and then the Hoare logic rules are *derived* as theorems. From a pragmatic point of view, this has many desirable consequences. For

example, the user may freely mix semantic reasoning and reasoning with Hoare logic. This technique can be extended further, since rules for other logics can implemented, such as VDM, Dynamic Logic and Dijkstra's weakest preconditions, and these logics can be freely mixed together in proofs.

In Gordon's presentation, programs are given denotations as relations between initial and final states. Huisman in [Hui01] also uses a shallow embedding of programs, but in this case uses a coalgebraic denotation of programs (of a fragment of Java). Furthermore, translation of concrete programs to their denotations in the theorem prover is automated using the LOOP tool of Nijmegen [vdBJ01, Hui01]. Similarly, Huisman *derives* the rules of a Hoare logic as theorems directly from the semantics.

Gordon also leverages the tactics facility of HOL and defines a verification condition generator (VCG) which automatically applies the Hoare logic rules for annotated programs. (Note that we describe a similar exercise for AL in Chapter 5.) As what can be seen as an extension of Gordon's work, Homeier [Hom95, HM94] implements a deep embedding of a while-language. The deep embedding not only allows him to state as a theorem (vs. metatheorem in Gordon's case), the Hoare rule for assignment (which cannot be expressed in Gordon's approach), but also allowed the VCG to be encoded, and proved correct. Homeier describes his VCG as a "trustworthy tool for trustworthy programs".

Note that in the approaches of Huisman and Gordon, soundness is guaranteed since the rules are derived theorems. Furthermore, completeness is not a theoretical issue since the user can always resort to the semantics if necessary—there is no completeness issue. Purists may argue that this is not in the spirit of Hoare logic, to which one can concede that the theorems of Gordon and Huisman are merely modelled after the rules of Hoare logic. Pragmatically, completeness is useful to minimise the need to use the lower-level semantics.

Kleymann, in [Kle98], embeds Hoare logic and VDM into LEGO, using deep embeddings of the program syntax and the logic rules, but a shallow embedding of the assertion language. He then uses the embedding to construct mechanically-verifiable proofs of soundness and completeness.

Similarly, von Oheimb in [Ohe01] describes a deep embedding of (a fragment of) Java in Isabelle/HOL. He defines an operational semantics from which he proves soundness of the type system. He then formulates a Hoare logic for Java, which he proves to be not only sound, but also complete. The rules of the Hoare logic are given a deep embedding, but, like in Kleymann's work, the assertion language is given a shallow embedding.

Note that in the works of Kleymann and von Oheimb, the motivation is to prove metatheoretical properties, namely soundness and completeness, of Hoare Logic. On the other hand, Huisman's use of Hoare logic was primarily as a device for easing the task for proving correctness statements. In this sense, our motivation is closer to hers. An important difference of our approach to those previously mentioned is that we use a HOAS encoding of the program syntax.

The work of Gordon, Huisman, Kleymann and von Oheimb, like ours, also use the logic of the theorem prover both as a framework logic and the logic for assertions. In contrast, Homeier uses a deep embedding of the assertion language, and in that sense, he uses the theorem prover (HOL) only as a framework logic.

3.2 Metalanguage

We now introduce the metalanguage which is used to present the implementation of the program logic. The metalanguage is based on a higher-order simply-typed lambda calculus.

We have the following types

$$\begin{array}{rcl} \sigma, \tau & ::= & \mathsf{Prop} \mid \sigma \times \tau \mid \sigma \to \tau \mid \mathsf{Rcd}^{\tau}_{\sigma} \mid A \\ A & ::= & \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{Var} \mid \mathsf{Fn} \mid \mathsf{Mn} \mid \mathsf{Val} \mid \mathsf{Sp} \mid \mathsf{Prg} \ . \end{array}$$

Here Prop is the type of propositions, $\sigma \times \tau$ is the product type, $\sigma \to \tau$ is function space, $\mathsf{Rcd}^{\tau}_{\sigma}$ is the type of σ-indexed records and A represents the base types. For base types, we have the natural numbers nat, booleans bool, variables Var, field names Fn, method names Mn, values (results) Val, specifications Sp and program terms Prg. Records in $\mathsf{Rcd}^{\tau}_{\sigma}$ are intended to be interpreted as partial functions

with finite domain from σ into τ, but we require only restricted combinations of σ and τ: namely $\mathsf{Rcd}_{\mathsf{Fn}}^{\mathsf{Var}}$, $\mathsf{Rcd}_{\mathsf{Fn}}^{\mathsf{Sp}}$, $\mathsf{Rcd}_{\mathsf{Mn}}^{\mathsf{Var}\to\mathsf{Prg}}$ and $\mathsf{Rcd}_{\mathsf{Mn}}^{(\mathsf{Val}\to\mathsf{Sp})\times(\mathsf{Val}\to\mathsf{TR})}$.

The type of transition relations TR is defined to be $\mathsf{Val} \to (\mathsf{Val} \to \mathsf{Fn} \to \mathsf{Val})^2 \to (\mathsf{Val} \to \mathsf{Prop})^2 \to \mathsf{Prop}$, where we write $\cdots \to A^2 \to \cdots$ as shorthand for $\cdots \to A \to A \to \cdots$. Thus $T : \mathsf{TR}$ is a predicate over a return value $r : \mathsf{Val}$, and two stores $\grave{\sigma}, \acute{\sigma} : \mathsf{Val} \to \mathsf{Fn} \to \mathsf{Val}$. Note that we already have more typing information in the transition relation logic than there is in that of locus classicus; within our transition relation logic, for example, we have types for field names Fn, method names Mn and values Val; the transition relation logic of locus classicus is untyped first-order logic. We could introduce even more typing information, for example, by introducing a type Locn of locations, which would give the natural type for stores $\mathsf{Locn} \to \mathsf{Fn} \to \mathsf{Val}$. Extra types makes some things easier but can also make some things more difficult. The formalisation without a type Locn suffices as an embedding of AL, and it is unclear whether its addition is truly useful.

We have the following terms

$$e \quad ::= \quad x \mid ee' \mid \lambda x.e \mid c \ .$$

That is, terms are built from variables x, applications ee', abstractions $\lambda x.e$ and constants c (each of type τ^c) which we describe later. We sometimes annotate abstractions—for example, $\lambda x^\tau.e$ —to emphasise that variable x has type τ.

For *context* $\Gamma = x_1{:}\tau_1, \ldots x_n{:}\tau_n$, term e and type τ, the judgement

$$\Gamma \rhd e : \tau$$

informally means that assuming x_i has type τ_i (for $i = 1..n$), term e has type τ. Also, for term P, provided $\Gamma \rhd P : \mathsf{Prop}$, then the judgement

$$\Gamma \rhd P$$

informally means that P is true. The first of these two judgements are formalised

by the following standard rules for simply-typed lambda calculus:

$$\frac{}{\Gamma, x{:}\sigma \triangleright x : \sigma} \quad \text{(Var)}$$

$$\frac{}{\Gamma \triangleright c : \tau^c} \quad \text{(Const)}$$

$$\frac{\Gamma, x{:}\sigma \triangleright t : \tau}{\Gamma \triangleright \lambda x^\sigma . t : \sigma \to \tau} \quad \text{(Abs)}$$

$$\frac{\Gamma \triangleright t : \sigma \to \tau \quad \Gamma \triangleright t' : \sigma}{\Gamma \triangleright tt' : \tau} \quad \text{(App)}$$

Our metalanguage includes a classical higher-order logic by way of constants

$$\forall_\tau : (\tau \to \mathsf{Prop}) \to \mathsf{Prop}$$

$$\supset : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \ .$$

We write the more natural $\forall_\tau x.P$ for $(\forall_\tau)(\lambda x^\tau.P)$ and take standard classical higher-order logic encodings of absurdity (false), negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), iff ($\equiv$), existential quantification ($\exists$) and Leibniz equality $=^\tau$: $\tau \to \tau \to \mathsf{Prop}$, as displayed in Table 3.1. (We omit type annotations wherever possible, use parentheses with commas for (possibly) repeated application of terms, and infix notation where appropriate, e.g., $x =^\tau y$, or even $x = y$, for $=^\tau(x)(y)$.) We have standard rules for classical higher-order logic, displayed in Table 3.2.

$$\begin{aligned}
\mathsf{false} &\stackrel{\text{def}}{=} \forall_{\mathsf{Prop}} P.P \\
\neg P &\stackrel{\text{def}}{=} P \supset \mathsf{false} \\
P \wedge Q &\stackrel{\text{def}}{=} \neg(P \supset \neg Q) \\
P \vee Q &\stackrel{\text{def}}{=} \neg P \supset Q \\
P \equiv Q &\stackrel{\text{def}}{=} (P \supset Q) \wedge (Q \supset P) \\
\exists_\tau x.P &\stackrel{\text{def}}{=} \neg \forall_\tau x. \neg P \\
x =^\tau y &\stackrel{\text{def}}{=} \forall_{\tau \to \mathsf{Prop}} P.P(x) \supset P(y)
\end{aligned}$$

Table 3.1: Standard classical encodings of logical connectives.

$$\frac{\Gamma \rhd P : \mathsf{Prop} \quad \Gamma \rhd Q : \mathsf{Prop} \quad \Gamma \rhd R : \mathsf{Prop}}{\Gamma \rhd (P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R} \quad \text{(S)}$$

$$\frac{\Gamma \rhd P : \mathsf{Prop} \quad \Gamma \rhd Q : \mathsf{Prop}}{\Gamma \rhd P \supset Q \supset P} \quad \text{(K)}$$

$$\frac{\Gamma \rhd P : \mathsf{Prop}}{\Gamma \rhd \neg\neg P \supset P} \quad \text{(DN)}$$

$$\frac{\Gamma \rhd P : \sigma \rightarrow \mathsf{Prop} \quad \Gamma \rhd t : \sigma}{\Gamma \rhd \forall_\sigma P \supset Pt} \quad \text{(Forall-Elim)}$$

$$\frac{\Gamma \rhd P \supset Q \quad \Gamma \rhd P}{\Gamma \rhd Q} \quad \text{(Modus-Ponens)}$$

$$\frac{\Gamma \rhd P : \mathsf{Prop} \quad \Gamma, x{:}\sigma \rhd P \supset Q}{\Gamma \rhd P \supset \forall x^\sigma . Q} \quad \text{(Generalisation)}$$

$$\frac{\Gamma, x{:}\sigma \rhd t : \tau \quad \Gamma \rhd t' : \sigma}{\Gamma \rhd (\lambda x^\sigma . t)t' =^\tau t[t'/x]} \quad (\beta)$$

$$\frac{\Gamma \rhd t : \sigma \rightarrow \tau}{\Gamma \rhd \lambda x^\sigma . tx =^\tau t} \quad \text{provided } x \notin \mathrm{FV}(t) \quad (\eta)$$

$$\frac{\Gamma, x{:}\sigma \rhd t =^\tau t'}{\Gamma \rhd \lambda x^\sigma . t =^{\sigma \rightarrow \tau} \lambda x^\sigma . t'} \quad (\xi)$$

Table 3.2: Standard rules for classical higher-order logic.

For predicates $P, Q : \tau \to \mathsf{Prop}$, we write $P \subseteq Q$ as shorthand for $\forall_\tau x.(P(x) \supset Q(x))$. In particular, for sets A and B, the formula $A \subseteq B$ simply means than A is a subset of B, as expected. We define composition of $T : \mathsf{TR}$ and $U : \mathsf{Val} \to \mathsf{TR}$, written $T; U$, by

$$(T; U)(r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc}) \stackrel{\text{def}}{=} \exists \check{r}, \check{\sigma}, \check{alloc}.\ T(\check{r}, \grave{\sigma}, \check{\sigma}, \grave{alloc}, \check{alloc}) \land U(\check{r})(r, \check{\sigma}, \acute{\sigma}, \check{alloc}, \acute{alloc}) \quad .$$

For the other constants, we have: the normal constants for natural numbers (0, succ, $+$, $\ldots$; but no general recursor giving terms of type $\mathsf{nat} \to \tau$); booleans ($\mathtt{ff}$, $\mathtt{tt}$, and, $\ldots$); product types ($\pi_1^{\tau_1,\tau_2}$, $\pi_2^{\tau_1,\tau_2}$, $\langle -, - \rangle_{\tau_1,\tau_2}$); and the record manipulation constants:

$$\begin{aligned} \mathsf{lookup}_{\tau_1,\tau_2} &: \mathsf{Rcd}_{\tau_1}^{\tau_2} \to \tau_1 \to \tau_2 \to \mathsf{Prop} \\ \mathsf{update}_{\tau_1,\tau_2} &: \mathsf{Rcd}_{\tau_1}^{\tau_2} \to \tau_1 \to \tau_2 \to \mathsf{Rcd}_{\tau_1}^{\tau_2} \\ \mathsf{empty}_{\tau_1,\tau_2} &: \mathsf{Rcd}_{\tau_1}^{\tau_2} \quad . \end{aligned}$$

The intended interpretation of elements of record types are partial functions with finite domain. Constant $\mathsf{lookup}(r, i, a)$ is intended to be interpreted as i is in the domain of r and $r(i) = a$. Constant $\mathsf{domain}(r, -)$ is intended to be interpreted as the domain of r. Constant $\mathsf{update}(r, i, a)$ is intended to be interpreted as r overridden at i to take value a. We, thus, have the following axioms for records.

$$\forall r, i, a, a'.\mathsf{lookup}(r, i, a) \supset \mathsf{lookup}(r, i, a') \supset a = a' \qquad \text{(rcd_parfun)}$$

$$\forall i, a.\neg\mathsf{lookup}(\mathsf{empty}, i, a) \qquad \text{(rcd_empty)}$$

$$\begin{aligned} \forall r, i, i', a. \quad & i = i' \supset \mathsf{lookup}(\mathsf{update}(r, i, a), i', a) \\ \land \quad & i \neq i' \supset \forall a'.\mathsf{lookup}(\mathsf{update}(r, i, a), i', a') \equiv \mathsf{lookup}(r, i', a') \quad . \end{aligned} \qquad \text{(rcd_update)}$$

For convenience, we use the more succinct notation $[f_1{=}a_1, \ldots, f_k{=}a_k]$ for record

$$\mathsf{update}(\cdots \mathsf{update}(\mathsf{empty}, f_1, a_1), \cdots, f_k, a_k) \quad .$$

Also, we introduce the abbreviation $\mathsf{dom}(r)$ defined by

$$i \in \mathsf{dom}(r) \quad \text{iff} \quad \exists a.\mathsf{lookup}(r, i, a) \quad .$$

3.2.1 Program logic

The remaining constants are those for the program logic. The following constants allow us to construct specifications.

$$\begin{aligned} \mathsf{Nat} &: \mathsf{Sp} \\ \mathsf{Bool} &: \mathsf{Sp} \\ \mathsf{Obj} &: \mathsf{Rcd}_{\mathsf{Fn}}^{\mathsf{Sp}} \rightarrow \mathsf{Rcd}_{\mathsf{Mn}}^{\mathsf{Val} \rightarrow \mathsf{Sp} \times \mathsf{Val} \rightarrow \mathsf{TR}} \rightarrow \mathsf{Sp} \ . \end{aligned}$$

Note that this directly reflects the abstract syntax for specifications: namely a specification is either *Bool*, *Nat* or $[f_i{:}A_i\ ^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}U_j\ ^{j=1..\ell}]$ for specifications A_i, B_j and transition relations U_j, where y_j can occur free in B_j, U_j.

The following constants allow us to construct programs.

$$\begin{aligned} \mathsf{false} &: \mathsf{Prg} \\ \mathsf{true} &: \mathsf{Prg} \\ \mathsf{nat} &: \mathsf{nat} \rightarrow \mathsf{Prg} \\ \mathsf{let} &: \mathsf{Prg} \rightarrow (\mathsf{Var} \rightarrow \mathsf{Prg}) \rightarrow \mathsf{Prg} \\ \mathsf{obj} &: \mathsf{Rcd}_{\mathsf{Fn}}^{\mathsf{Var}} \rightarrow \mathsf{Rcd}_{\mathsf{Mn}}^{\mathsf{Var} \rightarrow \mathsf{Prg}} \rightarrow \mathsf{Prg} \\ \mathsf{if} &: \mathsf{Var} \rightarrow \mathsf{Prg} \rightarrow \mathsf{Prg} \rightarrow \mathsf{Prg} \\ \mathsf{var} &: \mathsf{Var} \rightarrow \mathsf{Prg} \\ \mathsf{fsel} &: \mathsf{Var} \rightarrow \mathsf{Fn} \rightarrow \mathsf{Prg} \\ \mathsf{minv} &: \mathsf{Var} \rightarrow \mathsf{Mn} \rightarrow \mathsf{Prg} \\ \mathsf{fupd} &: \mathsf{Var} \rightarrow \mathsf{Fn} \rightarrow \mathsf{Var} \rightarrow \mathsf{Prg} \ . \end{aligned}$$

Constants let and obj give our encoding of syntax its higher-order nature. For example, returning to our illustration, program

$$\textit{let } x{=}\textit{true in if } x \textit{ then false else true}$$

has encoding

$$\mathsf{let}(\mathsf{true}, \lambda x.\mathsf{if}(x, \mathsf{false}, \mathsf{true})) \ .$$

We generalise the encoding procedure and write $\ulcorner a \urcorner$ for the encoding of a.

Finally, we have the Val-valued constants

$$\begin{aligned} \mathsf{bool}_{\mathsf{Val}} &: \mathsf{bool} \to \mathsf{Val} \\ \mathsf{nat}_{\mathsf{Val}} &: \mathsf{nat} \to \mathsf{Val} \\ \mathsf{var}_{\mathsf{Val}} &: \mathsf{Var} \to \mathsf{Val} \ . \end{aligned}$$

We need constants $\mathsf{bool}_{\mathsf{Val}}$ and $\mathsf{nat}_{\mathsf{Val}}$ because we have extra typing information in our transition relation language. Their presence forces types $\mathsf{bool}_{\mathsf{Val}}$ and $\mathsf{nat}_{\mathsf{Val}}$ to be subtypes of Val. We need $\mathsf{var}_{\mathsf{Val}}$ because of the deep embedding of program syntax together with the shallow embedding of the assertion logic, the combination of which gives us the type of variables Var which is distinct from that of values Val. Whereas, supposing we have a shallow embedding of programs, then the distinction between Var and Val would no longer be needed; program variables would be identified with variables of the metalanguage.

We note that since $\mathsf{var}_{\mathsf{Val}}$ maps program variables to values, it is in essence a stack. Considering transition relations as relations over a return value, initial and final stores, and the values of the variables in the context, a natural type for transition relations would be a predicate over a value, two stores and a stack:

$$\mathsf{Val} \to (\mathsf{Val} \to \mathsf{Fn} \to \mathsf{Val})^2 \to (\mathsf{Var} \to \mathsf{Val}) \ .$$

Then each transition relation is parameterised with its own stack, thus allowing us to dispense with constant $\mathsf{var}_{\mathsf{Val}}$. However, this does not fit with the essence of HOAS. The stack-parameterised transition relations approach would perhaps be obligatory if we were to use a first-order, deep embedding of the program syntax since we cannot guarantee that the bound variables are pairwise distinct; it is possible for two different bound variables to have the same name, in which case, constant $\mathsf{var}_{\mathsf{Val}}$ assigns the same value to both of them when, in fact, they should take different values. Our HOAS embedding guarantees uniqueness of bound variables, thus this is no longer a problem. The reason why the same trick cannot be applied to the other parameters (namely, return value, initial and final stores) is because the formulation of the rules in AL require us to substitute for these parameters, whereas we do not need substitution of stacks.

Since $\mathsf{var}_{\mathsf{Val}}$ is a constant, a standard interpretation would interpret it as a function mapping variables to values. Thus a consequence of this implementation decision is: to define a model, we must interpret $\mathsf{var}_{\mathsf{Val}}$ as a stack for which all the axioms are true. This is certainly counter-intuitive.

Since these Val-valued constants are effectively type-coercion functions, they will be omitted from this presentation when there is no risk of ambiguity.

We introduce predicate symbols for: specification subsumption

$$<: : \mathsf{Sp} \to \mathsf{Sp} \to \mathsf{Prop} \ ;$$

assumption for variables[1]

$$[- : -] : \mathsf{Var} \to \mathsf{Sp} \to \mathsf{Prop} \ ;$$

and derivability (verification) judgement

$$[- : - :: -] : \mathsf{Prg} \to \mathsf{Sp} \to \mathsf{TR} \to \mathsf{Prop} \ .$$

We encode the rules so that if

$$x_1{:}A_1, \ldots , x_n{:}A_n \vdash a : A :: T$$

is derivable, then

$$\rhd \forall_{\mathsf{Var}} x_1, \ldots x_n . [x_1 : \ulcorner A_1 \urcorner] \supset \cdots \supset [x_n : \ulcorner A_1 \urcorner] \supset [\ulcorner a \urcorner : \ulcorner A \urcorner :: \ulcorner T \urcorner]$$

is derivable.

Hereafter, unless explicitly typed otherwise, the following symbols and their decorated variants are metavariables of the following types: n have type nat, x and y have type Var; f have type Fn; m have type Mn; a have type Prg; b have type Var $\to$ Prg; A have type Sp; B have type Val $\to$ Sp; T have type TR; and U have type Val $\to$ TR.

[1]In [HT00] we have *defined* $[- : -]$ to be $[x : A] \stackrel{\text{def}}{=} [\mathsf{var}(x) : A :: \mathsf{Res}(x)]$. This is not compatible with the interpretation given here. However, the interpretation can be modified to accommodate this.

3.2.1.1 Subsumption axioms

The following axioms are for the subsumption relation. The only subspecification of Bool (resp. Nat) is Bool (resp. Nat).

$$\mathsf{Bool} <: \mathsf{Bool} \tag{ss_bool}$$

$$\mathsf{Nat} <: \mathsf{Nat} \tag{ss_nat}$$

Object type A is a subspecification of A' if A has all fields and methods of A', and furthermore, (1) for any field f, the specification in A along f matches that of A' and (2) for any method m, the specification in A along m is a subspecification of A'. Note that point (2) is precisely the fact that the subspecification relation is covariant along methods, and point (1) is precisely the fact that it is invariant along fields.

$$\begin{aligned}
&\forall_{\mathsf{Rcd}_{\mathsf{Fn}}^{\mathsf{Sp}}} \vec{A}, \vec{A}'. \forall_{\mathsf{Rcd}_{\mathsf{Mn}}^{(\mathsf{Val} \rightarrow \mathsf{Sp}) \times (\mathsf{Val} \rightarrow \mathsf{TR})}} \vec{B}, \vec{B}'. \\
&\quad (\forall f. \forall A. \mathsf{lookup}(\vec{A}', f, A) \supset \mathsf{lookup}(\vec{A}, f, A)) \supset \\
&\quad (\forall h. \forall m. \forall B'_m. \mathsf{lookup}(\vec{B}', m, B'_m) \supset \exists B_m. \\
&\qquad\qquad \mathsf{lookup}(\vec{B}, m, B_m) \wedge \\
&\qquad\qquad \pi_1(B_m)(h) <: \pi_1(B'_m)(h) \wedge \\
&\qquad\qquad \pi_2(B_m)(h) \subseteq \pi_2(B'_m)(h)) \supset \\
&\quad\ \mathsf{Obj}(\vec{A}, \vec{B}) <: \mathsf{Obj}(\vec{A}', \vec{B}')
\end{aligned} \tag{ss_obj}$$

3.2.1.2 Program axioms

The remaining axioms are those of the program logic. A program a with specification A and transition relation T is said to be *well specified* (ws) if we can derive the judgement $[a : A :: T]$. The rules to derive judgements stating well specified triples are encoded as follows.

The first rule is the subsumption rule (as mentioned in Chapter 2). Note that to prove $[a : A' :: T']$, it suffices to find weaker A, T (i.e. to prove that $A <: A'$

and $T \subseteq T'$) and prove $[a : A :: T]$.

$$\begin{aligned}&\forall a.\forall A'.\forall T'.\forall A.\forall T.\\&\quad (A <: A') \supset\\&\quad (T \subseteq T') \supset\\&\qquad [a : A :: T] \supset [a : A' :: T']\end{aligned} \qquad \text{(ws_subs)}$$

The following are straightforward rules for constants.

$$[\mathsf{false} : \mathsf{Bool} :: \mathsf{Res}(\mathtt{ff})] \qquad \text{(ws_constf)}$$

$$[\mathsf{true} : \mathsf{Bool} :: \mathsf{Res}(\mathtt{tt})] \qquad \text{(ws_constt)}$$

$$\forall n.[\mathsf{nat}(n) : \mathsf{Nat} :: \mathsf{Res}(n)] \qquad \text{(ws_nat)}$$

The rule for variables states this: provided $x{:}A$ is in our context, i.e. we know $[x : A]$, then we can prove $[x : A :: \mathsf{Res}(x)]$.

$$\forall x.\forall A.[x : A] \supset [x : A :: \mathsf{Res}(x)] \qquad \text{(ws_var)}$$

As an example of functions over constants, for any binary natural number operation *op*, we have

$$\begin{aligned}&\forall x_0, x_1.\\&\quad [x_0 : \mathsf{Nat} :: \mathsf{Res}(x_0)] \supset [x_1 : \mathsf{Nat} :: \mathsf{Res}(x_1)] \supset\\&\qquad [op(x_0, x_1) : \mathsf{Nat} :: \mathsf{Res}(op(x_0, x_1))]\end{aligned} \qquad \text{(ws_natop)}$$

The conditional rule is a straightforward translation from locus classicus.

$$\begin{aligned}&\forall x.\forall a_0, a_1.\forall B.\forall U.\forall B_0, B_1.\forall U_0, U_1.\\&\quad [x : \mathsf{Bool} :: \mathsf{Res}(x)] \supset\\&\quad [a_0 : B_0(x) :: U_0(x)] \supset\\&\quad (B_0(\mathtt{tt}) = B(\mathtt{tt}) \wedge U_0(\mathtt{tt}) \equiv U(\mathtt{tt})) \supset\\&\quad [a_1 : B_1(x) :: U_1(x)] \supset\\&\quad (B_1(\mathtt{ff}) = B(\mathtt{ff}) \wedge U_1(\mathtt{ff}) \equiv U(\mathtt{ff})) \supset\\&\qquad [\mathsf{if}(x, a_0, a_1) : B(x) :: U(x)]\end{aligned} \qquad \text{(ws_cond)}$$

In the rule for let, to prove $[\mathsf{let}(a,b) : A' :: T']$, we are required to prove three conditions, of which the second one has form $\forall x.[x : A] \supset [b(x) : A' :: U(x)]$. To prove this, we prove $[b(x) : A' :: U(x)]$ but have the extra assumption $[x : A]$. Note further that the universal quantification of x ensures that x is a fresh variable name.

$$\begin{array}{l} \forall a.\forall b.\forall A'.\forall T''.\forall A.\forall T.\forall U. \\ \quad [a : A :: T] \supset \\ \quad (\forall x.[x : A] \supset [b(x) : A' :: U(x)]) \supset \\ \quad (T; U \subseteq T'') \supset \\ \quad\quad [\mathsf{let}(a,b) : A' :: T''] \end{array} \tag{ws_let}$$

The rule for method invocation is a straightforward translation from that of locus classicus.

$$\begin{array}{l} \forall x.\forall m.\forall B.\forall U. \\ \quad [x : \mathsf{Obj}([], [m{=}\langle B, U\rangle]) :: \mathsf{Res}(x)] \supset \\ \quad\quad [\mathsf{minv}(x,m) : B(x) :: U(x)] \end{array} \tag{ws_minv}$$

Note the form of conclusion of the rule for field selection: we can derive the triple $[\mathsf{fsel}(x,f) : A :: T]$ for arbitrary T, provided we can prove $T \equiv T_{\mathsf{fsel}}(x,f)$. The rule formulated as such allows it to be applied immediately to a goal $[\mathsf{fsel}(x,f) : A :: T]$ when T is equivalent to, but possibly syntactically different from, $T_{\mathsf{fsel}}(x,f)$. Without this trick, certainly in a prover such as LEGO, the user must first manipulate the goal $[\mathsf{fsel}(x,f) : A :: T]$ until T is syntactically equal to $T_{\mathsf{fsel}}(x,y)$.

$$\begin{array}{l} \forall x.\forall f.\forall A.\forall T. \\ \quad [x : \mathsf{Obj}([f{=}A], []) :: \mathsf{Res}(x)] \supset \\ \quad T \equiv T_{\mathsf{fsel}}(x,f) \supset \\ \quad\quad [\mathsf{fsel}(x,f) : A :: T] \end{array} \tag{ws_fsel}$$

where $T_{\mathsf{fsel}}(x,f)(r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc}) \stackrel{\text{def}}{=} \mathsf{Res}(\acute{\sigma}(x,f))(r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc})$.

The rule for object creation applies both tricks mentioned above. Note the extra pathological conditions to ensure that the domains of records $\vec{x}$ and $\vec{A}$, and $\vec{b}$ and $\vec{B}$ match. This matching is implicit in the notation used in the locus classicus.

$$
\begin{array}{l}
\forall_{\mathsf{Rcd}^{\mathsf{Var}}_{\mathsf{Fn}}}\vec{x}.\forall_{\mathsf{Rcd}^{\mathsf{Var}\to\mathsf{Prg}}_{\mathsf{Mn}}}\vec{b}.\forall_{\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}}\vec{A}.\forall_{\mathsf{Rcd}^{(\mathsf{Val}\to\mathsf{Sp})\times(\mathsf{Val}\to\mathsf{TR})}_{\mathsf{Mn}}}\vec{B}.\forall T. \\
\quad (\mathsf{dom}(\vec{x}) = \mathsf{dom}(\vec{A}) \wedge \mathsf{dom}(\vec{b}) = \mathsf{dom}(\vec{B})) \supset \\
\quad (\forall f.\forall A_f.\forall x_f.\mathsf{lookup}(\vec{A}, f, A_f) \supset \mathsf{lookup}(\vec{x}, f, x_f) \supset [x_f : A_f :: \mathsf{Res}(x_f)]) \supset \\
\quad (\forall m.\forall y.[y : \mathsf{Obj}(\vec{A}, \vec{B})] \supset \\
\qquad \forall b_m.\forall B_m.\mathsf{lookup}(\vec{b}, m, b_m) \supset \mathsf{lookup}(\vec{B}, m, B_m) \supset \\
\qquad\quad [b_m(y) : \pi_1(B_m)(y) :: \pi_2(B_m)(y)]) \supset \\
\quad T \equiv T_{\mathsf{obj}}(\vec{x}) \supset \\
\qquad [\mathsf{obj}(\vec{x}, \vec{b}) : \mathsf{Obj}(\vec{A}, \vec{B}) :: T]
\end{array}
\qquad \text{(ws_obj)}
$$

where
$$
\begin{array}{rl}
T_{\mathsf{obj}}(\vec{x})(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \stackrel{\text{def}}{=} & (\forall z.z \neq r \supset al\grave{l}oc(z) \equiv al\acute{l}oc(z)) \wedge \\
& (\forall f \in \mathsf{dom}(\vec{x}).\acute{\sigma}(r, f) = x_f) \wedge \\
& (\forall z.\forall w.z \neq r \supset \grave{\sigma}(z, w) = \acute{\sigma}(z, w)) \ .
\end{array}
$$

Finally, the rule for field update uses the same trick as that in the rule for field selection.

$$
\begin{array}{l}
\forall x, y.\forall f.\forall_{\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}}\vec{A}.\forall_{\mathsf{Rcd}^{(\mathsf{Val}\to\mathsf{Sp})\times(\mathsf{Val}\to\mathsf{TR})}_{\mathsf{Mn}}}\vec{B}.\forall T. \\
\quad [x : \mathsf{Obj}(\vec{A}, \vec{B}) :: \mathsf{Res}(x)] \supset \\
\quad [y : A_f :: \mathsf{Res}(y)] \supset \\
\quad T \equiv T_{\mathsf{fupd}}(x, f, y) \supset \\
\qquad [\mathsf{fupd}(x, f, y) : \mathsf{Obj}(\vec{A}, \vec{B}) :: T]
\end{array}
\qquad \text{(ws_fupd)}
$$

where
$$
\begin{array}{rl}
T_{\mathsf{fupd}}(x, f, y)(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \stackrel{\text{def}}{=} & r = x \wedge \acute{\sigma}(x, f) = y \wedge \\
& (\forall z.\forall w.\neg(z = x \wedge w = f) \supset \\
& \quad \grave{\sigma}(z, w) = \acute{\sigma}(z, w)) \wedge \\
& al\grave{l}oc = al\acute{l}oc \ .
\end{array}
$$

3.3 Practical considerations

The embedding formalisation detailed above has been implemented in two theorem provers: LEGO and PVS. The two theorem provers are quite different and each has its strengths and weaknesses. In this section, we highlight some of these differences and how they affect the usability of the implementation.

The LEGO implementation used a direct concretisation of the formalisation presented above. A nice feature of LEGO is its treatment of logic variables. For example, assuming

$$a \stackrel{\text{def}}{=} \mathsf{let}(\mathsf{true}, \lambda x.\mathsf{false})$$
$$T_0 \stackrel{\text{def}}{=} \lambda r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc}.r = \mathtt{ff}$$

suppose we want to prove

$$[a : \mathsf{Bool} :: T_0] \ .$$

First we apply the subsumption rule ws_subs using LEGO's `Refine` command. LEGO automatically instantiates a, A', T' by matching the conclusion of ws_subs with our goal. It then presents us with the following remaining goals:

$$?_1 : \mathsf{Sp}$$
$$?_2 : \mathsf{TR}$$
$$?_3 : ?_1 <: \mathsf{Bool}$$
$$?_4 : ?_2 \subseteq T_0$$
$$?_5 : [a : ?_1 :: ?_2] \ ,$$

where $?_i$ is LEGO notation for the current goals, i.e. remaining things we have to "prove". It postpones instantiation of A, T by introducing logic variables $?_1, ?_2$. The first two goals provide the constraints that these two logic variables must satisfy. The useful part is that we can postpone instantiating these variables until later, for it might be possible that using the `Refine` command, some of these variables get automatically instantiated. For example, we could concentrate on goal $?_3$ and apply rule ss_bool. LEGO then automatically instantiates $?_1$ with

Bool based on the form of rule ss_bool. Similarly, we can concentrate on goal $?_5$ which would give us an instantiation for $?_2$.

However, a significant weakness of LEGO is its lack of automation. Once we have instantiated $?_2$ (either by explicit instantiation, or LEGO's automatic instantiation) we still have to prove the goal $?_4$. In this particular example, the goal is trivial, but using LEGO, we must issue successive commands to discharge this goal.

In search of more automation, we tried the implementation with PVS. As an optimisation, rather than introducing record types Rcd as an uninterpreted type (i.e. introduce Rcd as a type and then provide axioms), we defined it as an interpreted type as follows:

```
Rcd : TYPE = [d: pred[dom], [(d) -> ran]]
```

using PVS notation. (Technical note: since PVS doesn't provide proper parameterised types, we had to put this definition into a module, parameterised by types `dom` and `ran`.) This defines `Rcd` as the type of partial functions from `dom` to `ran`, implemented as a (dependent) pair (d, f) consisting of a subset d of `dom` and a (total) function f from d to `ran`. We then gave the natural definitions to the other record manipulation constants. This implementation is consistent with the presentation above in the sense that the record axioms are derivable. Furthermore, we believe that the interpretation of the metalanguage (to follow) still models the implementation. This interpreted definition of `Rcd` allows PVS to automatically discharge more proof obligations involving records.

In use, the PVS implementation provided a noticable improvement in usability over the LEGO implementation, specifically because many proof obligations could be (almost) automatically discharged. However, there were still many proof obligations that intuitively could be automaticallly discharged, but could not in practice; such proof obligations typically required instantiation of universally quantified variables.

On the other hand, in use, we noticed that PVS's automatic instantiations were not as accurate as those of LEGO (in the sense that the instantiations were often wrong). Worse still, the lack of proper logic variables meant that in practice,

explicit instantiation of the universally quantified variables in the AL rules gave more reliable results.

Overall, the PVS implementation was certainly easier to use. However, having to continually provide instantiations for specifications and transition relations became a real chore, especially since theoretically they can be automatically inferred. (This is effectively what the type inference and VCG algorithms do in Chapters 4 and 5.)

It is also unclear whether it is possible to make other modifications to the formalisation (similar to the interpreted definition of Rcd) which would give improved usability.

3.4 Examples

We have attempted several examples in our implementations. Initially, we followed the development in [AL98] and introduced some abbreviations to make our programs more succinct. Furthermore we derived, using the existing axioms, theorems (or equivalently, derivable rules) for these syntactic abbreviations.

For example, in the pure language, field selection strictly has the form $x.f$ where x is a variable. We introduce an abbreviation so that for an arbitrary program a, the program $a.f$ is an abbreviation for *let* $x{=}a$ *in* $x.f$, for some x not free in a. This is encoded into our formal system by defining

$$\mathsf{fsel}'(a, f) \stackrel{\text{def}}{=} \mathsf{let}(a, \lambda x.\mathsf{fsel}(x, f)) \ .$$

We simply overload our existing notation and write fsel for fsel$'$. Crucially, we then prove the following theorem.

$$\begin{array}{l} \rhd \forall a.\forall f.\forall A.\forall T', T. \\ \qquad [a : \mathsf{Obj}([f{=}A], []) :: T] \supset \\ \qquad T' \equiv T; \lambda x.T_{\mathsf{fsel}}(x, f) \supset \\ \qquad\quad [\mathsf{fsel}(a, f) : A :: T'] \end{array}$$

This theorem allows us to directly derive judgements about programs using fsel$'$ without expanding its definition.

We continue to extend and overload the remaining program constructors and derive corresponding "higher-level" rules. Using these rules, we successfully prove the examples given in Sec. 4.1–2 of [AL98]. We then derive an easier-to-use rule for the let construction before attempting two larger examples: the greatest common divisor program (Sec. 4.3 in [AL98]), and an original example based on the dining philosophers scenario. We consider these two examples in more detail after the new rule for let.

3.4.1 Reversible let rule

In locus classicus, Rule (Let) is formulated as

$$\frac{E \vdash a : A :: T \quad E, x{:}A \vdash b : B :: U}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : B :: T'} \ ,$$

provided x does not occur free in B, and the first-order theorem $T;_x U \subseteq T'$ is derivable. In essence, the subsumption rule has been incorporated into this rule by default; this is necessary because $T; U$ is not a first-order formula. In particular, the rule is not reversible: in general, we cannot recover T, U from T'.

In practice, when building proofs in a goal-directed style (as is the default in LEGO and PVS), one finds that whenever we apply the let rule, we must find instantiations for T, U, and typically this involves some loss of information. (If we choose T, U to be too weak, then we cannot discharge the side condition; so typically we choose them to be stronger, but we must be careful not to choose them too strong or else we might not be able to discharge a side condition further up the proof tree.) Since sequential composition of programs is defined using the let constructor, it occurs extensively in typical programs and it quickly becomes cumbersome to find these instantiations.

Using our implementation, we can derive the following substitute for the let

rule.

$$\begin{array}{l}\forall a.\forall b.\forall A'.\forall T''.\forall A.\forall T.\forall U.\\ \quad [a : A :: T] \supset \\ \quad (\forall x.[x : A] \supset [b(x) : A' :: U(x))] \supset \\ \quad (T; U \equiv T'') \supset \\ \quad\quad [\mathsf{let}(a, b) : B :: T'']\end{array} \qquad \text{(wsq_let)}$$

This rule does not lose any information. In proof derivations, it is particularly useful because information loss can be postponed until later using an explicit application of the subsumption axiom.

3.4.2 Greatest common divisor

The gcd program from [AL98], can be written using notation closer to that of popular OO languages, as follows. It is a simple exercise to translate this program into our formal language.

$$\begin{array}{llll}\mathsf{calc_gcd}(y) \stackrel{\text{def}}{=} & \texttt{if} & (y.f < y.g) & \{y.g = y.g - y.f; y.m()\} \\ & \texttt{else if} & (y.g < y.f) & \{y.f = y.f - y.g; y.m()\} \\ & \texttt{else} & & \{y.f\}\end{array}$$

$$\mathsf{gcd} \stackrel{\text{def}}{=} \mathsf{obj}([f{=}1, g{=}1], [m{=}\mathsf{calc_gcd}])$$

This program creates an object with one method m, such that if the fields have nonzero values a and b, invoking m will reduce both fields to the gcd of a and b. This is the intuition behind the formal specification given in loc. cit. First we strengthened the transition relation given in loc. cit, and then we proved gcd satisfies the stronger specification. The subsumption axiom can be used to weaken this to the original statement.

We introduced the constant $\mathsf{gcd}_{\mathsf{Val}} : \mathsf{Val} \to \mathsf{Val} \to \mathsf{Val}$ and postulated axioms consistent with its interpretation as the gcd function over natural numbers. Of course, we could have define gcd axiomatically, and then derived our postulated axioms as theorems about gcd; however, this is not the purpose of the exercise.

Explicitly, we defined

$$\begin{aligned} U_{\mathsf{gcd}}(y)(r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc}) \stackrel{\text{def}}{=} \ & (1 \leq \grave{\sigma}(y,f) \wedge 1 \leq \grave{\sigma}(y,g)) \supset \\ & r = \acute{\sigma}(y,f) \wedge r = \acute{\sigma}(y,g) \wedge \\ & r = \mathsf{gcd}_{\mathsf{Val}}(\acute{\sigma}(y,f), \acute{\sigma}(y,g)) \wedge \\ & 1 \leq \acute{\sigma}(y,f) \wedge 1 \leq \acute{\sigma}(y,g) \\ A_{\mathsf{gcd}} \stackrel{\text{def}}{=} \ & \mathsf{Obj}([f = \mathsf{Nat}, g = \mathsf{Nat}], [m = \langle \mathsf{Nat}, U_{\mathsf{gcd}} \rangle]) \ . \end{aligned}$$

Using these definitions, we proved

$$\rhd [\mathsf{gcd} : A_{\mathsf{gcd}} :: T_{\mathrm{triv}}]$$

where T_{triv} is a trivial transition relation.

3.4.3 Dining philosophers

Object-oriented languages have shown to be particularly suitable for writing simulations. In the next example, we consider a simulation in an OO language for a formalisation of the dining philosophers scenario. The formalisation we choose is based on that presented in Roscoe's book [Ros97], where a general description of the scenario can be found. Our implementation follows Roscoe's observation that the important events that we should model are when the forks get picked up and put down. To make the example more manageable, we only consider the case for three philosophers at the table.

We simulate the scenario by creating an object for each fork, and an object for each philosopher. The philosophers interact with the forks by invoking their methods. In our example, two of the philosophers pick up their forks "left then right" and one picks up his forks "right then left". The resulting system is known not to deadlock. We prove this using a suitable formalisation of "does not deadlock".

Here is code to create a fork object.

```
Fork ≝ obj([on_table=true],
          [try_pick_up=λs. if (s.on_table) {s.on_table = false; true}
                          else {false} ,
           put_down=λs.s.on_table = true; false])
```

A philosopher object invokes the try_pick_up method to pick up a fork. The method returns true after updating the fork object's state if this is possible. It returns false if the fork is not on the table.

We introduce the following definitions for creating the two types of philosophers.

```
phil_tick ≝ λs. if (s.n_forks == 0 and s.hungry) {
                  if (s.fork1.try_pick_up()) {s.n_forks = 1; false}
                  else {false}
                } else if (s.n_forks == 1 and s.hungry) {
                  if (s.fork2.try_pick_up())
                    {s.n_forks = 2; s.hungry = false; false}
                  else {false}
                } else if (s.n_forks == 2) {
                  s.fork2.put_down(); s.n_forks = 1; false
                } else {
                  s.fork1.put_down(); s.n_forks = 0; s.hungry = true; false
                }
 LRPhil ≝ λfork_l, fork_r.Phil(fork_l, fork_r)
 RLPhil ≝ λfork_l, fork_r.Phil(fork_r, fork_l)
   Phil ≝ λfork_1, fork_2.
            obj([hungry=true, n_forks=nat(0),
              fork1=var(fork_1), fork2=var(fork_2)],
              [tick=phil_tick])
```

A philosopher has four internal states: (1) he is hungry and is holding no forks;

(2) he is hungry and he is holding one fork; (3) he is no longer hungry[2] and is holding two forks; and (4) he is not hungry and holding one fork. Each state transition corresponds exactly to a fork being either picked up or put down.

Finally, we put the whole system together as follows by creating a "table" object.

$$
\begin{aligned}
\mathsf{Table} \stackrel{\text{def}}{=}\ & \mathsf{let}(\mathit{fk}_1 = \mathsf{Fork}, \mathit{fk}_2 = \mathsf{Fork}, \mathit{fk}_3 = \mathsf{Fork},\\
& \mathit{ph}_1 = \mathsf{LRPhil}(\mathit{fk}_1, \mathit{fk}_2),\\
& \mathit{ph}_2 = \mathsf{LRPhil}(\mathit{fk}_2, \mathit{fk}_3),\\
& \mathit{ph}_3 = \mathsf{RLPhil}(\mathit{fk}_3, \mathit{fk}_1),\\
& \mathsf{obj}(\\
& \quad [\mathsf{f1}{=}\mathit{fk}_1, \mathsf{f2}{=}\mathit{fk}_2, \mathsf{f3}{=}\mathit{fk}_3,\\
& \quad \mathsf{p1}{=}\mathit{ph}_1, \mathsf{p2}{=}\mathit{ph}_2, \mathsf{p3}{=}\mathit{ph}_3],\\
& \quad [\mathsf{tick1}{=}\lambda s.s.\mathsf{p1.tick}(),\\
& \quad \mathsf{tick2}{=}\lambda s.s.\mathsf{p2.tick}(),\\
& \quad \mathsf{tick3}{=}\lambda s.s.\mathsf{p3.tick}()]))
\end{aligned}
$$

The table should be considered as a "black box" with three buttons, one for each of the tick methods. To complete the simulation, one must compose this program with another program that plays out the possible traces of the system.

Note how we use aliasing in this example: philosopher objects ph_1 and ph_2 both store references to the same fork object fk_2; one can check that $\mathit{ph}_1.\mathsf{fork2}$ and $\mathit{ph}_2.\mathsf{fork1}$ both store the value of fk_2.

With the dining philosopher scenario simulated by these code fragments, we can prove that this system does not deadlock, which we will now formalise. We say that a philosopher is *blocked* whenever he needs to pick up a fork to perform a state transition but cannot (exactly when the fork in question is not on the table.) The system is *deadlocked* precisely when all the philosophers on the table are blocked.

Given the store σ, we can determine whether any particular philosopher is blocked by inspecting the values of the fields of the philosopher and its forks. To

[2]Here we assume that the philosopher instantaneously eats as soon as he picks up the second fork and so is no longer hungry. The point is that the event corresponding to a philosopher eating is not important with respect to deadlock considerations.

assist our intuitions, we define the following predicates. "Philosopher p is holding fork *fork*",

$$\begin{aligned}\mathsf{is_holding}(p, \mathit{fork}, \sigma) \stackrel{\text{def}}{=} \; & \sigma(p, \mathsf{n_forks}) = 1 \wedge \mathit{fork} = \sigma(p, \mathsf{fork1}) \vee \\ & \sigma(p, \mathsf{n_forks}) = 2 \wedge \mathit{fork} = \sigma(p, \mathsf{fork1}) \vee \\ & \sigma(p, \mathsf{n_forks}) = 2 \wedge \mathit{fork} = \sigma(p, \mathsf{fork2}) \quad .\end{aligned}$$

"Philosopher p is waiting for fork *fork*,"

$$\begin{aligned}&\mathsf{waiting_for}(p, \mathit{fork}, \sigma) \stackrel{\text{def}}{=} \\ &\quad (\sigma(p, \mathsf{n_forks}) = 0 \wedge \sigma(\sigma(p, \mathsf{fork1}), \mathsf{on_table}) = \mathtt{ff} \wedge \mathit{fork} = \sigma(p, \mathsf{fork1})) \\ &\vee \quad (\sigma(p, \mathsf{n_forks}) = 1 \wedge \sigma(\sigma(p, \mathsf{fork2}), \mathsf{on_table}) = \mathtt{ff} \wedge \mathit{fork} = \sigma(p, \mathsf{fork2})) \ .\end{aligned}$$

Assuming that $F(t)$ is the set of forks, and $P(t)$ the set of philosophers on table t, for $p \in P(t)$, the predicate "philosopher p is blocked,"

$$\mathsf{blocked}(t, p, \sigma) \stackrel{\text{def}}{=} \exists_{F(t)} \mathit{fork}. \mathsf{waiting_for}(p, \mathit{fork}, \sigma)$$

and "all philosophers are blocked,"

$$\mathsf{all_blocked}(t, \sigma) \stackrel{\text{def}}{=} \forall_{P(t)} p. \mathsf{blocked}(p, \sigma) \ .$$

We can use the following proof technique to show that the system does not deadlock. Let $\prec$ be a total (irreflexive) order such that $\mathit{fk}_1 \prec \mathit{fk}_2 \prec \mathit{fk}_3$. It is the case that the philosophers pick up their forks in this order; if a philosopher picks up f_1 before f_2, then $f_1 \prec f_2$. It is then straightforward to prove that the system does not deadlock[3].

For table t, order relation *orel* and store σ, if we define InvTable by

$$\begin{aligned}\mathsf{InvTable}(t, \mathit{orel}, \sigma) \stackrel{\text{def}}{=} \; & \forall_{P(t)} p. \mathsf{InvPhil}(p, \mathit{orel}, \sigma) \wedge \\ & (\forall_{F(t)} f. \sigma(f, \mathsf{on_table}) = \mathtt{ff} \supset \\ & \quad \exists_{P(t)} p. \mathsf{is_holding}(p, f, \sigma)) \wedge \\ & \mathsf{bg_pred}\end{aligned}$$

[3]This is in fact a special case of Roscoe's rules for avoiding deadlock in [Ros97].

where bg_pred is a required strengthening that states facts such as the fork and philosopher objects are pairwise distinct, and the fork fields of the philosophers are pointing to the intended forks. Furthermore,

$$\begin{aligned}\mathsf{InvPhil}(p, orel, \sigma) \stackrel{\text{def}}{=}\ & orel\,(\sigma(p, \mathsf{fork1}), \sigma(p, \mathsf{fork2})) \wedge \\ & (\mathsf{state}_0(p) \vee \mathsf{state}_1(p) \vee \mathsf{state}_2(p) \vee \mathsf{state}_3(p))\end{aligned}$$

where each $\mathsf{state}_j(p)$ states the values of the n_forks and hungry fields of philosopher p at the corresponding state. It follows that if we define

$$\begin{aligned}\mathsf{Spec}_{\mathsf{Table}} \stackrel{\text{def}}{=} \mathsf{Obj}(\ & [\mathsf{f1}{=}\mathsf{Spec}_{\mathsf{Fork}}, \mathsf{f2}{=}\mathsf{Spec}_{\mathsf{Fork}}, \mathsf{f3}{=}\mathsf{Spec}_{\mathsf{Fork}}, \\ & \mathsf{p1}{=}\mathsf{Spec}_{\mathsf{Phil}}, \mathsf{p2}{=}\mathsf{Spec}_{\mathsf{Phil}}, \mathsf{p3}{=}\mathsf{Spec}_{\mathsf{Phil}}], \\ & [\mathsf{tick1}{=}\mathsf{TR}_{\mathsf{tick}}, \mathsf{tick2}{=}\mathsf{TR}_{\mathsf{tick}}, \mathsf{tick3}{=}\mathsf{TR}_{\mathsf{tick}}])\end{aligned}$$

and

$$\mathsf{TR}_{\mathsf{tick}} \stackrel{\text{def}}{=} \lambda s, r.\lambda\grave{\sigma}, \acute{\sigma}.\lambda \grave{alloc}, \acute{alloc}.\mathsf{InvTable}(s, \grave{\sigma}) \supset \mathsf{InvTable}(s, \acute{\sigma})$$

we can prove in our logic,

$$\rhd [\mathsf{Table} : \mathsf{Spec}_{\mathsf{Table}} :: \lambda r.\lambda\grave{\sigma}, \acute{\sigma}.\lambda \grave{alloc}, \acute{alloc}.\mathsf{InvTable}(r, \prec, \acute{\sigma})]\ .$$

That is, InvTable is an *invariant* of the system. It is an invariant in the sense that it holds immediately after the table object is created, and it is invariant with respect to the actions of the three "buttons." Of course, $\mathsf{Spec}_{\mathsf{Fork}}$ and $\mathsf{Spec}_{\mathsf{Phil}}$ are specifications that are strong enough to describe the behaviour of fork and philosopher objects respectively.

Furthermore, we can prove, for table t, philosopher p, forks *fork*, *fork'* and store σ,

$$\mathsf{blocked}(p, \sigma) \supset \mathsf{is_holding}(p, fork, \sigma) \supset \sigma(p, \mathsf{fork1}) = fork \tag{3.1}$$

$$\mathsf{is_holding}(p, fork, \sigma) \supset \mathsf{waiting_for}(p, fork', \sigma) \supset \sigma(p, \mathsf{fork2}) = fork' \tag{3.2}$$

$$\begin{aligned}&\mathsf{InvTable}(t, orel, \sigma) \supset \\ &\quad \forall_{F(t)} fork.(\sigma(fork, \mathsf{on_table}) = \mathtt{ff} \supset \exists_{P(t)} p.\mathsf{is_holding}(p, fork, \sigma))\end{aligned} \tag{3.3}$$

and

$$\mathsf{InvTable}(t, orel, \sigma) \supset \forall_{P(t)} p.\ orel\,(\sigma(p, \mathsf{fork1}), \sigma(p, \mathsf{fork2}))\ . \tag{3.4}$$

Assuming this, it is straightforward to prove the corollary

$$\mathsf{InvTable}(t, orel, \sigma) \supset \neg \mathsf{all_blocked}(t, \sigma)\ , \tag{3.5}$$

as required.

3.5 Soundness

We now proceed to the main contribution of this chapter: soundness of our embedding. The basic concept is as follows. We construct an interpretation of our metalanguage, including the predicate and function symbols for our embedding of AL, and show that it models higher-order logic. Using this interpretation, we show that the axioms introduced for AL are indeed admissible.

First we give names to some sets which will be used throughout this section, followed by an overview of the notation for presheaf categories, which we use to construct our interpretation. It is well known that presheaf categories, since they form a topos, model intuitionistic higher-order logic, and, via restriction to double negation closed presheaves, also classical logic. However, we found no presheaf category that models our logic; a new construction was required. We then construct the category $\mathcal{D}$ in which we give the interpretation. Subsequently we explicitly interpret terms from our metalanguage as objects and morphisms in $\mathcal{D}$. Next we check that this interpretation also models the axioms of the program logic. Finally, we prove a soundness result in the spirit of that found in [AL98] using the interpretation.

3.5.1 Preliminaries

First we name some sets that will be used throughout the rest of this section. Transition relations are predicates over a return value and an initial and final

store. We introduce the following convenient abbreviation. (Recall, in the logic, a store is determined by a total function and a subset its domain.)

$$\mathsf{trans} \stackrel{\text{def}}{=} \mathrm{Pow}(\mathsf{val} \times ((\mathsf{val} \times \mathsf{fname}) \to \mathsf{val})^2 \times \mathrm{Pow}(\mathsf{val})^2) \ .$$

Also, we follow closely the syntactic definition of specifications and define the set of specifications inductively as the smallest set **specs** closed under the following rules.

$$\frac{}{\mathit{Bool} \in \mathsf{specs}}$$

$$\frac{}{\mathit{Nat} \in \mathsf{specs}}$$

$$\frac{A_i \in \mathsf{specs}\ ^{i=1..k} \quad B_j : \mathsf{locn} \to \mathsf{specs}\ ^{j=1..\ell} \quad U_j : \mathsf{locn} \to \mathsf{trans}\ ^{j=1..\ell}}{[f_i{=}A_i{}^{i=1..k}, m_j{=}\langle B_j, U_j\rangle^{j=1..\ell}] \in \mathsf{specs}} \ ,$$

where we write $[f_i{=}A_i{}^{i=1..k}, m_j{=}\langle B_j, U_j\rangle^{j=1..\ell}]$ to denote the record that maps f_i to A_i (for $i = 1..k$) and m_j to the pair $\langle B_j, U_j\rangle$ (for $j = 1..\ell$).

3.5.2 Presheaves

Let $\mathbb{C}$ be a small category, and **Set** be the category of sets and functions.

A (covariant) functor, or *presheaf*, F is given by a family of sets $(F_X)_{X \in \mathbb{C}}$ indexed by objects in $\mathbb{C}$, and for each morphism $f \in \mathbb{C}(X, Y)$, a function $F_f : F_X \to F_Y$ such that $F_{\mathrm{id}} = \mathrm{id}$ and $F_{g \circ f} = F_g \circ F_f$. If $x \in F_X$, we may write $x\restriction_Y$ for $F_f(x)$ when $f : X \to Y$ is clear from the context.

A *natural transformation* m from presheaf F to presheaf G, written $m : F \to G$ (or maybe $m : G \leftarrow F$), is given by a family $(m_X)_{(X \in \mathbb{C})}$ of maps $m_X : F_X \to G_X$ such that, for $f : X \to Y$, $G_f \circ m_X = m_Y \circ F_f$ (*naturality*). If $x \in F_X$, we may write $m(x)$ for $m_X(x)$.

We write $\hat{\mathbb{C}}$ for the functor category of presheaves from $\mathbb{C}$ into **Set** and natural transformations between them. The terminal object is the constant presheaf defined $\mathbf{1}_X \stackrel{\text{def}}{=} \mathbf{1}$ and $\mathbf{1}_f \stackrel{\text{def}}{=} \mathrm{id}$. For presheaves F, G, we define their product $F \times G$ pointwise as a product in **Set** as follows: $(F \times G)_X \stackrel{\text{def}}{=} F_X \times G_X$ and $(F \times G)_f \stackrel{\text{def}}{=} F_f \times G_f$. Category $\hat{\mathbb{C}}$ is cartesian closed and, for X in $\mathbb{C}$, an element f in the function space at X, $(F{\Rightarrow}G)_X$ (sometimes denoted $(G^F)_X$), is a family,

indexed by morphisms $u : X \to Y$, of natural transformations $f_{u,Y} : F_Y \to G_Y$, such that f is *compatible*:

$$(F \Rightarrow G)_X \stackrel{\text{def}}{=} \{(f_{u,Y})_{u:X\to Y} | (f_{u,Y})_u \text{ is compatible}\}$$

where compatibility is precisely commutativity of the square

$$\begin{array}{ccc} F_Y & \xrightarrow{f_{u,Y}} & G_Y \\ {\scriptstyle F_v}\downarrow & & \downarrow{\scriptstyle G_v} \\ F_Z & \xrightarrow[f_{v \circ u,Z}]{} & G_Z \end{array}$$

for all $u : X \to Y$ and $v : Y \to Z$. And for morphism $\rho : X \to W$, family $(f_{u,Y})_u \in (F \Rightarrow G)_X$, we define

$$(F \Rightarrow G)_\rho((f_{u,Y})_u) \stackrel{\text{def}}{=} (g_{u,Y})_u$$

where $(g_{u,Y})$ is defined pointwise $g_{u,Y}(x) \stackrel{\text{def}}{=} f_{u \circ \rho,Y}(x)$, since u now ranges over $W \to Y$. We shall write f_Y for $f_{u,Y}$ in case where u is clear from the context.

A presheaf F is said to be a *constant presheaf* exactly when $F_X = S$ for some set S, independent of X, and $F_u = \mathrm{id}_S$. Given a set S, we write ∇S for its corresponding constant presheaf. When F is a constant presheaf ∇S, the function space $F \Rightarrow G$ degenerates, and $f \in (F \Rightarrow G)_X$ is determined by a single function from S to G_X. This last fact is particularly convenient when giving interpretations to our constants.

For any presheaf F, a global element is a natural transformation $\mathbf{1} \to F$. In particular, using the characteristic property of function space $\hat{\mathbb{C}}(F, G \Rightarrow H) \cong \hat{\mathbb{C}}(F \times G, H)$, we have the isomorphisms

$$\hat{\mathbb{C}}(\mathbf{1}, F \Rightarrow G) \cong \hat{\mathbb{C}}(F, G)$$

and

$$\hat{\mathbb{C}}(\mathbf{1}, F \Rightarrow (G \Rightarrow H)) \cong \hat{\mathbb{C}}(F \times G, H) \ .$$

That is, finding a global element of a function space amounts to finding a natural transformation.

3.5.3 Category of interpretation

As mentioned previously, it is well known that presheaf categories model higher-order logic. However, we now show how the two obvious presheaf categories do not give intuitive models of our axioms. We then show how we use both these categories to construct the category in which we can give our interpretation, and then proceed to show that it models our axioms.

3.5.3.1 Categories $\mathbb{I}$ and $\mathbb{X}$

Let $\mathbb{I}$ be the category of finite sets and injective maps. The presheaf category $\hat{\mathbb{I}} \stackrel{\text{def}}{=} \mathbf{Set}^{\mathbb{I}}$ appears to be a sensible candidate category in which to interpret our metalanguage. As described in [Hof99], we can certainly interpret the higher-order syntax in a natural way. However, when we try to give an interpretation of $\mathsf{var}_{\mathsf{Val}}$, which, we recall, should be interpreted as the stack, we run into problems: any interpretation of $\mathsf{var}_{\mathsf{Val}}$ in $\hat{\mathbb{I}}$ is independent of the stack. We run into similar problems when trying to give interpretations to the predicates $[- : -]$ and $[- : - :: -]$, which should depend on the environment—a function that maps variables to specifications. To allow us to interpret these constants and predicates, we introduce the category $\mathbb{X}$.

Recall that locn is the set of locations and val is the set of values (constants and locations). Let the objects of $\mathbb{X}$ be worlds XSE, which are triples (X, S, E) where X is a set of variables, $S : X \to \mathsf{val}$ and $E : X \to \mathsf{specs}$ are functions; and let morphisms $u : XSE \to X'S'E'$ be injective maps $u : X \to X'$ such that, for all $x \in X$, both $S'(u(x)) = S(x)$ and $E'(u(x)) = E(x)$. Given a world XSE, we think of S as the stack, and E as the environment. Thus in $\hat{\mathbb{X}} \stackrel{\text{def}}{=} \mathbf{Set}^{\mathbb{X}}$, we can give natural interpretations to symbols such as $\mathsf{var}_{\mathsf{Val}}$, $[- : -]$ and $[- : - :: -]$. However, we now have further problems: the natural interpretation of programs in $\hat{\mathbb{X}}$ (as explained in [Hof99]) includes "programs" that can depend on the stack (and environment); in particular, such programs can be constructed if we have a function from F into programs, where presheaf F depends on the stack or environment.

Instead, we use categories $\hat{\mathbb{I}}$ and $\hat{\mathbb{X}}$ to construct a new category $\mathcal{D}$ in which

we give our interpretation. Before we present the construction, we take this opportunity to state the following property of $\mathbb{X}$.

Lemma 1 *For worlds $XSE, X'S'E', X''S''E'' \in \mathbb{X}$, and morphisms $f : XSE \to X'S'E'$, $g : XSE \to X''S''E''$, there is world $X'''S'''E''' \in \mathbb{X}$ and morphisms $u : X'S'E' \to X'''S'''E'''$, $v : X''S''E'' \to X'''S'''E'''$ such that $uf = vg$.*

To put this another way, given two divergent morphisms f, g, there is a world $X'''S'''E'''$ and morphisms u, v which complete the rhombus and give the commuting diagram diagram below.

$$\begin{array}{ccccc} & & XSE & & \\ & \swarrow^{f} & & \searrow^{g} & \\ X'S'E' & & & & X''S''E'' \\ & \searrow_{u} & & \swarrow_{v} & \\ & & X'''S'''E''' & & \end{array}$$

The proof is a straightforward, pointwise construction in **Set** and can be found in Section A.1.

3.5.3.2 Category $\mathcal{D}$

We define $\mathcal{D}$ to be the category whose objects are pairs $F = (F^{(0)}, F^{(1)})$, where $F^{(0)} \in \hat{\mathbb{I}}$, and $F^{(1)} \in \hat{\mathbb{X}}$. A morphism $f : F \to G$ is given by a pair $(f^{(0)}, f^{(1)})$ of morphisms $f^{(0)} \in \hat{\mathbb{I}}(F^{(0)}, G^{(0)})$, $f^{(1)} \in \hat{\mathbb{X}}(F^{(01)}, G^{(1)})$, where $-^{(01)} : \mathcal{D} \to \hat{\mathbb{X}}$ is defined on objects $F \in \mathcal{D}$ by ${F^{(01)}}_{XSE} \stackrel{\text{def}}{=} {F^{(0)}}_X \times {F^{(1)}}_{XSE}$ and ${F^{(01)}}_u \stackrel{\text{def}}{=} {F^{(0)}}_u \times {F^{(1)}}_u$ (since if $u \in \mathbb{X}(XSE, X'S'E')$ then, in particular, $u \in \mathbb{I}(X, X')$). Composition, in $\mathcal{D}$, of $f : F \to G$, $g : G \to H$ is defined at XSE by

$$\begin{aligned} {(g \circ f)^{(0)}}_X &\stackrel{\text{def}}{=} {g^{(0)}}_X \circ {f^{(0)}}_X \\ {(g \circ f)^{(1)}}_{XSE} &\stackrel{\text{def}}{=} {g^{(1)}}_{XSE} \circ \langle {f^{(0)}}_X \circ \pi_1, {f^{(1)}}_{XSE} \rangle \end{aligned}$$

and identity

$$\mathrm{id}_F{}^{(0)} \stackrel{\mathrm{def}}{=} \mathrm{id}_{F^{(0)}}$$
$$\mathrm{id}_F{}^{(1)} \stackrel{\mathrm{def}}{=} \pi_2 \ ,$$

where $\pi_2 : F^{(01)} \rightarrow F^{(1)}$ is a projection. It is easy to check this definition of composition is indeed associative, i.e., $\mathcal{D}$ really is a category. Thus, for $f : F \rightarrow G$, $f^{(01)}$ at XSE is defined by

$$f^{(01)}{}_{XSE} \stackrel{\mathrm{def}}{=} \langle f^{(0)}{}_X \circ \pi_1, f^{(1)}{}_{XSE} \rangle \ ,$$

which can easily be shown to be a natural transformation, i.e., a morphism of $\hat{\mathbb{X}}$. It is also straightforward to check that $\mathrm{id}_F{}^{(01)} = \mathrm{id}_{F^{(01)}}$ and $(gf)^{(01)} = g^{(01)} f^{(01)}$, i.e., $-^{(01)}$ is a functor.

For objects $F, G \in \mathcal{D}$, and morphism $m : F \rightarrow G$, we sometimes write F_{XSE} and m_{XSE} for respectively $F^{(01)}{}_{XSE}$ and $m^{(01)}{}_{XSE}$ in the absence of ambiguity.

We define product by $(F^{(0)}, F^{(1)}) \times (G^{(0)}, G^{(1)}) \stackrel{\mathrm{def}}{=} (F^{(0)} \times G^{(0)}, F^{(1)} \times G^{(1)})$. It immediately follows that $-^{(0)}$, $-^{(1)}$ and $-^{(01)}$ preserve products. For $F, G \in \mathcal{D}$, we define function space by $(F{\Rightarrow}G) \stackrel{\mathrm{def}}{=} (F^{(0)}{\Rightarrow}G^{(0)}, F^{(01)}{\Rightarrow}G^{(1)})$. Note that the exponent of the right component of the function space is $F^{(01)}$. And

$$\begin{array}{ll} \mathcal{D}(E \times F, G) & \\ \cong \hat{\mathbb{I}}(E^{(0)} \times F^{(0)}, G^{(0)}) \times \hat{\mathbb{X}}(E^{(01)} \times F^{(01)}, G^{(1)}) & \text{by def. of morphism in } \mathcal{D} \\ \cong \hat{\mathbb{I}}(E^{(0)}, F^{(0)}{\Rightarrow}G^{(0)}) \times \hat{\mathbb{X}}(E^{(01)}, F^{(01)}{\Rightarrow}G^{(1)}) & \text{since } \hat{\mathbb{I}}, \hat{\mathbb{X}} \text{ are cartesian closed} \\ \cong \mathcal{D}(E, F{\Rightarrow}G) & \text{by def. of exponential in } \mathcal{D}, \end{array}$$

that is, $\mathcal{D}$ is cartesian closed.

3.5.3.3 Properties of $\mathcal{D}$

To aid presentation, we introduce functors $\Delta : \hat{\mathbb{I}} \rightarrow \mathcal{D}$ and $\Psi : \hat{\mathbb{X}} \rightarrow \mathcal{D}$, defined as follows, where **1** denotes the terminal object (and we overload notation by using

the same symbol for different terminal objects).

$$\Delta F \stackrel{\text{def}}{=} (F, \mathbf{1})$$
$$\Delta u \stackrel{\text{def}}{=} (u, \mathbf{1})$$
$$\Psi G \stackrel{\text{def}}{=} (\mathbf{1}, G)$$
$$\Psi u \stackrel{\text{def}}{=} (\mathbf{1}, u \circ \pi)$$

where $\pi : F \times G \to G$ is a projection. The following theorem states some useful properties of $\mathcal{D}$.

Theorem 1 *1. Functor Δ is full and faithful. That is, $\mathcal{D}(\Delta F, \Delta G) \cong \hat{\mathbb{I}}(F, G)$.*

2. Functor Δ is right adjoint to $-^{(0)}$. That is, we have a natural isomorphism $\hat{\mathbb{I}}(F^{(0)}, G) \cong \mathcal{D}(F, \Delta G)$.

3. Functor Δ preserves products. That is $\Delta(F \times G) \cong \Delta F \times \Delta G$.

4. $F \Rightarrow \Delta G \cong \Delta(F^{(0)} \Rightarrow G)$

5. Functor Ψ is right adjoint to $-^{(01)}$.

6. Functor Ψ preserves products.

7. $\Psi F \Rightarrow \Psi G \cong \Psi(F \Rightarrow G)$

8. $F \Rightarrow \Psi G \cong \Psi(F^{(01)} \Rightarrow G)$

9. $\Delta F \Rightarrow \Psi G \cong \Psi((\Delta F)^{(01)} \Rightarrow G)$

10. $\Psi F \Rightarrow \Delta G \cong \Delta G$

The proofs are straightforward. For example, we can prove 3 as follows. For arbitrary $F \in \mathcal{D}$,

$$\begin{aligned}
\mathcal{D}(F, \Delta(G \times H)) &\cong \hat{\mathbb{I}}(F^{(0)}, G \times H) && \text{since } \Delta \text{ is r. adj. to } -^{(0)} \\
&\cong \hat{\mathbb{I}}(F^{(0)}, G) \times \hat{\mathbb{I}}(F^{(0)}, H) && \text{characteristic property of } \times \\
&\cong \mathcal{D}(F, \Delta G) \times \mathcal{D}(F, \Delta H) && \text{since } \Delta \text{ is r. adj. to } -^{(0)} \\
&\cong \mathcal{D}(F, \Delta G \times \Delta H) && \text{characteristic property of } \times.
\end{aligned}$$

And since clearly $F^{(01)} \times G \cong (F \times \Psi G)^{(01)}$, we can prove 7 as follows. For arbitrary $F \in \mathcal{D}$,

$$\begin{aligned}
\mathcal{D}(F, \Psi(G \Rightarrow H)) &\cong \hat{\mathbb{X}}(F^{(01)}, G \Rightarrow H) && \text{since } \Psi \text{ is r. adj. to } -^{(01)} \\
&\cong \hat{\mathbb{X}}(F^{(01)} \times G, H) && \text{since } \hat{\mathbb{X}} \text{ is a ccc} \\
&\cong \hat{\mathbb{X}}((F \times \Psi G)^{(01)}, H) && \text{by earlier observation} \\
&\cong \mathcal{D}(F \times \Psi G, \Psi H) && \text{since } \Psi \text{ is r. adj. to } -^{(01)} \\
&\cong \mathcal{D}(F, \Psi G \Rightarrow \Psi H) && \text{since } \mathcal{D} \text{ is a ccc.}
\end{aligned}$$

Part 4 is an immediate consequence of part 2, as can be seen below. For arbitrary $H \in \mathcal{D}$, since we know $(F \times G)^{(0)} = F^{(0)} \times G^{(0)}$,

$$\begin{aligned}
\mathcal{D}(H, F \Rightarrow \Delta G) &\cong \mathcal{D}(H \times F, \Delta G) && \text{since } \mathcal{D} \text{ is a ccc} \\
&\cong \hat{\mathbb{I}}(H^{(0)} \times F^{(0)}, G) && \text{by part 2 and previous observation} \\
&\cong \hat{\mathbb{I}}(H^{(0)}, F^{(0)} \Rightarrow G) && \text{since } \hat{\mathbb{I}} \text{ is a ccc} \\
&\cong \mathcal{D}(H, \Delta(F^{(0)} \Rightarrow G)) && \text{by part 2.}
\end{aligned}$$

Similarly, parts 8 and 9 are immediate consequences of part 5.

In particular, Theorem 1 tells us the following facts. From (1), we know that to give a morphism of type $\Delta F \to \Delta G$ is to give a morphism in $\hat{\mathbb{I}}(F, G)$. Similarly, to give a morphism $\Psi F \to \Psi G$ is to give a global element of $\Psi F \Rightarrow \Psi G$, which, by (7), is to give a global element of $\Psi(F \Rightarrow G)$, which is to give a global element of $F \Rightarrow G$, i.e., a morphism in $\hat{\mathbb{X}}(F, G)$. And, from (10), we know that there are only constant morphisms of type $\Psi F \to \Delta G$. This last fact allows us to give interpretations to our types so that the only nontrivial morphisms into $[\![\mathsf{Prg}]\!]$ are those from $[\![\mathsf{Fn}]\!]$, $[\![\mathsf{Mn}]\!]$ and $[\![\mathsf{Var}]\!]$; specifically, there are no nontrivial maps $[\![\mathsf{nat}]\!]$ nor $[\![\mathsf{bool}]\!]$ to $[\![\mathsf{Prg}]\!]$.

Finally we also have the following naturality lemma for morphisms in $\mathcal{D}$. The proof falls out immediately from the fact that $m = (m^{(0)}, m^{(1)})$ and $m^{(0)}, m^{(1)}$ are natural transformations.

Lemma 2 *For all $F, G \in \mathcal{D}$ and $f : XSE \to X'S'E'$, the morphism $m : G \to F$ is natural in the sense that ${F^{(01)}}_f \circ m_{XSE} = m_{X'S'E'} \circ {G^{(01)}}_f$.*

3.5.4 Interpretation

We now give interpretations to the terms and types of our metalanguage.

Each type τ is interpreted as an object $[\![\tau]\!]$ in $\mathcal{D}$. Function space in the metalanguage is interpreted as function space in $\mathcal{D}$; namely $[\![\sigma \to \tau]\!] \stackrel{\text{def}}{=} [\![\sigma]\!] \Rightarrow [\![\tau]\!]$. Similarly, products in the metalanguage are products in $\mathcal{D}$: namely $[\![\sigma \times \tau]\!] \stackrel{\text{def}}{=} [\![\sigma]\!] \times [\![\tau]\!]$.

For typing context $\Gamma = x_1{:}\tau_1, \ldots, x_k{:}\tau_k$, we define its interpretation by $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{x_i \in \mathsf{dom}\,\Gamma} [\![\tau_i]\!]$. For context Γ, term a and type τ, the typing judgement $\Gamma \rhd a : \tau$ is interpreted as a morphism (of $\mathcal{D}$) $[\![\Gamma \rhd a : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$. When there is no risk of confusion, we simply denote its interpretation as $[\![a]\!]$.

Each constant c of type τ has interpretation $[\![c]\!]$, a global element of $[\![\tau]\!]$, that is, a morphism $\mathbf{1} \to [\![\tau]\!]$.

For terms e, e' supposing $t = [\![\Gamma \rhd e : \sigma \to \tau]\!]$ and $t' = [\![\Gamma \rhd e' : \sigma]\!]$, we define $[\![\Gamma \rhd ee' : \tau]\!] \stackrel{\text{def}}{=} \mathrm{ev} \circ \langle t', t\rangle$ where ev is an evaluation morphism of $\mathcal{D}$ (which we know is a cartesian-closed category). Also, for term e, supposing $t = [\![\Gamma, x{:}\sigma \rhd e : \tau]\!] : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\tau]\!]$, we define $[\![\Gamma \rhd \lambda x^{\sigma}.e : \sigma \to \tau]\!] \stackrel{\text{def}}{=} \mathrm{curry}(t) : [\![\Gamma]\!] \to [\![\sigma]\!] \Rightarrow [\![\tau]\!]$. And for $[\![x]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$, world XSE and $\eta \in [\![\Gamma]\!]$, we define $[\![x]\!]_{XSE}(\eta) \stackrel{\text{def}}{=} \eta(x)$.

3.5.4.1 Predicates

We now construct a contravariant functor $\mathbf{Pred} : \mathcal{D}^{\mathrm{Op}} \to \mathbf{Poset}$, which we show satisfies criteria amounting to a certain model of higher-order logic, more precisely a tripos in the sense of [HJP80, Pit02]. We then appeal to a theorem of triposes (namely "soundness"), and conclude that our interpretation models higher-order logic.

Definition 1 *Let G be a presheaf in $\mathcal{D}$. A* predicate *over G is given by a subset $P_{XSE} \subseteq G^{(01)}{}_{XSE}$ for each world XSE, such that for all worlds $X'S'E'$, whenever $f : XSE \to X'S'E'$,*

$$g \in P_{XSE} \quad \textit{iff} \quad G^{(01)}{}_f(g) \in P_{X'S'E'} \;.$$

We define

$$\mathbf{Pred}(G) \stackrel{\text{def}}{=} \{P \mid P \text{ is a predicate over } G\} \;.$$

And we define a partial order $\leq$ over $\mathbf{Pred}(G)$ by a pointwise lifting of subset inclusion: for $U, V \in \mathbf{Pred}(G)$, we define $U \leq V$ iff $U_{XSE} \subseteq V_{XSE}$ for all worlds XSE. Suppose $m \in \mathcal{D}^{\mathrm{op}}(F, G)$, that is, $m : F \leftarrow G$. For $P \in \mathbf{Pred}(F)$ and world XSE, define

$$Q_{XSE} \stackrel{\mathrm{def}}{=} \{x \in G^{(01)}{}_{XSE} \mid m_{XSE}(x) \in P_{XSE}\} \ .$$

We now proceed to show that $Q \in \mathbf{Pred}(G)$.

Suppose $x \in Q_{XSE}$, that is, $m_{XSE}(x) \in P_{XSE}$. Since P is a predicate, we know that $m_{XSE}(x)\restriction_{X'S'E'} \in P_{X'S'E'}$. Thus by naturality, we know that $m_{X'S'E'}(x\restriction_{X'S'E'}) \in P_{X'S'E'}$. So by definition of Q, we conclude $x\restriction_{X'S'E'} \in Q_{X'S'E'}$.

Conversely, we show that if $x\restriction_{X'S'E'} \in Q_{X'S'E'}$ then $x \in Q_{XSE}$ by using the same arguments as above but backwards.

So we conclude that $Q \in \mathbf{Pred}(G)$. We define $\mathbf{Pred}(m)(P) \stackrel{\mathrm{def}}{=} Q$, which gives $\mathbf{Pred}(m) : \mathbf{Pred}(G) \to \mathbf{Pred}(F)$, as required.

Given $m : G \to F$, we define a function $\forall_m : \mathbf{Pred}(G) \to \mathbf{Pred}(F)$ as follows. For $U \in \mathbf{Pred}(G)$ and world XSE, define

$$(\forall_m(U))_{XSE} \stackrel{\mathrm{def}}{=} \{x \in F^{(01)}{}_{XSE} \mid \begin{array}{l} \text{for all } f : XSE \to X'S'E', \\ \text{for all } a \in G^{(01)}{}_{X'S'E'}, \\ m(a) = x\restriction_{X'S'E'} \text{ implies } a \in U_{X'S'E'}\} \ . \end{array}$$

We now show that $\forall_m(U) \in \mathbf{Pred}(F)$.

Let $f : XSE \to X'S'E'$.

Assume $x \in (\forall_m U)_{XSE}$. We can show $F^{(01)}{}_f(x) \in (\forall_m U)_{X'S'E'}$ as follows. Let $X''S''E'' \in \mathbb{X}$, $g : X'S'E' \to X''S''E''$ and $a \in G^{(01)}{}_{X''S''E''}$. Suppose $m_{X''S''E''}(a) = F^{(01)}{}_g(F^{(01)}{}_f(x))$. Since $x \in (\forall_m U)_{XSE}$ and (i) $gf : XSE \to X''S''E''$, (ii) $a \in G^{(01)}{}_{X''S''E''}$, and (iii) $m(a) = F^{(01)}{}_g(F^{(01)}{}_f(x)) = x\restriction_{X''S''E''}$, we conclude that $a \in U_{X''S''E''}$. That is, $F^{(01)}{}_f(x) \in (\forall_m U)_{X'S'E'}$.

Conversely, assume $F^{(01)}{}_f(x) \in (\forall_m U)_{X'S'E'}$. We show that $x \in (\forall_m U)_{XSE}$ using Lemma 1, as follows. Let $X''S''E'' \in \mathbb{X}$, $g : XSE \to X''S''E''$ and $a \in G^{(01)}{}_{X''S''E''}$. Suppose $m_{X''S''E''}(a) = F^{(01)}{}_g(x)$. By appealing to Lemma 1, we

know there is world $X'''S'''E'''$ and morphisms $u : X'S'E' \to X'''S'''E'''$ and $v : X''S''E'' \to X'''S'''E'''$ such that $uf = vg$. We calculate:

$F^{(01)}{}_{uf}(x)$	
$= F^{(01)}{}_{vg}(x)$	by assumption
$= F^{(01)}{}_{v}(m_{X''S''E''}(a))$	since $m_{X''S''E''}(a) = F^{(01)}{}_{g}(x)$
$= m_{X'''S'''E'''}(G^{(01)}{}_{v}(a))$	by naturality (Lemma 2)

Since $F^{(01)}{}_f(x) \in (\forall_m U)_{X'S'E'}$ and (i) $u : X'S'E' \to X'''S'''E'''$, (ii) $G^{(01)}{}_v(a) \in G^{(01)}{}_{X'S'E'}$, and (iii) $F^{(01)}{}_u(F^{(01)}{}_f(x)) = m^{X'''S'''E'''}(G^{(01)}{}_v(a))$, we conclude that $G^{(01)}{}_v(a) \in U_{X'''S'''E'''}$. And since U is a predicate, we have $a \in U_{X''S''E''}$, as required.

So we conclude that $\forall_m(U) \in \mathbf{Pred}(F)$.

Functor $\mathbf{Pred} : \mathcal{D}^{\mathrm{Op}} \to \mathbf{Poset}$ satisfies particular properties listed in the next theorem. Note that these are essentially those of a *tripos* as defined in [HJP80] except (1) the Beck condition has been weakened and (2) $\mathbf{Pred}(F)$ forms a *Boolean algebra* (vs. Heyting algebra) for $F \in \mathcal{D}$. This is because we are interested in a model of *classical higher-order logic* (vs. intuitionistic) and, for our purposes, the Beck condition in its restricted form as in the theorem below suffices for our purposes. We abuse convention (not just notation!) and call this a tripos.[4]

Theorem 2 *Functor* $\mathbf{Pred} : \mathcal{D}^{\mathrm{Op}} \to \mathbf{Poset}$ *forms a* tripos. *That is, it has the following properties:*

- *the sets* $\mathbf{Pred}(F)$ *form a Boolean algebra with respect to subset inclusion, and this structure is preserved by the morphism part of* $\mathbf{Pred}$*;*
- *the functor* $\mathbf{Pred}$ *is* representable, *i.e. there is an object* $Prop \in \mathcal{D}$ *so that* $\mathbf{Pred}(F) \cong \mathcal{D}(F, Prop)$*;*
- *for each morphism* $m : G \to F$*, there is a function* $\forall_m : \mathbf{Pred}(G) \to \mathbf{Pred}(F)$ *so that* $U \leq \forall_m(V)$ *iff* $\mathbf{Pred}(m)(U) \leq V$*, where* $\leq$ *is the order of the boolean algebra; and*

[4]We note that the definitions of tripos differ in [HJP80] and [Pit02] (as explained in Remark 4.6 of the latter).

- *quantification commutes with substitution in the sense that: for* $A, B, C \in \mathcal{D}$, $f : A \to B$ *and* $U \in \mathbf{Pred}(B \times C)$, *we have* $\mathbf{Pred}(f)(\forall_{\pi_B}(U)) = \forall_{\pi_A}(\mathbf{Pred}(f \times id)(U))$, *where* $\pi_B : B \times C \to B$ *and* $\pi_A : A \times C \to A$ *are projections.*

The proof can be found in Section A.1.

We explicitly name the bijections in the proof of $\mathbf{Pred}(F) \cong \mathcal{D}(F, \mathit{Prop})$:

$$\kappa_F : \mathbf{Pred}(F) \leftarrow \mathcal{D}(F, \mathit{Prop})$$
$$\chi_F : \mathbf{Pred}(F) \to \mathcal{D}(F, \mathit{Prop}) \ .$$

And, as defined in the proof, we recall

$$\mathit{Prop} \stackrel{\text{def}}{=} \Psi\nabla\{\mathtt{ff}, \mathtt{tt}\} \ .$$

Now, provided we interpret Prop as *Prop* and $\forall$ and $\supset$ appropriately, we may conclude that our interpretation models higher-order logic. More explicitly, we note that

$$\mathrm{ev}_{F,\mathit{Prop}} : (F \Rightarrow \mathit{Prop}) \times F \to \mathit{Prop}$$

and for projection $\pi : (F \Rightarrow \mathit{Prop}) \times F \to F \Rightarrow \mathit{Prop}$,

$$\forall_\pi : \mathbf{Pred}((F \Rightarrow \mathit{Prop}) \times F) \to \mathbf{Pred}(F \Rightarrow \mathit{Prop}) \ ,$$

and so we define

$$\mathit{forall}_F \stackrel{\text{def}}{=} \forall_\pi(\kappa(\mathrm{ev}_{F,\mathit{Prop}})) \in \mathbf{Pred}(F \Rightarrow \mathit{Prop})$$
$$[\![\forall_\sigma]\!] \stackrel{\text{def}}{=} \mathrm{curry}(\chi(\mathit{forall}_{[\![\sigma]\!]})) \in \mathcal{D}(\mathbf{1}, ([\![\sigma]\!] \Rightarrow \mathit{Prop}) \Rightarrow \mathit{Prop}) \ .$$

We define pointwise, for $x, y \in \{\mathtt{ff}, \mathtt{tt}\}$:

$$[\![\supset]\!](x, y) \stackrel{\text{def}}{=} \begin{cases} \mathtt{tt} & \text{if } x = \mathtt{ff} \text{ or } y = \mathtt{tt} \\ \mathtt{ff} & \text{otherwise.} \end{cases}$$

To show that $[\![\supset]\!]$ is well-defined, we note that, formally, we require

$$\begin{aligned} [\![\supset]\!] &\in \mathcal{D}(\mathbf{1}, \mathit{Prop}{\Rightarrow}\mathit{Prop}{\Rightarrow}\mathit{Prop}) && \text{(global elt. in } \mathit{Prop}{\Rightarrow}(\mathit{Prop}{\Rightarrow}\mathit{Prop})) \\ &\cong \mathcal{D}(\mathit{Prop} \times \mathit{Prop}, \mathit{Prop}) && \text{since } \mathcal{D} \text{ is a ccc} \\ &\cong \hat{\mathbb{X}}(\nabla\{\mathtt{ff},\mathtt{tt}\}^2, \nabla\{\mathtt{ff},\mathtt{tt}\}) && \text{since } \mathit{Prop} \stackrel{\text{def}}{=} \Psi\nabla\{\mathtt{ff},\mathtt{tt}\}. \end{aligned}$$

However, for sets A, B and presheaf category $\hat{\mathbb{C}}$, we know that $\hat{\mathbb{C}}(\nabla A, \nabla B) \cong \mathbf{Set}(A, B)$. So it suffices to define $[\![\supset]\!]$ as a function $\{\mathtt{ff},\mathtt{tt}\}^2 \to \{\mathtt{ff},\mathtt{tt}\}$.

So we conclude that our interpretation models higher-order logic. That is, using any reasonable axioms and rules for higher-order logic, we have $\Gamma \rhd P$, if and only if $[\![P]\!] : [\![\Gamma]\!] \to \mathit{Prop}$ is the constant morphism $\eta \mapsto \mathtt{tt}$.

3.5.4.2 Forcing

We now introduce *forcing judgements* which help us reason about interpretations of predicates.

Let XSE be a world, Γ be a context and suppose $\eta \in [\![\Gamma]\!]_{XSE}$, i.e. $\eta(x) \in [\![\Gamma]\!]_{XSE}(x)$ for all $x \in \mathsf{dom}\,\Gamma$. If $\Gamma \rhd P : \mathsf{Prop}$ then we write

$$XSE \Vdash_{\Gamma,\eta} P$$

to mean

$$[\![P]\!]_{XSE}(\eta) = \mathtt{tt} \ .$$

We overload notation again and define, for $u : XSE \to X'S'E'$, the notation $\eta\restriction_u(x) \stackrel{\text{def}}{=} [\![\Gamma(x)]\!]_u(\eta(x))$. A consequence of this definition and that of predicate is the following *invariant*: whenever $u : XSE \to X'S'E'$, then $XSE \Vdash_{\Gamma,\eta} P$ if and only if $X'S'E' \Vdash_{\Gamma,\eta\restriction_u} P$. Moreover, a relation that satisfies this invariant corresponds to a predicate.

Using this notation, it follows that

$$\begin{aligned} & XSE \Vdash_{\Gamma,\eta} \forall_\tau x.P \\ \text{iff} \quad & \forall u : XSE \to X'S'E'.\forall a \in [\![\tau]\!]_{X'S'E'}.X'S'E' \Vdash_{\Gamma[x\mapsto\tau],(\eta\restriction_u)[x\mapsto a]} P \end{aligned}$$

and

$$XSE \Vdash_{\Gamma,\eta} P \supset Q \quad \text{iff} \quad XSE \Vdash_{\Gamma,\eta} P \text{ implies } XSE \Vdash_{\Gamma,\eta} Q \ .$$

It is a straightforward calculation to show that these two equivalences follow from our definitions of $[\![\forall_\sigma]\!]$ and $[\![\supset]\!]$.

We note that $U_{XSE} \stackrel{\text{def}}{=} \emptyset$ defines a predicate and so there is no world XSE, Γ, η such that $XSE \Vdash_{\Gamma,\eta} \mathsf{false}$ since $\mathsf{false} \stackrel{\text{def}}{=} \forall_{\mathsf{Prop}} P.P$ and we can always choose U as an instantiation for P.

Thus we obtain the corresponding results for the other logical connectives, namely

1. $XSE \Vdash_{\Gamma,\eta} P \wedge Q$ iff $XSE \Vdash_{\Gamma,\eta} P$ and $XSE \Vdash_{\Gamma,\eta} Q$;

2. $XSE \Vdash_{\Gamma,\eta} P \vee Q$ iff $XSE \Vdash_{\Gamma,\eta} P$ or $XSE \Vdash_{\Gamma,\eta} Q$;

3. $XSE \Vdash_{\Gamma,\eta} \neg P$ iff not $XSE \Vdash_{\Gamma,\eta} P$; and

4. $XSE \Vdash_{\Gamma,\eta} \exists_\tau x.P$ iff there is world $X'S'E'$, $u : XSE \to X'S'E'$ and $a \in [\![\tau]\!]_{X'S'E'}$ such that $X'S'E' \Vdash_{\Gamma',\eta'} P$, where $\Gamma' \stackrel{\text{def}}{=} \Gamma[x \mapsto \tau]$ and $\eta' \stackrel{\text{def}}{=} (\eta\restriction_u)[x \mapsto a]$.

3.5.4.3 Leibniz equality

We now see our first application of the forcing notation by showing that the interpretation of Leibniz equality is equality in the interpretation.

First we show that equality over certain presheaves in the interpretation is a predicate. Let $F \in \mathcal{D}$ and suppose that F preserves injectivity of morphisms in $\mathbb{X}$, viz F_u is injective for morphism u in $\mathbb{X}$ (which are, in particular, injective functions).

We define a family $(\delta_F)_{XSE} \subseteq (F^{(01)} \times F^{(01)})_{XSE}$ indexed by worlds XSE by

$$(\delta_F)_{XSE} \stackrel{\text{def}}{=} \{(x,x) \mid x \in F^{(01)}{}_{XSE}\} \ .$$

We observe, for morphism $u : XSE \to X'S'E'$: if $(x,x) \in (\delta_F)_{XSE}$ then $(F^{(01)} \times F^{(01)})_u(x,x) = (F^{(01)}{}_u(x), F^{(01)}{}_u(x)) \in (\delta_F)_{X'S'E'}$; conversely, if $(F^{(01)} \times$

$F^{(01)})_u(x,y) \in (\delta_F)_{X'S'E'}$,then $F^{(01)}{}_u(x) = F^{(01)}{}_u(y)$, and so by supposition, we conclude $x = y$, i.e., $(x,y) \in (\delta_F)_{XSE}$. That is, δ_F is a predicate over F^2.

And since

$$\delta_F \in \mathbf{Pred}(F^2) \cong \mathcal{D}(F^2, Prop) \cong \mathcal{D}(F, F \Rightarrow Prop) \ ,$$

we may consider δ_F as a morphism δ'_F from F into $\mathcal{D}(F, Prop) \cong \mathcal{D}(\mathbf{1}, F \Rightarrow Prop)$.

Suppose $\Gamma \rhd a =^{\tau} b$ and $[\![\tau]\!]$ preserves injectivity of morphisms. By definition of Leibniz equality, this is precisely $\Gamma \rhd \forall_{\tau \to \mathsf{Prop}} P.P(a) \supset P(b)$. Since we know our interpretation models higher-order logic, we know for all XSE and $\eta \in [\![\Gamma]\!]_{XSE}$,

$$XSE \Vdash_{\Gamma,\eta} \forall_{\tau \to \mathsf{Prop}} P.P(a) \supset P(b) \ .$$

That is, for $u : XSE \to X'S'E'$ and $p \in (\mathbf{Pred}[\![\tau]\!])_{X'S'E'}$,

$$X'S'E' \Vdash_{\Gamma',\eta'} P(a) \quad \text{implies} \quad X'S'E' \Vdash_{\Gamma',\eta'} P(b)$$

where $\Gamma' \stackrel{\text{def}}{=} \Gamma[P \mapsto (\tau \to \mathsf{Prop})]$ and $\eta' \stackrel{\text{def}}{=} (\eta\restriction_u)[P \mapsto p]$.

So, choosing

$$\text{id} : XSE \to XSE \quad \text{and} \quad p \stackrel{\text{def}}{=} (\delta'_{[\![\tau]\!]})_{XSE}([\![a]\!]_{XSE}(\eta)) \subseteq [\![\tau]\!]^{(01)}{}_{XSE}(\eta) \ ,$$

since we know $[\![a]\!]_{XSE}(\eta) \in \{[\![a]\!]_{XSE}(\eta)\} = (\delta'_{[\![\tau]\!]})_{XSE}([\![a]\!]_{XSE})$, i.e., $X'S'E' \Vdash_{\Gamma',\eta'} P(a)$, we must have $X'S'E' \Vdash_{\Gamma',\eta'} P(b)$, that is, $[\![b]\!]_{XSE} \in \{[\![a]\!]_{XSE}\}$. That is, $[\![b]\!]_{XSE}(\eta) = [\![a]\!]_{XSE}(\eta)$.

Conversely, suppose $[\![a]\!]_{XSE}(\eta) = [\![b]\!]_{XSE}(\eta)$. So by naturality, for any $u : XSE \to X'S'E'$, $[\![a]\!]_{X'S'E'}(\eta') = [\![b]\!]_{X'S'E'}(\eta')$, where $\eta' \stackrel{\text{def}}{=} \eta\restriction u$. So certainly, $X'S'E' \Vdash_{\Gamma',\eta'} P(a)$ implies $X'S'E' \Vdash_{\Gamma',\eta'} P(b)$.

Thus, provided $[\![\tau]\!]$ preserves injectivity of morphisms, $\Gamma \rhd a =^{\tau} b$ iff $[\![a]\!] = [\![b]\!]$.

So, when does $F \in \mathcal{D}$ satisfy this injectivity requirement? The answer is, if and only if $F^{(0)}$ and $F^{(1)}$ both preserve injectivity of morphisms: for $u : XSE \to X'S'E'$,

> $F^{(0)}{}_u$ is injective (for u considered as a morphism of $\mathbb{I}$) and
>
> $F^{(1)}{}_u$ is injective.

This is straightforward to show.

In particular, for any set A, the constant presheaf ∇A preserves injectivity of morphisms. Thus *Prop* preserves injectivity of morphisms.

Furthermore, we can show for presheaves F, G, H where F, G preserve injectivity of morphisms, then so do $F \times G$ and $H \Rightarrow F$. Thus we obtain the corresponding results about objects in $\mathcal{D}$.

In fact, we interpret all the types introduced in Section 3.2 as objects in $\mathcal{D}$ which preserve injectivity of morphisms.

3.5.4.4 Base types

We interpret $[\![\mathsf{Var}]\!] \stackrel{\text{def}}{=} \Delta V$ and $[\![\mathsf{Prg}]\!] \stackrel{\text{def}}{=} \Delta P$, where V and P are presheaves defined as follows. For $X, Y \in \mathbb{I}$ and $u : X \to Y$,

$$\begin{aligned} V_X &\stackrel{\text{def}}{=} X \\ V_u(x) &\stackrel{\text{def}}{=} u(x) \\ P_X &\stackrel{\text{def}}{=} \text{programs with free variables among } X\text{, modulo } \alpha\text{-conversion} \\ P_u(a) &\stackrel{\text{def}}{=} a[u] \ , \end{aligned}$$

where we write $a[u]$ to denote the program a with all free occurrences of variables x replaced by $u(x)$. For example, given

$$\begin{aligned} & a \stackrel{\text{def}}{=} \textit{let } z\textit{=false in if } z \textit{ then } x \textit{ else } y \\ & \quad \in P_{\{x,y\}} \\ & u : \{x, y\} \to \{p, q, r\} \\ & u \stackrel{\text{def}}{=} \begin{cases} x \mapsto p \\ y \mapsto q \ . \end{cases} \end{aligned}$$

we define $a[u] \stackrel{\text{def}}{=} \textit{let } z\textit{=false in if } z \textit{ then } p \textit{ else } q$, thus, $a[u] \in P_{\{p,q,r\}}$. Clearly, V, P preserve injectivity of morphisms since morphisms in $\mathbb{X}$ are, in particular, injective functions.

It suffices for us to give only special cases for the definition of $[\![\mathsf{Rcd}_{\tau_1}^{\tau_2}]\!]$. First of all, let us introduce two auxiliary presheaves for the two special cases. Supposing

that S is a set, $F \in \hat{\mathbb{I}}$ and $G \in \hat{\mathbb{X}}$, we define pointwise

$$(Q_S^F)_X \stackrel{\text{def}}{=} S \rightharpoonup (F_X)$$
$$(Q_S^F)_u(f) \stackrel{\text{def}}{=} F_u \circ f$$

where $u : X \rightarrow X'$ and $f : S \rightharpoonup F_{X'}$, and

$$(R_S^G)_{XSE} \stackrel{\text{def}}{=} S \rightharpoonup (G_{XSE})$$
$$(R_S^G)_v(g) \stackrel{\text{def}}{=} G_v \circ g$$

where $v : XSE \rightarrow X'S'E'$ and $g : S \rightharpoonup G_{X'S'E'}$. Now supposing $[\![\tau_1]\!] = \Delta\nabla S$ for some set S, $[\![\tau_2]\!] = \Delta F$ for some $F \in \hat{\mathbb{I}}$ and $[\![\tau_3]\!] = \Psi G$ for some $G \in \hat{\mathbb{X}}$, we can define the two special cases by

$$[\![\mathsf{Rcd}_{\tau_1}^{\tau_2}]\!] \stackrel{\text{def}}{=} \Delta(Q_S^F)$$
$$[\![\mathsf{Rcd}_{\tau_1}^{\tau_3}]\!] \stackrel{\text{def}}{=} \Psi(R_S^G) \ .$$

It can easily be checked that these two special cases are sufficient for us to interpret all our defined types and axioms. Clearly, provided $[\![\tau]\!]$ preserves injectivity of morphisms, then so does $[\![\mathsf{Rcd}_\sigma^\tau]\!]$.

The remaining types are defined as the following constant objects of $\mathcal{D}$.

$$[\![\mathsf{Fn}]\!] \stackrel{\text{def}}{=} \Delta\nabla\mathsf{fname}$$
$$[\![\mathsf{Mn}]\!] \stackrel{\text{def}}{=} \Delta\nabla\mathsf{mname}$$
$$[\![\mathsf{Val}]\!] \stackrel{\text{def}}{=} \Psi\nabla\mathsf{val}$$
$$[\![\mathsf{Sp}]\!] \stackrel{\text{def}}{=} \Psi\nabla\mathsf{specs}$$
$$[\![\mathsf{nat}]\!] \stackrel{\text{def}}{=} \Psi\nabla\mathbb{N}$$
$$[\![\mathsf{bool}]\!] \stackrel{\text{def}}{=} \Psi\nabla\mathbb{B} \ ,$$

where, of course, $\mathbb{N}$ is the set of natural numbers, and $\mathbb{B}$ is the set $\{\texttt{ff}, \texttt{tt}\}$. Immediately, we see the interpretations of types defined above preserve injectivity of morphisms, since we know that constant presheaves do.

Recall TR in the metalogic is an abbreviation for $\mathsf{Val} \to (\mathsf{Val} \to \mathsf{Fn} \to \mathsf{Val})^2 \to (\mathsf{Val} \to \mathsf{Prop})^2 \to \mathsf{Prop}$. Thus,

$$\begin{aligned}[\![\mathsf{TR}]\!] &= [\![\mathsf{Val}]\!] \Rightarrow ([\![\mathsf{Val}]\!] \Rightarrow [\![\mathsf{Fn}]\!] \Rightarrow [\![\mathsf{Val}]\!])^2 \Rightarrow ([\![\mathsf{Val}]\!] \Rightarrow \mathit{Prop})^2 \Rightarrow \mathit{Prop} \\ &\cong [\![\mathsf{Val}]\!] \times (([\![\mathsf{Val}]\!] \times [\![\mathsf{Fn}]\!]) \Rightarrow [\![\mathsf{Val}]\!])^2 \times ([\![\mathsf{Val}]\!] \Rightarrow \mathit{Prop})^2 \Rightarrow \mathit{Prop}\end{aligned}$$

and since $[\![\mathsf{Val}]\!]$ and $[\![\mathsf{Val}]\!] \times [\![\mathsf{Fn}]\!]$ are constant,

$$\cong \mathsf{val} \times (\mathsf{val} \times \mathsf{fname} \to \mathsf{val})^2 \times \mathrm{Pow}(\mathsf{val})^2 \to \{\mathtt{ff}, \mathtt{tt}\}$$

a function space in **Set**.

3.5.4.5 Transition relations

Since we know that each syntactically constructed type is interpreted as an object of $\mathcal{D}$ which preserves injectivity of morphisms, we may use the equivalences in Section 3.5.4.2 and 3.5.4.3, and, provided our axioms are consistent: for T, T', U such that $\Gamma \rhd T : \mathsf{TR}$, $\Gamma \rhd T' : \mathsf{TR}$, $\Gamma \rhd U : \mathsf{Val} \to \mathsf{TR}$,

1. $\Gamma \rhd T \subseteq T'$ iff for all worlds XSE and $\eta \in [\![\Gamma]\!]_{XSE}$, we have
$$[\![T]\!]_{XSE}(\eta) \subseteq [\![T']\!]_{XSE}(\eta) \ ;$$
2. for all XSE, $\eta \in [\![\Gamma]\!]_{XSE}$, we have $[\![\mathit{Res}(e)]\!]_{XSE}(\eta) = R([\![e]\!]_{XSE}(\eta))$;
3. for all XSE, $\eta \in [\![\Gamma]\!]_{XSE}$, we have
$$[\![T_{\mathsf{fsel}}(x, f)]\!]_{XSE}(\eta) = R^{\mathsf{fname}}([\![x]\!]_{XSE}(\eta), [\![f]\!]_{XSE}(\eta)) \ ;$$
4. for all XSE, $\eta \in [\![\Gamma]\!]_{XSE}$, we have
$$[\![T_{\mathsf{fupd}}(x, f, y)]\!]_{XSE}(\eta) = T^{\mathsf{fupd}}([\![x]\!]_{XSE}(\eta), [\![f]\!]_{XSE}(\eta), [\![y]\!]_{XSE}(\eta)) \ ;$$
5. for all XSE, $\eta \in [\![\Gamma]\!]_{XSE}$, we have $[\![T_{\mathsf{obj}}(\vec{x})]\!]_{XSE}(\eta) = T^{\mathsf{obj}}(\vec{f}, \vec{z})$ where $\vec{f}$ is an enumeration of $\{[\![f]\!]_{XSE}(\eta) \mid f \in \mathsf{dom}(\vec{x})\}$ and $\vec{z}$ is an enumeration of $\{[\![x_f]\!]_{XSE}(\eta) \mid f \in \mathsf{dom}(\vec{x})\}$ which is consistent with $\vec{f}$; and
6. for all XSE, we have $\eta \in [\![\Gamma]\!]_{XSE}$, $[\![T; U]\!]_{XSE}(\eta) = [\![T]\!]_{XSE}(\eta); [\![U]\!]_{XSE}(\eta)$.

3.5.4.6 Interpretation of record constants and axioms

Formally, for the record manipulation constants, we require

$$\begin{aligned}
[\![\mathsf{lookup}_{\sigma,\tau}]\!] &\in \mathcal{D}(\mathbf{1}, [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\tau]\!] \Rightarrow \mathsf{Prop}) \\
[\![\mathsf{update}_{\sigma,\tau}]\!] &\in \mathcal{D}(\mathbf{1}, [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\tau]\!] \Rightarrow [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!]) \\
[\![\mathsf{empty}_{\sigma,\tau}]\!] &\in \mathcal{D}(\mathbf{1}, [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!]) \ .
\end{aligned}$$

It suffices to consider, for S a set, $F \in \hat{\mathbb{I}}$, $G \in \hat{\mathbb{X}}$ and F, G preserve injectivity of morphisms, cases (1) $[\![\sigma]\!] = \Delta\nabla S$ and $[\![\tau]\!] = \Delta F$ and (2) $[\![\sigma]\!] = \Psi\nabla S$ and $[\![\tau]\!] = \Psi G$.

Suppose case (1), that is, suppose $[\![\sigma]\!] = \Delta\nabla S$ and $[\![\tau]\!] = \Delta F$. We note that

$$\mathcal{D}(\mathbf{1}, [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!] \Rightarrow [\![\sigma]\!] \Rightarrow [\![\tau]\!] \Rightarrow \mathsf{Prop}) \cong \mathbf{Pred}([\![\mathsf{Rcd}^{\tau}_{\sigma}]\!] \times [\![\sigma]\!] \times [\![\tau]\!]) \ .$$

So up to isomorphism, it suffices to define $[\![\mathsf{lookup}_{\sigma,\tau}]\!]_{XSE} \stackrel{\text{def}}{=} L_{XSE}$ for (L_{XSE}) a family of sets with

$$\begin{aligned}
L_{XSE} &\subseteq [\![\mathsf{Rcd}^{\tau}_{\sigma}]\!]_{XSE} \times [\![\sigma]\!]_{XSE} \times [\![\tau]\!]_{XSE} \\
&\cong (S \rightharpoonup F_X) \times S \times F_X \ ,
\end{aligned}$$

and satisfying the characteristic property of predicate. So we define

$$L_{XSE} \stackrel{\text{def}}{=} \{(r, i, a) \mid i \in \mathsf{dom}(r) \wedge r(i) = a\} \ .$$

Let $f : XSE \to X'S'E'$ be a morphism. Since F preserves injectivity of morphisms, clearly

$$(F_f \circ r, i, F_f(a)) \in L_{X'S'E'} \quad \text{iff} \quad (r, i, a) \in L_{XSE} \ ,$$

the left-hand side of which is precisely

$$([\![\mathsf{Rcd}^{\tau}_{\sigma}]\!] \times [\![\sigma]\!] \times [\![\tau]\!])_f(r, i, a) \in L_{X'S'E'} \ .$$

That is, L is a predicate.

Up to isomorphism, we require

$$\begin{aligned}
[\![\mathsf{update}_{\sigma,\tau}]\!] &\in \hat{\mathbb{I}}(Q^F_S \times \nabla S \times F, Q^F_S) \\
[\![\mathsf{empty}_{\sigma,\tau}]\!] &\in \hat{\mathbb{I}}(\mathbf{1}, Q^F_S) \ .
\end{aligned}$$

So we define, for $r : S \rightharpoonup F_X$, $i \in S$, $a \in F_X$,

$$\llbracket \mathsf{update}_{\sigma,\tau} \rrbracket_X(r,i,a) \stackrel{\text{def}}{=} r[i \mapsto a]$$
$$\llbracket \mathsf{empty}_{\sigma,\tau} \rrbracket_X \stackrel{\text{def}}{=} \emptyset \;, \quad \text{the everywhere undefined partial function.}$$

It remains to check that these definitions are natural transformations. Since $i = i\restriction_u$, we have $(r,i,a)\restriction_u = (r\restriction_u, i, a\restriction_u)$ and so

$$\begin{aligned}
&(\llbracket \mathsf{update}_{\sigma,\tau} \rrbracket_X(r,i,a))\restriction_u \\
&= (r[i \mapsto a])\restriction_u && \text{by definition} \\
&= (F_u \circ r[i \mapsto a]) && \text{by definition of } (Q^F_S)_u \\
&= (r\restriction_u)[i \mapsto (a\restriction_u)] && \text{by definition of composition.}
\end{aligned}$$

We note that $\emptyset\restriction_u \stackrel{\text{def}}{=} F_u \circ \emptyset = \emptyset$. Thus $\emptyset$ is a global element of Q^F_S.

Suppose case (2), that is, suppose $\llbracket\sigma\rrbracket = \Psi\nabla S$ and $\llbracket\tau\rrbracket = \Psi G$. Using a similar argument as before, it suffices to define $\llbracket \mathsf{lookup}_{\sigma,\tau} \rrbracket_{XSE} \stackrel{\text{def}}{=} L_{XSE}$ for a predicate L. In this case, we define L so that $L_{XSE} \subseteq (S \rightharpoonup G_{XSE}) \times S \times G_{XSE}$. Let $f : XSE \to X'S'E'$ be a morphism. Since G preserves injectivity of morphisms, clearly

$$(G_f \circ r, i, G_f(a)) \in L_{X'S'E'} \quad \text{iff} \quad (r,i,a) \in L_{XSE} \;,$$

the left-hand side of which is precisely

$$(\llbracket \mathsf{Rcd}^\tau_\sigma \rrbracket \times \llbracket\sigma\rrbracket \times \llbracket\tau\rrbracket)_f(r,i,a) \in L_{X'S'E'} \;.$$

That is, L is a predicate.

Up to isomorphism, we require

$$\llbracket \mathsf{update}_{\sigma,\tau} \rrbracket \in \hat{\mathbb{X}}(R^G_S \times \nabla S \times G, R^G_S)$$
$$\llbracket \mathsf{empty}_{\sigma,\tau} \rrbracket \in \hat{\mathbb{X}}(\mathbf{1}, R^G_S) \;.$$

So we define, for $r : S \rightharpoonup G_{XSE}$, $i \in S$, $a \in G_{XSE}$,

$$\llbracket \mathsf{update}_{\sigma,\tau} \rrbracket_{XSE}(r,i,a) \stackrel{\text{def}}{=} r[i \mapsto a]$$
$$\llbracket \mathsf{empty}_{\sigma,\tau} \rrbracket_{XSE} \stackrel{\text{def}}{=} \emptyset \;, \quad \text{the everywhere undefined partial function.}$$

We can show that these definitions give natural transformations as in the previous case.

We check the axioms rcd_parfun, rcd_update and rcd_empty using the forcing notation as follows. Both of the above cases use the same reasoning. Let XSE be a world and suppose we have Γ, η such that $\eta(x) \in [\![\Gamma]\!]_{XSE}(x)$ for all $x \in \mathsf{dom}(\Gamma)$. For convenience, let us also identify some variables with metavariables, namely suppose $\eta(r) = r$, $\eta(i) = i$, $\eta(a) = a$ and $\eta(a') = a'$. Certainly, if $i \in \mathsf{dom}(r)$, $r(i) = a$ and $r(i) = a'$, then since r is a partial function, it must be the case that $a = a'$. Thus $XSE \Vdash_{\Gamma,\eta} \mathsf{lookup}(r, i, a)$ and $XSE \Vdash_{\Gamma,\eta} \mathsf{lookup}(r, i, a')$ imply $XSE \Vdash_{\Gamma,\eta} a = a'$. Thus

$$XSE \Vdash_{\Gamma,\eta} \mathsf{lookup}(r, i, a) \supset \mathsf{lookup}(r, i, a') \supset a = a' \ .$$

Thus axiom rcd_parfun is valid.

For axiom rcd_update, it is certainly the case that if $i = i'$ then $i' \in \mathsf{dom}(r[i \mapsto a])$ and $(r[i \mapsto a])(i') = a$, by the definition of overloading $-[- \mapsto -]$. Thus

$$XSE \Vdash_{\Gamma,\eta} i = i' \supset \mathsf{lookup}(\mathsf{update}(r, i, a), i', a) \ .$$

Also, if $i \neq i'$, then

$$\begin{array}{ll} & i' \in \mathsf{dom}(r[i \mapsto a]) \quad \text{and} \quad (r[i \mapsto a])(i') = a' \\ \text{iff} & i' \in \mathsf{dom}(r) \quad \text{and} \quad r(i') = a' \ . \end{array}$$

Thus

$$XSE \Vdash_{\Gamma,\eta} i \neq i' \supset \mathsf{lookup}(\mathsf{update}(r, i, a), i', a') \equiv \mathsf{lookup}(r, i', a') \ .$$

So we conclude that axiom rcd_update is valid.

Finally, there are no i, a such that $i \in \mathsf{dom}(\emptyset)$. Thus

$$XSE \Vdash_{\Gamma,\eta} \neg \mathsf{lookup}(\mathsf{empty}, i, a) \ .$$

That is, axiom rcd_empty is valid.

3.5.4.7 Program logic global elements

Let us now consider the specification constructors. Formally, we require

$$\begin{aligned}
[\![\mathsf{Bool}]\!] &\in \mathcal{D}(1, [\![\mathsf{Sp}]\!]) \\
[\![\mathsf{Nat}]\!] &\in \mathcal{D}(1, [\![\mathsf{Sp}]\!]) \\
[\![\mathsf{Obj}]\!] &\in \mathcal{D}(1, [\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}} \to \mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}} \to \mathsf{Sp}]\!])
\end{aligned}$$

But since $[\![\mathsf{Sp}]\!] \stackrel{\text{def}}{=} \Psi\nabla\mathsf{specs}$, and using the fact that Ψ is right adjoint to $-^{(01)}$, we have $\mathcal{D}(1, [\![\mathsf{Sp}]\!]) \cong \hat{\mathbb{X}}(1, \nabla\mathsf{specs}) \cong \mathsf{specs}$. Hence we can define $[\![\mathsf{Bool}]\!] \stackrel{\text{def}}{=} \mathit{Bool}$ up to isomorphism. Similarly, we define $[\![\mathsf{Nat}]\!] \stackrel{\text{def}}{=} \mathit{Nat}$. Using the characteristic property of function space and the fact that Ψ is right adjoint to $-^{(01)}$, we have

$$\begin{aligned}
&\mathcal{D}(1, [\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}} \to \mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}} \to \mathsf{Sp}]\!]) \\
&\cong \mathcal{D}([\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}]\!] \times [\![\mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}}]\!], [\![\mathsf{Sp}]\!]) && \text{char. ppty of f. space} \\
&\cong \hat{\mathbb{X}}([\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}]\!]^{(01)} \times [\![\mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}}]\!]^{(01)}, \nabla\mathsf{specs}) && \text{by Theorem 1 part 9.}
\end{aligned}$$

Thus, up to isomorphism, $[\![\mathsf{Obj}]\!]$ is a natural transformation in $\hat{\mathbb{X}}$ and

$$[\![\mathsf{Obj}]\!]_{XSE} : [\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}]\!]^{(01)}_{XSE} \times [\![\mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}}]\!]^{(01)}_{XSE} \to \mathsf{specs}$$

It is straightforward to check that

$$\begin{aligned}
[\![\mathsf{Rcd}^{\mathsf{Sp}}_{\mathsf{Fn}}]\!]^{(01)}_{XSE} &\cong (\mathsf{fname} \rightharpoonup \mathsf{specs}) \\
[\![\mathsf{Rcd}^{\mathsf{Val}\to\mathsf{Sp}\times\mathsf{TR}}_{\mathsf{Mn}}]\!]^{(01)}_{XSE} &\cong (\mathsf{mname} \rightharpoonup \mathsf{val} \to \mathsf{specs} \times \mathsf{trans})
\end{aligned}$$

That is, $[\![\mathsf{Obj}]\!]_{XSE} : (\mathsf{fname} \rightharpoonup \mathsf{specs}) \times (\mathsf{mname} \rightharpoonup \mathsf{val} \to \mathsf{specs} \times \mathsf{trans}) \to \mathsf{specs}$, up to isomorphism. For $A : \mathsf{fname} \rightharpoonup \mathsf{specs}$ and $B : \mathsf{mname} \rightharpoonup (\mathsf{val} \to \mathsf{specs}) \times (\mathsf{val} \to \mathsf{trans})$, we can thus define

$$\begin{aligned}
[\![\mathsf{Bool}_{\mathsf{Sp}}]\!]_{XSE} &\stackrel{\text{def}}{=} \mathit{Bool} \\
[\![\mathsf{Nat}_{\mathsf{Sp}}]\!]_{XSE} &\stackrel{\text{def}}{=} \mathit{Nat} \\
[\![\mathsf{Obj}_{\mathsf{Sp}}]\!]_{XSE}(A, B) &\stackrel{\text{def}}{=} [f{=}A(f)^{f\in\mathsf{dom}\,A}, m{=}B(m)^{m\in\mathsf{dom}\,B}]
\end{aligned}$$

We give interpretations of program term constructors by first defining natural transformations in $\hat{\mathbb{I}}$. Provided we are happy these definitions really give natural

transformations, we know that their images under Δ are morphisms in $\mathcal{D}$. And since $\mathcal{D}$ is cartesian closed, this is tantamount to finding the required global elements of $\mathcal{D}$.

For $x, y \in X$, $z \notin X$, $f \in \mathsf{fname}$, $m \in \mathsf{mname}$, $a, a_0, a_1 \in P_X$, $b \in (V{\Rightarrow}P)_X$, $\vec{x} : \mathsf{fname} \rightharpoonup V_X$ and $\vec{b} : \mathsf{mname} \rightharpoonup (V{\Rightarrow}P)_X$, we define natural transformations f^{false}, f^{true}, f^{let}, f^{obj}, f^{if}, f^{var}, f^{fsel}, f^{minv}, f^{fupd} of $\hat{\mathbb{I}}$ by

$$\begin{aligned}
{f^{\mathsf{false}}}_X &\stackrel{\text{def}}{=} \mathtt{ff} \\
{f^{\mathsf{true}}}_X &\stackrel{\text{def}}{=} \mathtt{tt} \\
{f^{\mathsf{let}}}_X(a,b) &\stackrel{\text{def}}{=} \mathit{let}\ z{=}a\ \mathit{in}\ b_{X\cup\{z\}}(z) \\
{f^{\mathsf{obj}}}_X(\vec{x},\vec{b}) &\stackrel{\text{def}}{=} [f{=}\vec{x}(f)^{f\in\mathsf{dom}\,\vec{x}}, m{=}\varsigma(z)\vec{b}(m)_{X\cup\{z\}}(z)^{m\in\mathsf{dom}(\vec{b})}] \\
{f^{\mathsf{if}}}_X(x,a_0,a_1) &\stackrel{\text{def}}{=} \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 \\
{f^{\mathsf{var}}}_X(x) &\stackrel{\text{def}}{=} x \\
{f^{\mathsf{fsel}}}_X(x,f) &\stackrel{\text{def}}{=} x.f \\
{f^{\mathsf{minv}}}_X(x,m) &\stackrel{\text{def}}{=} x.m() \\
{f^{\mathsf{fupd}}}_X(x,f,y) &\stackrel{\text{def}}{=} x.f{:=}y
\end{aligned}$$

Except, perhaps, for ${f^{\mathsf{let}}}_X$ and ${f^{\mathsf{obj}}}_X$, these definitions clearly give natural transformations.

Let us look more closely at the definition of f^{let} and see why it is a natural transformation. We need to show that f^{let} is well defined since $z \notin X$ is chosen arbitrarily. Suppose $a \in P_X$, $b \in (V{\Rightarrow}P)_X$ and $z \notin X$. Recall $(V{\Rightarrow}P)_X$ is a set of families $(f_{u,Y})_{u:X\to Y}$ of compatible morphisms. Hence b is one such family. Thus $b_{X\cup\{z\}}$ is a function from $V_{X\cup\{z\}} = X \cup \{z\}$ into $P_{X\cup\{z\}}$. In particular, we can form $b_{X\cup\{z\}}(z)$ giving an element of $P_{X\cup\{z\}}$. That is, $b_{X\cup\{z\}}(z)$ is a program with free variables in $X \cup \{z\}$. And so ${f^{\mathsf{let}}}_X(a,b) \stackrel{\text{def}}{=} \mathit{let}\ z{=}a\ \mathit{in}\ b_{X\cup\{z\}}(z)$ is in P_X.

Now we show that the definition of $f^{\mathsf{let}}(a,b)$ is independent of the choice of z. Consider, for some $z' \notin X$, the substitution $b_{X\cup\{z\}}(z)[z'/z]$. Since $b_{X\cup\{z\}}(z) \in P_{X\cup\{z\}}$, the substitution is precisely the image of $b_{X\cup\{z\}}(z)$ under the action of P_ρ, where $\rho : X\cup\{z\} \to X\cup\{z'\}$ is the substitution map. Since b is a compatible

family of transformations,

$$\begin{array}{ccc}
V_{X\cup\{z\}} & \xrightarrow{b_{X\cup\{z\}}} & P_{X\cup\{z\}} \\
{\scriptstyle V_\rho}\downarrow & & \downarrow{\scriptstyle P_\rho} \\
V_{X\cup\{z'\}} & \xrightarrow[b_{X\cup\{z'\}}]{} & P_{X\cup\{z'\}}
\end{array}$$

commutes. Thus

$$b_{X\cup\{z\}}(z)[z'/z] = P_\rho(b_{X\cup\{z\}}(z)) = b_{X\cup\{z'\}}(V_\rho(z)) = b_{X\cup\{z'\}}(z') \ . \tag{3.6}$$

Hence *let* $z'{=}a$ *in* $b_{X\cup\{z'\}}(z') \equiv$ *let* $z'{=}a$ *in* $b_{X\cup\{z\}}(z)[z'/z]$. Finally, by α-conversion, we conclude that *let* $z{=}a$ *in* $b_{X\cup\{z\}}(z) =$ *let* $z'{=}a$ *in* $b_{X\cup\{z'\}}(z')$, precisely the fact that the definition of f^{let} is independent of the choice of z.

It remains to show that f^{let} is a natural transformation. Namely,

$$\begin{array}{ccc}
P_X \times (V{\Rightarrow}P)_X & \xrightarrow{f^{\mathsf{let}}{}_X} & P_X \\
{\scriptstyle P_\rho \times (V\Rightarrow P)_\rho}\downarrow & & \downarrow{\scriptstyle P_\rho} \\
P_Y \times (V{\Rightarrow}P)_Y & \xrightarrow[f^{\mathsf{let}}{}_Y]{} & P_Y
\end{array}$$

commutes for $\rho : X \to Y$. Let $a \in P_X$ and $b \in (V{\Rightarrow}P)_X$. Chasing the diagram first across then down, we have (*let* $z{=}a$ *in* $b_{X\cup\{z\}}(z))[\rho]$. Chasing the diagram first down then across, we have *let* $z'{=}a[\rho]$ *in* $b[\rho]_{Y\cup\{z'\}}(z')$. We show that these two expressions are equal using naturality and the following commutativity property:

$$\begin{array}{ccc}
X & \hookrightarrow & X \cup \{z\} \\
{\scriptstyle \rho}\downarrow & & \downarrow{\scriptstyle \rho^+} \\
Y & \hookrightarrow & Y \cup \{z'\}
\end{array}$$

where ρ^+ is an extension of ρ that maps z to z'.

We can use a similar argument to show that f^{obj} is a natural transformation.

Similarly, to find global elements $\mathcal{D}(\mathbf{1}, F{\Rightarrow}[\![\mathsf{Val}]\!])$, we find natural transformations $\hat{\mathbb{X}}(F^{(01)}, \nabla\mathsf{val})$. So, for $v \in \mathbb{B}$, $n \in \mathbb{N}$ and world XSE, we define functions $e^{\mathbb{B}}_{XSE} : \mathbb{B} \to \mathsf{val}$ and $e^{\mathbb{N}}_{XSE} : \mathbb{N} \to \mathsf{val}$ in $\hat{\mathbb{X}}$ by

$$e^{\mathbb{N}}_{XSE}(v) \stackrel{\text{def}}{=} v$$
$$e^{\mathbb{B}}_{XSE}(n) \stackrel{\text{def}}{=} n$$

and so $e^{\mathbb{B}} : \nabla\mathbb{B} \to \nabla\mathsf{val}$ and $e^{\mathbb{N}} : \nabla\mathbb{N} \to \nabla\mathsf{val}$ are natural transformations (morphisms of $\hat{\mathbb{X}}$). Thus, we can define, up to isomorphism,

$$[\![\mathsf{bool}_{\mathsf{Val}}]\!] \stackrel{\text{def}}{=} \Psi e^{\mathbb{B}}$$
$$[\![\mathsf{nat}_{\mathsf{Val}}]\!] \stackrel{\text{def}}{=} \Psi e^{\mathbb{N}}$$

and since $[\![\mathsf{Var}]\!]^{(01)}{}_{XSE} = X$ and $[\![\mathsf{Val}]\!] \stackrel{\text{def}}{=} \nabla\mathsf{val}$, it suffices to define $[\![\mathsf{var}_{\mathsf{Val}}]\!]$ as a family of functions $X \to \mathsf{val}$:

$$[\![\mathsf{var}_{\mathsf{Val}}]\!]_{XSE} \stackrel{\text{def}}{=} S\ .$$

This definition gives a natural transformation by the definition of morphism in $\mathbb{X}$. (I.e. A morphism $u : XSE \to X'S'E'$ is an injective function $u : X \to X'$ such that for all $x \in X$, both $S'(u(x)) = S(x)$ and $E'(u(x)) = E(x)$.)

3.5.4.8 Program logic predicates

We define auxiliary relations corresponding to subsumption ($<:$) and derivability ($[- : - :: -]$). These auxiliary definitions will not only help us to interpret the corresponding relations of the embedding, but also allow us to prove the main result in two manageable steps (finding a model and then proving soundness in the model.) Tables 3.3 and 3.4 give these auxiliary definitions of $- \Vdash - : - :: -$ and $- <:_{\text{aux}} -$.

The definitions use the functions: $R : \mathsf{val} \to \mathsf{trans}$ defined by

$$R(v') \stackrel{\text{def}}{=} \{(r, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc}) \mid Res(v')\}\ ;$$

function $R^{\mathsf{fname}} : \mathsf{val} \times \mathsf{fname} \rightarrow \mathsf{trans}$ defined pointwise by

$$R^{\mathsf{fname}}(h, f) \stackrel{\text{def}}{=} \{(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \mid T_{\mathsf{fsel}}(h, f)\} \ ;$$

function $T^{\mathsf{obj}} : \mathsf{fname}^k \times \mathsf{val}^k \rightarrow \mathsf{trans}$ defined by

$$T^{\mathsf{obj}}(\vec{f}, \vec{x}) \stackrel{\text{def}}{=} \{(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \mid T_{\mathsf{obj}}(\vec{y})\} \ ,$$

where the domain of $\vec{y}$ is the set of field names in $\vec{f}$ and for $f_i \in \vec{f}$, we define $y_{f_i} \stackrel{\text{def}}{=} x_{f_i}$; function $T^{\mathsf{fupd}} : \mathsf{val} \times \mathsf{fname} \times \mathsf{val} \rightarrow \mathsf{trans}$ defined by

$$T^{\mathsf{fupd}}(x, f, y) \stackrel{\text{def}}{=} \{(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \mid T_{\mathsf{fsel}}(x, f, y)\} \ .$$

We overload R and write $R(h.f)$ for $R^{\mathsf{fname}}(h, f)$.

Also, given $T \in \mathsf{trans}$ and $U : \mathsf{val} \rightarrow \mathsf{trans}$, we define their composition $T; U \in \mathsf{trans}$ by

$$\begin{aligned}(T; U) \stackrel{\text{def}}{=} \{(r, \grave{\sigma}, \acute{\sigma}, al\grave{l}oc, al\acute{l}oc) \mid \exists \breve{\sigma}, al\check{l}oc, x.\ & (x, \grave{\sigma}, \breve{\sigma}, al\grave{l}oc, al\check{l}oc) \in T \wedge \\ & (r, \breve{\sigma}, \acute{\sigma}, al\check{l}oc, al\acute{l}oc) \in U(x) \quad \} \ .\end{aligned}$$

This is the semantic counterpart of the syntactic composition operator.

We can check that these relations are predicates, i.e.

$$U_{XSE} \stackrel{\text{def}}{=} \{(A, A') \mid A <:_{\text{aux}} A'\}$$

gives $U \in \mathbf{Pred}([\![\mathsf{Sp}]\!]^2)$, and

$$V_{XSE} \stackrel{\text{def}}{=} \{(a, A, T) \mid XSE \Vdash a : A :: T\}$$

gives $V \in \mathbf{Pred}([\![\mathsf{Prg}]\!] \times [\![\mathsf{Sp}]\!] \times [\![\mathsf{TR}]\!])$. For $- <:_{\text{aux}} -$, this is trivial since its definition is independent of XSE and $[\![\mathsf{Sp}]\!]^2$ is constant. The case for $- \Vdash - : - :: -$ is a bit more involved since $[\![\mathsf{Prg}]\!]$ is not a constant presheaf. We show that it is a predicate by induction over program syntax: given $u : XSE \rightarrow X'S'E'$, to show $XSE \Vdash a : A :: T$ iff $X'S'E' \Vdash a[u] : A :: T$ we may assume $XSE \Vdash a' : A' :: T'$ iff $X'S'E' \Vdash a'[u] : A' :: T'$ for all (strict) subterms a' of a. (Recall that $a[u]$ denotes a whose free variables are renamed by u, i.e. the definition $[\![\mathsf{Prg}]\!]_u(a)$.) The following lemma is useful.

Lemma 3 *For* $A \in \mathsf{specs}$, $B \in \mathsf{val} \to \mathsf{specs}$, $U \in \mathsf{val} \to \mathsf{trans}$, $a \in P_X$, *given* $u : XSE \to X'S'E'$: *(1)*

$$\begin{aligned}&\forall v : XSE \to X''S''E''.\forall x \in X''.\\&\quad E''(x) = A \implies X''S''E'' \Vdash a[v] : B(S''(x)) :: U(S''(x))\end{aligned}$$

implies

$$\begin{aligned}&\forall w : X'S'E' \to X''S''E''.\forall x \in X''.\\&\quad E''(x) = A \implies X''S''E'' \Vdash a[u][w] : B(S''(x)) :: U(S''(x)) \ ;\end{aligned}$$

and (2)

$$\begin{aligned}&\forall w : X'S'E' \to X''S''E''.\forall x \in X''.\\&\quad E''(x) = A \implies X''S''E'' \Vdash a[u][w] : B(S''(x)) :: U(S''(x))\end{aligned}$$

implies

$$\begin{aligned}&\forall v : XSE \to X''S''E''.\forall x \in X''.\exists g : X''S''E'' \to X'''S'''E'''.\\&\quad E''(x) = A \implies X'''S'''E''' \Vdash a[v][g] : B(S''(x)) :: U(S''(x)) \ .\end{aligned}$$

Proof Suppose

$$\begin{aligned}&\forall v : XSE \to X''S''E''.\forall x \in X''.\\&\quad E''(x) = A \implies X''S''E'' \Vdash a[v] : B(S''(x)) :: U(S''(x))\end{aligned}$$

Let $w : X'S'E' \to X''S''E''$, $x \in X''$ and $E''(x) = A$. Since (i) $wu : XSE \to X''S''E''$, (ii) $x \in X''$, and (iii) $E''(x) = A$, we know $X''S''E'' \Vdash a[wu] : B(S''(x)) :: U(S''(x))$. But $a[wu] = a[u][w]$.

Conversely, suppose

$$\begin{aligned}&\forall w : X'S'E' \to X''S''E''.\forall x \in X''.\\&\quad E''(x) = A \implies X''S''E'' \Vdash a[u][w] : B(S''(x)) :: U(S''(x)) \ .\end{aligned}$$

Let $v : XSE \to X''S''E''$, $x \in X''$ and $E''(x) = A$. By Lemma 1, we know that there is world $X'''S'''E'''$, $f : X'S'E' \to X'''S'''E'''$ and $g : X''S''E'' \to X'''S'''E'''$ such that $fu = gv$. Since (i) $f : X'S'E' \to X'''S'''E'''$, (ii) $g(x) \in X'''$, and (iii) $E'''(g(x)) = E''(x) = A$, we know $X'''S'''E''' \Vdash a[u][f] : B(S'''(g(x))) ::$

$U(S'''(g(x)))$. But $a[u][f] = a[fu] = a[gv] = a[v][g]$ and $S'''(g(x)) = S''(x)$. Thus, in fact, $X'''S'''E''' \Vdash a[v][g] : B(S''(x)) :: U(S''(x))$. □

We can prove that $- \Vdash - : - :: -$ is a predicate by induction over AL syntax. To demonstrate how the previous lemma is useful, we give the case $c \equiv \textit{let } x{=}a \textit{ in } b$.

Suppose $XSE \Vdash c : A'' :: T''$. So we may assume that there are A, A', T, U such that

$$A' <:_{\text{aux}} A''$$

$$T; U \subseteq T''$$

$$XSE \Vdash a : A :: T$$

$$\forall v : XSE \to X''S''E''. \forall x \in X''.$$
$$E''(x) = A \implies X''S''E'' \Vdash b[v] : A' :: U(S''(x)) \ .$$

By part (1) of our lemma, we can conclude

$$\forall w : X'S'E' \to X''S''E''. \forall x \in X''.$$
$$E''(x) = A \implies X''S''E'' \Vdash b[u][w] : A' :: U(S''(x)) \ .$$

By our induction hypothesis, we can conclude

$$X'S'E' \Vdash a[u] : A :: T \ .$$

So by the definition of $- \Vdash - : - :: -$, we know $XSE \Vdash c[u] : A'' :: T''$.

Conversely, suppose $X'S'E' \Vdash c[u] : A'' :: T''$. So we may assume that there are A, A', T, U such that

$$A' <:_{\text{aux}} A''$$

$$T; U \subseteq T''$$

$$X'S'E' \Vdash a[u] : A :: T$$

$$\forall w : X'S'E' \to X''S''E''. \forall x \in X''.$$
$$E''(x) = A \implies X''S''E'' \Vdash b[u][w] : A' :: U(S''(x)) \ .$$

Let $v : XSE \to X''S''E''$, $x \in X''$ and $E''(x) = A$. By part (2) of our lemma, we know there is $g : X''S''E'' \to X'''S'''E'''$ such that $X'''S'''E''' \Vdash a[v][g]$:

$$\frac{}{Bool <:_{\text{aux}} Bool}$$

$$\frac{}{Nat <:_{\text{aux}} Nat}$$

$$\frac{\mathsf{dom}\,\vec{A}' \subseteq \mathsf{dom}\,\vec{A} \quad A_f = A'_f \ {}^{f \in \mathsf{dom}(A')} \quad \mathsf{dom}\,\vec{B}' \subseteq \mathsf{dom}\,\vec{B} \quad B_m \leq B'_m \ {}^{m \in \mathsf{dom}(B')}}{[\vec{A}, \vec{B}] <:_{\text{aux}} [\vec{A}', \vec{B}']}$$

$$\text{where } \langle B, U\rangle \leq \langle B', U'\rangle \stackrel{\text{def}}{=} \forall h \in \mathsf{locn}.B(h) <:_{\text{aux}} B'(h) \wedge U(h) \subseteq U'(h)$$

Table 3.3: Auxiliary subsumption relation. For $A, A' \in$ specs, we define $A <:_{\text{aux}} A'$ to be the smallest relation closed under these rules. Note: we write A_f for the unique x such that lookup$(\vec{A}, f, x)$, and similarly for A'_f, B_m, B'_m.

$B(S''(x)) :: U(S''(x))$. So by our induction hypothesis, we know $X''S''E'' \Vdash a[v] : B(S''(x)) :: U(S''(x))$. That is, we know

$$\begin{aligned} &\forall v : XSE \to X''S''E''.\forall x \in X''. \\ &\quad E''(x) = A \implies X''S''E'' \Vdash a[v] : B(S''(x)) :: U(S''(x)) \ . \end{aligned}$$

By our induction hypothesis, we also know

$$XSE \Vdash a : A :: T \ .$$

And so by the definition of $- \Vdash - : - :: -$, we know

$$XSE \Vdash c : A'' :: T'' \ .$$

The case for $c \equiv [f_i{=}x_i, m_j{=}\varsigma(y_j)b_j]$ is similar. The remaining cases are straightforward to show using the induction hypothesis.

Since the auxiliary relations defined in Tables 3.4 and 3.3 are predicates, we can interpret the assumption relation $[- : -]$ and the derivability relation $[- : - :: -]$ as follows

$$\begin{aligned} XSE \Vdash_{\Gamma,\eta} [a : A :: T] \quad &\text{iff} \quad XSE \Vdash [\![a]\!]_{XSE}(\eta) : [\![A]\!]_{XSE}(\eta) :: [\![T]\!]_{XSE}(\eta) \\ XSE \Vdash_{\Gamma,\eta} A <: A' \quad &\text{iff} \quad [\![A]\!]_{XSE}(\eta) <:_{\text{aux}} [\![A']\!]_{XSE}(\eta) \end{aligned}$$

$$\frac{XSE \Vdash a : A :: T \quad A <:_{\text{aux}} A' \quad T \subseteq T'}{XSE \Vdash a : A' :: T'}$$

$$\frac{}{XSE \Vdash x : E(x) :: R(S(x))}$$

$$\frac{}{XSE \Vdash \mathsf{false} : \mathsf{bool} :: R(\mathtt{ff})} \qquad \frac{}{XSE \Vdash \mathsf{true} : \mathsf{bool} :: R(\mathtt{tt})}$$

$$\frac{\begin{array}{l} XSE \Vdash x : \mathsf{bool} :: R(S(x)) \\ XSE \Vdash a_0 : B_0(S(x)) :: U_0(S(x)) \quad B_0(\mathtt{tt}) = B(\mathtt{tt}) \quad U_0(\mathtt{tt}) = U(\mathtt{tt}) \\ XSE \Vdash a_1 : B_1(S(x)) :: U_1(S(x)) \quad B_1(\mathtt{tt}) = B(\mathtt{tt}) \quad U_1(\mathtt{tt}) = U(\mathtt{tt}) \end{array}}{XSE \Vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : B(S(x)) :: U(S(x))}$$

$$\frac{\begin{array}{l} XSE \Vdash a : A :: T \\ \forall u : XSE \to X'S'E'.\forall x \in X'. \\ \quad E'(x) = A \implies X'S'E' \Vdash b[u] : A' :: U(S'(x)) \end{array}}{XSE \Vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : A' :: T;U}$$

$$\frac{\begin{array}{l} A \stackrel{\text{def}}{=} [f_i{=}A_i{}^{i=1..k}, m_j{=}\langle B_j, U_j\rangle^{j=1..\ell}] \quad S(\vec{x}) \stackrel{\text{def}}{=} S(x_1), \ldots, S(x_\ell) \\ XSE \Vdash x_i : A_i :: R(S(x_i)) \\ \forall u : XSE \to X_jS_jE_j.\forall y_j \in X'. \\ \quad E_j(y_j) = A \implies X_jS_jE_j \Vdash b_j[u] : B_j(S_j(y_j)) :: U_j(S_j(y_j)) \end{array}}{XSE \Vdash [f_i{=}x_i, m_j{=}\varsigma(y_j)b_j(y_j)] : A :: T^{\mathsf{obj}}(\vec{f}, S(\vec{x}))}$$

$$\frac{XSE \Vdash x : [f{=}A] :: R(S(x))}{XSE \Vdash x.f : A :: R(S(x).f)}$$

$$\frac{XSE \Vdash x : [m{=}\langle B, U\rangle] :: R(S(x))}{XSE \Vdash x.m() : B(S(x)) :: U(S(x))}$$

$$\frac{XSE \Vdash x : A :: R(S(x)) \quad XSE \Vdash y : A_k :: R(S(y))}{XSE \Vdash x.f_k{:=}y : A :: T^{\mathsf{fupd}}(S(x), f_k, S(y))}$$

Table 3.4: Auxiliary derivability relation. For world XSE, program $a \in P_X$, $A \in$ specs, and $T \in$ trans, we define $XSE \Vdash a : A :: T$ to be the smallest relation closed under these rules.

and since $U_{XSE} \stackrel{\text{def}}{=} \{(x, A) \in X \times \mathsf{specs} \mid E(x) = A\}$ gives a predicate U by the definition of morphism in $\mathbb{X}$, we can define

$$XSE \Vdash_{\Gamma,\eta} [x : A] \quad \text{iff} \quad E([\![x]\!]_{XSE}(\eta)) = [\![A]\!]_{XSE}(\eta) \ .$$

3.5.4.9 Interpretation $[\![-]\!]$ models the axioms for AL

Our interpretation models the axioms introduced in Section 3.2.1, as stated in the next lemma.

Lemma 4 *Record axioms rcd_parfun, rcd_empty and rcd_update; subspecification axioms ss_obj, ss_bool and ss_nat; and program logic axioms ws_subs, ws_constf, ws_constt, ws_nat, ws_natop, ws_cond, ws_let, ws_minv, ws_fsel, ws_obj, ws_fupd introduced in Section 3.2.1 are true in our interpretation.*

The proof of this lemma requires computing the interpretation of each axiom and checking that it does take value tt everywhere. We use forcing judgements to make this task more manageable. We use the definition in Table 3.4 and find that all the axioms are true. (This is no coincidence!) However, the following lemma proves useful.

Lemma 5 *We have*

$$XSE \Vdash_{\Gamma,\eta} \forall_{\mathsf{Var}} x.[x : A'] \supset \phi$$

iff, for all $u : XSE \to X'S'E'$ *and* $x \in X'$,

$$E'(x) = [\![A']\!]_{X'S'E'}(\eta') \implies X'S'E' \Vdash_{\Gamma',\eta'} \phi$$

where

$$\Gamma' \stackrel{\text{def}}{=} \Gamma[x \mapsto \mathsf{Var}]$$
$$\eta' \stackrel{\text{def}}{=} (\eta\restriction_u)[x \mapsto x] \ .$$

Note that we are abusing notation here since x is both a formal variable of the metalanguage and also an element of $[\![\mathsf{Var}]\!]_{X'S'E'} = X'$.

Proof (Lemma 5) We observe

$$XSE \Vdash_{\Gamma,\eta} \forall_{\mathsf{Var}} x.[x : A'] \supset \phi$$

iff for worlds $X'S'E'$, morphism $u : XSE \to X'S'E'$ and $x \in [\![\mathsf{Var}]\!]_{X'S'E'}$

$$X'S'E' \Vdash_{\Gamma',\eta'} [x : A'] \quad \text{implies} \quad X'S'E' \Vdash_{\Gamma',\eta'} \phi$$

where Γ', η' are defined as in the statement of the lemma.

But, by definition

$$X'S'E' \Vdash_{\Gamma',\eta'} [x : A'] \quad \text{iff} \quad E'(y) = [\![A']\!]_{X'S'E'}(\eta')$$

and

$$[\![\mathsf{Var}]\!]_{X'S'E'} = X' \ .$$

□

Proof (Lemma 4) The proof proceeds by giving meaning to the axioms using forcing judgements. The cases are straightforward and we give the case for ws_let as an example. Suppose

$$\begin{aligned} \emptyset \Vdash_{\Gamma,\eta} \ \forall a, b, A', T'', A, T, U.& \\ &[a : A :: T] \supset \\ &(\forall x.[x : A] \supset [b(x) : A' :: U(x)]) \supset \\ &(T; U \subseteq T'') \supset \\ &\quad [\mathsf{let}(a, b) : B :: T''] \ . \end{aligned}$$

Thus, using the equivalences for $\supset$ and $\forall$ introduced in Section 3.5.4.2,

$$\begin{aligned} &\forall a, b, A', T'', A, T, U. \\ &\quad XSE \Vdash a : A :: T \\ &\quad \text{and} \\ &\quad XSE \Vdash_{\Gamma,\eta} \forall x.[x : A] \supset [b(x) : A' :: U(x)] \\ &\quad \text{and} \\ &\quad XSE \Vdash_{\Gamma,\eta} T; U \subseteq T'' \\ &\quad \text{implies} \\ &\quad\quad XSE \Vdash \mathit{let}\ x{=}a\ \mathit{in}\ b(x) : B :: T'' \end{aligned}$$

where $\eta = (a \mapsto a, b \mapsto b, A' \mapsto A', T'' \mapsto T'', A \mapsto A, T \mapsto T, U \mapsto U)$. So by Lemma 5 and the equivalences in Section 3.5.4.5, we have

$$\begin{array}{l} \forall a, b, A', T'', A, T, U. \\ \quad XSE \Vdash a : A :: T \\ \quad \text{and} \\ \quad \forall u : XSE \to X'S'E'. \forall x \in X'. E'(x) = A \implies X'S'E' \Vdash b(x) : A' :: U(x) \\ \quad \text{and} \\ \quad T; U \subseteq T'' \\ \quad \text{implies} \\ \quad\quad XSE \Vdash \textit{let } x{=}a \textit{ in } b(x) : B :: T'' \ . \end{array}$$

This is true by the definition of $- \Vdash - : - :: -$ in Table 3.4. □

3.5.5 Justification

Soundness in our model is expressed as the following theorem which is reminiscent of Theorem 1 of [AL98].

Theorem 3 *Assume that the operational semantics says that program b yields result v when run with an empty stack and an empty initial store. If $\Vdash [\![\ulcorner b \urcorner]\!]_\emptyset : [\![\mathsf{bool}]\!]_\emptyset :: [\![\mathsf{Res}(\mathtt{tt})]\!]_\emptyset$ then v is boolean $\mathtt{tt}$. Similarly, if $\Vdash [\![\ulcorner b \urcorner]\!]_\emptyset : [\![\mathsf{bool}]\!]_\emptyset :: [\![\mathsf{Res}(\mathtt{ff})]\!]_\emptyset$ then v is boolean $\mathtt{ff}$.*

We now take a step back and consider the importance of this theorem in relation to the proof of soundness of our embedding. Since we know that our interpretation models higher-order logic and the axioms of our embedding, we know that if we can derive a judgement of the form

$$\rhd [a : A :: T]$$

it follows that, for all XSE,

$$XSE \Vdash [\![a]\!]_{XSE} : [\![A]\!]_{XSE} :: [\![T]\!]_{XSE} \ .$$

So now Theorem 3 gives us a soundness result in the style of the soundness theorem in [AL98].

To prove Theorem 3, we follow [AL98], and prove a more general theorem (namely Theorem 4) which we can prove directly. We follow loc. cit. and use the technique of store specifications and introduce similar auxiliary definitions relating to store specifications.

Within the logic, stores are partial mappings from location and field name pairs into the set of results; closures of methods are omitted. We call these stores *flat stores* to distinguish them from stores that are used in the operational semantics. Recall, val is the set of values (which includes locn, the set of locations) and fname is the set of field names. We define

$$\mathsf{flstore} \stackrel{\text{def}}{=} \mathsf{val} \rightharpoonup (\mathsf{fname} \rightharpoonup \mathsf{val}) \ .$$

We let π be the projection mapping a store $\sigma \in \mathsf{store} \stackrel{\text{def}}{=} \mathsf{val} \rightharpoonup ((\mathsf{fname} \rightharpoonup \mathsf{val}) \times (\mathsf{mname} \rightharpoonup \mathsf{val}))$ to the flatstore $\sigma' \in \mathsf{flstore}$ which is σ with all information about methods omitted. We define an embedding $\mathsf{fl} : (\mathsf{val} \times \mathsf{flstore} \times \mathsf{flstore}) \rightarrow (\mathsf{val} \times \mathsf{store}^2 \times \mathrm{Pow}(\mathsf{val})^2)$ as follows.

$$\mathsf{fl}(v, \sigma, \sigma') \stackrel{\text{def}}{=} (v, \grave{\sigma}, \acute{\sigma}, \grave{alloc}, \acute{alloc})$$

where

$$\begin{aligned}
\grave{alloc} &\stackrel{\text{def}}{=} \mathsf{dom}(\sigma) \\
\acute{alloc} &\stackrel{\text{def}}{=} \mathsf{dom}(\sigma') \\
\grave{\sigma}(h, f) &\stackrel{\text{def}}{=} \begin{cases} \sigma(h)(f) & \text{if defined} \\ \bot & \text{otherwise} \end{cases} \\
\acute{\sigma}(h, f) &\stackrel{\text{def}}{=} \begin{cases} \sigma'(h)(f) & \text{if defined} \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

and $\bot$ is some arbitrary but fixed element of val (which certainly is not empty).

The following lemma about fl is straightforward to prove.

Lemma 6 *The embedding* fl *satisfies the following properties*

1. *If* $\mathsf{fl}(\hat{v}, \sigma, \hat{\sigma}) \in T$ *and* $\mathsf{fl}(v, \hat{\sigma}, \sigma') \in U(\hat{\sigma})$ *then* $\mathsf{fl}(v, \sigma, \sigma') \in T; U$.

2. *For any* $v \in \mathsf{val}$, *we have* $\mathsf{fl}(v, \sigma, \sigma) \in R(v)$

3. *For any* x, f *such that* $\sigma(x)(f)$ *is defined, we have* $\mathsf{fl}(\sigma(x)(f), \sigma, \sigma) \in R(x.f)$.

4. *For any* $\vec{f}, \vec{x}, h$ *such that* $h \notin \mathsf{dom}(\sigma)$, *we have*

$$\mathsf{fl}(h, \sigma, \sigma[h \mapsto (f_1 \mapsto x_1, \dots, f_k \mapsto x_k)]) \in T^{\mathsf{obj}}(\vec{f}, \vec{x}) \ .$$

5. *For any* x, f, y *such that* $\sigma(x)$ *is defined, we have* $\mathsf{fl}(x, \sigma, \sigma[x \mapsto (\sigma(x)[f \mapsto y])]) \in T^{\mathsf{fupd}}(x, f, y)$.

Theorem 4 *If*

1. $\Sigma \models \sigma$

2. $\Sigma \models S : E$

3. $XSE \Vdash a : A :: T$ *and*

4. $\sigma, S \vdash a \leadsto v, \sigma'$

then there is Σ' *such that*

i. $\mathsf{fl}(v, \sigma, \sigma') \in T$

ii. $\Sigma' \geq \Sigma$

iii. $\Sigma' \models \sigma'$ *and*

iv. $\Sigma' \models v : A$.

Here Σ and Σ' are *store specifications*, maps from val into specs, that in particular map the constants of base type to their corresponding specification. Informally: we write $\Sigma \models v : A$ to mean that given a store satisfying Σ, value v has specification A; we write $\Sigma \models S : E$ to mean that stack S satisfies the environment (which can be considered as a stack specification) E; and we write $\Sigma \models \sigma$ to mean that σ satisfies Σ. The reader is referred to [AL98] for the motivations and intuitions for these definitions.

Definition 2 *We have the following auxiliary definitions relating to store specifications.*

1. $\Sigma \models v : A$ *if and only if* $\Sigma(v) <:_{\mathsf{aux}} A$
2. $\Sigma \models S : E$ *is defined inductively by*
   - $\Sigma \models \emptyset : \emptyset$
   - $\Sigma \models S : E$ *and* $\Sigma \models v : A$ *implies* $\Sigma \models S[x \mapsto v] : E[x \mapsto A]$
3. $\Sigma \models \sigma$ *exactly when* $\mathsf{dom}\,\Sigma = \mathsf{dom}\,\sigma$ *and for all* v, *supposing* $\Sigma(v) = [f_i = A_i, m_j = \langle B, U \rangle] = A$
   - $\Sigma \models \sigma(v, f_i) : A_i$
   - *for* $\sigma(v, m_j) = \langle \varsigma(y_j) b_j, S_j \rangle$ *there is* E_j *such that*

     $$\Sigma \models S_j : E_j$$

     and

     $$X'S'E' \Vdash b_j : B_j(v) :: U_j(v)$$

     where $X' = X \cup \{y_j\}$, $E' = E[y_j \mapsto A]$ *and* $S' = S[y_j \mapsto v]$.

Proof (Theorem 4) We proceed by induction over the operational semantics: induction over the depth of the derivation of $\sigma, S \vdash a \rightsquigarrow v, \sigma'$. Hence we consider each operational rule in turn.

- Case os_let. Let us write c for *let x=a in b*. We have as our assumptions

$$\Sigma \models \sigma \tag{3.7}$$

$$\Sigma \models S : E \tag{3.8}$$

$$XSE \Vdash c : C :: V \tag{3.9}$$

$$\sigma, S \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b \rightsquigarrow v, \sigma' \tag{3.10}$$

Our induction hypothesis is exactly the text of Theorem 4 for those σ, S, a, v and σ' such that $\sigma, S \vdash a \rightsquigarrow v, \sigma'$ has appeared in the derivation of

assumption 3.10. Using the definition in Table 3.4, assumption 3.9 implies that there are A, A', T, U such that

$$A' <:_{\mathsf{aux}} C \tag{3.11}$$

$$T;U \subseteq V \tag{3.12}$$

and

$$XSE \Vdash a : A :: T \tag{3.13}$$

$$\forall u : XSE \to X'S'E'.\forall x \in X'.E'(x) = A \implies X'S'E' \Vdash b[u] : A' :: U(S'(x)) \ . \tag{3.14}$$

By inspecting the operational semantics, it is clear that to derive 3.10 we must have derived, for some $\hat{v}, \hat{\sigma}$,

$$\sigma, S \vdash a \leadsto \hat{v}, \hat{\sigma} \tag{3.15}$$

$$\hat{\sigma}, S[x \mapsto \hat{v}] \vdash b \leadsto v, \sigma' \ . \tag{3.16}$$

In particular, if we choose $x \notin X$, $X' \stackrel{\text{def}}{=} X \cup \{x\}$, $E' \stackrel{\text{def}}{=} E[x \mapsto A]$ and $u(y) \stackrel{\text{def}}{=} y$ and let $\hat{v} \stackrel{\text{def}}{=} S'(x)$ then from 3.14 we know

$$X'S'E' \Vdash b : A' :: U(\hat{v}) \ . \tag{3.17}$$

The conclusion from applying our induction hypothesis to conclusions 3.15, 3.7, 3.8 and 3.13 is: there exists $\hat{\Sigma}$ such that

$$\mathsf{fl}(\hat{v}, \sigma, \hat{\sigma}) \in T \tag{3.18}$$

$$\hat{\Sigma} \geq \Sigma \tag{3.19}$$

$$\hat{\Sigma} \models \hat{\sigma} \tag{3.20}$$

$$\hat{\Sigma} \models \hat{v} : A \ . \tag{3.21}$$

Since conclusion 3.19 merely states that $\hat{\Sigma}$ is an extension of Σ, it immediately follows that $\hat{\Sigma} \models S : E$. Together with conclusion 3.21, we thus have

$$\hat{\Sigma} \models S[x \mapsto \hat{v}] : E[x \mapsto A] \tag{3.22}$$

We thus conclude, by applying the induction hypothesis to conclusions 3.16, 3.20, 3.22 and 3.17: there is Σ' such that

$$\mathsf{fl}(v, \hat{\sigma}, \sigma') \in U(\hat{v}) \tag{3.23}$$

$$\Sigma' \geq \hat{\Sigma} \tag{3.24}$$

$$\Sigma' \models \sigma' \tag{3.25}$$

$$\Sigma' \models v : A' \ . \tag{3.26}$$

Remember that we require to prove:

i. $\mathsf{fl}(v, \sigma, \sigma') \in V$

ii. $\Sigma' \geq \Sigma$

iii. $\Sigma' \models \sigma'$ and

iv. $\Sigma' \models v : C$.

We prove these goals as follows: (i) using Lemma 6 part 1 and conclusions 3.18 and 3.23, and the transitivity of $\subseteq$ and conclusion 3.12; (ii) using the transitivity of $\geq$ and conclusions 3.19 and 3.24; (iii) is precisely conclusion 3.25; and (iv) using the transitivity of $- <:_{\text{aux}} -$ and conclusions 3.26 and 3.11.

- Case os_minv. Assume

$$\frac{S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(m) = \langle \varsigma(y)b, S' \rangle \qquad \sigma, S'[y \mapsto h] \vdash b \rightsquigarrow v, \sigma'}{\sigma, S \vdash x.m() \rightsquigarrow v, \sigma'} \tag{3.27}$$

$$\Sigma \models \sigma \tag{3.28}$$

$$\Sigma \models S : E \tag{3.29}$$

$$XSE \Vdash x.m() : C :: V \ . \tag{3.30}$$

From assumption 3.30 we conclude that there are B and U such that

$$B(h) <:_{\text{aux}} C$$

$$U(h) \subseteq V$$

$$XSE \Vdash x : [m : B :: U] :: R(S(x)) \ .$$

And so $E(h) = [\cdots, m : B' :: U', \cdots]$, where $B'(h) <:_{\mathrm{aux}} B(h)$ and $U'(h) \subseteq U(h)$. From assumption 3.29, we conclude that $\Sigma \models S(x) : E(x)$, that is $\Sigma \models h : E(x)$. Therefore $\Sigma(h) <:_{\mathrm{aux}} E(x)$. And so $\Sigma(h) = [\cdots, m : \hat{B} : \hat{U}, \cdots]$ where $\hat{B}(h) <:_{\mathrm{aux}} B'(h)$ and $\hat{U}(h) \subseteq U'(h)$. From assumption 3.28, we conclude that there is E' such that $\Sigma \models S' : E'$ and

$$X' \cup \{y\} S'[y \mapsto h] E'[y \mapsto \Sigma(h)] \Vdash b : \hat{B}(h) :: \hat{U}(h) \ . \tag{3.31}$$

Furthermore

$$\Sigma \models S'[y \mapsto h] : E'[y \mapsto \Sigma(h)] \ . \tag{3.32}$$

Now, we conclude, using the induction hypothesis on the derivation of b, assumption 3.28 and conclusions 3.32 and 3.31, that there is Σ' such that

$$\mathsf{fl}(v, \sigma, \sigma') \in \hat{U}(h) \tag{3.33}$$

$$\Sigma' \geq \Sigma \tag{3.34}$$

$$\Sigma' \models \sigma' \tag{3.35}$$

$$\Sigma' \models v : \hat{B}(h) \ . \tag{3.36}$$

Thus we conclude: (i) $\mathsf{fl}(v, \sigma, \sigma') \in V$ by conclusion 3.33 and $\hat{U}(h) \subseteq U'(h) \subseteq U(h) \subseteq V$; (ii) $\Sigma' \geq \Sigma$ is precisely conclusion 3.34; (iii) $\Sigma' \models \sigma'$ is precisely 3.35; and (iv) $\Sigma' \models v : C$ by conclusion 3.36 and $\hat{B}(h) <:_{\mathrm{aux}} B'(h) <:_{\mathrm{aux}} B(h) <:_{\mathrm{aux}} C$.

- Case os_var. Assume

$$\frac{S(x) = v}{\sigma, S \vdash x \rightsquigarrow v, \sigma} \tag{3.37}$$

$$\Sigma \models \sigma \tag{3.38}$$

$$\Sigma \models S : E \tag{3.39}$$

$$XSE \Vdash x : C :: V \ . \tag{3.40}$$

We know from 3.40 that $E(x) <: C$ and $R(S(x)) \subseteq V$. So we conclude: (i) by Lemma 6 part 2, $\mathsf{fl}(v, \sigma, \sigma) \in R(S(x)) \subseteq V$; (ii) trivially, $\Sigma \geq \Sigma$; (iii) from 3.38, $\Sigma \models \sigma$; and (iv) since Σ is a store specification and $E(x) <:_{\mathrm{aux}} C$, we deduce $\Sigma \models x : C$.

- Case os_const. Assume

$$\frac{}{\sigma, S \vdash \mathsf{false} \rightsquigarrow \mathtt{ff}, \sigma} \tag{3.41}$$

$$\Sigma \models \sigma \tag{3.42}$$

$$\Sigma \models S : E \tag{3.43}$$

$$XSE \Vdash \mathit{false} : C :: V \ . \tag{3.44}$$

We know from 3.44 that $Bool <: C$ and $R(\mathtt{ff}) \subseteq V$. So we conclude: (i) by Lemma 6 part 2, $\mathsf{fl}(\mathtt{ff}, \sigma, \sigma) \in R(\mathtt{ff}) \subseteq V$; (ii) trivially, $\Sigma \geq \Sigma$; (iii) from 3.42, $\Sigma \models \sigma$; and (iv) since Σ is a store specification and $Bool <:_{\text{aux}} C$, we deduce $\Sigma \models \mathtt{ff} : C$.

The case for true is similar.

- Case os_cond. Assume $c \equiv \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1$,

$$\frac{S(x) = \mathtt{tt} \quad \sigma, S \vdash a_0 \rightsquigarrow v, \sigma'}{\sigma, S \vdash c \rightsquigarrow v, \sigma'} \tag{3.45}$$

$$\Sigma \models \sigma \tag{3.46}$$

$$\Sigma \models S : E \tag{3.47}$$

$$XSE \Vdash c : C :: V \tag{3.48}$$

We consider the case where $S(x) = \mathtt{tt}$. The other case is exactly similar. From 3.48, we know that there are B, U, B_0, U_0 such that

$$XSE \Vdash c : B(S(x)) :: U(S(x))$$

$$B(S(x)) <: C$$

$$U(S(x)) \subseteq V$$

and

$$B_0(\mathtt{tt}) = B(\mathtt{tt}) \tag{3.49}$$

$$U_0(\mathtt{tt}) = U(\mathtt{tt}) \ . \tag{3.50}$$

From the induction hypothesis on the derivation of a_0, we know

$$\mathsf{fl}(v, \sigma, \sigma') \in U_0(S(x)) \tag{3.51}$$

$$\Sigma' \geq \Sigma \tag{3.52}$$

$$\Sigma' \models \sigma' \tag{3.53}$$

$$\Sigma' \models v : B_0(S(x)) \ . \tag{3.54}$$

So we conclude: (i) from 3.51, 3.50, $\mathsf{fl}(v, \sigma, \sigma') \in U(S(x)) \subseteq V$; (ii) from 3.52, $\Sigma' \geq \Sigma$; (iii) from 3.53, $\Sigma' \models \sigma'$; and (iv) from 3.54, 3.49 and $B(S(x)) <:_{\mathrm{aux}} C$, $\Sigma' \models v : C$.

- Case os_obj. Assume $c \equiv [f_i{=}x_i{}^{i=1..k}, m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}]$,

$$\frac{\begin{array}{c} S(x_i) = v_i{}^{i=1...n} \quad h \notin \mathsf{dom}(\sigma) \quad h \in \mathsf{locn} \\ \sigma' = \sigma[h \mapsto (f_i \mapsto v_i{}^{i=1..k}, m_j \mapsto \langle \varsigma(y_j)b_j, S\rangle^{j=1..\ell})] \end{array}}{\sigma, S \vdash c \rightsquigarrow h, \sigma'} \tag{3.55}$$

$$\Sigma \models \sigma \tag{3.56}$$

$$\Sigma \models S : E \tag{3.57}$$

$$XSE \Vdash c : C :: V \ . \tag{3.58}$$

From 3.58, there exist A_i, B_j, U_j ($i = 1..k, j = 1..\ell$) and T' such that, assuming

$$A' \equiv [f_i{=}A_i{}^{i=1..k}, m_j{=}\langle B_j, U_j\rangle^{j=1..\ell}]$$
$$T' \equiv T^{\mathsf{obj}}(\vec{f}, S(\vec{x}))$$
$$T' \subseteq V$$

we have

$$XSE \Vdash x_i : A_i :: R(S(x_i)) \tag{3.59}$$

$$\begin{array}{l} \forall u : XSE \rightarrow X_jS_jE_j . \forall y_j \in X_j . \\ \quad E_j(y_j) = A \implies X_jS_jE_j \Vdash b_j : B_j(S_j(y_j)) :: U_j(S_j(y_j)) \ . \end{array} \tag{3.60}$$

In particular, for any $h \in \mathsf{locn}$, if, for each j, we choose $y_j \notin X$, $X_j = X \cup \{y_j\}$, $S_j = S[y_j \mapsto h]$, $E_j = E[y_j \mapsto A']$ and $u(y) \stackrel{\mathsf{def}}{=} y$, we have

$$X_jS_jE_j \Vdash b_j : B_j(h) :: U_j(h) \ . \tag{3.61}$$

So defining $\Sigma' \stackrel{\text{def}}{=} \Sigma[h \mapsto A']$, we conclude: (i) by Lemma 6, $\mathsf{fl}(h, \sigma, \sigma') \in T' \subseteq V$; (ii) by definition, $\Sigma' \geq \Sigma$; and (iv) by definition, $\Sigma'(h) = A'$, hence $\Sigma \models h : A'$. To show (iii), we argue as follows. From 3.59, $\Sigma \models S(x_i) : A_i$, and so $\Sigma' \models S(x_i) : A_i$ since Σ' is an extension of Σ. Now $\sigma'(h, f_i) = S(x_i)$, $\Sigma' \models S(x_i) : A_i$ and $\sigma'(h, m_j) = \langle \varsigma(y_j)b_j, S \rangle$ and so from 3.59, $\Sigma' \models h : A'$.

- Case os_fsel. Assume

$$\frac{S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(f) = v}{\sigma, S \vdash x.f \rightsquigarrow v, \sigma} \tag{3.62}$$

$$\Sigma \models \sigma \tag{3.63}$$

$$\Sigma \models S : E \tag{3.64}$$

$$XSE \Vdash x.f : C :: V \ . \tag{3.65}$$

From 3.65, there is $A <: [f{:}A']$ such that

$$A' <: C$$
$$R(x.f) \subseteq V$$
$$XSE \Vdash x.f : A' :: R(x.f) \ .$$

So, if we define $\Sigma' \stackrel{\text{def}}{=} \Sigma$, then we conclude: (i) since $\sigma(h)(f) = v$, by Lemma 6, $\mathsf{fl}(v, \sigma, \sigma) \in R(h.f)$; (ii) trivially $\Sigma' \geq \Sigma$; (iii) from 3.63, $\Sigma' \models \sigma$; and (iv) from 3.63, $\Sigma \models \sigma(S(x), f) : \Sigma(S(x)).f$, and since $v = \sigma(S(x), f)$, $A' = \Sigma(S(x)).f$, $\Sigma' \geq \Sigma$ and $A' <: C$, we have $\Sigma' \models v : C$.

- Case os_fupd. Assume $c \equiv x.f{:=}y$,

$$\frac{\begin{array}{c} S(x) = h \quad h \in \mathsf{locn} \quad \sigma(h)(f) \text{ is defined} \\ S(y) = v \quad \sigma' = \sigma[h \mapsto (\sigma(h)[f \mapsto v])] \end{array}}{\sigma, S \vdash c \rightsquigarrow h, \sigma'} \tag{3.66}$$

$$\Sigma \models \sigma \tag{3.67}$$

$$\Sigma \models S : E \tag{3.68}$$

$$XSE \Vdash c : C :: V \ . \tag{3.69}$$

From 3.69, there are A, A', T such that

$$A' <: [f{:}A]$$
$$T = T^{\mathsf{fupd}}(S(x), f, S(y))$$
$$T \subseteq V$$
$$A <:_{\mathsf{aux}} C$$

and

$$XSE \Vdash c : A :: T \tag{3.70}$$
$$XSE \Vdash x : A :: R(x) \tag{3.71}$$
$$XSE \Vdash y : A' :: R(S(y)) \ . \tag{3.72}$$

From 3.68, $\Sigma \models S(y) : E(y)$, and from 3.71 implies $E(y) <: A'$, thus

$$\Sigma \models v : A' \ . \tag{3.73}$$

Now pick $\Sigma' = \Sigma$. We conclude: (i) by Lemma 6, $\mathsf{fl}(h, \sigma, \sigma') \in T$; and (ii) trivially $\Sigma' \geq \Sigma$. To show (iii), from 3.72, $\Sigma \models \sigma'(h, f) : A'$, and σ' is defined to be the same as σ' at those points other than (h, f), and so from 3.67, we conclude $\Sigma' \models \sigma'$. Finally, to show (iv), from 3.68, $\Sigma \models S(x) : E(x)$, and since $h = S(x)$, we conclude that $\Sigma(x) <: E(x)$. But $E(x) <: A$, and since we have 3.71, we conclude $\Sigma(h) <: A$, i.e. $\Sigma \models S(x) : A$.

□

3.6 Conclusions

The use of HOAS and a direct embedding of the assertion logic allows one to embed AL into a theorem prover with minimal overhead: the use of HOAS means we inherit variable binding; and a direct embedding of the assertion language means we inherit all the theorem-proving facilities of the theorem prover, such as tactics. One can argue that it is only necessary to embed AL into one's favourite theorem prover once and use it forever after; however, we point out

that embeddings are invariably tied to the syntax of the underlying theorem prover, and, should a better prover one day become available, it is reassuring to know that only a minimal investment has been put into the embedding into last year's theorem prover.

Formalisation of AL in a theorem prover also means we automatically inherit a notion of machine proof, at the very least, as a proof script. Crucially, the machine proof can be automatically checked, a property that pencil-and-paper proofs do not enjoy. However, since we use a direct embedding of the assertion language, there is no obvious way to make such machine proofs portable: I cannot use my favourite proof checker that I trust beyond all reasonable doubt.

The embedding has been used to construct proofs for two extended examples, namely an implementation of Euclid's algorithm and a simulation of the dining philosophers example. The exercises have shown that such endeavours are possible, but, writing on a personal note, the author does not claim such experiences to be particularly enjoyable. One finds, when using the embedding (in both LEGO and PVS), that a seemingly unnecessary amount of effort is required in choosing and then instantiating proof rules. The majority of these choices do not require much ingenuity. Worse still, the choices that do require ingenuity are buried deep amongst the others, and so, one must be careful to recognise them when one encounters them during proof construction.

Furthermore, the resulting proof scripts are very sensitive to the syntax of the program that we are considering. Even simple syntactic changes may break a proof script in the sense that it no longer proves the verification statement. In practice, the proof script must be fixed by interactively debugging it using the theorem prover.

In Chapter 5, we develop a *verification condition generator* (VCG) algorithm which can reduce the tedium of constructing proofs.

Though the choice of HOAS for programs and a direct embedding for transition relations allows us to present an embedding of AL with minimal overhead, the repercussions of these choices are felt when we come to justify the implementation: what do we really prove using the encoding?

If we use a first-order, deep embedding of the syntax, then we can use a standard interpretation of the metalanguage and conclude that programs really are programs and the rules really are the rules of AL. Thus the soundness proof in locus classicus carries over directly to the embedding. However, with a HOAS encoding of program syntax and a standard interpretation, programs may not be the same as programs in first-order syntax. (For example, the term $l(a, f)$ has type P for *arbitrary* functions $f : V \to P$.)

Supposing we choose a deep embedding of transition relations, e.g. embed the rules of first-order logic, then it may be possible to normalise a verification proof in the embedding, and then translate it into a proof of AL. However this is not possible in our case, since the side conditions are proved directly using the logic of the underlying theorem prover: there is no longer a correspondence between the logic rules and axioms of the theorem prover and those of the assertion logic.

Instead, to justify our embedding, we find a non-standard interpretation of the metalanguage which models our axioms. In particular, the interpretation of HOAS-style programs is precisely first-order programs and so, for example, allows us to interpret the derivability relation as an auxiliary derivability relation defined by induction over the first-order syntax. We then prove the soundness result for this auxiliary derivability relation, in the spirit of the proof in loc. cit. The technique used in the construction of the model of HOAS is an application of some ideas presented in [Hof99] and is not specific to AL. The approach used here may be useful for justifying HOAS embeddings of other program logics.

Chapter 4

Type inference

In this chapter we describe a type inference algorithm for AL. In the development of this thesis, the algorithm may be considered to be an important component of the *verification condition generator* algorithm to be described in Chapter 5. However, because there is a close relationship between the typing and verification rules of AL (specifically, the typing rules may be considered to be a special case of the verification rules where we have trivial instantiations for the transition relations), it so happens that this algorithm is precisely that of type inference, which interesting in its own right.

4.1 An overview

Following the techniques used by Palsberg et al. [Pal95, PWO97], we take the route whereby we introduce variables for the types we would like to find and derive constraints over these. The constraint system is then considered as a *constraint graph*, using which we construct an *automaton* (in fact, a family of automata) to recognise a solution.

Since automata recognise languages, we must find a language-based representation of types. In fact we work with more general languages which we call *pretypes*. Pretypes allow us to overcome some technical obstacles caused by base types, and give rise to the natural notion of presolution which possess the useful property of directedness.

First we give a presentation of a type system for AL. In fact we give variants of the rules that can be found in [AL98]. The typing rules can be considered to be the verification rules where all the transition relations have been omitted, but since this chapter develops a type inference algorithm, it is useful to present them specifically as typing rules. We then give the language-based representation of (pre)types in Section 4.3. Section 4.4 describes how we derive the constraints system given a program to type check. Before we present the automaton in Section 4.6, we define *constraint graphs* and *(pre)solutions* in Section 4.5. Finally, we introduce the key automaton that allows us to solve the problem in Section 4.6. In this section, we also introduce the notion of well-formedness which we show corresponds precisely to typability. In Section 4.7, we describe in the detail the more complicated algorithms that we require and provide an analysis of their efficiency.

4.1.1 Related work

Since our work was inspired by that of Palsberg et al. [PWO97, Pal95], naturally there are many similarities. However, there are several notable differences also.

In [Pal95], Palsberg considers a calculus (AC) [AC96] of Abadi and Cardelli, which is related to but simpler than that (AL) of Abadi and Leino which we consider. Since AC allows method update operations, fields can be, and are, encoded as methods. Initially it would appear that AC is a more expressive language and thus would have a harder type inference problem. The absence of method update in AL allows for a weaker type system; i.e., there are more well-typed AL programs. More precisely, we have a larger subtype relation, since we allow for covariant method return types. This difference manifests itself in the closure rules for constraint graphs: our closure rules are more complicated. Nevertheless, closure is still computable in the same asymptotic time complexity. Also in [Pal95], there are no base types such as booleans and integers. Since we do have base types, we must have a more involved characterisation of types. Furthermore, we generalise this definition of type to that of pretype, which has useful mathematical properties that types lack and allows for simpler presentation.

In [Hen97], Henglein exploits the fact that object subtyping in those variants of AC, considered by Palsberg in [Pal95], is invariant along fields, and improves on Palsberg's cubic-polynomial time complexity solution. The improvement is possible since it is no longer necessary to maintain the transitive closure of a partial order. Unfortunately, our subtyping relation does not enjoy the same invariance property, and so Henglein's optimisation is not applicable.

In [PWO97], Palsberg et al. consider type inference with non-structural subtyping. The only types are bottom $\bot$, top $\top$ and arrow type $\tau_1 \rightarrow \tau_2$. Since they have both bottom and top, they can find a least shape solution (typing) by finding the smallest set of subterm occurrences to which an arrow type must be assigned. This can be done by accumulating lower and upper bounds for each subterm occurrence using a product automaton. Our subtype relation hints at being non-structural since a subtype can have fewer components (fields and methods). Yet it is also structural in the sense that subtypes of object types are always objects types, and subtypes of base types are always base types. In any case, the type inference problem cannot be solved by simply accumulating lower and upper bounds.

In [PJ97], Palsberg and Jim show that type inference for an object-calculus with simple self types is NP-complete. In contrast, AL does not have self types.

It has recently been brought to the attention of the author through personal communications with Jens Palsberg that Palsberg et al. have independently extended the same type inference framework to a typed object calculus with covariant fields [PZJ02]. Furthermore, in loc. cit., there is no syntactic distinction between covariant and invariant fields, but rather, fields are determined to be covariant by inference. One also notices, in loc. cit., a striking similarity between their definition of "R is satisfaction-closed" and our closure rules in Table 4.5. Their algorithm (unlike ours) does not handle base types.

$$A, B ::= Bool \mid [f_1{:}A_1, \ldots, f_k{:}A_k, m_1{:}B_1, \ldots, m_\ell{:}B_\ell]$$

Table 4.1: Syntax of AL types

$$\frac{}{Bool <: Bool}$$

$$\frac{B_j <: B'_j{}^{j=1..\ell'}}{[f_i{:}A_i{}^{i=1..k}, m_j{:}B_j{}^{j=1..\ell}] <: [f_i{:}A_i{}^{i=1..k'}, m_j{:}B'_j{}^{j=1..\ell'}]} \quad \{k' \leq k \quad \text{and} \quad \ell' \leq \ell\}$$

Table 4.2: Definition of subtype

4.2 Type derivations

Abadi and Leino present in [AL98] a type system deriving judgements of the form

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A$$

where a is a program, each x_i a variable and A and each A_i a type, the syntax of which is defined in Table 4.1. Such a judgement can be understood to mean that assuming each x_i has type A_i, then the program a has type A.

These judgements are derived using rules such as

$$\frac{E \vdash x : [f{:}A]}{E \vdash x.f : A}$$

which means, if we can derive $E \vdash x : [f : A]$, then we can derive $E \vdash x.f : A$. Of particular note is the subsumption rule

$$\frac{E \vdash a : A}{E \vdash a : A'} \quad \{A <: A'\}$$

where we have as a side condition $A <: A'$. The definition of the subtype relation $<:$ can be found in Table 4.2. Of course, the presence of the side condition means that this rule can be applied only when the side condition is true.

A closed program a is said to be type correct (or typable) if there is a type A and a type derivation for $\vdash a : A$. Type derivations are typically presented as trees.

However, because of the subsumption rule, which can be applied at any point of the type derivation, the derivation trees for a given program do not have unique shape. For our purposes, we take the equivalent collection of typing rules presented in Table 4.3. The rules in Table 4.3 are equivalent to those found in locus classicus in the following sense: for program a and type A,

$$\vdash_0 a : A \quad \text{iff} \quad \vdash a : A$$

where we write $\vdash_0 a : A$ to denote typability using the rules of locus classicus. Note that there is no longer a separate subsumption rule: it has been incorporated into the other rules. Note also that it is not necessary to incorporate subtyping into the rules for if, let and method invocation because the subtyping can be pushed further up the derivation tree. Of course, one can incorporate the subsumption rule into *all* the rules, in which case, we can prove equivalence of the two collection of rules using the straightforward lemma: for typing context E, program a and type A,

$$E \vdash_0 a : A \quad \text{iff} \quad E \vdash a : A \ .$$

However, this produces more variables and more constraints. These extra constraints are redundant because the subtype relation is transitive. To prove equivalence of the rules in Table 4.3 and those of locus classicus, we need to strengthen the lemma above to: for typing context E, program a and type A,

$$E \vdash_0 a : A \quad \text{implies} \quad E \vdash a : A', \text{ for all } A' \text{ such that } A <: A'$$
$$E \vdash a : A \quad \text{implies} \quad E \vdash_0 a : A \ .$$

The proof of this lemma falls out, again, from the fact that the subtype relation is transitive.

Using these new rules, it is clear that the shape of a type derivation for program term a is determined by the term's syntactic structure.

4.3 Pretypes and types

Since we use automata to solve the type inference problem, it is convenient to have a presentation of types based on formal languages. Our automaton will

$$\frac{}{E \vdash x : A} \quad \{E(x) <: A\} \qquad \text{(var)}$$

$$\frac{}{E \vdash \mathit{false} : A} \quad \{Bool <: A\} \qquad \text{(constf)}$$

$$\frac{}{E \vdash \mathit{true} : A} \quad \{Bool <: A\} \qquad \text{(constt)}$$

$$\frac{E \vdash x : Bool \quad E \vdash a_0 : A \quad E \vdash a_1 : A}{E \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A} \qquad \text{(cond)}$$

$$\frac{E \vdash a : A \quad E, x{:}A \vdash b : A'}{E \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : A'} \qquad \text{(let)}$$

$$\frac{E \vdash x_i : A_i{}^{i=1..k} \quad E, y_j{:}A \vdash b_j : B_j{}^{j=1..\ell}}{E \vdash [f_i{=}x_i{}^{i=1..k} m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}] : A'} \quad \left\{ \begin{array}{l} A = [f_i{:}A_i{}^{i=1..k}, m_j{:}B_j{}^{j=1..\ell}] \\ A <: A' \end{array} \right\} \qquad \text{(obj)}$$

$$\frac{E \vdash x : [f : A]}{E \vdash x.f : A'} \quad \left\{ A <: A' \right\} \qquad \text{(fsel)}$$

$$\frac{E \vdash x : [m : A]}{E \vdash x.m() : A} \qquad \text{(minv)}$$

$$\frac{E \vdash x : A \quad E \vdash y : A'}{E \vdash x.f_i{:=}y : A''} \quad \left\{ \begin{array}{l} A <: [f{:}A'] \\ A <: A'' \end{array} \right\} \qquad \text{(fupd)}$$

Table 4.3: Typing rules with built-in subsumption

actually recognise pretypes, which one may consider as incomplete types. We now proceed to introduce these definitions and show their correspondence with types as used by Abadi and Leino in loc. cit.

First, we recall that sets fname and mname are disjoint and we define an alphabet $\Sigma_0 \stackrel{\text{def}}{=} \mathsf{fname} \cup \mathsf{mname}$. Already, one can see that if we consider languages over this alphabet, it is possible, in some way, to capture the structure of object types. In fact, in [Pal95], types are defined precisely to be such languages. However, in our case, the problem is complicated by the fact that we have base types.

We notice that prefix-closed languages over Σ_0 represent trees whose branches are labelled with field and method names, but whose nodes and leaves are unlabelled. We want to represent trees whose leaves are decorated with names of base types. In fact, we generalise this further and consider trees whose nodes are also decorated, but insist they are decorated with a special decoration $\star$, which one understands as "object type".

One notes that such trees can be represented as a pair (L, f) where L is a language and f a mapping from L into a set Δ of decorations. Though this is certainly a reasonable approach to the problem at hand, it appears the following representation of trees allows for a cleaner presentation.

Alternatively, such trees can be represented as languages of words whose letters alternate between node labels (decorations) and edge labels (field and method names). We thus define a set of decorations $\Delta \stackrel{\text{def}}{=} \mathcal{B} \cup \{\star\}$, the set of base type names plus object type, and define a new alphabet $\Sigma \stackrel{\text{def}}{=} \Sigma_0 + \Delta$, meaning the disjoint union[1] of Σ_0 and Δ.

Definition 3 *Let L be a subset of Σ^*.*

1. *A word in Σ^* is said to be* alternating *if for any two adjacent letters, one is in Σ_0 and the other is in Δ.*

2. *A word α is said to be* maximal in L *if $\alpha \in L$ and for all α' in L*

$$\alpha \preceq \alpha' \quad \textit{implies} \quad \alpha = \alpha' \ ,$$

where $\preceq$ denotes prefix ordering.

3. *Given a word α, and a set of words L, we define $L{\downarrow}\alpha$ read "L at α" by*

$$L{\downarrow}\alpha \stackrel{\text{def}}{=} \{\beta \mid \alpha\beta \in L\} \ .$$

4. *For word α and a set of words L, we define αL by*

$$\alpha L \stackrel{\text{def}}{=} \{\alpha\beta \mid \beta \in L\} \ .$$

[1] To all intents and purposes, $\Delta + \Sigma_0$ can be considered to be the same as $\Delta \cup \Sigma_0$ except that we use the former notation to emphasise the disjointness of the union and in particular the fact that we can *decide* whether an element in $\Delta + \Sigma_0$ is in Δ or Σ_0.

It is easy to check that for any word α, the operation $\downarrow\alpha$ distributes over intersection, i.e. for X a set of languages, $(\bigcap X)\downarrow\alpha = \bigcap_{L\in X}(L\downarrow\alpha)$.

We now define pretypes and types as languages satisfying certain properties, as follows.

Definition 4 *A set of words $t \subseteq \Sigma^*$ is said to be a* pretype *if*

LANG *t is nonempty and prefix closed;*

ALT *all non-empty words in t are alternating and start in Δ, that is, if $d\alpha \in t$ then $d\alpha$ is alternating and $d \in \Delta$;*

LEAF *for decoration d, if $d \in \mathcal{B}$ and $\alpha d \in t$ then αd is maximal in t; and*

TREE *for decorations d, d', if $\alpha d \in t$ and $\alpha d' \in t$ then $d = d'$.*

Let P to be the set of all pretypes. Let a type *be a finite*[2] *pretype whose maximal words are all of odd length, and let T to denote the set of all types.*

The following lemma is easy to check. For our purposes, a set of words is said to be a *language* if it has property LANG.

Lemma 7 *1. For languages t, t' in Σ^* with $t \subseteq t'$, if t' is a pretype, then so is t.*

2. Pretypes are closed under intersections, i.e., for X a set of pretypes, $\bigcap X$ is also a pretype.

3. For α an even length word, $t\downarrow\alpha$ is a pretype if $\alpha \in t$ and empty otherwise.

Note that types, however, are not closed under intersection. Consider, for example, the intersection of types $t = \{\varepsilon, b_1\}$ and $t' = \{\varepsilon, b_2\}$: the intersection $t \cap t' = \{\varepsilon\} \notin T$ since ε is maximal in $t \cap t'$ and ε is not of odd length.

[2]The finiteness restriction here can be relaxed to incorporate recursive types.

4.3.1 Constructions on pretypes

Now that we have defined what a pretype is, let us introduce some standard constructions. For base type b, there is the corresponding pretype

$$\overline{b} \stackrel{\text{def}}{=} \{\varepsilon, b\}$$

which, in fact, is a type. Also, given pretypes $t_1, \ldots, t_k, u_1, \ldots, u_\ell$, we introduce notation

$$[f_i{:}t_i{}^{i=1..k}, m_j{:}u_j{}^{j=1..\ell}] \stackrel{\text{def}}{=} \{\varepsilon, \star\} \cup \bigcup_{i=1}^{k} \star f_i t_i \cup \bigcup_{j=1}^{\ell} \star m_j u_j \ .$$

It is straightforward to check that if t_i (for $i = 1..k$), u_j (for $j = 1..\ell$) are types then $[f_i{:}t_i{}^{i=1..k}, m_j{:}u_j{}^{j=1..\ell}]$ is also a type.

Thus, what we have shown is that given a type (of AL), A, there is a type $\overline{A} \in T$. Conversely, the set of types T is characterised by these constructors: that is

$$T = \{\overline{A} \mid A \text{ a type of AL}\} \ .$$

To show this, we present an algorithm that, given a type t, computes an AL-type A such that $t = \overline{A}$. We write $\underline{t}$ for this A.

Case $b \in t$, for some b in $\mathcal{B}$. By LEAF, b is a maximal word in t and so by TREE, $t = \overline{b}$.

Case $\star \in t$. If $\star$ is maximal in t, then by TREE, we conclude $t = \overline{[]}$. Otherwise, by TREE again, we may assume that there is no word b in t, ($b \in \mathcal{B}$). So suppose $\star f_1, \ldots, \star f_k, \star m_1, \ldots, \star m_\ell$ are all the two-element words in t. We recursively compute $\underline{t{\downarrow}\star f_i}$ and $\underline{t{\downarrow}\star m_j}$, and by expanding the definition above, we trivially conclude

$$\underline{t} = [f_i{:}\underline{t{\downarrow}\star f_i}, m_j{:}\underline{t{\downarrow}\star m_j}] \ .$$

Thus, we know that our new definition of type coincides with the definition of AL-type, and therefore, we can define $<: \subseteq T \times T$ using the syntactic rules in Table 4.2. Explicitly, $t <: t'$ if and only if $\underline{t} <: \underline{t'}$.

4.3.2 Ordering pretypes

We have a binary relation $<:$ defined over types. We now define a partial order over pretypes which we show is an extension of $<:$.

Define a relation $\leq \subseteq P \times P$ over pretypes as follows. Given $t, t' \in P$, we write

$$t \leq t'$$

precisely when

$$\text{both} \quad \alpha f \in t' \implies t{\downarrow}\alpha f = t'{\downarrow}\alpha f \quad \text{and} \quad t \supseteq t' \ .$$

That is, broadly, branches of tree t' are branches of t, but, when descending the branches of t', if we traverse a branch labelled by a field f, the subtree rooted at the (far) end of this branch occurs as a subtree of t. This corresponds exactly to the fact that there is covariant subtyping along methods, but not fields. Some properties that are satisfied by this definition are listed in the theorem below.

Theorem 5 *1. $(P, \leq)$ is a partial order.*

2. For pretypes t, t', if $t <: t'$ then $t \leq t'$.

3. For types t, t', if $t \leq t'$ then $t <: t'$.

Note that in property 2, if $t <: t'$ holds for pretypes t, t', then they are necessarily types, by definition of $<:$. Proofs of parts 1 and 3 can be found in Section A.2.

We also have the following useful property about this ordering on pretypes.

Lemma 8 *For pretypes t, t', if $t \leq t'$ then $t{\downarrow}\beta \leq t'{\downarrow}\beta$.*

Proof By the standard result that ${\downarrow}\beta$ preserves inclusion, we immediately conclude $t{\downarrow}\beta \supseteq t'{\downarrow}\beta$.

Now suppose $\alpha f \in t'{\downarrow}\beta$. This is precisely $\beta\alpha f \in t'$. And since $t \leq t'$, we know that $t{\downarrow}\beta\alpha f = t'{\downarrow}\beta\alpha f$. This is can be written as $(t{\downarrow}\beta){\downarrow}\alpha f = (t'{\downarrow}\beta){\downarrow}\alpha f$. $\square$

4.4 Generating constraints

Now that we have the preliminary developments out of the way, we are ready to consider the problem of type inference. We follow broadly the approach of Palsberg [Pal95, PWO97] and proceed as follows.

Given a program a, we introduce type variables for subterm occurrences (and variables) and derive constraints that express necessary and sufficient conditions for typability. The problem is thus reduced to solving this system of constraints. We consider these constraints as edges in a graph, and then use this to define a transition function for a family of automata; we get an (non-deterministic finite-state) automaton by taking each unknown type variable as a start state. We show that given any well-formed constraint graph, the solution space is directed, and thus has a least upper bound. We show that the languages (pretypes), accepted by the automata, give us precisely this least upper bound.

Assume that we are given a term a whose bound variables are pairwise distinct. We introduce the following type variables:

- for each subterm occurrence a', we introduce a type variable $[\![a']\!]$;
- for each variable x, we introduce a type variable $[x]$; and
- for each occurrence of subterm $x.f$, we introduce a type variable $\langle x.f \rangle$.

Note that for (program) variable x and field name f, we have distinct variables $[x]$ and $[\![x]\!]$, and distinct variables $[\![x.f]\!]$ and $\langle x.f \rangle$. Intuitively, for typing judgement

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A \ ,$$

we write $[x_i]$ for A_i and $[\![a]\!]$ for A. Although the pair $x_i{:}A_i$ may appear to the left of the turnstile of many typing judgements, $[x_i]$ is still well-defined since A_i must be syntactically the same in each case. The distinction between $[x]$ and $[\![x]\!]$ is made clear in the special case where $a \equiv x$. In the case where $a \equiv x.f$, the rule

$$\frac{E \vdash x : [f{:}A]}{E \vdash x.f : A'} \quad \{A <: A'\}$$

must occur, and, in this case, we write $\langle x.f \rangle$ for A and $[\![x.f]\!]$ for A'.

Constraints have the form $A \leq A'$ where A and A' are type expressions with possible occurrences of type variables. In fact, for our purposes, it suffices to consider the following restricted class of type expressions

$$A \quad ::= \quad X \mid b \mid [f_i{:}X_i{}^{i=1,\dots,k}, m_j{:}Y_j{}^{j=1,\dots,\ell}]$$

where X, Y and their decorated variants are type variables, and b are the base types. A solution of a system of constraints is an assignment of types to type variables. Our immediate objective is to find a system of constraints whose solutions are precisely the typings of a.

We consider all subterm occurrences of a, each of which gives constraints as listed in Table 4.4, where we write $t = t'$ to mean $t \leq t'$ and $t' \leq t$. Note that when we write $[\![a']\!]$, strictly speaking, a' is a program occurrence. However, we shall often write $[\![a']\!]$ when a' is in fact a subterm of a; we take care in cases where syntactically equivalent subterms occur more than once in a.

The constraints listed in Table 4.4 are derived directly from the side conditions of the rules in Table 4.3. Thus a solution of the resulting system of constraints is a typing of a and vice versa.

Lemma 9 *Given a program a, let C be the set of constraints determined by Table 4.4. The solutions to C are in one-one correspondence with typings of a.*

4.5 Constraint graphs

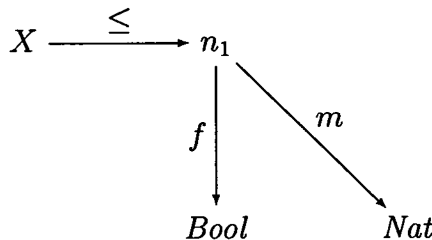
On the problem of solving constraints, we first develop the notion of constraint graph. A type tree itself is a special instance of a constraint graph: the field and method name labelled branches correspond to the *structural* edges of the constraint graph. Constraint graphs also have subtype edges which represent the subtyping constraints between nodes.

For example, for type variable X, the constraint $X \leq [f{:}Bool, m{:}Nat]$ could

Subterm occurrence	Constraints
x	$[x] \leq [\![x]\!]$
true	$Bool = [\![true]\!]$
false	$Bool = [\![false]\!]$
if x *then* a_0 *else* a_1	$[\![a_0]\!] = [\![a_1]\!] = [\![if\ x\ then\ a_0\ else\ a_1]\!]$, $[\![x]\!] = Bool$
let x=a *in* b	$x = [\![a]\!]$, $[\![b]\!] = [\![let\ x{=}a\ in\ b]\!]$
$[f_i{=}x_i{}^{i=1..k}, m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}]$	$A \equiv [f_i{:}[\![x_i]\!]^{i=1..k}, m_j{:}[\![b_j]\!]^{j=1..\ell}]$ $A = [y_1] = \cdots = [y_\ell]$, $A \leq [\![[f_1{=}\cdots m_1{=}\cdots]]\!]$
$x.f$	$[\![x]\!] = [f{:}\langle x.f\rangle], \langle x.f\rangle \leq [\![x.f]\!]$
$x.m()$	$[\![x]\!] = [m{:}[\![x.m()]\!]]$
$x.f{:=}y$	$[\![x]\!] \leq [f{:}[\![y]\!]], [\![x]\!] \leq [\![x.f{:=}y]\!]$

Table 4.4: Constraints generation. Given a program a, we consider all its subterm occurrences. For each occurrence we look up its corresponding constraints in the table above.

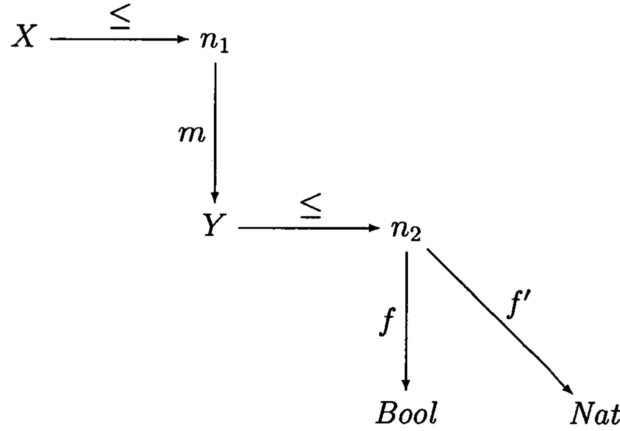
be represented by the following constraint graph.



We have named the root node of $[f{:}Bool, m{:}Nat]$ as n_1. We can use this constraint graph to solve for X as follows: start at X, and walk along the subtype edge to n_1; then from n_1 we see that we can walk along either the f or m labelled edges, and arrive at node *Bool* or *Nat*, respectively; thus we conclude that a solution is $X = [f{:}Bool, m{:}Nat]$.

Consider a more complicated example. For X, Y type variables, the con-

straints $X \leq [m{:}Y]$ and $Y \leq [f{:}Bool, f'{:}Nat]$ can be represented by the following constraint graph.



We can use this constraint graph to solve for X and Y as follows. Starting from Y, we can walk along the subtype edge to n_2, and as in the previous example, we can walk along f or f', so a possible solution for Y is $Y = [f{:}Bool, f'{:}Nat]$. Starting from X, we can walk along the subtype edge to n_1 and walk along the m labelled edge to Y, whence we can walk along the same edges as before. Thus a possible solution for X is $X = [m{:}[f{:}Bool, f'{:}Nat]]$.

Constraint graphs can be considered to be a generalised type trees, where, as well as the possibility of descending structural edges (labelled by field and method names), we can also walk along subtype edges.

Now we turn to the formal definition of constraint graph. Recall that we define $\Sigma_0 \stackrel{\text{def}}{=} \mathsf{fname} \cup \mathsf{mname}$, decorations $\Delta \stackrel{\text{def}}{=} \mathcal{B} \cup \{\star\}$ and alphabet $\Sigma \stackrel{\text{def}}{=} \Sigma_0 + \Delta$. A *constraint graph* $G = (N, C, \mathcal{B}; E, \leq)$ is a directed graph consisting of three disjoint sets of nodes N, C and $\mathcal{B}$, a set of labelled edges $E \subseteq C \times \Sigma_0 \times N$, and edges $\leq \subseteq G \times G$ (writing G as shorthand for the set of all nodes $N \cup C \cup \mathcal{B}$), satisfying the following additional property:

- every node has at most one outgoing edge of label ℓ.

Intuitively, one can think of C and $\mathcal{B}$ as the nodes whose types we already know—the (compound) object types $[f_i{:}X_i, m_j{:}Y_j]$, and base types b; and think of N as the nodes whose types we have to infer. It suffices for the edges in E to have type

$C{\times}\Sigma_0{\times}N$ since expressions in our constraints systems are of the restricted form. A more general notion of constraint graph might insist that $E \subseteq C{\times}\Sigma_0{\times}G$.

Consider the constraint $[f{:}X, m{:}Y] \leq [f{:}X', m{:}Y']$. By the definition of subtype (see Table 4.2), we see that this induces the following further constraints: $X = X'$ and $Y \leq Y'$. The constraint $X = X'$ can be equivalently expressed as two constraints $X \leq X'$ and $X' \leq X$. To account for this, we introduce the following notion of closure.

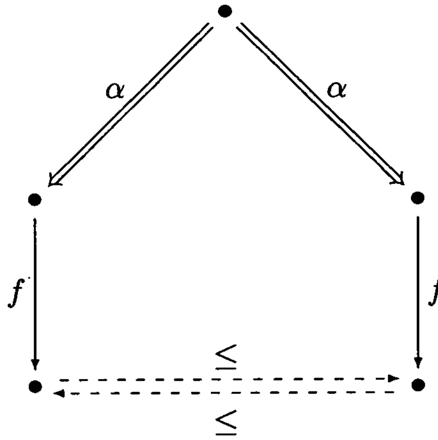
We introduce the notation $u \xrightarrow{l} v$, for l a label (i.e. a field or method name) and nodes u, v to denote that $(u, l, v) \in E$. We also introduce the notation $u \overset{\alpha}{\Rightarrow} v$, defined inductively as follows.

$$\overset{\varepsilon}{\Rightarrow} \overset{\text{def}}{=} \overset{\leq}{\rightarrow}$$

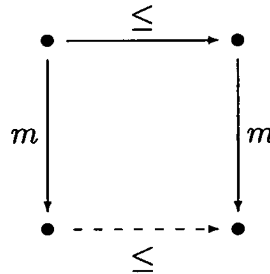
$$\overset{l\alpha}{\Rightarrow} \overset{\text{def}}{=} \overset{\leq}{\rightarrow}; \overset{l}{\rightarrow}; \overset{\alpha}{\Rightarrow} \; .$$

Definition 5 *A constraint graph is said to be* closed *if* $\leq$ *is reflexive, transitive and closed under the following rules*

1. *for any field name* f,

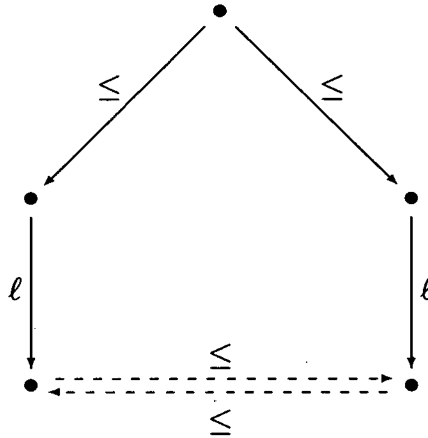


2. *for any method name* m,



where existence of the dashed arrows are implied by the existence of the solid arrows.

We note that our closure rules are already more complicated than those in [Pal95]. Therein is a closure rule of the form



which is a special case of rule 1 and also more general case of rule 2. The apparent contradiction is explained by the fact in AL there is a distinction between fields and methods: the rule for fields is a more general case, whereas the rule for methods is a special case.

More alarmingly, it appears that rule 1 requires us to perform a breadth-first search for each closure step. Fortunately there is an efficient closure algorithm, asymptotically no worse than that in [Pal95], which is detailed in Section 4.7.

4.5.1 Presolutions

A function $h : G \rightarrow P$ is said to be a *presolution* if

1. for b in $\mathcal{B}$, we have $h(b) = b$;
2. for u in C, we have $\star \in h(u)$;
3. for $\ell \in \Sigma_0$, if $u \xrightarrow{\ell} v$ then $h(u){\downarrow}\star\ell = h(v)$ and conversely, if $u \in C$ then $\star\ell \in h(u)$ implies there is some v such that $u \xrightarrow{\ell} v$;
4. if $u \leq v$ then $h(u) \leq h(v)$.

A presolution is said to be a *solution* if its range is included in T. That is, $h(u)$ is in fact a type for each u.

We now consider the space of presolutions ordered by pointwise reverse inclusion. As it turns out, this partially ordered space is closed under pointwise intersection.

Theorem 6 *The space of presolutions is closed under intersections. That is, given a set of presolutions H, the intersection $\bigcap H \stackrel{\text{def}}{=} \bigcap_{h \in H} h$ is a presolution.*

The proof of this theorem can be found in Section A.2.

Since the space of presolutions is closed under pointwise intersection, it is therefore directed. And so, supposing there is at least one presolution to a constraint graph, we know that there is a least upper bound which is a presolution itself.

4.5.2 Presolutions of closures

Conveniently, the closure (with respect to Definition 5) of a constraint graph has the same solutions as the original graph.

Theorem 7 *Given a constraint graph G, if h is a presolution of G then it is a presolution of G', the closure of G with respect to Definition 5.*

The proof can be found in Section A.2.

Conversely, if h is a presolution of G', then it is also a presolution of G since G' has more constraints. Thus we have the following simple corollary of the above theorem.

Corollary 1 *A constraint graph and its closure have the same presolutions.*

4.5.3 From presolutions to solutions

Later, we construct an automaton that computes the least upper bound presolution (Corollary 3). Though a presolution is better than nothing, we really desire

a solution. Fortunately, it is a small step from presolution to solution. We now consider "closing" a presolution to give us a solution.

Intuitively, a pretype (presolution) is a type (solution) that is incomplete in a particular way; some of the the leaf decorations may be missing. However, the space of presolutions has the advantage that it is directed, and so, there exists a most general presolution. In contrast, there is not necessarily a most general solution. Here the most general presolution in some way encompasses solutions of least shape as described by Palsberg.

Given a finite presolution h, we consider the following closure construction for defining h'. For each u in G, define $h'(u)$ to be $h(u)$ except whenever there is some even α, maximal in $h(u)$, we ensure that $\alpha\star \in h'(u)$.

Theorem 8 *The function h', so constructed, is a solution of G.*

Proof The result falls out easily from case by case analysis of the properties of presolution. □

4.5.4 From constraints to graphs

Given our constraints, we take N to be the set of all type variables. For each subterm in the constraints system of the form

$$[f_1{:}A_1, \ldots, f_k{:}A_k, m_1{:}B_1, \ldots, m_\ell{:}B_\ell]$$

with type variables A_i and B_j, we create a node u in C and define the edges $u \xrightarrow{f_i} A_i$ and $u \xrightarrow{m_j} B_j$. Finally, we define $\leq$ edges for the inequalities, where $A = A'$ is interpreted as $A \leq A'$ and $A' \leq A$.

Clearly the resulting constraint graph is solvable if and only if the constraints system is solvable.

4.6 A family of automata

In this section, we construct an automaton $\mathcal{A}$ (more precisely, a transition function giving a family of automata, one for each start state) that accepts the least

presolution. Given a closed constraint graph $G = (N, C, B; E, \leq)$, we take as the set of states,

$$2 \times G \stackrel{\text{def}}{=} \{(i, u) \mid i \in \{0, 1\}, u \in G\} \ ,$$

and define transitions using the following rules

$$\begin{array}{lll} (i, u) \xrightarrow{\varepsilon} (i, v) & \text{if} & u \leq v \\ (0, u) \xrightarrow{\star} (1, u) & \text{if} & u \in C \\ (0, b) \xrightarrow{b} (1, b) & \text{if} & b \in B \\ (1, u) \xrightarrow{\ell} (0, v) & \text{if} & u \xrightarrow{\ell} v \ . \end{array}$$

We follow convention and write $(i, u) \xrightarrow{\alpha} (i', u')$ for the sequence of transition steps accepting the letters in α starting from state (i, u) and finishing in state (i', u'). Here, ε denotes an internal move; in particular, the automaton accepts a word α if and only if it accepts letters $l_1 \cdots l_n$ in sequence such that $\alpha = l_{i_1} \ldots l_{i_k}$ where $i_1 \cdots i_k$ is the subsequence consisting of those indices $i \in 1..n$ for which $l_i \neq \varepsilon$.

Immediately we see that our automaton has the following invariant property. If $(i, u) \xrightarrow{\alpha} (i', u')$, then α has even length precisely when $i = i'$ and odd length otherwise.

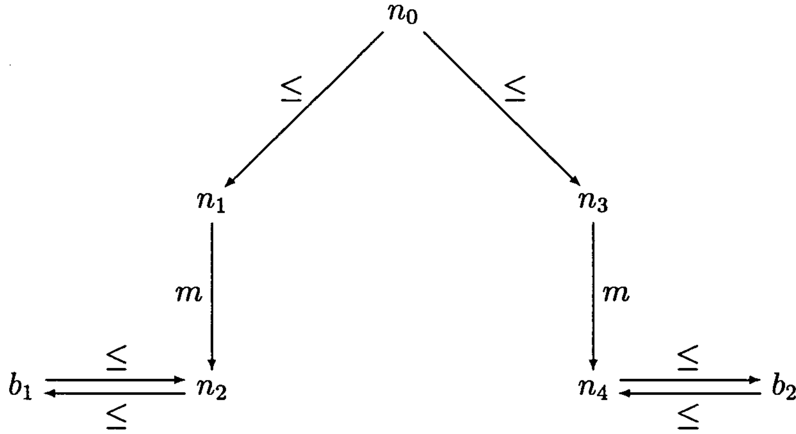
Let us write $\mathcal{L}_s$ for the language accepted by $\mathcal{A}$ from initial state $(0, s)$. We define the function $\psi : G \to \text{Pow}(\Sigma^*)$ as follows

$$\psi(s) \stackrel{\text{def}}{=} \mathcal{L}_s \ .$$

We now proceed to show that ψ is precisely the least upper bound of the space of presolutions, if it exists.

4.6.1 Well-formed constraint graphs

Note that $\mathcal{L}_s$ itself may not be a pretype. Consider the following constraint graph for $b_1, b_2 \in B$ with $b_1 \neq b_2$



and its corresponding automaton. Starting from $s \stackrel{\text{def}}{=} (0, n_0)$, the automaton can perform an internal move to $(0, n_1)$ and eventually accept $\star m b_1$:

$$s \xrightarrow{\varepsilon} (0, n_1) \xrightarrow{\star} (1, n_1) \xrightarrow{m} (0, n_2) \xrightarrow{\varepsilon} (0, b_1) \xrightarrow{b_1} (1, b_1) \ .$$

Alternatively it can perform an internal move to $(0, n_3)$ and eventually accept $\star m b_2$:

$$s \xrightarrow{\varepsilon} (0, n_3) \xrightarrow{\star} (1, n_3) \xrightarrow{m} (0, n_4) \xrightarrow{\varepsilon} (0, b_2) \xrightarrow{b_2} (1, b_2) \ .$$

Immediately we see that $\mathcal{L}_s$ is not a pretype because it does not satisfy property TREE.

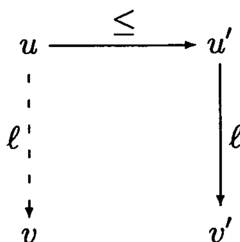
However, such constraint graphs have no solutions. Given a graph G, its automaton must satisfy certain properties for G to have a solution. If the automaton of G satisfies these properties, then G is said to be *well-formed.*[3]

Definition 6 *Let G be a constraint graph and $\mathcal{A}$ its corresponding automaton. Constraint graph G is said to be* well-formed *if it satisfies the following properties:*

[3]Note that an immediate consequence of property 1 of Definition 6 is that no program requiring recursive types gives a well-formed constraint graph. To allow recursive types, this property must be removed and possibly replaced by a weaker property. The correct weakening is necessarily dependent on the definition of the subtype relation.

1. *$\mathcal{A}$ accepts no infinite word;*

2. *for u, u' in C, and ℓ in* mname ∪ fname



the dashed arrow exists whenever the solid arrows do, in the diagram above; and

3. *for each s in G, the language $\mathcal{L}_s$ is a pretype.*

We see immediately that for the previous example, if we choose $s = (0, n_0)$ then, since $\mathcal{L}_s$ is not a pretype, the graph is not well-formed since it does not satisfy property 3 above.

4.6.2 Completeness

We now proceed to show that our definition of well-formedness is complete: a constraint graph with at least one solution is well-formed.

Firstly, we show the stronger result that ψ is an upper bound of all presolutions.

Theorem 9 *For any presolution h, we have $h(s) \supseteq \mathcal{L}_s$ for all s.*

The proof can be found in Section A.2.

And now the completeness results follows.

Corollary 2 *A constraint graph with at least one solution is well-formed.*

Proof Suppose G has solution h. Thus h is, in particular, a finite presolution and from Theorem 9, we know that $h(s) \supseteq \mathcal{L}_s$. So we conclude that $\mathcal{A}$ only accepts finite words and also $\mathcal{L}_s$ is a pretype. We have thus shown that G satisfies respectively properties 1 and 3 of well-formedness.

To show that G also satisfies property 2, suppose $u, u' \in C$, $u \leq u'$, and $u' \xrightarrow{\ell} v'$ for some label ℓ. Since h is a presolution, $\star\ell \in h(u')$. We know that $h(u) \leq h(u')$ and in particular, $h(u) \supseteq h(u')$, therefore $\star\ell \in h(u)$ also. Finally, since h is a presolution, it must be that $u \xrightarrow{\ell} v$. □

4.6.3 Soundness

Conversely (and fortunately), our definition of well-formedness is also sound; a well-formed constraint graph has a solution. We prove this by first exhibiting a presolution, namely that recognised by our automaton, and then applying Theorem 8 to obtain a solution.

First we have the show that ψ as recognised by our automaton really is a presolution.

Theorem 10 *Assuming that G is well-formed, the function ψ is a presolution.*

The proof can be found in Section A.2.

Now we can use Theorem 9 to prove the following corollaries.

Corollary 3 *Given a well-formed G, the function ψ is the least upper bound presolution.*

Proof From Theorem 9 we know that ψ is an upper bound of all presolutions. Moreover, Theorem 10 tells us that ψ is a presolution, hence it is the least upper bound of all presolutions. □

Corollary 4 *A well-formed and closed constraint graph has a solution.*

Proof Supposing G is well-formed, from Theorem 9 we know that ψ is finite. So by Theorem 8, G has a solution. □

4.6.4 Type inference

A primary application of our automaton is, given a program occurrence or variable, to compute all paths in its inferred type tree. This is used in our VCG

algorithm described in Chapter 5. Since it is likely that this is applied to many program occurrences and variables, it is more efficient to first determinise our automaton. We can use the subset approach as described in [HU79] to define a deterministic automaton $\mathcal{A}'$, which has exponentially many states as $\mathcal{A}$. However, our automaton has specific properties which allow us to determinise it into one that has no more states and in cubic time complexity (with respect to the size of the automaton).

First let us define an ordering on states of $\mathcal{A}$ as follows: define

$$s \leq s' \quad \text{iff} \quad s \xrightarrow{\varepsilon} s' \ .$$

It is straightforward to check that $\leq$ over automaton states is a preorder (i.e. a reflexive and transitive order) because it is effectively a lifting of a preorder (namely $\leq$ over graph nodes).

Also, given a set of states X (i.e. a state of $\mathcal{A}'$), clearly $\mathcal{A}'$ accepts the same language starting from state

$$\{s \mid \exists s_0 \in X . s_0 \leq s\}$$

as starting from X itself. So we can restrict our concerns to only those states of $\mathcal{A}'$ which are upclosed.

Furthermore, if X is a chain (i.e. if $x, y \in X$ then $x \leq y$ or $y \leq x$), and $\mathcal{A}'$ can make the transition $X \xrightarrow{\ell} Y$, for $\ell \in \Sigma$, then Y is also a chain. To see this, assume $x, y \in X$, $x \leq y$ and $x \xrightarrow{\ell} x'$, $y \xrightarrow{\ell} y'$. To show that necessarily $y \leq y'$, we consider cases for $\ell \in \Sigma_0$ (i.e. ℓ is a field or method name) and $\ell = \star$. If $\ell \in \Sigma_0$, then this is a simple consequence of the closure rules for constraint graphs. Case $\ell = \star$ falls out immediately from the definition of transitions for $\mathcal{A}$.

Since the language accepted by $\mathcal{A}$ starting from s_0 is the same as that accepted by $\mathcal{A}'$ starting from $\{s_0\}$, and is the same as that starting from the upclosed chain $X(s_0) \stackrel{\text{def}}{=} \{s \mid s_0 \leq s\}$, it suffices for us to consider only those states of $\mathcal{A}'$ that are upclosed chains. Finally, since $\leq$ is a preorder, any upclosed chain Y has, and can be uniquely determined by, a least element $x \in Y$; for example, s_0 is a least element of $X(s_0)$ and s_0 determines the upclosed chain $X(s_0)$. (Note that since $\leq$ is a not a partial order, this least element is not necessarily unique.)

Thus we construct a deterministic automaton $\mathcal{A}_d$ as follows. The states of $\mathcal{A}_d$ are the same as those of $\mathcal{A}$. The transitions of $\mathcal{A}_d$ are defined by: for any state state s_0 of $\mathcal{A}$, if $\mathcal{A}'$ has the following transition

$$X(s_0) \xrightarrow{\ell} Y$$

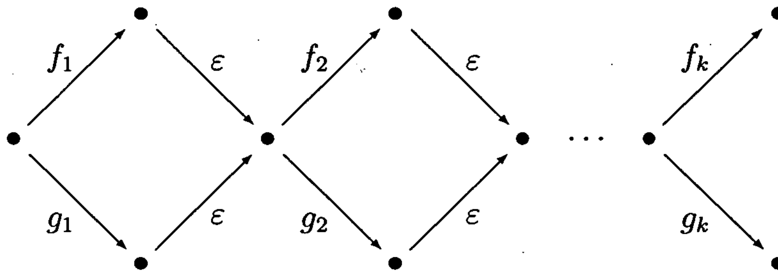
then $\mathcal{A}_d$ has the following transition

$$s_0 \xrightarrow{\ell} x$$

where x is an arbitrary least element of Y.

Assuming $x \leq y$ for states of $\mathcal{A}$ can be determined in constant time (which is indeed the case, if the data structure introduced in Section 4.7 is used), then the transition relation of $\mathcal{A}_d$ can be computed in $O(|\mathcal{A}|^2 \times |E|)$ time.

Note that although we have avoided an exponential blow-up of automaton states, the language accepted by $\mathcal{A}_d$ from any start state can still be exponential. Consider, for example, an automaton with the following transitions.



4.7 Algorithms

We now present efficient algorithms for inference of types for AL. Given a program a, we compute its system of constraints by induction over its syntax. The recipe in Section 4.5.4 allows us to derive a graph from these constraints. We then use an efficient algorithm to compute the closure of this graph, as defined in Definition 5. We construct an automaton from the closure and the well-formedness checking algorithm reuses some data structures used to compute the closure.

4.7.1 Efficient closure

Our definition of closure for constraint graphs is convenient for proving the properties of our algorithm. However, the naive algorithm suggests a breadth-first search for each closure step. We now introduce an alternative definition of closure, which, we show, coincides with the existing definition, but crucially, lends itself more naturally to an iterative algorithm.

Assume we have a constraint graph $G = (N, C, \mathcal{B}; E, \leq)$. As an auxiliary device, we consider a graph $\overline{G}$ which is a constraint graph which in addition has a symmetric relation $\triangle$. We shall graphically represent $x \triangle y$ as

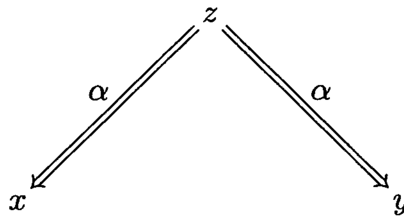
$$x \longleftrightarrow y \ ,$$

emphasising the fact that $\triangle$ is symmetric. We define $\overline{G}$ to be the smallest graph containing G and closed under the rules in Table 4.5.

Using economy as justification for abuse of notation, we write: G to mean the $\leq$ edges of G; G' for those of its closure (with respect to Definition 5); and $\overline{G}$ for the edges $(\leq \cup \triangle)$ of $\overline{G}$.

We have the following lemma about the auxiliary $\triangle$-edges.

Lemma 10 *In $\overline{G}$, we have $x \triangle y$ if and only if there exist α and z such that*



.

The proof can be found in Section A.2.

The new closure rules in Table 4.5 correspond to those in Definition 5 in the following way.

Theorem 11 *For nodes x and y, we have $x \leq y$ in $\overline{G}$ if and only if $x \leq y$ in G'.*

The proof can be found in Section A.2.

For X a set of (x,y) and $[x,y]$ pairs, we define inverse and symmetric closure,

$$X^{-1} \stackrel{\text{def}}{=} \{(x,y) \mid (y,x) \in X\} \cup \{[x,y] \mid [y,x] \in X\}$$
$$X^s \stackrel{\text{def}}{=} X \cup X^{-1} \ .$$

The set X is said to be *reflexive* if for all $x \in G$, both $(x,x) \in X$ and $[x,x] \in X$.

For a set X of (x,y) and $[x,y]$ pairs, where (x,y) represents $x \leq y$ and $[x,y]$ represents $x \bigtriangleup y$, let us write $F(X)$ for the set of pairs after one application of the rules in Table 4.5. More explicitly, we define, for reflexive X,

$$\begin{aligned} F(X) \stackrel{\text{def}}{=} & \quad X \\ \cup & \quad \{[x,y] \mid (x,y) \in X\}^s \\ \cup & \quad \{[x,z] \mid [x,y] \in X, (y,z) \in X\}^s \\ \cup & \quad \{[x',y'] \mid [x,y] \in X, x \xrightarrow{m} x', y \xrightarrow{m} y'\}^s \\ \cup & \quad \{(x',y') \mid (x,y) \in X, x \xrightarrow{m} x', y \xrightarrow{m} y'\} \\ \cup & \quad \{(x',y') \mid [x,y] \in X, x \xrightarrow{f} x', y \xrightarrow{f} y'\}^s \\ \cup & \quad \{(x,z) \mid (x,y) \in X, (y,z) \in X\} \ . \end{aligned}$$

Since $F(X)$ is symmetric with respect to its $[x,y]$ pairs provided X is symmetric with respect to its $[x,y]$ pairs, clearly $\overline{G}$ is the least fix-point of F containing the edges of G. We present and prove correct an algorithm that computes this fix point.

The algorithm uses the following data structures. All complexity measures are with respect to n equal to the number of nodes in G.

PE a graph for storing potential (x,y) edges. This is implemented as a matrix of lists, where entry $\mathbf{PE}[x,y]$ is the list of elements (x',y',ℓ) such that $x \xrightarrow{\ell} x'$ and $y \xrightarrow{\ell} y'$ for ℓ in Σ_0. It is initialised in the preprocessing phase of the algorithm and does not change thereafter.

ITC a graph for storing (x,y) edges. This is implemented as an iterative transitive closure data structure. Note that Palsberg et al. use the same in [Pal95]. Worthy of note is the fact that it maintains an adjacency matrix M such that $M[x,y] = 1$ if and only if there is a path from x to y. Furthermore,

$O(n^2)$ insert operations can be performed in $O(n^3)$ time [Ita86]. To maintain the transitive-closed property of the adjacency matrix, insertion of an edge may result in the addition of many other edges. An insert operation returns a list R of all new edges added.

HG a graph for storing $[x, y]$ edges. This is implemented as an adjacency matrix.

Q a set of (x, y) and $[x, y]$ edges representing the work set: those edges that still need to be considered. This is implemented as a data structure allowing constant-time insert, pick and delete, and membership test.

P a set of (x, y) and $[x, y]$ edges representing the edges that have been considered. This is an auxiliary device that simplifies the correctness argument of the algorithm and has no direct computational effect.

We have the following invariants.

$$\mathbf{P} = \mathbf{HG} + (\mathbf{ITC} - \mathbf{Q}) \tag{4.1}$$

$$\mathbf{P} \cap \mathbf{Q} = \emptyset \tag{4.2}$$

$$F(\mathbf{P}) \subseteq \mathbf{P} \cup \mathbf{Q} \wedge G \subseteq \mathbf{P} \cup \mathbf{Q} \tag{4.3}$$

$$\mathbf{P} \cup \mathbf{Q} \subseteq \overline{G}\ . \tag{4.4}$$

In the preprocessing stage of the algorithm, initialise **ITC**, **HG**, and **P** with the empty set. Initialise **Q** to be the reflexive closure of $\leq$, viz $\{(x, y) \mid x \leq y\} \cup \{(x, x) \mid x \in G\}$. These initialisation steps take $O(n^2)$ time. Initialise **PE** by starting with an empty list for each $\mathbf{PE}[x, y]$ and then iterating over all $x \xrightarrow{\ell} x'$, $y \xrightarrow{\ell'} y'$ in E, inserting (x', y', ℓ) into the list $\mathbf{PE}[x, y]$ whenever and $\ell = \ell'$. Initialisation of **PE** takes $O(|E|^2)$ time. Note that

$$(x', y', \ell) \in \mathbf{PE}[x, y] \iff x \xrightarrow{\ell} x', y \xrightarrow{\ell} y'\ . \tag{4.5}$$

Lemma 11 *After the initialisation step above, the invariant holds.*

Proof We check each of the invariants in turn. (1) $\mathbf{P} = \emptyset = \emptyset + (\emptyset - \mathbf{Q}) = \mathbf{HG} + (\mathbf{ITC} - \mathbf{Q})$. (2) $\mathbf{P} \cap \mathbf{Q} = \emptyset \cap \emptyset = \emptyset$. (3) $F(\mathbf{P}) = F(\emptyset) = \emptyset \subseteq \emptyset \cup \mathbf{Q}$. Also, $G \subseteq \mathbf{Q} \subseteq \mathbf{P} \cup \mathbf{Q}$. (4) $\mathbf{P} \cup \mathbf{Q} = \emptyset \cup Q \subseteq G'$ since G' is reflexive and contains G. □

Once the preprocessing stage has been done, we repeat the following step until the work list **Q** is empty.

Pick and remove some e from **Q**, and add e to **P**. Consider the cases where e is of the form (x, y) and $[x, y]$.

- Case (x, y).

  A1 Insert (x, y) into **ITC**, and let R be the list returned by the insert operation.

  A2 Add $[x, y]$ and $[y, x]$ into **Q**, if they are not in **HG**.

  A3 For each $[x_0, x]$ in **HG**, add $[x_0, y]$ and $[y, x_0]$ to **Q** if they are not in **HG**. Since **HG** is stored as an adjacency matrix, and insertion of a pair into **Q** can be done in constant time, this takes $O(n)$ time.

  A4 For each (x', y', m) in $\mathbf{PE}[x, y]$, add (x', y') to **Q** if it is not in **ITC**.

  A5 Append $R - (x, y)$ to **Q**.

- Case $[x, y]$.

  B1 Add $[x, y]$ to **HG**.

  B2 For each (y, y_0) in **ITC** but not in **Q**, add $[x, y_0]$ and $[y_0, x]$ to **Q** if they are not in **HG**.

  B3 For each (x', y', m) in $\mathbf{PE}[x, y]$, add $[x', y']$ and $[y', x']$ to **Q** if they are not in **HG**.

  B4 For each (x', y', f) in $\mathbf{PE}[x, y]$, add (x', y') and (y', x') to **Q** if they are not in **ITC**.

Informally, at each step of the iteration, we move one edge from **Q** into one of **ITC** or **HG** (A1 or B1), but certainly into **P**. By adding an extra edge into **P**, we might have added more edges to $F(\mathbf{P})$. We find out exactly which edges these are and put them into **Q** to be considered later. In the case where we are adding $[x, y]$ into **HG**, we only need to consider those edges required by the rules htr (B2), hm (B3) and stf (B4). Similarly, in the case where we are adding (x, y)

into **ITC**, we only need to consider those edges required by the rules hst (A2), htr (A3), stm (A4) and stt (A5).

Lemma 12 *The iteration step as defined above preserves the invariants.*

The proof can be found in Section A.2.

The algorithm terminates because of the following observation. Note that invariant 4.2 tells us that **P** and **Q** are disjoint. Note further that at each step of the iteration, we add exactly one edge from **Q** into **P**. Thus we know that **P** is strictly increasing. We also know that $\mathbf{P} \cup \mathbf{Q}$ is bounded above by the set of all (x, y) and $[x, y]$ pairs. Thus **Q** must take value $\emptyset$ eventually.

When the algorithm terminates, we know that $\mathbf{Q} = \emptyset$. From invariant 4.3 we know that $F(\mathbf{P}) \subseteq \mathbf{P}$ and **P** includes G. But F is monotone hence **P** is a fix-point of F including G. Since $\overline{G}$ is the least such fix-point, we conclude that **P** includes $\overline{G}$. Furthermore, from invariant 4.4 we know that $\overline{G}$ includes **P**. Taking these results together, we have proven that $\mathbf{P} = \overline{G}$. Invariant 4.1, states that $\mathbf{P} = \mathbf{ITC} + \mathbf{HG}$, and so we know that **ITC** is the $\leq$ relation of $\overline{G}$.

On the question of time efficiency, we note that **P** strictly increases on each iteration. Since **P** is bounded by $2n^2$, we know that the iteration step is executed at most $2n^2$ times. Insertion of $O(n^2)$ edges into **ITC** can be done in $O(n^3)$ time, since the insertion operation for **ITC** has linear amortised time complexity [Ita86]. We note that R returned by the insertion operation is a list of *new* edges added to **ITC**, and moreover the number of edges in **ITC** is bounded by n^2 so over the whole computation, step A5 appends at most n^2 elements to **Q**, taking $O(n^2)$ time. The matrix **PE** has at most $|E|^2$ elements in its lists, and since step A4 is executed at most once for each (x, y), it therefore inserts at most $|E|^2$ pairs into **Q**. Similarly for steps B3 and B4. The remaining steps in the iteration are clearly bounded by $O(n)$.

The preprocessing stage takes $O(n^2 + |E|^2)$ time but this is subsumed by the iteration part of the algorithm which takes $O(n(n^2 + |E|^2))$ time. Since the constraint graphs that we consider have $|E| \in O(n)$, we conclude that the closure algorithm takes $O(n^3)$ time.

4.7.2 Well-formedness

Recall in Definition 6, a constraint graph is said to be well-formed if: its automaton $\mathcal{A}$ accepts no infinite word; existence of u, u', ℓ, v' such that $u \leq u'$, $u' \xrightarrow{\ell} v'$ implies existence of $u \xrightarrow{\ell} v$; and for each s in G, the language accepted by $\mathcal{A}$ is a pretype.

The first of these three conditions can be checked by checking for cycles in $\mathcal{A}$. This can be done in total running time $O(n^2)$.

The second is the same as the well-formedness condition of Palsberg in [Pal95]. The check can be performed by an algorithm whose running time is $O(n^2)$.

We now consider more closely an algorithm for checking of the third property. Recall in Section 4.3, a pretype is a set of words satisfying properties LANG, ALT, LEAF and TREE. The first two properties are guaranteed by the definition of $\mathcal{A}$. In the case of LEAF, we must check that after the automaton has accepted αd, for $d \in \mathcal{B}$, then it further performs only silent moves. Now, after accepting αd, the only possibility then is to perform an ε-move to some $(1, u)$ (that is, $d \leq u$) and from there accept ℓ to $(0, v)$ (that is, $u \xrightarrow{\ell} v$). We recall from the definition of constraint graph, that the set of labelled edges E is a subset of $C \times \Sigma_0 \times G$, namely only C nodes have outgoing labelled edges. So it certainly suffices to check that no descendent of $d \in \mathcal{B}$ is in C. This is also a necessary condition, for even if $u \in C$ is some descendent of $d \in \mathcal{B}$ and u has no outgoing labelled edges, this graph is already not well-formed since $\mathcal{L}_d$ does not satisfy the TREE property. (Consider $(0, d) \xrightarrow{d} (1, d) \xrightarrow{\varepsilon} (1, u)$ and also $(0, d) \xrightarrow{\varepsilon} (0, u) \xrightarrow{\star} (1, u)$.) For each $d \in \mathcal{B}$, we can find all its descendents in $O(n)$ time by scanning the adjacency matrix in **ITC**, and since $|\mathcal{B}|$ is fixed, we conclude that this check can be performed in $O(n)$ time.

We note that property TREE, is equivalent to: for $x, y \in C \cup \mathcal{B}$, if $x \bigtriangleup y$ then x and y are both in C or both in $\mathcal{B}$ and $x = y$. So for each x in C, we check that there are only y in C such that $x \bigtriangleup y$. Similarly, for each x in $\mathcal{B}$, we check that if $x \bigtriangleup y$ then $x = y$. Since we have already computed the adjacency matrix for $\bigtriangleup$ in the data structure **HG**, the former check has running time $O(n|C|)$ and the latter has running time $O(n|B|)$. Since $|B|, |C| \in O(n)$, we conclude that these

checks have total running time $O(n^2)$.

4.8 Type inference summary

Here is a summary of the type inference algorithm presented in this chapter. Let a be an AL program. In Section 4.4, we construct a constraint system whose solutions are in 1-1 correspondence with typings of a. In Section 4.5, we show that the solutions of this constraint system are precisely the solutions of its constraint graph G, which are precisely the solutions of the closure of G. We provide an efficient algorithm to compute the closure of G in Section 4.7. Using the closure of G, we construct a family of non-deterministic automata $\mathcal{A}$.

In Section 4.6.1, we show that typability of a is equivalent to well-formedness of the closure of G, and, furthermore, in Section 4.7 we provide an efficient algorithm to decide well-formedness from $\mathcal{A}$.

We show that the solutions (more precisely, the presolutions) of the constraint graph can be recognised by $\mathcal{A}$. We provide an efficient algorithm for determinising $\mathcal{A}$ into a family of deterministic finite-state automata $\mathcal{A}_d$ in Section 4.6.4. The language accepted by $\mathcal{A}_d$ can thus be computed using a straightforward recursively-defined function.

4.9 Conclusions and further work

We have presented an algorithm that (1) determines typability of terms in AL and if the answer is positive, (2) constructs an automaton that computes a least shape typing. Along the way, we formulated an alternative definition of type, using prefix-closed languages.

Moreover, we introduced the notion of pretype, which encompass not only types with missing information in its leaves, but also infinite recursive types. Pretypes possess convenient mathematical properties allowing for a cleaner exposition, particularly in our hierarchy of lemmata. Furthermore, they also allow us to define cleanly, the notion of least-shape typing.

Using pretypes, we modified the techniques [Pal95, PWO97] as used by Palsberg et al. and obtained a solution to the type inference problem. Though the solution described in [Pal95] was later shown by Henglein [Hen97] to be too general for the subtype relations in the four AC calculi considered, the merits of the technique shine through when applied to AL. The modifications were not trivial though: our finer-grained subtype relation required a more elaborate definition of constraint graph closure. Nevertheless, it is still possible to compute the closure in asymptotic $O(n^3)$ time. Similarly, a modification of well-formedness, defined on the constraint graph, coincides with typability.

Though type inference for AL was, as far as the author is aware, previously unsolved and therefore the work described here makes a contribution in itself, the original motivation was the search for a verification condition generator (VCG) algorithm for the program logic of AL. In fact, the VCG algorithm uses the type inference algorithm described here as a black box, and is presented in Chapter 5. However, in order to be able to use the VCG tool in a modular fashion, using it to help separately check program components, the black box interface to the type inference algorithm described here is insufficient. Since we always infer a solution of least shape, for any program a (which is not a subprogram of a larger program) of object type, we infer $[\![a]\!] = []$, namely the object type with no fields nor methods, or equivalently the type of all objects. In the context of modular verification, type inference for a large program by separate inference on its components is not possible as the algorithm stands.

Typically, in the case where a is a component of a larger program *let* x=a *in* b, say, we would have preconceptions as to which fields and methods of a we intend to use, i.e. are likely to be used in b. That is, a will have an intended type. And so the question should no longer be "is a typable?", but "is there a type derivation concluding in $\vdash a : A$?". To answer this second question, we simply use exactly the same algorithm but with the extra constraint $[\![a]\!] = A$. At the other end, the question becomes "is b typable assuming x has type A?", which, again, can be answered using our algorithm but with a further constraint $[x] = A$.[4]

[4] As the formalisation stands, this constraint is valid only if A satisfies the restricted grammar displayed in Section 4.4, which precludes, for example, $[f{:}[g{:}A]]$. In such cases, it is necessary

The algorithm presented here assumes that there is no subtyping between the base types. If subtyping between base types is required, then modifications are certainly required in the definition of well-formedness. The details of such a modification have yet to be studied.

---

to introduce further variables, e.g., introduce $A' \stackrel{\text{def}}{=} [g{:}A]$ and define $A \stackrel{\text{def}}{=} [f{:}A']$.

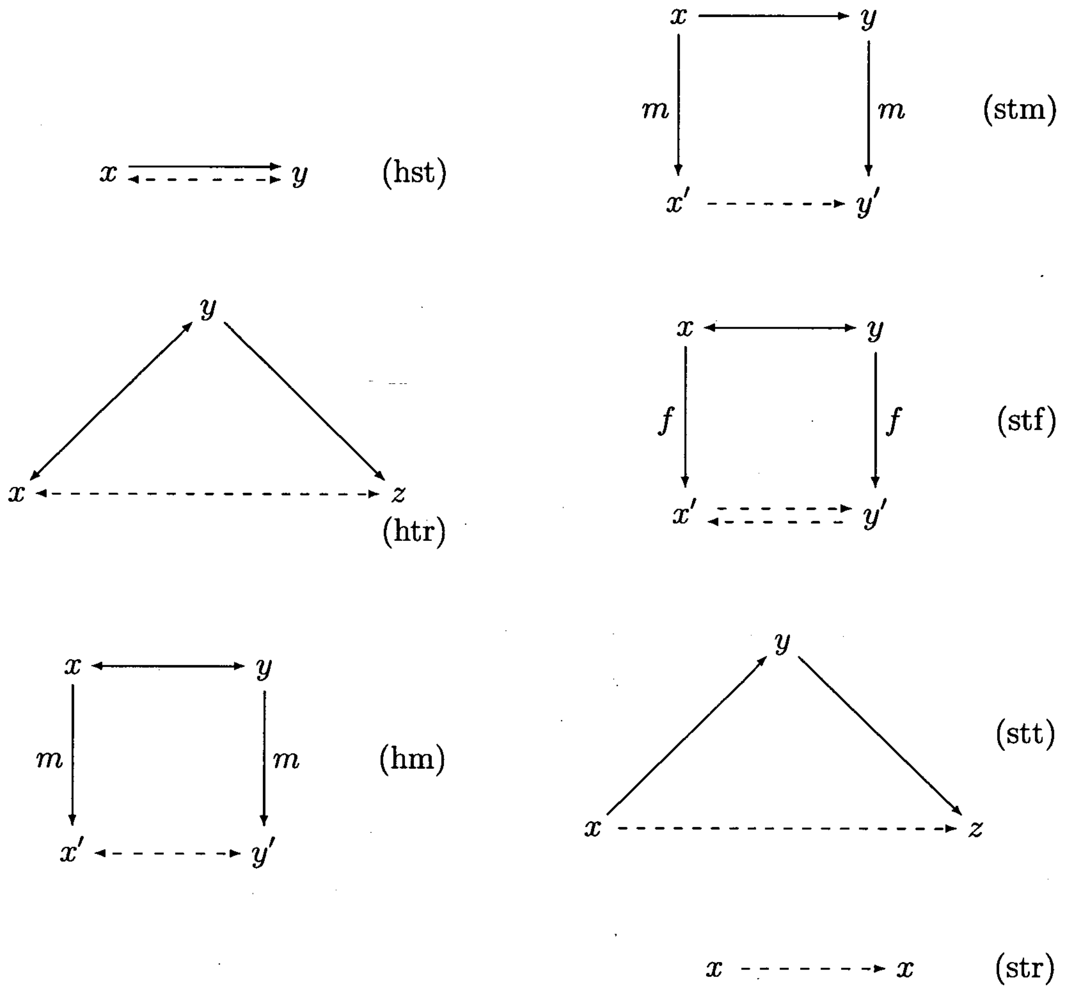


Table 4.5: More closure rules

Chapter 5

Verification Condition Generator

In this chapter, we develop a *verification condition generator* (VCG), which, informally, is an algorithm which assigns to each judgement $J \stackrel{\text{def}}{=} E \vdash a : A :: T$, a logical formula Ψ_0 called a *verification condition* (VC), such that, derivability of J follows from validity of Ψ_0. Thus the VCG reduces the task of finding a proof of J to proving Ψ_0, which one hopes is an easier task. This specification of a VCG can be considered to be its soundness property. Showing only soundness is not enough since we there is always a sound trivial VCG, namely one that always returns an unsatisfiable VC. However, our VCG algorithm also satisfies the converse, namely if Ψ_0 is valid, then J is derivable. Our VCG is thus referred to as being complete.

Finding a proof of J is to build a derivation tree. Firstly, this involves choosing the correct rule at each position, and secondly, instantiating each rule with suitable choices of specifications and transition relations. Here "suitability" not only requires that the hypotheses and conclusions of the instantiated rules "fit" in the proof tree, but also the instantiated side-conditions are verifiable.

The approach we describe can be summarised as follows. (1) We take an equivalent formulation of the verification rules with the crucial property that it is syntax-directed: namely the shape of a proof tree, concluding in $E \vdash a : A :: T$, is uniquely determined by the syntactic shape of a. (2) Thus given any a, we can immediately obtain a *skeleton proof*, which can be considered to be a proof tree with all specifications and transition relations omitted. We *flesh out* this skeleton

by introducing higher-order variables of two sorts: a sort for unknown specifications, and a sort for unknown transition relations. These variables must satisfy the constraints imposed by the rules in the skeleton proof, for example those in the side conditions. (3) From type inference, we know the component transition relations of any specification variable. By rewriting constraints involving specifications with the equivalent constraints on their component transition relations, we can thus eliminate all specification variables in favour of more transition relation variables. Thus we now have unknown transition relations $X_1 \cdots X_n$, say, and a set of constraints Φ. Thus the formula $\Psi \stackrel{\text{def}}{=} \exists \vec{X}.\Phi$ is a VC, which we refer to as the *second-order VC*, in the sense mentioned above, where Φ is understood to be the conjunction of the elements in Φ. (4) We simplify the VC obtained so far, by choosing suitable instantiations for each variable X_i, such that logical validity of the VC is preserved.

We show that, for a second-order VC Ψ, as described, we can instantiate all variables X_i with first-order, fix-point formulae to obtain a fix-point formula Ψ_1, which is free of higher-order quantifiers. Furthermore, by allowing annotations in the program a, we show that by ensuring our programs are suitably annotated, Ψ can in fact be instantiated with fix-point-free formulae, thus giving a VC Ψ_0 which is first-order in the sense that it contains no higher-order variables, nor fix-points.

At this point, one may question, why not approach the VCG problem as in previous work [Gor88, HM94, Hom95], and define a function by recursion over the program syntax, which gives a more local solution, as opposed to our global approach of collecting constraints and solving the resulting constraints system. We point out that in AL, by assuming variables satisfy static specifications, each method body is verified *once* with respect to *one* specification which is used for all its invocations. So, for example, in *let* $x{=}a$ *in* b program b may invoke method m, whose body is defined in a, and vice versa. Since in the recursively-defined VCG approaches, the VC for a method (procedure) invocation is recursively defined using the VC for the method body, we see there is a bidirectional dependency between the VCs for a and b. We also know from [AL98] that only well-typed

programs can be verified. That is, the VCG problem is at least as hard as type inference. A glance at the state-of-the-art [Pal95, Hen97] in type inference for object-calculi, suggests constraint systems.

We note that Ψ, the second-order VC, is no improvement over building a proof tree directly, since to prove Ψ one must find a witness for each existentially quantified variable. However, Ψ_1, the fix-point VC, is already better, since in the case where we have a finite model, fix-point formulae admit polynomial-time model checking algorithms. But, Ψ_0, the first-order VC, is better still since it may be possible to automatically prove such VCs in an automated theorem prover such as SPASS [W$^+$99], provided with cleverly formulated lemmata. On the other hand, extra work is required to provide annotations in the program.

5.1 Related work

The VCG approach described in this chapter depends on the type inference algorithm presented in Chapter 4. However, the algorithm is used only as a black box, the details of the algorithm are not important.

In [HM94], Homeier and Martin formally verify a VCG, for a simple while-language, notably without procedures. Subsequently, in Homeier's PhD dissertation [Hom95], procedures are a feature of the underlying language. However, the VCG considers only *well-formed* programs, the definition of which, in particular, bans the possibility of aliased variables (Sec. 10.4.1 in loc. cit.). In contrast, correctness of our algorithm is not machine checked and our OO-language embraces both aliasing and a higher-order store.

Compaq Research's ESC project [DLNS98] also uses VCG technology. However, they compile the source language (ESC/Java and, earlier, ESC/Modula-3) into the language of guarded commands before generating the VCs. These VCs are passed to an automatic theorem prover which attempts to prove them without further interaction from the user. The motivational lineage of ESC is pragmatic. Consequently, the tool aims to catch *more* errors (than type-checking alone), but not *all* errors. Thus the tool can afford to be unsound; i.e. a program passed by

the tool need not necessarily satisfy its specification. Our design goals are different in the following respects: soundness is not considered "harmful", but indeed, its availability considered crucial; also, the user might want to work directly with the resulting VCs, e.g. using an interactive theorem prover, and therefore it is important that they correspond in a direct way to the original program; in the same way, a C++ programmer does not expect to debug assembly code generated by the compiler.

The LOOP project [vdBJ01, JP01] of Nijmegen, is superficially similar to the work described here: the LOOP tool takes as input, annotated Java code (JML), compiling it to theorem prover theories (currently PVS and Isabelle), which contain theorems to be proved by the programmer. As our approach, soundness is maintained, and furthermore, the resulting proof obligations can be discharged using a program logic [JP01] defined directly on the source language itself. In some ways, LOOP is more ambitious since it considers many features of Java not present in AL, including exceptions. However, we suggest that our VCG automatically discharges more proof obligations, or equivalently, our VCs are "easier" to prove than the theorems from LOOP. Specifically, in our case, invariants can be specified in the input through annotations, and no longer need to be provided when proving the VCs. In contrast, at present, though loop invariants can be specified in the JML input, the LOOP tool does not propagate this information, and thus they must be provided again whilst interacting with the theorem prover.

The Proof Carrying Code (PCC) project [NL96, Nec97] of Necula and Lee applied many similar ideas to verification of programs in a fragment of DEC Alpha assembly language. A code consumer publishes a specification for a program called the *safety policy.* Given a program and this safety policy, a Floyd-style VCG is used to generate a *safety predicate.* Satisfiability of this safety predicate entails correctness of the program with respect to the safety policy. The code producer ships the code together with a proof of the safety policy; the code consumer regenerates the safety predicate and checks that it is proved by the provided "proof", and if so, the consumer executes the code which runs without

$$\frac{}{Bool <: Bool}$$

$$\frac{B_1 <: B'_1 \quad \cdots \quad B_{\ell'} <: B'_{\ell'} \quad U_1 \subseteq U'_1 \quad \cdots \quad U_{\ell'} \subseteq U'_{\ell'}}{[f_i{:}A_i{}^{i=1..k}, m_j{:}\varsigma(y_j)B_j{::}U_j{}^{j=1..\ell}] <: [f_i{:}A_i{}^{i=1..k'}, m_j{:}\varsigma(y_j)B'_j{::}U'^{j=1..\ell'}_j]} \quad \{k' \leq k,\ \ell' \leq \ell\}$$

Table 5.1: Subspecification relation. The subspecification relation together with the subsumption rule allows us to use a program of finer specification where one of coarser specification is expected. It is a natural extension of the subtype relation to specifications. We note that the subspecification relation is defined covariantly along methods.

any performance penalties.

In the following sense, the work on PCC is similar to that presented in this thesis. The specification language as presented in loc. cit. can express safety of memory accesses. More explicitly, the specification logic is first order logic with constants allowing it describe changes to contents of memory locations, which is very similar to the language of transition relations in AL. Also, the VCG insists annotation of loops with invariants. The annotations break a program with cycles into several smaller cycle-free programs.

The PCC technology works with assembly language. A consequence of this is that it works with code that is directly executed; there is no need to trust an interpretor or compiler; at most, we need to trust an assembler (and processor hardware). However, it is also well-known that assembly language is not ideal for many problems. This thesis considers a much richer language with objects, including its consequences such as higher-order store and dynamic-bound recursion. For example, in AL, it is not so clear exactly where annotations must be provided since the loops cannot be accurately and easily determined statically. Also, in AL, our specification language is more expressive, since we have specifications as well as transition relations. This extra expressiveness maintains modularity in the presence of objects.

5.2 Second-order verification conditions

The logic rules as presented by Abadi and Leino in [AL98] include a *subsumption* rule of the form

$$\frac{E \vdash a : A :: T}{E \vdash a : A' :: T'} \quad \{A <: A' \quad T \subseteq T'\}$$

Informally, it means that a program satisfying a finer specification can be used where one satisfying a coarser specification is required. Since its conclusion can match a program of any syntactic shape, it can be used anywhere within a proof. That is, proofs do not have a uniquely determined shape. However, if we incorporate subsumption into every rule, see Table 5.2 on page 178, then proofs do have uniquely determined shapes. The downside is that we now have subsumption constraints in the side conditions of every rule. In practice, this means that verifying proofs by hand becomes more laborious, but, with computer assistants, we view this as an acceptable compromise.

Let us recall some notation which we use to define the constraint system induced by a program. For typing judgement

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A \ ,$$

we write $[\![a]\!]$ for A, and $[x_i]$ for A_i. The reader is referred to Section 4.4 where this notation is introduced in more detail. Note, however, that in this chapter, $[\![a]\!]$ and $[x_i]$ denote the inferred types rather than type variables. For γ a sequence of field names and method names, and A a type, we overload notation and write $\gamma \in A$ to mean that γ is a path in the type A.[1] We implicitly assume that the symbol γ (and its decorated variants) ranges over sequences ending in a method name. Similarly, α ranges over non-empty sequences of method names, and β ranges over sequences ending in a method name but containing at least one field name. We write $\#\gamma$ for the number of method names in γ.

A variable renaming is an injective, order-preserving function $\sigma : \{1..m\} \rightarrow \{1..n\}$ between initial segments of $\mathbb{N}$. Let $\mathsf{dom}(\sigma)$ (domain) denote $\{1..m\}$ and

[1] We note that, in Chapter 4, by the definition of pretype (Definition 4), a type is a set of words whose letters, in particular, alternate between field and method names, and decorations. In this Chapter, when we write $\gamma \in A$ for A a type, we implicitly omit the decorations.

$\mathsf{cod}(\sigma)$ (codomain) denote $\{1..n\}$. Its application to a transition relation $T = T(x_1, \ldots, x_m)$ of arity m is written σT and is defined pointwise by

$$(\sigma T)(x_1, \ldots, x_n) \stackrel{\text{def}}{=} T(x_{\sigma(1)}, \ldots, x_{\sigma(m)}) \ .$$

Recall from the discussion in the overview: we take the inferred types as skeleton specifications, and flesh them out by introducing variables for all unknown specifications and transition relations; and then we eliminate specification variables in favour of transition relation variables. This is possible since, writing $A(\gamma)$ for the component transition relation of specification A along γ, by definition of subspecification in locus classicus and reproduced in Table 5.1 on page 137,

$$A <: A'$$

iff

$$\begin{aligned} A(\alpha) &\subseteq B(\alpha) && \text{for all } \alpha \in B \\ A(\beta) &= B(\beta) && \text{for all } \beta \in B \ . \end{aligned}$$

And since, for any subterm occurrence a, there is one transition relation for each $\gamma \in [\![a]\!]$, and we know $[\![a]\!]$ from type inference, we can introduce variables for the component transition variables of all specification variables, which can now be eliminated by appealing to the previous equivalence.

Thus for each judgement $E \vdash a : A :: T$ with unknown A and T, we introduce a variable $(\!|a|\!)$ for T and for each $\gamma \in [\![a]\!]$, a variable $(\!|a|\!)_\gamma$ for the component transition relations of A. (There is one exception to this: for subterm occurrence x in method invocation $x.m()$, we do not introduce variables $(\!|x|\!)$ nor $(\!|x|\!)_\gamma$; this variable can be and is more conveniently eliminated before we generate the constraints. This is further elaborated on later.)

The arity of $(\!|a|\!)$, written $\mathrm{ar}((\!|a|\!))$ is n, for

$$x_1{:}[x_1], \ldots, x_n{:}[x_n] \vdash a : [\![a]\!]$$

the derived typing judgement of a. Now consider $(\!|[m{=}\varsigma(y)b]|\!)_m$, which one can see, by studying the rule for object creation in Table 5.2 on page 178, is the

transition relation for method m, whose body is b. Since y occurs free in b, its arity is one more than that of $(\![m{=}\varsigma(y)b]\!)$. By generalising, the arity of $(\!a\!)_\gamma$ is, thus, $\mathrm{ar}((\!a\!)) + \#\gamma$.

For each variable x and $\gamma \in [x]$, we introduce a second-order variable $\langle x \rangle_\gamma$ to denote the component transition relation of the specification of x that occurs in the verification contexts. If x_j occurs in the context of verification judgement

$$x_1{:}A_1, \dots, x_j{:}A_j, \dots, x_n{:}A_n \vdash a : A :: T$$

then the arity of $\langle x_j \rangle_\gamma$ is $j - 1 + \#\gamma$. This is well-defined since a program variable must occur in the same position and have the same specification in all contexts; we cannot permute variables in contexts.

Table 5.3 on page 179 defines some useful renaming functions. The renaming $w_p^{q,\gamma}$ is a weakening/renaming used for variable instances. The idea is to map the first p variables into the first q slots, and then map the remaining $\#\gamma$ variables (which are introduced by the ς-binders) to the $\#\gamma$ slots after the first q slots. Similarly, the renaming $\sigma_p^{q,\gamma}$ is also a weakening, and differs from $w_p^{q,\gamma}$ only in the use of its formal parameters. It is used to shuffle the variables into the correct slots for method invocation.

We now construct a function Cons which, given a list $\vec{x}$ of free variables and a program a (whose free variables come from $\vec{x}$), computes a set of constraints on transition relation variables. In particular, for a closed program a, and ε the empty list, $\mathrm{Cons}(\varepsilon, a)$ is precisely the set of all constraints imposed by the side conditions of the skeleton proof.

For example, supposing the variables in E are $x_1, \dots, x_n$, consider the rule for variable instances.

$$\frac{}{E \vdash x_j : A :: T} \quad \left\{ \begin{array}{l} E(x_j) <: A \\ Res(x_j) \subseteq T \end{array} \right\}$$

The second of the two side conditions is precisely

$$Res(x_j) \subseteq (\!x_j\!) \ ,$$

for some suitable formalisation of $Res(x_j)$. Note that there is an implicit weakening in the first of the two side conditions. Recalling that the component transition

relations of A are $(\!|x_j|\!)_\gamma$ for all γ in $[\![x_j]\!]$ and similarly those of $E(x_j)$ are $\langle x_j\rangle_\gamma$ for all γ in $[x_j]$, we can use the earlier equivalence and write the constraint as

$$\begin{aligned} w^{n,\alpha}_{j-1}\langle x_j\rangle_\alpha &\subseteq (\!|x_j|\!)_\alpha \ , &&\text{for all } \alpha \text{ in } [\![x_j]\!], \\ w^{n,\beta}_{j-1}\langle x_j\rangle_\beta &= (\!|x_j|\!)_\beta \ , &&\text{for all } \beta \text{ in } [\![x_j]\!]. \end{aligned}$$

The variable $\langle x_j\rangle_\gamma$ has $j-1+\#\gamma$ variables, of which the last $\#\gamma$ variables are ς-bound variables and similarly the variable $(\!|x_j|\!)_\gamma$ has $n+\#\gamma$ variables, again the last $\#\gamma$ of which are the ς-bound variables. The implicit weakening allows the first $j-1$ variables of $\langle x_j\rangle_\gamma$ to be among the first n variables of $(\!|x_j|\!)_\gamma$. The remaining ς-bound variables must be considered separately. So the explicit weakening injects the first $j-1$ variables of $\langle x_j\rangle_\gamma$ into the first n variables; and translates the remaining $\#\gamma$ ς-bound variables. We thus have the following clause:

$$\begin{aligned} \mathrm{Cons}(\vec{x}, x_j) \stackrel{\text{def}}{=} \quad & \{w^{n,\alpha}_{j-1}\langle x_j\rangle_\alpha \subseteq (\!|x_j|\!)_\alpha \mid \alpha \in [\![x_j]\!]\} \\ & \cup \{w^{n,\beta}_{j-1}\langle x_j\rangle_\beta = (\!|x_j|\!)_\beta \mid \beta \in [\![x_j]\!]\} \\ & \cup \{Res(x_j) \subseteq (\!|x_j|\!)\} \end{aligned}$$

The case for field lookup is similar.

$$\begin{aligned} \mathrm{Cons}(\vec{x}, x_j.f) \stackrel{\text{def}}{=} \quad & \{w^{n,f\gamma}_{j-1}\langle x_j\rangle_{f\gamma} = (\!|x_j|\!)_{f\gamma} \mid f\gamma \in [\![x_j]\!]\} \\ & \cup \{(\!|x_j|\!)_{f\alpha} \subseteq (\!|x_j.f|\!)_\alpha \mid \alpha \in [\![x_j.f]\!]\} \\ & \cup \{(\!|x_j|\!)_{f\beta} = (\!|x_j.f|\!)_\beta \mid \beta \in [\![x_j.f]\!]\} \\ & \cup \{T_{\text{fsel}}(x_j, f) \subseteq (\!|x_j.f|\!)\} \end{aligned}$$

For field update, we have the rule

$$\frac{E \vdash x_j : A :: Res(x_j) \quad E \vdash x_k : A'' :: Res(x_k)}{E \vdash x_j.f{:=}x_k : A' :: T'} \ ,$$

and side conditions

$$A <: [f{:}A''], \quad A <: A', \quad T_{\text{fupd}}(x_j, f, x_k) \subseteq T' \ .$$

Noting that the transition relation components of A are $(\!|x_j|\!)_\gamma$ for γ in $[\![x_j]\!]$, and those of A'' are $(\!|x_k|\!)_\gamma$ for γ in $[\![x_k]\!]$, we expand the definition of subspecification

using the earlier equivalence. (Note also that the variables $(\!|x_j|\!)$ and $(\!|x_k|\!)$ are *internal* in the sense that they cannot be further constrained, and so in theory we could eliminate them from the constraint system altogether.)

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, x_j.f{:=}x_k) \stackrel{\text{def}}{=} \quad & \mathrm{Cons}(\vec{x}, x_j) \\
& \cup \{(\!|x_j|\!) = \mathit{Res}(x_j)\} \\
& \cup \mathrm{Cons}(\vec{x}, x_k) \\
& \cup \{(\!|x_k|\!) = \mathit{Res}(x_k)\} \\
& \cup \{(\!|x_j|\!)_{f\gamma} = (\!|x_k|\!)_\gamma \mid \gamma \in [\![x_k]\!]\} \\
& \cup \{(\!|x_j|\!)_\alpha \subseteq (\!|x_j.f{:=}x_k|\!)_\alpha \mid \alpha \in [\![x_j.f{:=}x_k]\!]\} \\
& \cup \{(\!|x_j|\!)_\beta = (\!|x_j.f{:=}x_k|\!)_\beta \mid \beta \in [\![x_j.f{:=}x_k]\!] \\
& \cup \{T_{\mathrm{fupd}}(x_j, f, x_k) \subseteq (\!|x_j.f{:=}x_k|\!)\}
\end{aligned}$$

When we consider the case for method invocation

$$\frac{E \vdash x : [m{:}\varsigma(y)B{::}U] :: \mathit{Res}(x)}{E \vdash x.m() : A :: T}$$

it is slightly worrying because the side conditions

$$B[x/y] <: A, \quad U[x/y] \subseteq T$$

require us to rename y to x, which at first appears to require a non-injective function. (Remember renamings are, by definition, injective.) Suppose the variables in E are $x_1, \ldots, x_n$. Supposing $x_j \equiv x$, writing x_{n+1} for y, and expanding the rule for variable x, we have the rule

$$\frac{}{E \vdash x_j.m() : A :: T}$$

with side conditions

$$\begin{aligned}
& E(x_j) <: [m{:}\varsigma(x_{n+1})B{::}U] \\
& B[x_j/x_{n+1}] <: A \\
& U[x_j/x_{n+1}] \subseteq T
\end{aligned}$$

which, once we expand the definition of the subspecification relation and use the fact that $(\!|x_j|\!)_\gamma$ are the component transition relations of $[m{:}\varsigma(x_{n+1})B{::}U]$, gives

$$\begin{array}{ll}
w^{n,m\alpha}_{j-1}\langle x_j\rangle_{m\alpha} \subseteq (\!|x_j|\!)_{m\alpha}\ , & \text{for } m\alpha \text{ in } [\![x_j]\!] \\
w^{n,m\beta}_{j-1}\langle x_j\rangle_{m\beta} = (\!|x_j|\!)_{m\beta}\ , & \text{for } m\beta \text{ in } [\![x_j]\!] \\
\sigma^{\alpha}(\!|x_j|\!)_{m\alpha} \subseteq (\!|x_j.m()|\!)_{\alpha}\ , & \text{for } \alpha \text{ in } [\![x_j.m()]\!] \\
\sigma^{\beta}(\!|x_j|\!)_{m\beta} \subseteq (\!|x_j.m()|\!)_{\beta}\ , & \text{for } \beta \text{ in } [\![x_j.m()]\!] \\
\sigma^{\varepsilon}(\!|x_j|\!)_{m} \subseteq (\!|x_j.m()|\!) &
\end{array}$$

where σ^γ is the non-injective renaming defined pointwise

$$\sigma^\gamma : \{1..(n+1+\#\gamma)\} \to \{1..(n+\#\gamma)\},$$

$$\sigma^\gamma(i) \stackrel{\text{def}}{=} \begin{cases} i, & \text{for } i < n+1 \\ j, & \text{for } i = n+1 \\ i-1, & \text{for } i > n+1. \end{cases}$$

The arity of $(\!|x_j|\!)_{m\gamma}$ is $n+\#m\gamma$, and the arity of $(\!|x_j.m()|\!)_\gamma$ is $n+\#\gamma$. We require to line up the remaining $\#\gamma$ variables of these two variables, and also identify x_j and x_{n+1}. This is precisely what the renamings σ^γ do.

Note that the composition $\sigma^\gamma \circ w^{q,\gamma}_p = \sigma^{q,\gamma}_p$, which is defined in Table 5.3 on page 179. Since the nodes of the form $(\!|x_j|\!)_\gamma$ are *internal* in the sense that they cannot be further constrained, we can eliminate them immediately. This is why $(\!|x_j|\!)_\gamma$ and $(\!|x_j|\!)$ are omitted from the set of constraint variables.

Our initial concern has now been addressed. Though the functions σ^γ are non-injective, they only appear composed with weakenings of the form $w^{q,n}_p$ to give renamings of the form $\sigma^{q,n}_p$ which are injective (and order-preserving). Thus we have the following clause:

$$\begin{array}{rl}
\text{Cons}(\vec{x}, x_j.m()) \stackrel{\text{def}}{=} & \{\sigma^{n,\alpha}_{j-1}\langle x_j\rangle_{m\alpha} \subseteq (\!|x_j.m()|\!)_\alpha \mid \alpha \in [\![x_j.m()]\!]\} \\
& \cup\, \{\sigma^{n,\beta}_{j-1}\langle x_j\rangle_{m\beta} = (\!|x_j.m()|\!)_\beta \mid \beta \in [\![x_j.m()]\!]\} \\
& \cup\, \{\sigma^{n}_{j-1}\langle x_j\rangle_{m} \subseteq (\!|x_j.m()|\!)\}
\end{array}$$

To justify the definition of $\text{Cons}(\vec{x}, \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1)$ we use the fact that for transition relations ϕ and ϕ', and term t, we have the equivalence

$$\phi[t/x] \subseteq \phi'[t/x] \quad \text{iff} \quad (x = t \wedge \phi) \subseteq \phi'\ .$$

Thus we have

$$\phi_0[\texttt{tt}/x] \subseteq \phi'[\texttt{tt}/x] \land \phi_1[\texttt{ff}/x] \subseteq \phi'[\texttt{ff}/x]$$

iff

$$((x = \texttt{tt} \land \phi_0) \lor (x = \texttt{ff} \land \phi_1)) \subseteq \phi' \ . \tag{5.1}$$

Thus we introduce new syntax $e_0 \lhd x \rhd e_1$, read "e_0 if x else e_1," and define semantically

$$(e_0 \lhd x \rhd e_1)(\vec{x}) \stackrel{\text{def}}{=} (x = \texttt{tt} \land e_0(\vec{x})) \lor (x = \texttt{ff} \land e_1(\vec{x})) \ .$$

Using the previous equivalence and the fact that $(x = \texttt{ff}) \lor (x = \texttt{tt})$ for any boolean variable x, we derive the following clause:

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, \mathit{if}\ x_j\ \mathit{then}\ a_0\ \mathit{else}\ a_1) \stackrel{\text{def}}{=}\ & \{\llparenthesis x_j \rrparenthesis = \mathit{Res}(x_j)\} \\
& \cup \mathrm{Cons}(\vec{x}, a_0) \cup \mathrm{Cons}(\vec{x}, a_1) \\
& \cup \{\llparenthesis a_0 \rrparenthesis_\alpha \lhd x \rhd \llparenthesis a_1 \rrparenthesis_\alpha \subseteq \llparenthesis a_{\text{if}} \rrparenthesis_\alpha \mid \alpha \in \llbracket a_{\text{if}} \rrbracket\} \\
& \cup \{\llparenthesis a_0 \rrparenthesis_\beta \lhd x \rhd \llparenthesis a_1 \rrparenthesis_\beta = \llparenthesis a_{\text{if}} \rrparenthesis_\beta \mid \beta \in \llbracket a_{\text{if}} \rrbracket\} \\
& \cup \{\llparenthesis a_0 \rrparenthesis \lhd x \rhd \llparenthesis a_1 \rrparenthesis \subseteq \llparenthesis a_{\text{if}} \rrparenthesis\}
\end{aligned}$$

In the constraints for the let case, we see another variant of the sequential composition syntax. We now write it simply as $T; U$ for transition relation expressions T and U, provided the arity of U is one more than that of T. The composition $T; U$ is defined implicitly to be "along the last variable slot of U". More precisely, we have semantically

$$(T; U)(\vec{x}) = \exists x, \breve{\sigma}. T(\vec{x})[\breve{\sigma}, x/\acute{\sigma}, r] \land U(\vec{x}x)[\breve{\sigma}/\grave{\sigma}] \ .$$

Note also that the variables $\langle x \rangle_\gamma$ are identified with $\llparenthesis a \rrparenthesis_\gamma$ as implicitly required in the let rule in Table 5.2 on page 178. Thus we have the clause:

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, \mathit{let}\ x{=}a\ \mathit{in}\ b) \stackrel{\text{def}}{=}\ & \mathrm{Cons}(\vec{x}, a) \cup \mathrm{Cons}(\vec{x}x, b) \\
& \cup \{\llparenthesis a \rrparenthesis_\gamma = \langle x \rangle_\gamma \mid \gamma \in \llbracket a \rrbracket\} \\
& \cup \{\llparenthesis b \rrparenthesis_\alpha \subseteq w_n^{n+1,\alpha} \llparenthesis a_{\text{let}} \rrparenthesis_\alpha \mid \alpha \in \llbracket a_{\text{let}} \rrbracket\} \\
& \cup \{\llparenthesis b \rrparenthesis_\beta = w_n^{n+1,\beta} \llparenthesis a_{\text{let}} \rrparenthesis_\beta \mid \beta \in \llbracket a_{\text{let}} \rrbracket\} \\
& \cup \{\llparenthesis a \rrparenthesis; \llparenthesis b \rrparenthesis \subseteq \llparenthesis a_{\text{let}} \rrparenthesis\}
\end{aligned}$$

Repeating the same exercise again, we obtain the following clause for object creation, writing $a_{\mathsf{obj}} \equiv [f_i = z_i{}^{i=1..k}, m_j = \varsigma(y_j)b_j{}^{j=1..\ell}]$:

$$\begin{aligned}
\mathrm{Cons}(\vec{x}, a_{\mathsf{obj}}) \stackrel{\mathrm{def}}{=} \quad & \bigcup_{i=1..k} \mathrm{Cons}(\vec{x}, z_i) \cup \bigcup_{j=1..\ell} \mathrm{Cons}(\vec{x}y_j, b_j) \\
& \cup \{ (\!| z_i |\!)_\gamma = \langle y_j \rangle_{f_i\gamma} \mid i = 1..k,\ j = 1..\ell,\ f_i\gamma \in [y_j] \} \\
& \cup \{ (\!| b_j |\!)_\gamma = \langle y_{j'} \rangle_{m_j\gamma} \mid j, j' = 1..\ell,\ m_j\gamma \in [y_j] \} \\
& \cup \{ (\!| b_j |\!) = \langle y_{j'} \rangle_{m_j} \mid j, j' = 1..\ell,\ m_j \in [y_j] \} \\
& \cup \{ (\!| z_i |\!)_\gamma = (\!| a_{\mathsf{obj}} |\!)_{f_i\gamma} \mid i = 1..k,\ f_i\gamma \in [\![a_{\mathsf{obj}}]\!] \} \\
& \cup \{ (\!| b_j |\!)_\alpha \subseteq (\!| a_{\mathsf{obj}} |\!)_{m_j\alpha} \mid j = 1..\ell,\ m_j\alpha \in [\![a_{\mathsf{obj}}]\!] \} \\
& \cup \{ (\!| b_j |\!)_\beta = (\!| a_{\mathsf{obj}} |\!)_{m_j\beta} \mid j = 1..\ell,\ m_j\beta \in [\![a_{\mathsf{obj}}]\!] \} \\
& \cup \{ (\!| b_j |\!) \subseteq (\!| a_{\mathsf{obj}} |\!)_{m_j} \mid j = 1..\ell,\ m_j \in [\![a_{\mathsf{obj}}]\!] \} \\
& \cup \{ T_{\mathsf{obj}}(z_1 \cdots z_k) \subseteq (\!| a_{\mathsf{obj}} |\!) \}
\end{aligned}$$

Thus the expression $\mathrm{Cons}(\varepsilon, a)$ computes precisely the side conditions of our skeleton proof. Supposing $\mathrm{Cons}(\vec{x}, a)$ is a set of constraints over variables $\vec{X}$, we will consider the second-order formula $\exists\vec{X}.\mathrm{Cons}(\vec{x}, a)$ where there are implicit conjunctions between the elements of $\mathrm{Cons}(\vec{x}, a)$. In fact, this second-order formula is a verification condition as we shall show in the following section.

5.3 Soundness and completeness

A useful factoring of the statement of completeness requires the following theorem.

Theorem 12 *Given a proof derivation of $E \vdash a : A :: T$ and a least shape typing t mapping subterm occurrences and variables to types, there is a proof derivation of $E'' \vdash a : A'' :: T$ such that for subterm a' of a, if $E' \vdash a' : A' :: T'$ occurs in the latter proof then A' has the same shape as $t(a')$. In particular, A'' has the same shape as $t(a)$, and also, for all variables x, if $E''(x)$ is defined then it has the same shape as $t(x)$, the type of x in the contexts.*

To paraphrase: if there is a proof derivation of some verification statement about a, then there is proof whose shape corresponds closely to a least-shape type

derivation of a.

The theorem can be proved by presenting an algorithm that takes an arbitrary proof of $E \vdash a : A :: T$ and constructs a proof with the desired property. The specifications in the constructed proof are precisely the types in the type derivation fleshed-out with the corresponding transition relations from the input proof. Since the specifications occurring in the input proof may be syntactically bigger (i.e. have more fields and methods) than those in the output proof, typically some transition relations are omitted from the input proof. To prove that this constructed "proof" really is a proof, we proceed by induction over the input proof and use the fact that the typing t is of least shape together with the following lemma, which is stated without proof.

Lemma 13 *For specifications A and A' from the input proof, let $\overline{A}$ and $\overline{A'}$ be the corresponding specifications in the constructed "proof". If $A <: A'$ then $\overline{A} <: \overline{A'}$.*

This lemma is true because of the close relationship between subspecification and subtype and also because $\overline{A'}$ is of lesser shape than A'.

Given Theorem 12, we can now state completeness with a slightly stronger assumption as follows.

Theorem 13 (Completeness) *Given a program a, specification A and transition relation T. If $\vdash a : A :: T$ has a proof whose component specifications have the same shape as their corresponding types in the inferred least shape typing, then*

$$Cons(\varepsilon, a) \cup \{(\!|a|\!)_\gamma = A(\gamma) \mid \gamma \in A\} \cup \{(\!|a|\!) = T\} \tag{5.2}$$

has a solution.

Intuitively, the proof is obvious: if there is a proof derivation of the required shape, then clearly the concrete transition relations therein satisfy the required constraints, and so they give us witnesses proving our second-order VC.

Note that we write $A(\gamma)$ for the component transition relation of specification A along path γ. The proof proceeds by induction, but in order to do so, we must strengthen the statement to that of the following lemma.

Lemma 14 *For program a, provided specification A has the same shape as the type $[\![a]\!]$ and each A_i has the same shape as $[x_i]$,*

$$x_1{:}A_1, \ldots, x_n{:}A_n \vdash a : A :: T$$

implies

$$\begin{array}{ll} Cons(\vec{x}, a) & \\ \langle x_i \rangle_\gamma = A_i(\gamma), & \text{for } i = 1..n \text{ and } \gamma \in [x_i] \\ (\!|a|\!)_\gamma = A(\gamma), & \text{for } \gamma \in [\![a]\!] \\ (\!|a|\!) = T & \end{array}$$

has a solution, where we write $\vec{x}$ for the sequence $x_1 \cdots x_n$.

Finally, we also have the following soundness theorem which is straightforward since Cons computes the set of all side conditions in a skeleton proof.

Theorem 14 (Soundness) *Given a, A, T, if*

$$Cons(\varepsilon, a) \cup \{(\!|a|\!)_\gamma = A(\gamma) \mid \gamma \in A\} \cup \{(\!|a|\!) = T\}$$

has solution, then $\vdash a : A :: T$ is provable.

5.4 First-order, fix-point verification conditions

We know from the previous theorem, that $\text{Cons}(\varepsilon, a)$ is a VC, albeit expressed as a second-order formula. However, this VC offers us no advantage over finding a proof directly, since to prove this formula, one must find witnesses for the existentially quantified transition relations. Fortunately, we can simplify this VC by automatically finding instantiations that preserve logical equivalence. We now present rules that find these instantiations and reduce this formula into a logically equivalent, first-order fix-point formula.

In the following sections, we observe the following notational disciplines.

- We assume that Ψ_0 ranges over first-order, fix-point formulae.

- We assume that v (and its decorated variants) range over *boolean assumptions* which are lists of pairs $x{=}b$ where $b \in \{\mathtt{tt}, \mathtt{ff}\}$.

- We assume that s (and its decorated variants) range over *switchings* which are functions mapping boolean variables (i.e. program variables of type boolean, as determined by type inference) to $\mathtt{tt}, \mathtt{ff}$. These are the semantic counterparts to boolean assumptions and we write $s \models v$ precisely when for each $x{=}b \in v$, we have $s(x) = b$.

- We assume that e (and its decorated variants) range over transition relation expressions built-up from the following syntax

$$e \quad ::= \quad \phi \mid X \mid e_0 \lhd x \rhd e_1 \mid \sigma e \ ,$$

  where ϕ are first-order transition relation expressions, X transition relation variables, σ ranges over (injective) renamings (of first-order variables occurring in transition relations).

- Similarly, we assume that L (and its decorated variants) range over transition relation expression built-up from the following more elaborate syntax

$$L \quad ::= \quad X \mid L_0 \lhd x \rhd L_1 \mid \sigma L \mid L_0 ; L_1 \mid \exists_\sigma L \ ,$$

  where $\exists_\sigma -$ is the left adjoint of the renaming $\sigma -$.

- Finally, we assume that T (and its decorated variants) range over transition relation expressions built-up from the following, yet more elaborate, syntax.

$$T \quad ::= \quad v, L \mid \exists_\sigma T \mid T_0 \vee T_1 \mid \mu X.T \ .$$

Semantically, $F : X \mapsto L$ (and also $X \mapsto T$) is said to be monotone when $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

Table 5.6 on page 181 shows a schematic diagram of our simplification algorithm, introducing the syntactic forms of the VCs we will be considering. A constraint of the form $e_0 = e_1$ is understood to mean both $e_0 \subseteq e_1$ and $e_1 \subseteq e_0$.

We interpret constraints of the form $v \Rightarrow e_0 = e_1$ to mean $e_0 = e_1$ assuming that $x = b$ for each $x{=}b$ in v. Similarly, $v, L \subseteq e$ means that $L \subseteq e$ assuming v.

We start at the top with our second-order verification condition, a formula of the form

$$\exists \vec{X}. \quad \bigwedge_{e,e'} e = e' \quad \wedge \quad \bigwedge_{i} L_i \subseteq X_i \ ,$$

We simplify this by eliminating constraints of the form $e_0 = e_1$, for e_0, e_1 dependent on $\vec{X}$ (i.e. constraints $\phi_0 = \phi_1$ can be left alone), using the rules displayed in Tables 5.11 on page 184 and 5.7 on page 182.

Once we have eliminated all such equality constraints, we are left with constraints of the form $L \subseteq e$ and $\phi = \phi'$ only. We simplify this VC by applying the rules displayed in Tables 5.8 to 5.10 on pages 182–183.

This approach of reducing a second-order formula into a logically equivalent first-order formula by finding instantiations for the existentially quantified second-order variables is related to the work of Bledsoe. In [Ble79], he presents a theory where maximal instantiations are found. For a formula of the form $\exists X.\Phi(X) \wedge \Phi'(X)$ where Φ is antitone and Φ' monotone, a maximal instantiation A for X makes $\Phi(A)$ trivial, reducing the formula to $\Phi'(A)$ which is logically equivalent to the original formula. However, unlike in our case, his instantiations are not guaranteed to be complete, i.e., it may be that the instantiated formula is no longer provable. Also he only considered instantiation of a single variable. The present improvements are of course partly enabled by the particular syntactic form of our constraints.

Let us consider the rules in Table 5.11 on page 184 each of which is both sound (conclusion implies premise) and, conversely, complete.

Rule (eq-sym) simply allows us to swap the expressions on either side of an equality constraint. Rule (eq-resp) allows us to rewrite equivalent expressions in equality constraints, where equivalence is defined in Table 5.7 on page 182.

Rule (eq-inst) is the rule that allows us to eliminate an equality constraint. Note that it can only eliminate constraints of the form $\varepsilon \Rightarrow X = e$: those with an empty assumption and a variable to the left of the equality. Note further that

variable X may not occur free in e.

In the case where X also occurs free in e, we can apply (eq-idem), which introduces an extra existentially quantified variable. A mapping $F : X \mapsto T$, for some formula T with possibly free occurrences of X, is said to be *idempotent* if $F(F(X))$ is logically equivalent $F(X)$. Note that the the rule is certainly complete, and it is sound provided that $F : X \mapsto e$ is idempotent. Fortunately, $F : X \mapsto e$ is idempotent for any e satisfying $e = X$, as the next two observations will show.

Firstly, since (eq-resp) (together with the relation defined in Table 5.7) allows us to pull renamings inside of $- \lhd - \rhd -$ expressions, we may assume without loss of generality, that for constraints $v \Rightarrow e = e'$ or $L \subseteq e$, the expression e has form either $\sigma\phi$, σX or $e_0 \lhd x \rhd e_1$. (Setting σ to be the identity function recovers cases X and ϕ.) Also, for expression $e \equiv X \lhd x \rhd X'$, we note that $X \mapsto e$ and $X' \mapsto e$ are idempotent. Secondly, if F, F' are idempotent, then $X \mapsto F(X) \lhd x \rhd F'(X)$ is idempotent.

Finally, let us assume $\varepsilon \Rightarrow X = e$ where X, amongst possibly other variables, occurs free in e. Using the first observation, we consider $e \equiv \sigma\phi, \sigma X, e' \lhd x \rhd e''$ in turn, and argue that $X \mapsto e$ is idempotent. The first case is trivial. The third case follows from the second observation. For the second case, since renamings are injective and order-preserving, the only renaming from $\{1..n\}$ to $\{1..n\}$ is the identity function, and therefore $e \equiv X$ after all. So, immediately, we conclude that $X \mapsto e$ is idempotent.

The motivation for (eq-idem) is to allow us to eliminate variables where (eq-inst) is not applicable, so, (eq-idem) is typically applied immediately before an application of (eq-inst). Now, supposing $\varepsilon \Rightarrow X = e$, then (eq-idem) introduces a new variable Y, say; an application of (eq-inst) eliminates X; but we can α-convert Y to X since X is no longer a bound variable. Thus we may assume without loss of generality, that whenever we have a constraint $\varepsilon \Rightarrow X = e$, we may apply (eq-inst) whether or not its side condition holds.

Rules (eq-pullback) and (eq-equaliser) allow us to rewrite equality constraints involving renamings. The former for renamings of different variables, and the

latter for renamings of the same variable.

Rule (eq-pullback) uses the *pullback* of renamings σ and τ which is defined as follows. Given two renamings σ, τ with equal codomain their pullback consists of two maps π_1, π_2 with $\mathsf{cod}(\pi_1) = \mathsf{dom}(\sigma)$, $\mathsf{cod}(\pi_2) = \mathsf{dom}(\tau)$ and $\sigma \circ \pi_1 = \tau \circ \pi_2$ such that whenever $\sigma \circ \rho_1 = \tau \circ \rho_2$ then there is a unique renaming ψ with $\rho_i = \pi_i \circ \psi$ $(i = 1, 2)$. The domain of the pullback is an enumeration of the pairs (i, j) such that $\sigma(i) = \tau(j)$. Notice that when (i, j) and (i, j') are two such pairs then $j = j'$ and analogously for the first component. The renaming π_1, resp., π_2 maps the number corresponding to pair (i, j) to i, resp., j. In this case the pullback is unique (not only up to isomorphism as usual).

Similarly, rule (eq-equaliser) uses the *equaliser* of σ and τ, where $\mathsf{dom}(\sigma) = \mathsf{dom}(\tau)$ and $\mathsf{cod}(\sigma) = \mathsf{cod}(\tau)$, which consists of a renaming ρ such that $\sigma \circ \rho = \tau \circ \rho$ and whenever $\sigma \circ \psi = \tau \circ \psi$ then ψ factors uniquely through ρ. The domain of ρ is an enumeration of the subset of $\mathsf{dom}(\sigma) = \mathsf{dom}(\tau)$ consisting of those i for which $\sigma(i) = \tau(i)$. The map ρ sends the number corresponding to such an i to i. Again, in this case the equaliser is unique.

The completeness of rules (eq-pullback) and (eq-equaliser) is not obvious. It is based on the following categorical lemma:

Lemma 15 *Let I be the category of renamings and I_p be the category of partial renamings. If C is any category and $F : I \longrightarrow C$ is a functor that extends to I_p then F preserves pullbacks and equalisers.*

Proof A left inverse of a map e is a map p such that $pe = \mathrm{id}$. Clearly, in this case $F(p)$ is a left inverse of $F(e)$ for any functor F. We notice that every map e in I has a canonical left inverse p in I_p which is undefined outwith the image of e.

Suppose that a, b, c, d are maps such that $ba = dc$. If all of a, b, c, d have left inverses a', b', c', d' with $b'd = ac'$ and $d'b = ca'$ then they form a pullback diagram. Indeed, if $bf = dg$ then $f = b'dg = ac'g$, so $f = ah$ for $h = c'g$. One continues to argue in this way. Being equationally defined such a pullback is preserved by any functor. Now let a, b, c, d with $ba = dc$ be a pullback in I and a', b', c', d' be the canonical left inverses in I_p. We show that $b'd = ac'$.

Let $x \in \mathsf{dom}(d) = \mathsf{dom}(c')$. If $dx = by$ for some $y \in \mathsf{dom}(b)$ then there is $z \in \mathsf{dom}(a) = \mathsf{dom}(c)$ with $x = cz, y = az$. Thus, $ac'x = az = y = b'dx$. If, on the other hand, $dx \notin \mathsf{im}(b)$ then $c'x = \bot = ac'x = b'dx$. Likewise, we have $d'b = ca'$ so that any pullback in I is of the aforementioned form when viewed in I_p so that it will be preserved by any functor.

For the equalisers we use the following generalisation of so-called *absolute equalisers*.

Suppose that e, u, v are maps in a category such that $ue = ve$ and that e, resp., u have left inverses p and k. If $(kv)^n = ep$ for some $n \in \mathbb{N}$ then e is the equaliser of u and v. This follows by straightforward equational reasoning. Being purely equationally defined such an equaliser is preserved by any functor. Notice also that for element x, if $(kv)^n x = epx$ then $(kv)^{n+1}x = kv(kv)^n x = kvepx = kuepx = epx$.

Any equaliser in I has this property with p, k being the canonical left-inverses of e and u and n greater than the cardinality of $\mathsf{dom}(u) = \mathsf{dom}(v)$. Suppose that $x \in \mathsf{dom}(u) = \mathsf{dom}(v)$. If $ux = vx$ then $epx = kux = kvx$. If $ux \neq vx$ and $vx \notin \mathsf{im}(u)$ then $epx = \bot = kvx$.

If, however, $ux \neq vx$, but $vx = uy$ then $kvx = y$ yet $epx = \bot$. Suppose w.l.o.g. that $ux < vx = uy$. Then also $x < y$. Now, $uy = vy$ is not possible as otherwise $vx = vy$ contradicting injectivity of v. If $vy \notin \mathsf{im}(u)$ then as seen above $kvy = \bot$, so $(kv)^2 x = \bot$. Finally, if again $vy = uz$ then $vx < vy$ as $x < y$, so $uy = vx < vy = uz$, so $y < z$. Continuing in this way we obtain a chain $x = x_0 < x_2 < x_3 < \cdots < x_{t-1}$ where $epx_t = \bot = kvx_t$ and $vx_i = ux_{i+1}$. It follows that $(kv)^{t-i}x_i = epx_i$ hence the result. □

We instantiate this lemma with C being the category of sets and F being the functor mapping $\{1..n\}$ to the set of n-ary transition relations and a renaming to the induced function on transition relations which extends to partial renamings by inserting dummy elements. The conclusion of the lemma is then tantamount to the respective equivalence of premise and conclusion of the two rules in question. To illustrate let us consider a special case: let $\mathsf{dom}(\sigma) = \mathsf{dom}(\tau) = \{1,2,3\}, \mathsf{cod}(\sigma) = \mathsf{cod}(\tau) = \{1,2,3,4\}$. Let $\sigma(1) = \tau(1) = 1$,

$\sigma(x) = x, \tau(x) = x+1$ otherwise. The equaliser ρ has domain $\{1\}$ and sends 1 to 1. Then $\sigma A = \tau A$ expands to $A(x_1, x_2, x_3) = A(x_1, x_3, x_4)$. The lemma asserts that there is a 1-ary transition relation C such that $A(x_1, x_2, x_3) = C(x_1)$, i.e., that A does not depend on its second and third variable. Indeed, this holds for $C(x_1) = A(x_1, d, d)$ where d is a dummy element. Namely, we have $A(x_1, d, d) = A(x_1, x_2, d) = A(x_1, x_2, x_3)$ by successive use of the assumption. Similar illustrations are possible for rule (eq-pullback).

Supposing we have a constraint $v \Rightarrow \sigma X_1 = \tau\phi$, then for the left-inverse σ^{-1} of σ, if $\sigma^{-1} \circ \tau$ is a renaming, then we can apply (eq-const). Note that in general, $\sigma^{-1} \circ \tau$ may be a partial function, in which case it is not a renaming. In such a case, there is no X_1 satisfying this constraint, and so we conclude that the VC is false.

Rules (eq-if-elim), (eq-beta1) and (eq-beta2) allow us to rewrite equality constraints involving the if-else expression $e \triangleleft x \triangleright e'$. Together they consider the cases for the assumption list containing x and otherwise.

Finally rules (eq-if-intro1) and (eq-if-intro2) allow us to shorten the list of assumptions v in a constraint of the form $v \Rightarrow X = e'$.

The rules in Table 5.10 on page 183 eliminate variables X when they occur in a constraint of the form $T \subseteq X$. For a transition relation T which is monotone w.r.t. X, we write $\mu X.T$ to denote the least transition relation X such that $X = T$, remembering that X typically occurs free in T. By Knaster-Tarski we have the explicit formula $\mu X.T = \bigcap_{T[A/X] \subseteq A} A$ and the derived formulae: $T[\mu X.e/X] \subseteq \mu X.T$ and $\mu X.T \subseteq A$ whenever $T \subseteq A$. In particular, when T is independent of X, then the fix-point $\mu X.T$ is simply T itself, and this is precisely the second rule in Table 5.10.

The rules in Table 5.8 on page 182 simplify $- \triangleleft - \triangleright -$ expressions. Rules (if-betar1), (if-betar2), (if-betal1) and (if-betal2) are trivially sound and complete by definition of our interpretation of boolean assumptions. The remaining rules (if-eliml) and (if-elimr) can be shown to be sound and complete by using the fact $(x = \mathtt{ff}) \vee (x = \mathtt{ff})$.

In Table 5.9 on page 183, (collate) is trivially sound and complete. In (adj-

ren), we find the left adjoint $\exists_\sigma$ of renaming $\sigma : \{1..m\} \to \{1..n\}$, where adjointness is defined, for transition relations T, U of arities n, m:

$$\exists_\sigma T \subseteq U \quad \text{iff} \quad T \subseteq \sigma U \ .$$

Such an adjoint exists, and furthermore can be defined thus:

$$(\exists_\sigma T)(x_1, \ldots, x_m) \stackrel{\text{def}}{=} \exists y_1, \ldots, y_n. \quad T(y_1, \ldots, y_n) \wedge \bigwedge_{j=1..m} x_j = y_{\sigma(j)} \ .$$

Proposition 1 *If Ψ is obtained from Φ by application of any of the instantiation or equality rules (Tables 5.8 to 5.11 on pages 182–184) then Φ and Ψ are equivalent.*

This follows by straightforward reasoning in all cases except the equality rules involving pullback and equaliser. For those two the equivalence has been established above.

5.4.1 Success

We will now show that provided the rules are used in the right order, it is always possible to reduce a formula of the form

$$\exists \vec{X}. \quad \bigwedge_{e,e'} e = e' \quad \wedge \quad \bigwedge_i L_i \subseteq X_i$$

to a first-order (fix-point) verification condition. The proof provides a strategy for applying the rules. The correctness of this strategy relies on a key observation about compatibility of boolean assumptions v and v'. Formally, for two lists of boolean assumptions v, v', we say that v and v' are *incompatible* precisely if there is no switching s such that both $s \models v$ and $s \models v'$. Using this definition, we state and prove the following lemma.

Lemma 16 *Suppose from*

$$\exists \vec{X}. \quad v \Rightarrow e = e' \quad \wedge \quad \Phi$$

we obtain

$$\exists \vec{X}. \quad v_1 \Rightarrow e_1 = e'_1 \quad \cdots \quad v_n \Rightarrow e_n = e'_n \quad \wedge \quad \Phi$$

by repeated applications of (eq-if-elim), (eq-beta1) and (eq-beta2). Then the boolean assumptions $v_1, \ldots, v_n$ are pairwise incompatible.

Proof Since applications of (eq-if-elim) just add more constraints without changing the list of existentially-bound variables, nor modify the other constraints, we shall consider only the changing constraints, in isolation.

We prove the following more general result. Suppose

$$v_1 \Rightarrow e_1 = e'_1 \wedge \cdots \wedge v_n \Rightarrow e_n = e'_n \ ,$$

and $v_1, \ldots, v_n$ are pairwise incompatible. Now suppose constraint $v_i \Rightarrow e_i = e'_i$ has form $v_i \Rightarrow e''_i \triangleleft x \triangleright e'''_i = e$.

If $x \in v_i$, then w.l.o.g., we may assume $x = \mathtt{tt}$ and so we apply (eq-beta1), replacing the constraint by $v_i \Rightarrow e''_i = e'_i$.

If $x \notin v_i$, then we can apply (eq-if-elim) and the constraint is replaced by two constraints: $v_i, x = \mathtt{tt} \Rightarrow e''_i = e'_i$ and $v_i, x = \mathtt{ff} \Rightarrow e'''_i = e'_i$. We note that $v_i, x = \mathtt{tt}$ and $v_i, x = \mathtt{ff}$ are incompatible. Furthermore, for $j \neq i$, since v_i and v_j are incompatible, $v_i, x = \mathtt{tt}$ and v_j are incompatible (and similarly for $v_i, x = \mathtt{ff}$.) (To see this, consider some s such that $s \models v_j$, then if $s \models v_i, x = \mathtt{tt}$ we can conclude that $s \models v_i$ also, contradicting the incompatibility of v_i and v_j.)

In either case, we obtain new constraints whose assumptions are again pairwise incompatible. Now the statement of the lemma follows by repeatedly applying this more general result to $v \Rightarrow e = e'$. □

To help us reason with (eq-if-intro1) and (eq-if-intro2), we overload some notation.

Definition 7 *Suppose from $v \Rightarrow X = e$ we obtain $\varepsilon \Rightarrow X = e'$ by repeated applications of (eq-if-intro1) and (eq-if-intro2). We overload notation and write $e \triangleleft v \triangleright X$ to denote e'.*

In fact, $e \lhd v \rhd X$ is a nested expression of $- \lhd - \rhd -$ constructs. Considering this as a syntax tree, where nodes are labelled with the first-order variables in v, and the branches are labelled tt and ff, we have $|v| + 1$ leaves, $|v|$ of which are labelled with X, and one is labelled with e. Now reading v backwards gives us a path from the root of this tree to the leaf e. We now state the following lemma without proof, but which can be found by formalising this explicit description of $e \lhd v \rhd X$.

Lemma 17 *We can repeatedly apply (eq-beta1) and (eq-beta2) to $\exists \vec{X}.v \Rightarrow e = e_0 \lhd v' \rhd X \wedge \Psi$ and obtain $\exists \vec{X}.v \Rightarrow e = e_0 \wedge \Psi$, if and only if, for every switching s, $s \models v'$ implies $s \models v$; otherwise, we obtain $\exists \vec{X}.v \Rightarrow e = X \wedge \Psi$.*

As a simple corollary, if v and v' are incompatible, then from $\exists \vec{X}.v \Rightarrow e = \sigma(e_0 \lhd v' \rhd X) \wedge \Psi$ we can obtain $\exists \vec{X}.v \Rightarrow e = \sigma X \wedge \Psi$ by repeated applications of (eq-beta1) and (eq-beta2).

Theorem 15 (Success) *Given a formula of the form*

$$\exists \vec{X}. \quad \bigwedge_{e,e'} e = e' \quad \wedge \quad \bigwedge_i L_i \subseteq X_i \ ,$$

it is always possible to repeatedly apply the rules displayed in Tables 5.8 to 5.11 on pages 182–184 to obtain a logically equivalent, first-order, fix-point formula Φ.

Proof Recall we use the convention of letting Ψ_0 denote a first-order, fix-point formula.

Given a formula of the form

$$\exists \vec{X}. \quad \bigwedge_{e,e'} e = e' \quad \wedge \quad \bigwedge_i L_i \subseteq X_i \ ,$$

we rewrite it as

$$\exists \vec{X}. \quad \bigwedge_i \varepsilon \Rightarrow e_{0,i} = e_{1,i} \quad \wedge \quad \bigwedge_i \varepsilon, L_i \subseteq X_i \ ,$$

that is, choosing the empty list ε for our assumptions v_i and v_i'.

Next we reveal that the strategy first eliminates all equality constraints by applying the rules in Table 5.11 on page 184, until we obtain a formula of the form

$$\exists \vec{X}. \quad \bigwedge_i v_i', L_i \subseteq e_i' \quad \wedge \quad \Psi_0$$

where we have expressions e_i' on the right-hand side of the implication constraints. We then apply the rules in Table 5.8 on page 182 until we have eliminated all $- \lhd - \rhd -$ subexpressions, leaving a formula of the form

$$\exists \vec{X}. \quad \bigwedge_i v_i', L_i \subseteq e_i' \quad \wedge \quad \Psi_0 \ .$$

Now the expressions e_i' appearing on the right can be renamings, variables or predicates, and also there are no occurrences of $- \lhd - \rhd -$ subexpressions in any L_i. We then apply the rules in Table 5.9 on page 183 until we obtain a VC of the form

$$\exists \vec{X}. \quad \bigwedge_i T_i \subseteq X_i \quad \wedge \quad \Psi_0 \ .$$

where each variable $X \in \vec{X}$ occurs at most once on the right of a constraint $T \subseteq X$, and also there are no occurrences of $- \lhd - \rhd -$ subexpressions in any T_i. Now we can apply the rules in Table 5.10 on page 183 until all higher-order variables are eliminated.

It is clear that we can apply the rules displayed in Table 5.8 on page 182 until all $- \lhd - \rhd -$ expressions are eliminated. Similarly, it is clear that we can always apply the rules in Table 5.9 on page 183 to obtain a formula of the form

$$\exists \vec{X}. \quad \bigwedge_{i=1..|\vec{X}|} L_i \subseteq X_i \quad \wedge \quad \Phi \ ,$$

where each $X \in \vec{X}$ occurs at most once on the right of a constraint. Finally, it is also clear that we can apply the rules displayed in Table 5.10 on page 183 to eliminate all higher-order variables to leave a first-order, fix-point formula.

The only difficult part of the proof concerns showing that the rules in Table 5.11 on page 184 allow us to eliminate all equality constraints, and we concentrate on this by presenting an explicit strategy.

STEP1. Suppose we start with a formula

$$\exists\vec{X}.\quad v \Rightarrow e = e' \quad \wedge \quad \Phi\ .$$

We iteratively apply (eq-if-elim), (eq-beta1) and (eq-beta2) until all $-\lhd-\rhd-$ expressions deriving from $v \Rightarrow e = e'$ have been eliminated. Thus the resulting formula has form

$$\exists\vec{X}.\quad v_1 \Rightarrow e_1 = e_1' \wedge \cdots \wedge v_n \Rightarrow e_n = e_n' \quad \wedge \quad \Phi$$

for some n, and v_i, e_i, e_i' $(i = 1..n)$, and there are no $-\lhd-\rhd-$ expressions in any of e_i, e_i'. Furthermore, by Lemma 16, we know also that $v_1, \ldots, v_n$ are pairwise incompatible.

STEP2. We now show that we can iteratively eliminate each of these n equality constraints in turn. Let us consider constraint $v_1 \Rightarrow e_1 = e_1'$. We proceed by cases, the number of which is reduced since there are no $-\lhd-\rhd-$ expressions in either e_1 nor e_1'.

- Case $v_1 \Rightarrow \sigma X = \tau Y$. We apply (eq-pullback) and this constraint is replaced by two constraints: $v_1 \Rightarrow X = \pi_1 Z$ and $v_1 \Rightarrow Y = \pi_2 Z$, where π_1, π_2 are the pullbacks of σ, τ. For each of these two constraints, we repeatedly apply (eq-if-intro1) and (eq-if-intro2) until they are replaced by constraints: $\varepsilon \Rightarrow X = \pi_1 Z \lhd v_1 \rhd X$ and $\varepsilon \Rightarrow Y = \pi_2 Z \lhd v_1 \rhd Y$. Now we can apply (eq-idem) followed by (eq-inst) to each of these, eliminating both of them.

  Now there may be free occurrences of X (resp. Y) in e_i, e_i' $(i = 2..n)$. Suppose that X occurs free in e_2. By assumption, there are no $-\lhd-\rhd-$ expressions in e_2, and so $e_2 \equiv \rho_2 X_1$ for some ρ. Thus after the substitution for X, the constraint $v_2 \Rightarrow e_2 = e_2'$ is replaced by $v_2 \Rightarrow \rho_2(\pi_1 Z \lhd v_1 \rhd X) = e'$, where e' is the substitution applied to e_2'. Now from our hypotheses, we know v_1, v_2 are incompatible so by the corollary of Lemma 17 we can apply (eq-beta1), (eq-beta2) to simplify this constraint to $v_2 \Rightarrow \rho_2 X = e''$. Now we can repeat the same argument for expression e'', and conclude that eliminating $v_1 \Rightarrow e_1 = e_1'$ has not changed the constraint $v_2 \Rightarrow e_2 = e_2'$.

Similarly, we can show that the remaining constraints $v_i \Rightarrow e_i = e'_i$ $(i = 2..n)$ also stay the same.

Thus we have eliminated $v_1 \Rightarrow e_1 = e'_1$ without changing $v_i \Rightarrow e_i = e'_i$ $(i = 2..n)$. That is we have simplified the VC to

$$\exists \vec{X}. \quad v_2 \Rightarrow e_2 = e'_2 \wedge \cdots \wedge v_n \Rightarrow e_n = e'_n \quad \wedge \quad \Phi_1 \ ,$$

where $\Phi_1 \stackrel{\text{def}}{=} \Phi[\pi_1 Z/X][\pi_2 Z/Y]$.

- Case $v_1 \Rightarrow \sigma X = \tau X$. We apply (eq-equaliser) and this constraint is replaced by $v_1 \Rightarrow X = \rho Z$. Repeated applications of (eq-if-intro1), (eq-if-intro2) simplifies this constraint to $\varepsilon \Rightarrow X = \rho Z \lhd v_1 \rhd X$. Now we can use rules (eq-idem), (eq-inst) and eliminate the constraint.

  Again, the other constraints $v_i \Rightarrow e_i = e'_i$ $(i = 2..n)$ may have changed because of the substitution, but we use the same argument as in the previous case to show that we can simplify these constraints to their form before the substitution. Thus we have simplified the VC to

$$\exists \vec{X}. \quad v_2 \Rightarrow e_2 = e'_2 \wedge \cdots \wedge v_n \Rightarrow e_n = e'_n \quad \wedge \quad \Phi_1 \ ,$$

  where $\Phi_1 \stackrel{\text{def}}{=} \Phi[\rho Z/X]$.

- Case $v_1 \Rightarrow \sigma X = \phi$. We apply (eq-const) followed by (eq-inst). The other constraints $v_i \Rightarrow e_i = e'_i$ $(i = 2..n)$ may have changed because of the substitution, but they certainly do not contain $- \lhd - \rhd -$ expressions.

- Case $v_i \Rightarrow \phi = \phi'$. This constraint has no free occurrences of higher-order variables and so we may ignore it. Explicitly, simplify the VC to

$$\exists \vec{X}. \quad v_2 \Rightarrow e_2 = e'_2 \wedge \cdots \wedge v_n \Rightarrow e_n = e'_n \quad \wedge \quad \Phi_1 \ ,$$

  where $\Phi_1 \stackrel{\text{def}}{=} \Phi \wedge v \Rightarrow \phi = \phi'$.

- The remaining cases are symmetric instances of the above, and can be reduced to the previous cases by applying (eq-sym).

Thus, in any case, we have shown that we can simplify

$$\exists \vec{X}. \quad v_1 \Rightarrow e_1 = e'_1 \wedge \cdots \wedge v_n \Rightarrow e_n = e'_n \quad \wedge \quad \Phi \ ,$$

to

$$\exists \vec{X}. \quad v_2 \Rightarrow e''_2 = e'''_2 \wedge \cdots \wedge v_n \Rightarrow e''_n = e'''_n \quad \wedge \quad \Phi_1 \ .$$

Importantly, we note that $v_2, \ldots, v_n$ are also pairwise incompatible, and also e''_i, e'''_i do not contain $- \triangleleft - \triangleright -$ expressions. Thus we can apply this elimination process repeatedly to eliminate the constraints $v_i \Rightarrow e_i = e'_i$ for $i = 1..n$.

Note that once we have eliminated these n constraints, we have reduced the number of equality constraints in the VC by one. We can thus repeat this procedure (at STEP1) until all the equality constraints are eliminated. □

5.5 First-order verification conditions

While the possibility of eliminating all second-order quantification in favour of fix-points is perhaps surprising, the presence of fix-points in a verification condition seems quite natural. After all, when verifying a program with looping behaviour, be it from a while construct, or recursive method invocation, if one were to obtain a purely first-order verification condition, then one would expect to be required to provide an invariant of some sort. This last analogy, suggests that if we allow the programmer to provide further hints to the verification condition generator, in the form of annotations in the program, then we might be able to eliminate some of the fix-point operators.

Closer examination of the simplification rules reveals that fix-point operators are introduced only by (freeinst), that is, whenever we find a constraint of the form $L \subseteq X$ and X occurs free in L, and we instantiate for X. Supposing the programmer can provide an explicit instantiation of X to the simplification process as a hint, then at least this fix-point occurrence can be eliminated.

Taking this as our motivation, we extend the syntax of the programming language by allowing the user to provide annotations. For our purposes, an

annotation ψ is a partial function mapping sequences γ (including ε in this case) to transition relations. The precise syntax used to write such partial functions is not important. Annotated programs, which for the sake of brevity will also be denoted by the symbols a and b, are defined as before except we now allow annotations $a{::}\psi$. Furthermore, annotations of variables can still be used as variables, so for example, we can write $x.f{:=}(y{::}\psi)$.

We now add the following clause to the definition of Cons.

$$\begin{array}{rcl}
\mathrm{Cons}(\vec{x}, a{::}\psi) & \stackrel{\mathrm{def}}{=} & \mathrm{Cons}(\vec{x}, a) \\
& \cup & \{(\!|a|\!)_\gamma = \psi(\gamma) \mid \gamma \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!) = \psi(\varepsilon) \mid \varepsilon \in \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\alpha) \subseteq (\!|a{::}\psi|\!)_\alpha \mid \alpha \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\beta) = (\!|a{::}\psi|\!)_\beta \mid \beta \in [\![a]\!] \cap \mathsf{dom}(\psi)\} \\
& \cup & \{\psi(\varepsilon) \subseteq (\!|a{::}\psi|\!) \mid \varepsilon \in \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!)_\gamma = (\!|a{::}\psi|\!)_\gamma \mid \gamma \notin \mathsf{dom}(\psi)\} \\
& \cup & \{(\!|a|\!) = (\!|a{::}\psi|\!) \mid \varepsilon \notin \mathsf{dom}(\psi)\}
\end{array}$$

Here we have extended $[\![-]\!]$ to annotated programs. Note that we use the information in ψ whenever it is defined, and simply identify $(\!|a|\!)_\gamma$ and $(\!|a{::}\psi|\!)_\gamma$ whenever it is not. We ignore $\psi(\gamma)$ for $\gamma \notin [\![a]\!]$.

5.5.1 Dependency graphs

The question still remains as to whether we can avoid all fix-point operators by providing enough annotations. Furthermore, a related and interesting question is how many annotations are required to avoid all fix-point operators. In particular, we would like to be able to express a closed-form description of where annotations are required for a given unannotated program.

Inspection of the constraints generating function reveals that constraints of the form $L \subseteq X$ with X free in L do not occur initially. They arise in the intermediate steps of the instantiating procedure. We have the following observation: given such a constraint, we instantiate for X, the expression $\mu X.L$; thus those constraints containing X, will also contain free occurrences of the other free variables in L. This observation suggests that these questions should be answered by

considering cycles in some sensible notion of dependency graph. We recall that given a graph $(G, \rightarrow)$, a path is a sequence of edges $(a_j \rightarrow a_{j+1})$, and a cycle is precisely a path with the same start and end nodes.

Rule (notfreeinst) suggests that a possible notion of dependency graph for VC $\exists \vec{X}.\Phi$ has $\vec{X}$ for nodes and an edge $X \rightarrow Y$ whenever there is a constraint in Φ of the form $L \subseteq Y$ and X occurs free in L. In fact, this is adequate for dependency graphs without equality constraints.

The problem appears to be much harder once we take into account the equality constraints. A natural approach would be to define an equivalence relation relating those variables involved in an equality constraint. Problems arise when we consider the case for *if* x *then* a_0 *else* a_1. Here we have a constraint of the form

$$(\!|a_0|\!)_\beta \lhd x \rhd (\!|a_1|\!)_\beta = (\!|\mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1|\!)_\beta \ .$$

This would suggest insisting on

$$(\!|a_0|\!)_\beta \sim (\!|\mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1|\!)_\beta \sim (\!|a_1|\!)_\beta \ .$$

Unfortunately, the transitivity of $\sim$ will force $(\!|a_0|\!)_\beta \sim (\!|a_1|\!)_\beta$ too, which can give too many cycles. Consider, for example, the program

$$\mathit{if}\ x\ \mathit{then}\ y_1\ \mathit{else}\ \mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z] \ ,$$

where the two occurrences of y have been decorated with different subscripts. We get constraints

$$\langle y\rangle_{f\alpha} \sim (\!|y_1|\!)_{f\alpha}$$

for the first branch of the conditional and

$$\langle y\rangle_{f\alpha} \sim (\!|y_2|\!)_{f\alpha} \rightarrow (\!|y_2.f|\!)_\alpha \sim \langle z\rangle_\alpha \rightarrow (\!|z|\!)_\alpha \rightarrow (\!|[f{=}z]|\!)_{f\alpha}$$
$$(\!|[f{=}z]|\!)_{f\alpha} \rightarrow (\!|\mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z]|\!)_{f\alpha}$$

for the second branch. By insisting

$$(\!|y_1|\!)_{f\alpha} \sim (\!|\mathit{if}\ x\ \mathit{then}\ y_1\ \mathit{else}\ \mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z]|\!)_{f\alpha}$$
$$\sim (\!|\mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z]|\!)_{f\alpha}$$

we find that we now have a cycle. However, applying the reduction rules to this system of constraints does not give us a fix-point operator.

Careful consideration shows that variables $(\!|y_1|\!)_\alpha$ and $(\!|\mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z]|\!)_\alpha$ should not be equivalent. Instead $(\!|\mathit{if}\ x\ \mathit{then}\ y_1\ \mathit{else}\ \mathit{let}\ z{=}y_2.f\ \mathit{in}\ [f{=}z]|\!)$ should be equivalent to $(\!|y_1|\!)_\alpha$ only when x is true, and conversely for the else branch. So it appears we have to consider cycles for cases where x is true and false. More generally, we should consider cycles for each *switching* s.

This example motivates our final definition of dependency graph.

Definition 8 *Given a set N, and binary relations $E_s, R_s \subseteq N \times N$, for each switching s, the triple $G = (N, (E_s), (R_s))$ is said to be* dg-closed *if,*

- *E_s is an equivalence relation, and*
- *$E_s R_s E_s \subseteq R_s$,*

and said to be tdg-closed *if furthermore, for each switching s,*

- *R_s is transitive.*

Notationally, given a tuple G of a set of nodes and two families of binary relations over N indexed by switchings s, we write G^{dg} and G^{tdg} to denote their dg- and tdg-closures, defined below

Definition 9 *For a triple $B = (N, (E_s), (R_s))$ where $(E_s), (R_s)$ are families of relations indexed by switchings s, we define,*

$$B^{\mathrm{dg}} \stackrel{\mathrm{def}}{=} (N, (E'_s), (R'_s))$$
$$B^{\mathrm{tdg}} \stackrel{\mathrm{def}}{=} (N, (E'_s), (R''_s))$$

where

$$E'_s \stackrel{\mathrm{def}}{=} \textit{least equivalence relation including } E_s$$
$$R'_s \stackrel{\mathrm{def}}{=} E'_s R_s E'_s$$
$$R''_s \stackrel{\mathrm{def}}{=} R'^{+}_s \quad \textit{the transitive closure of } R'_s.$$

Definition 10 (Dependency graph) *A* dependency graph *(resp. transitive dependency graph) is a dg-closed (resp. tdg-closed) tuple of nodes, and two families of relations. A dependency graph $G = (N, (E_s), (R_s))$ is said to be acyclic if R_s is acyclic for all switchings s.*

We frequently choose symbols such as $(\sim_s), (\rightarrow_s)$, which are suggestive of their properties, for the two families of relations of a dependency graph.

Definition 11 (Dependency graph homomorphism) *Suppose we have dependency graphs $C = (M, (E_s), (Q_s))$ and $D = (N, (F_s), (R_s))$. A function $f : M \rightarrow N$ is said to be a* dg-homomorphism *from C to D, written $f : C \rightarrow D$, if for all switchings s and nodes a, b,*

- *$a \; E_s \; b$ implies $f(a) \; F_s \; f(b)$, and*
- *$a \; Q_s \; b$ implies $f(a) \; R_s \; f(b)$.*

A dg-homomorphism $C \rightarrow D$ is said to be a tdg-homomorphism *if C, D are, in particular, transitive dependency graphs.*

We shall often drop the prefixes and simply write homomorphism for both dg- and tdg-homomorphism, when there is no risk of confusion.

We note that homomorphisms are morphisms in the categorical sense since:

1. homomorphisms $f : B \rightarrow C$, $g : C \rightarrow D$, whenever $a \rightarrow_s b$ in B, by the definition of homomorphism, we have $f(a) \rightarrow_s f(b)$ in C, and also $g(f(a)) \rightarrow_s g(f(b))$ in D, for switching s, i.e. $g \circ f$ is a homomorphism;
2. composition of homomorphisms is associative since they are in particular functions; and
3. for dependency graph D the identity function defined on its nodes is a homomorphism $D \rightarrow D$.

Proposition 2 *For transitive dependency graphs C, D, if $f : C \rightarrow D$ is a tdg-homomorphism, then C is acyclic provided D is acyclic.*

Proof Suppose $C = (M, (E_s), (Q_s))$, $D = (N, (F_s), (R_s))$ and $f : C \longrightarrow D$ is a homomorphism. Let us assume that C is not acyclic. Since Q_s is transitive, there are s, a such that

$$a\ Q_s\ a\ ,$$

and since f is a homomorphism

$$f(a)\ R_s\ g(a)$$

which is a cycle in D. That is, D is not acyclic. □

Thus in the case of transitive dependency graphs, acyclicity of C can be shown by exhibiting a homomorphism $f : C \longrightarrow D$ for some D known to be acyclic. (In fact this is also the case for general dependency graphs.)

At this point, we reveal to the reader that we will define a function which constructs a transitive dependency graph from any given VC, with the crucial property that if the VC contains a fix-point expression, then its dependency graph is not acyclic. We then show that the rules in Tables 5.9 to 5.11 on pages 183–184 preserve acyclicity in the following sense: supposing a rule can simplify Φ to Φ', then if the dependency graph of Φ is acyclic, then that of Φ' is also acyclic. To prove this we exhibit a homomorphism.

We state the following simple lemma without proof which gives us useful ways to construct tdg-homomorphisms.

Lemma 18 *Given triples, $C = (M, (E_s), (Q_s)), D = (N, (F_s), (R_s))$ which are not necessarily dependency graphs, a function $f : M \longrightarrow N$ satisfying*

- *$a\ E_s\ b$ implies $f(a)\ F_s\ f(b)$, and*
- *$a\ Q_s\ b$ implies $f(a)\ R_s\ f(b)$.*

is a dg-homomorphism $C^{dg} \longrightarrow D^{dg}$. Also, assuming C, D are dependency graphs, a dg-homomorphism $f : C \longrightarrow D$, in particular, is a tdg-homomorphism $C^{tdg} \longrightarrow D^{tdg}$.

To allow us to define a dependency graph for a given VC Ψ, we introduce an auxiliary definition $\Psi{\downarrow}s$ whose definition is displayed in Table 5.4 on page 179. Informally, $\Psi{\downarrow}s$ is Ψ assuming boolean variables take values as determined by s. Given a VC Ψ, we write $\mathsf{DG}(\Psi)$ to denote its transitive dependency graph which is defined in Table 5.5 on page 180, using the auxiliary definitions (in particular G_0) in Table 5.4 on page 179. Note that by definition, if Φ contains a fix-point expression, its dependency graph $\mathsf{DG}(\Phi)$ is not acyclic.

The following lemma, stated without proof, is required for the next proposition.

Lemma 19 *Let s be a switching.*

1. *If $e \leadsto e'$ (as defined in Table 5.7 on page 182) then $e{\downarrow}s$ and $e'{\downarrow}s$ have the same free higher-order variables.*
2. $(e[e'/X]){\downarrow}s \equiv (e{\downarrow}s)[e'{\downarrow}s/X]$
3. $(L[L'/X]){\downarrow}s \equiv (L{\downarrow}s)[L'{\downarrow}s/X]$
4. $(\Phi[L/X]){\downarrow}s \equiv (\Phi{\downarrow}s)[L{\downarrow}s/X]$

Proposition 3 *Acyclicity is preserved by the rules displayed in Tables 5.8 to 5.11 on pages 182–184 in the following sense. If applications of the rules in those Tables can simplify Ψ to Ψ' then $\mathsf{DG}(\Psi)$ is acyclic implies $\mathsf{DG}(\Psi')$ is acyclic.*

The proof proceeds by considering each rule in turn. Supposing a rule has form

$$\frac{\Psi}{\Psi'}\ P$$

for VCs Ψ, Ψ' and side condition P, we show that assuming P, there is a homomorphism $\mathsf{DG}(\Psi') \to \mathsf{DG}(\Psi)$. Since the composition of two homomorphisms is a homomorphism, this is sufficient to show that for any sequence of applications of rules simplifying Ψ to Ψ', there is a homomorphism $\mathsf{DG}(\Psi) \to \mathsf{DG}(\Psi')$. By appealing to Proposition 2, we conclude that $\mathsf{DG}(\Psi)$ is acyclic implies $\mathsf{DG}(\Psi')$ is acyclic, as required.

The proof can be found in Section A.3.

5.5.2 Bridges

Let us overload notation and write $\mathsf{DG}(\vec{x}, a)$ for $\mathsf{DG}(\mathrm{Cons}(\vec{x}, a))$. The dependency graph $\mathsf{DG}(\vec{x}, x.f{:=}y)$ includes the paths

$$\begin{array}{ccc} \langle x\rangle_{f\alpha} & \overset{\sim}{\text{———}} & (\!|x|\!)_{f\alpha} \\ & & \Big| \sim \\ \langle y\rangle_{\alpha} & \longrightarrow & (\!|y|\!)_{\alpha} \end{array} \quad \text{and} \quad \begin{array}{ccc} \langle x\rangle_{f\beta} & \overset{\sim}{\text{———}} & (\!|x|\!)_{f\beta} \\ & & \Big| \sim \\ \langle y\rangle_{\beta} & \overset{\sim}{\text{———}} & (\!|y|\!)_{\beta} \end{array}$$

The important aspect of these paths is that they join up the variables $\langle x\rangle_{f\gamma}$ and $\langle y\rangle_{\gamma}$ by forming a *bridge* between them. The presence of bridges creates possible cycles, as we shall see later.

Note that the presence of these bridges depends on there being a variable $(\!|y|\!)_{\gamma}$, that is, $\gamma \in [\![y]\!]$. Since γ is a sequence ending in some m, we conclude that this is precisely when y is higher-order, i.e. stores the location of an object which (hereditarily) has methods. Conversely, if the type $[\![y]\!]$ has paths containing only fields, i.e. y is not a higher-order variable, then there is no $\gamma \in [\![y]\!]$ and therefore, we do not get these bridges. To summarise, we get bridges when we have a higher-order field update.

Now consider an annotated field update, $x.f{:=}(y{::}\psi)$. The edges of the first of the two bridges in the previous dependency graph now become

$$\begin{array}{ccc} \langle x\rangle_{f\alpha} & \overset{\sim}{\text{———}} & (\!|x|\!)_{f\alpha} \\ & & \Big| \sim \\ & & (\!|y{::}\psi|\!)_{\alpha} \end{array} \quad \text{and} \quad \langle y\rangle_{\alpha} \longrightarrow (\!|y|\!)_{\alpha}$$

and similarly for the second bridge. Note that there is no edge between nodes $(\!|y{::}\psi|\!)_{\alpha}$ and $(\!|y|\!)_{\alpha}$. The annotation has effectively broken the bridge.

5.5.3 Cycles

The dependency graph $D = \mathsf{DG}(\vec{x}, \textit{if } x \textit{ then } a_0 \textit{ else } a_1)$ is displayed in Figure 5.1 where the barrel-shaped units denote subgraphs for a_0 and a_1 whose internal

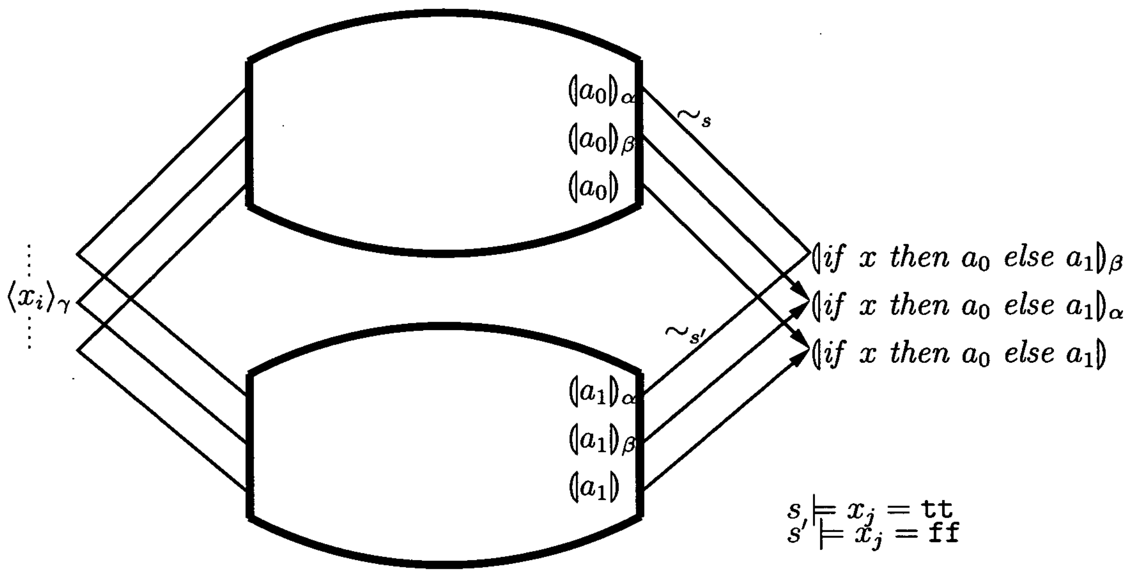


Figure 5.1: Dependency graph for *if* x *then* a_0 *else* a_1. The two barrel-shaped units denote the dependency graphs for a_0 and a_1, abstracting the internal paths. The V-shaped paths on the left emphasise that the node $\cdots \langle x_i \rangle_\gamma \cdots$ are shared between the two units. The paths labelled $\sim_s$ on the right emphasise that they are only present for switchings $s \models x_j = \texttt{tt}$, similarly for s'.

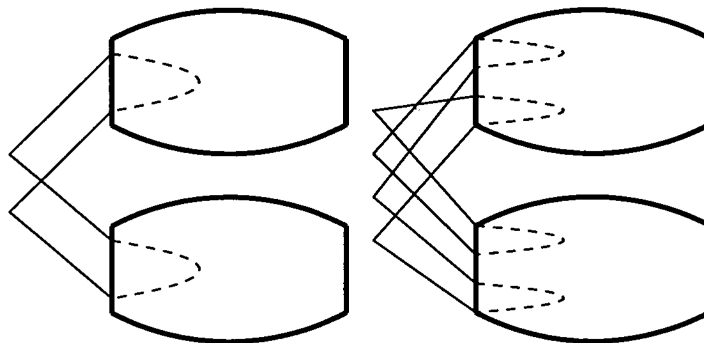


Figure 5.2: Two possible ways of creating a cycles in *if* x_j *then* a_0 *else* a_1

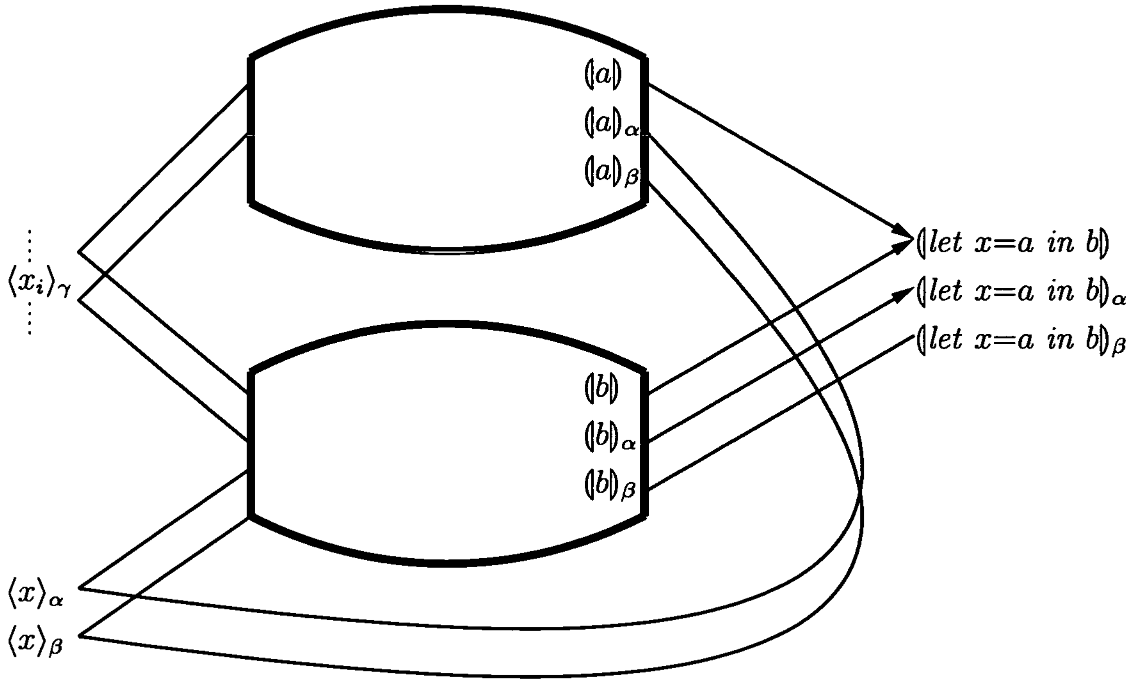


Figure 5.3: Dependency graph for *let* $x{=}a$ *in* b. The two barrel-shaped units denote the dependency graphs for a and b, abstracting the internal paths. The paths from the right of the graph of a to $\langle x\rangle_\alpha$, $\langle x\rangle_\beta$ emphasise the fact that the nodes $(\!|a|\!)_\gamma$ are identified with the nodes $\langle x\rangle_\gamma$.

paths have been abstracted. Two examples of possible cycles arising in D from interactions between the paths of the subgraphs are shown in Figure 5.2. Here we see bridges in the subgraphs joining to form a cycle. An example of the first type of cycle is

$$\textit{if } x' \textit{ then } (\textit{let } z{=}x.f \textit{ in } y.f{:=}x) \textit{ else } (\textit{let } z{=}y.f \textit{ in } x.f{:=}y)$$

which creates a cycle through $\langle x\rangle_{f\alpha}$ and $\langle y\rangle_{f\alpha}$.

Similarly, the dependency graph D is displayed in Figure 5.3, and again the paths in the subgraphs have been abstracted. Of course the two cycles displayed in Figure 5.1 are also possible in $D = \mathsf{DG}(\vec{x}, \textit{let } x{=}a \textit{ in } b)$, but furthermore, we also have the possibility of creating a cycle depicted by the second diagram in

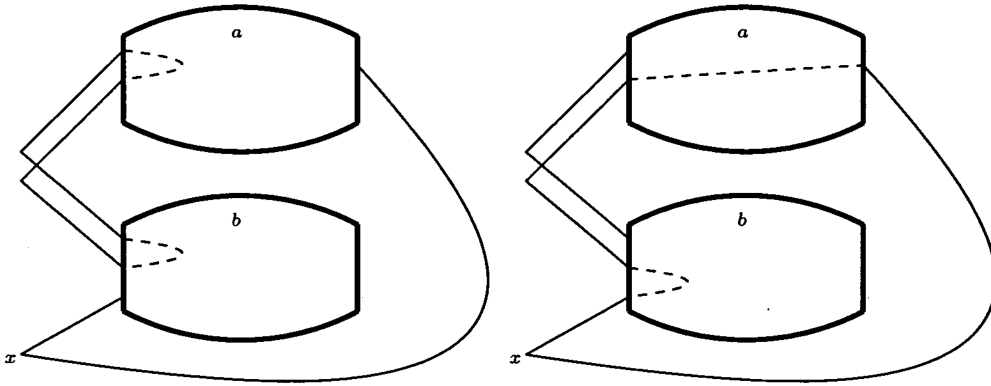


Figure 5.4: Two possible ways of creating a cycle in *let* $x{=}a$ *in* b

Figure 5.4. In this case, we can get a cycle if, in b, we create a bridge between x and some variable that is used in a. An example of such a cycle can be seen in

$$let\ x{=}y.f\ in\ y.f{:=}x\ .$$

This example creates a cycle through $\langle x\rangle_{\alpha}$ and $\langle y\rangle_{f\alpha}$.

Though the previous example is the simplest one demonstrating such a cycle, the resulting fix-point expression can easily be solved. A reasonable question is whether there are really cycles of such form that produce fix-point expressions that are not easily solvable. Unfortunately, the answer is yes.

Our store is updatable and higher-order in the sense that we can store function closures. Therefore we can, for example, construct two objects that in isolation do not exhibit recursion, but when composed together via the let construct and a field update, can become mutually recursive. Consider

$$\begin{array}{l} let\ u{=}[f{=}[m'{=}\varsigma(y')true], m{=}\varsigma(y)a; y.f.m'()]\ in \\ \quad u.f{:=}[m'{=}\varsigma(y')u.m()]; u.m() \end{array}$$

with an arbitrary program a. (Note that we are using syntactic sugar; a strictly legal program requires many more let constructors.) Here we create objects in both branches of the let construct. In isolation, neither seems to have recursive methods. However, when combined with the let construct and the field update, we get mutual recursion, albeit bound at runtime.

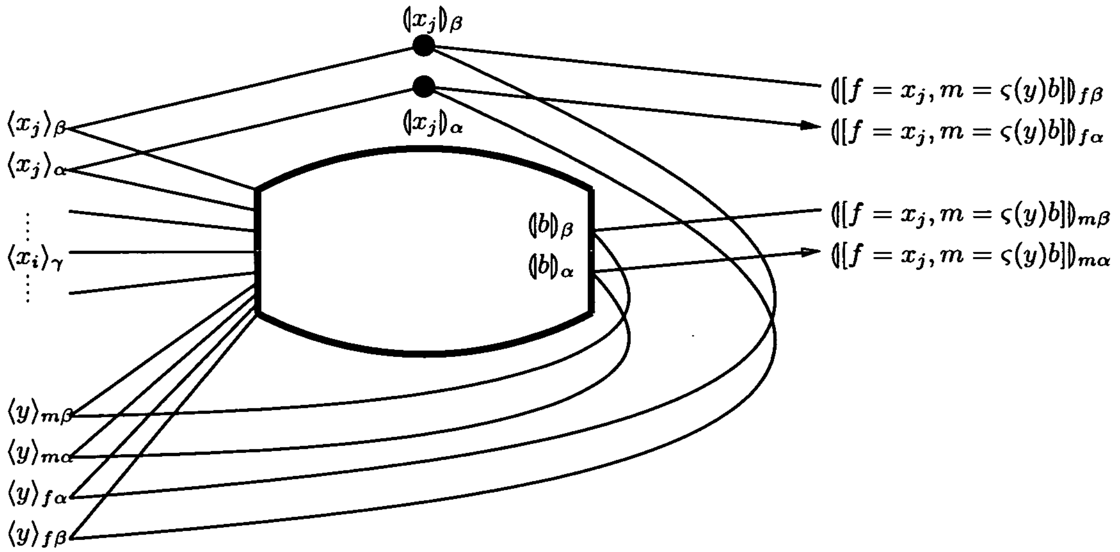


Figure 5.5: Dependency graph for $[f = x_j, m = \varsigma(y)b]$. The barrel-shaped unit denotes the dependency graph for b. The paths from $(\!|x_j|\!)_\alpha, (\!|x_j|\!)_\beta$ to $\langle y\rangle_{f\alpha}, \langle y\rangle_{f\beta}$ emphasise that these nodes are identified. Similarly for the paths from the right of the graph of b to $\langle y\rangle_{m\alpha}, \langle y\rangle_{m\beta}$.

In the case of object creation, we first consider objects with exactly one field and one method. We display $D = \mathsf{DG}(\vec{x}, [f{=}x_j, m{=}\varsigma(y)b])$ in Figure 5.5. Already there are new possibilities for creating cycles. Figure 5.6 shows four possibilities where internal paths of the subgraph can cause cycles. The possibility shown in the bottom right corresponds to static-bound method recursion. Unlike previous examples, we have a cycle that does not pass through a bridge. Nevertheless, if we annotate the program with a suitable annotation ψ as follows:

$$[f{=}x_j, m{=}\varsigma(y)(b{::}\psi)]$$

then we can also break this cycle.

So far, it may still appear feasible to prove acyclicity of dependency graphs by induction over programs and exhaustive case analysis as we have hinted at above. If the reader believes we have exhaustively covered all possibilities for creating cycles in the previous examples, then perhaps he/she should consider

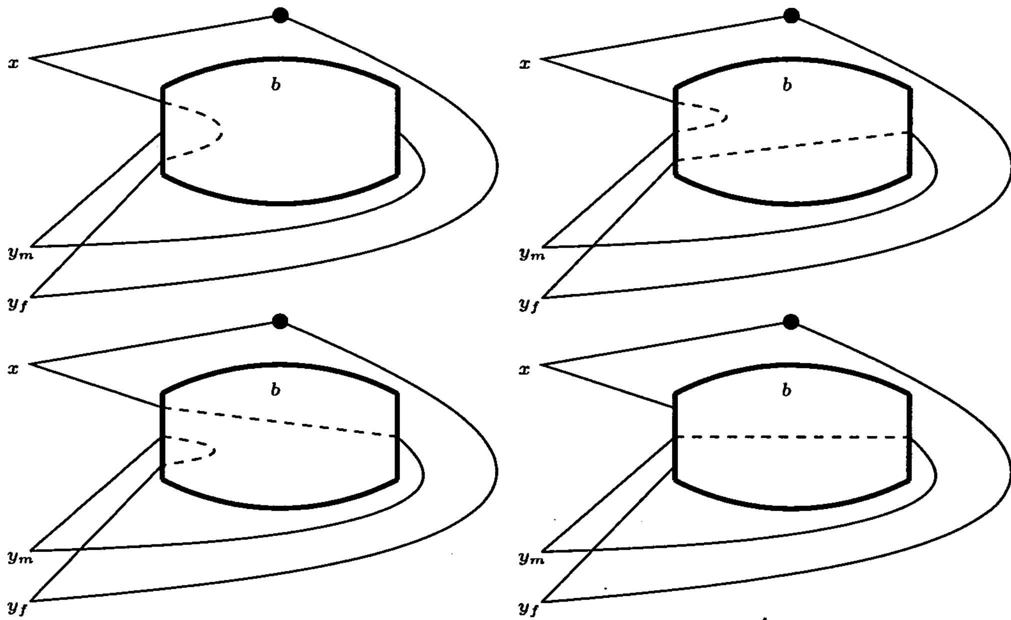


Figure 5.6: Four possible ways of creating a cycle in $[f{=}x_j, m{=}\varsigma(y)b]$

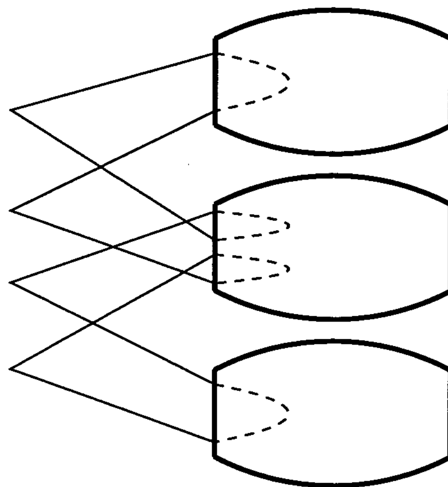


Figure 5.7: A possible cycle in an object with three method bodies

another special case of object creation, namely an object with three methods. Figure 5.7 shows a possible cycle caused by interaction between bridges within the subgraphs for the method bodies. Clearly in the general case of ℓ method bodies, there can only be more possibilities. Thus it is hoped the reader is convinced that exhaustive case analysis is not an ideal proof technique. In the sequel, we will propose exhibiting homomorphisms as a more suitable proof technique for proving acyclicity.

5.5.4 How many annotations?

Ideally we would like to provide as few annotations as possible to allow us to obtain a fix-point-free verification condition. A pragmatic solution would be to apply the instantiation rules and inform and hint the programmer whenever we introduce a fix-point operator. Hopefully each hint will allow the programmer to provide an annotation in the right place, breaking a particular cycle. Nevertheless, it would be more effective to be able to state, a priori, exactly where annotations are needed to be able to obtain a fix-point-free verification condition.

Our initial discussion about the need for annotations suggested that loops need invariants. We do not have while, for or other loop constructs, but we can create loops with method recursion. But are there other ways to create loops? The answer, as mentioned before, is yes.

Our store is both updatable and higher-order since we can update objects which contain methods, and it is well known that loops can occur in programs that do not have recursive functions (methods). Consider the following example

$$\begin{aligned} &\mathit{let}\ u{=}[f{=}[m{=}\varsigma(y)\mathit{true}]]\ \mathit{in} \\ &\quad u.f{:=}[m{=}\varsigma(y)u.f.m()];\ u.f.m()\ . \end{aligned}$$

Here we initially create an object, which we call u, with one field f initialised to an object with one method m that computes something trivial. Thereafter, within the main body of the let construct, we update field $u.f$ with another object with method m which invokes method $u.f.m$. When run, the program exhibits non-termination. and in a similar vein it is possible to write programs computing factorial, Fibonacci numbers, etc., recursively.

We could not have written

$$let\ u{=}[f{=}[m{=}\varsigma(y)u.f.m()]]\ in\ \cdots$$

since the scoping rules of the let construct forbids this. However, we can write essentially the same program by updating $u.f$ afterwards. This is allowed by the scoping rules, and is precisely runtime-bound recursion.

The point is that if we insist on annotating only static-bound (mutually) recursive methods, then we will not be able to eliminate all cycles from the dependency graph.

5.6 Conclusions and further work

We have presented an algorithm which can infer large parts of a proof in AL. There exists a prototype implementation and in Chapter 6 we give an account of the examples we have attempted with the prototype implementation.

For the future we intend to develop an implementation which could display loops in dependency graphs and so ask the user for more annotations when needed. Alternatively, one could strive for a static type system capable of guaranteeing cycle freeness of dependency graphs. Our experience so far, in particular the presence of implicit recursion described in Section 5.5.4 leads us to a pessimistic view as to the existence of such a system which would at the same time be reasonably strong and comprehensible. However, recent work on alias analysis and linearity [WSR00, OP00, Hof00] might be useful.

One of the goals of our VCG was to provide increased automation for finding verification proofs. The main bulk of this presentation is to show that our VCG algorithm is correct. An alternative approach towards the same goal is to embed the program logic into a theorem prover, as detailed in Chapter 3. Then one can use the automation provided by the theorem prover, for example by implementing specialised tactics (e.g. like Gordon's VCG in [Gor88]) to reduce some of the resulting proof burden. In particular, unsound tactics cannot allow incorrect "proofs" to be derived, since the proof checking facility of the theorem prover guards us against such an eventuality. Thus *soundness* of the VCG is then of

reduced importance. Nevertheless, we believe that completeness of the VCG algorithm is still important: in its absence, should one obtain an unprovable subgoal from a tactic, one cannot conclude that the goal itself is unprovable. If one chooses carefully how AL is embedded into a suitable theorem prover such as Isabelle/HOL, we believe that the VCG algorithm presented here can be implemented as a tactic.

Our method is currently restricted to closed programs. However, due to the compositional nature of AL, and our constraints-solving approach to VCG, we foresee no difficulty to extend it to open programs which would allow us to apply verification to certain crucial parts of a program and simply assume correctness of others. Suppose, for example, x occurs free in program b. If in b we use method m of x, then type inference infers $m \in [x]$. However, as it stands, since there is no constraint of the form $T \subseteq \langle x \rangle_m$ (where $\langle x \rangle_m$ is the transition relation that m of x must satisfy) and since we choose minimal instantiations for transition relations, variable $\langle x \rangle_m$ is instantiated with the strongest (empty) transition relation; the result of this exercise has no practical use since the answer effectively means that the program is correct provided m of x satisfies this strongest transition relation (in fact, any non-terminating method can satisfy this relation). However, if we allow annotations of variables (not simply variable occurrences) within a program, then we can impose further constraints on $\langle x \rangle_m$. For example, if we can add the constraint $U_0 \subseteq \langle x \rangle_m$, then solvability of the resulting set of constraints implies that *let* $x{=}a$ *in* b is correct provided we can show $\vdash a : [m{:}\varsigma(y)B{::}U_0] :: T$ for some B and T. We may simply assume that a satisfies this proviso[2] or we may prove that it does using our VCG or otherwise.

Extensions of the AL logic either by recursive types or with a view to address its incompleteness will lead to new challenges. For example, the equivalence allowing us to rewrite $A <: A'$ as an equivalent collection of constraints over the component transition relations of A, A' is only possible if $[\![A']\!]$ is finite.

In a finite model fix-points can be computed quickly (polynomial in the size of the model) by iteration. Thus, if we were able to detect statically, whether the

[2]because we are very confident of the correctness of a, or, more likely, because it is too expensive to do so

required model was indeed finite then we could use model-checking, i.e., iterative computation of fix-points to discharge verification conditions automatically. We believe that the model, i.e., the state space is indeed finite in examples like the dining philosophers example.

From a point of view of programming methodology, we may use unsound but complete approximations to proving the VC, and thus detect possible mistakes in the code and/or specification. If these complete methods are "push-button" (i.e. automatic) then, for example, they may be applied off-line, e.g. overnight, during the development of a project. For example, we could use model-checking on an arbitrarily sized, finite model and in this way possibly detect unsatisfiability of the verification conditions. Also, we can eliminate fix-point expressions by replacing a constraint $L \subseteq Y$, where L contains a fix-point, with finitely many approximations: for example, supposing $L \equiv L'[\mu X.F(X)/Z]$ for some L', the approximations can be

$$L'[F^i(\bot)/Z] \subseteq Y$$

for $i = 0..n$. To see why this is complete, we recall that fix-points only arise from applications of (notfreeinst) and so $Z \mapsto L'$ is monotone. Certainly $F^i(\bot) \subseteq \mu X.F(X)$, so

$$L'[F^i(\bot)/Z] \subseteq L'[\mu X.F(X)/X] \ .$$

Therefore $L'[F^i(\bot)/Z] \subseteq Y$ is a consequence of $L \subseteq Y$. Since these approximations are fix-point-free, they are more likely to be automatically discharged, or disproved, again giving a push-button check. We note that ESC [DLNS98] uses approximations of VCs for while loops.

In [FS01], Flanagan and Saxe address the problem of exponential blow up of VCs. It is clear that our VCG can experience the same complexity issues and it would be interesting to see if modifications of their solutions can be applied to ours.

Our global approach to verification condition generation appears to be to quite some degree independent of the program logic at hand though, of course, for simpler logics such as plain Hoare logic,and possibly those of Poetzsch-Heffter

and Müller (Mojave) [PHM99] and von Oheimb [Ohe01], the well-known straightforward computation of the verification condition by structural recursion works just as well. It would be interesting to apply our approach to other logics, for example Leino's later variant [Lei98] of AL (LRO), which admits recursively typed objects.

In LRO [Lei98], preliminary investigations suggest that since subtyping is defined by *name matching* (like in Java, as opposed to *structural type matching* as present in AL), the VCG problem is easier. Firstly, type inference is much simpler, and also, since each method has exactly one specification for all implementations (thus specifications are no longer covariant along methods), far fewer constraints (and thus second-order variables) are generated. Of course, these advantages are at the expense of a more restrictive type system, though it appears to be sufficient for modelling Java.

It appears that neither Mojave nor von Oheimb's logic features static specifications in the same spirit as found in AL and LRO. Thus it may be that the generality of our approach to VCG is unnecessary. However, we believe static specifications to be useful, especially because it allows us to internalise lemmata and thus reduce the size of formal proofs. For example, in a formal Mojave proof p, it may be necessary to prove $\{P\}\, T{:}m\, \{Q\}$ several times, for virtual method $T{:}m$. Formally, there is a (possibly different) derivation for each occurrence. However, in practice, one introduces a meta-lemma whose proof provides a derivation concluding in $\{P\}\, T{:}m\, \{Q\}$, and this derivation is inserted into p wherever it is needed. It may be possible to extend Mojave and von Oheimb's logic to feature static specifications and an internalisation of lemmata. For such a logic, our VCG approach may prove to be useful. Of course, the extended logic may become less complete because all possible occurrences of $\{P\}\, T{:}m\, \{Q\}$ must now be a weakening of one specific triple $\{P_0\}\, T{:}m\, \{Q_0\}$, and it may be that the assertion language cannot express P_0, Q_0. However, in von Oheimb's logic, there are such assertions and in fact they are used in the proof of completeness in the form of the "Most General Formula"; and so in this case, we believe that the logic would be no less complete.

$$\frac{}{E \vdash x_j : A :: T} \quad \left\{ \begin{array}{l} E(x_j) <: A \\ Res(x_j) \subseteq T \end{array} \right\}$$

$$\frac{}{E \vdash true : Bool :: T} \quad \left\{ Res(\mathtt{tt}) \subseteq T \right\}$$

$$\frac{}{E \vdash false : Bool :: T} \quad \left\{ Res(\mathtt{ff}) \subseteq T \right\}$$

$$\frac{}{E \vdash n : Nat :: T} \quad \left\{ Res(n) \subseteq T \right\}$$

$$\frac{\begin{array}{c} E \vdash x : Bool :: Res(x) \\ E \vdash a_0 : A_0 :: T_0 \quad E \vdash a_1 : A_1 :: T_1 \end{array}}{E \vdash if\ x\ then\ a_0\ else\ a_1 : A' :: T'} \quad \left\{ \begin{array}{l} T_0[\mathtt{tt}/x] \subseteq T[\mathtt{tt}/x] \\ T_1[\mathtt{ff}/x] \subseteq T[\mathtt{ff}/x] \\ A_0[\mathtt{tt}/x] <: A[\mathtt{tt}/x] \\ A_1[\mathtt{ff}/x] <: A[\mathtt{ff}/x] \end{array} \right\}$$

$$\frac{E \vdash a : A :: T \quad E, x : A \vdash b : B :: U}{E \vdash let\ x{=}a\ in\ b : A' :: T'} \quad \left\{ \begin{array}{l} T;_x U \subseteq T' \\ B <: A' \end{array} \right\}$$

$$\frac{\begin{array}{c} E \vdash x_i : A_i :: Res(x_i)^{1 \le i \le k} \\ E, y_j{:}A \vdash b_j : B_j :: U_j{}^{1 \le j \le \ell} \end{array}}{E \vdash [f_i{=}x_i{}^{i=1..k}, m_j{=}\varsigma(y_j)b_j{}^{j=1..\ell}] : A' :: T'} \quad \left\{ \begin{array}{l} A = [\ f_i{:}A_i{}^{i=1..k}, \\ \qquad m_j{:}\varsigma(y_j)B_j{::}U_j{}^{j=1..\ell}\ \ ] \\ A <: A' \\ T_{\mathsf{obj}}(x_1, \ldots, x_k) \subseteq T' \end{array} \right\}$$

$$\frac{E \vdash x : [f{:}A] :: Res(x)}{E \vdash x.f : A' :: T'} \quad \left\{ \begin{array}{l} A <: A' \\ T_{\mathsf{fsel}}(x, f) \subseteq T' \end{array} \right\}$$

$$\frac{E \vdash x : [m{:}\varsigma(y)B{::}U] :: Res(x)}{E \vdash x.m() : A :: T} \quad \left\{ \begin{array}{l} B[x/y] <: A \\ U[x/y] \subseteq T \end{array} \right\}$$

$$\frac{E \vdash x_j : A :: Res(x_j) \quad E \vdash x_k : A'' :: Res(x_k)}{E \vdash x_j.f{:=}x_k : A' :: T'} \quad \left\{ \begin{array}{l} A <: [f{:}A''] \\ A <: A' \\ T_{\mathsf{fupd}}(x_j, f, x_k) \subseteq T' \end{array} \right\}$$

Table 5.2: Rules for Abadi-Leino program logic. Definitions of predicate symbols Res, T_{obj}, T_{fsel} and T_{fupd} can be found in Table 2.3 on page 20.

$$w_p^{q,\gamma} : \{1..(p + \#\gamma)\} \to \{1..(q + \#\gamma)\}$$

$$w_p^{q,\gamma}(i) \stackrel{\text{def}}{=} \begin{cases} i & i \le p \\ i + (q - p) & i > p \end{cases}$$

$$\sigma_p^{q,\gamma} : \{1..(p + 1 + \#\gamma)\} \to \{1..(q + \#\gamma)\}$$

$$\sigma_p^{q,\gamma}(i) \stackrel{\text{def}}{=} \begin{cases} i & i \le p + 1 \\ i + (q - p - 1) & i > p + 1 \end{cases}$$

Table 5.3: Some basic renamings

$$\phi{\downarrow}s \stackrel{\text{def}}{=} \phi \quad \text{for no } X \text{ occurring free in } \phi$$

$$X{\downarrow}s \stackrel{\text{def}}{=} X \quad \text{for } X \in \vec{X}$$

$$(L_0 \lhd x \rhd L_1){\downarrow}s \stackrel{\text{def}}{=} \begin{cases} L_0{\downarrow}s & s(x) = \texttt{tt} \\ L_1{\downarrow}s & s(x) = \texttt{ff} \end{cases}$$

$$(\sigma L){\downarrow}s \stackrel{\text{def}}{=} \sigma(L{\downarrow}s)$$

$$(L_0; L_1){\downarrow}s \stackrel{\text{def}}{=} (L_0{\downarrow}s); (L_1{\downarrow}s)$$

$$(\exists_\sigma L){\downarrow}s \stackrel{\text{def}}{=} \exists_\sigma(L{\downarrow}s)$$

Supposing $\Phi \equiv \bigwedge_i v_i \Rightarrow e_{0,i} = e_{1,i} \ \wedge \ \bigwedge_i v_i', L_i \subseteq e_i'$,

$$\begin{aligned} \Phi{\downarrow}s \stackrel{\text{def}}{=} & \quad \bigwedge\{e_{0,i}{\downarrow}s = e_{1,i}{\downarrow}s \mid s \models v_i\} \\ & \wedge \ \bigwedge\{L_i{\downarrow}s = e_i'{\downarrow}s \mid s \models v_i'\} \end{aligned}$$

and for $\Psi \equiv \exists\vec{X}.\Phi \wedge \Psi_0$,

$$\Psi{\downarrow}s \stackrel{\text{def}}{=} \Phi{\downarrow}s$$

Table 5.4: "Evaluating" a verification condition at a specific switching s. This is an auxiliary device required to define the dependency graph of a VC.

For $\Phi \equiv (\bigwedge_i v_i \Rightarrow e_{0,i} = e_{1,i}) \wedge (\bigwedge_i v'_i, L_i \subseteq e'_i)$,

$$E_s(\Phi) \stackrel{\text{def}}{=} \{(X,Y) \mid e_0 = e_1 \in \Phi{\downarrow}s,\ X \text{ free in } e_0,\ Y \text{ free in } e_1\}$$
$$R_s(\Phi) \stackrel{\text{def}}{=} \{(X,Y) \mid L \subseteq e' \in \Phi{\downarrow}s,\ X \text{ free in } L,\ Y \text{ free in } e'\}$$

and for $\Phi \equiv \bigwedge_i T_i \subseteq X_i$,

$$E_s(\Phi) \stackrel{\text{def}}{=} \emptyset$$
$$R_s(\Phi) \stackrel{\text{def}}{=} \{(Y, X_i) \mid T_i \subseteq X_i \in \Phi,\ Y \text{ free in } T_i\} \cup \{(X,X) \mid T_i \subseteq X_i \in \Phi,\ \mu X.T' \text{ is a subexpression of } T_i\} \ .$$

And in either case,

$$G_0(N,\Phi) \stackrel{\text{def}}{=} (N, (E_s(\Phi)), (R_s(\Phi)))$$

and for $\Psi \equiv \exists \vec{X}.\Phi \wedge \Psi_0$,

$$G_0(\Psi) \stackrel{\text{def}}{=} G_0(\vec{X}, \Phi) \ .$$

Using these definitions, we define the dependency graph $D(\Psi)$, by

$$D(\Psi) \stackrel{\text{def}}{=} G_0(\Psi)^{\text{dg}}$$

and the transitive dependency graph $\mathsf{DG}(\Psi)$, by

$$\mathsf{DG}(\Psi) \stackrel{\text{def}}{=} D(\Psi)^{\text{tdg}}$$

Table 5.5: Dependency graph for constraints. For the two different forms of Φ, we define, for each s, two relations E_s, R_s which form the two families $(E_s), (R_s)$ used to define the triple $G_0(N, \Phi)$. Note that E_s need not be an equivalence relation.

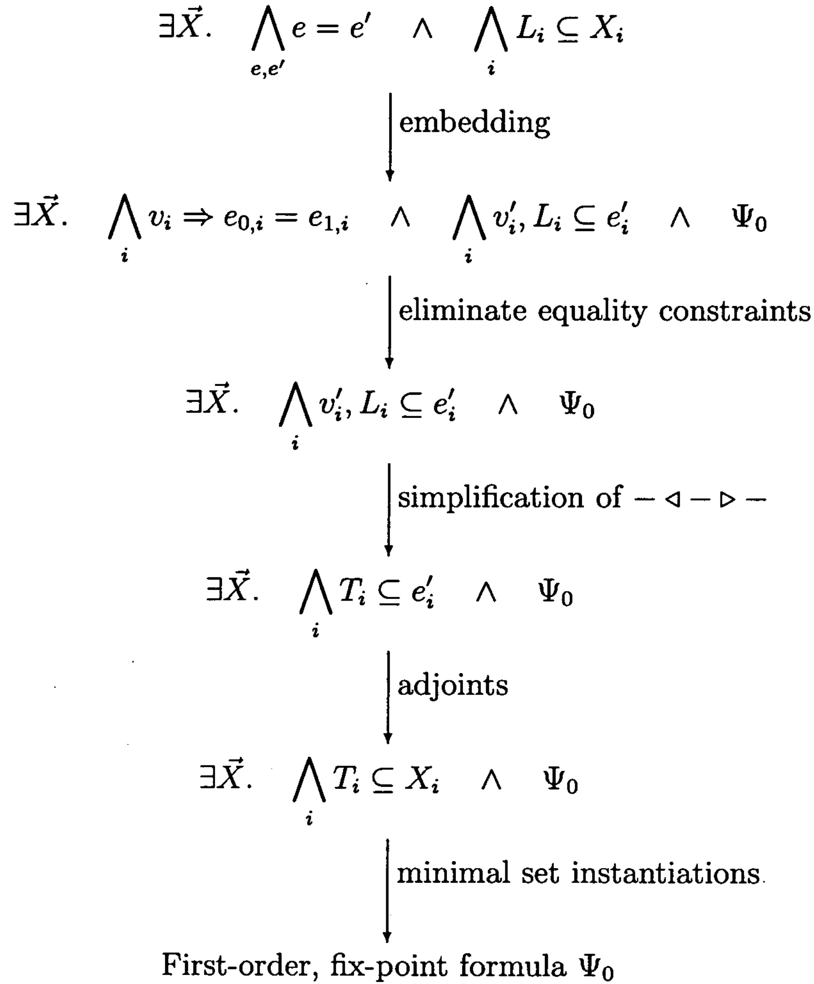


Table 5.6: Schematic representation of VC simplification algorithm. In the first step, we choose ε for each v_i, v_i' to transform the generated VC in to the form suitable our simplification rules. We first apply the equality elimination rules in Table 5.11, until they have all been eliminated. Then we eliminate all $- \lhd - \rhd -$ expressions using the rules in Table 5.8. Then we apply the rules in Table 5.9 to transform the VC into a form suitable for the rules in Table 5.10.

$$\frac{}{\sigma(\sigma' e) \leadsto (\sigma \circ \sigma')e} \qquad \text{(rw-renren)}$$

$$\frac{}{\sigma(e_0 \lhd x \rhd e_1) \leadsto (\sigma e_0) \lhd x \rhd (\sigma e_1)} \qquad \text{(rw-renif)}$$

Table 5.7: Expression rewriting. These rules allow us to pull renamings inside of $- \lhd - \rhd -$ expressions.

For $v_t \stackrel{\text{def}}{=} v, x{=}\mathtt{tt}$ and $v_f \stackrel{\text{def}}{=} v, x{=}\mathtt{ff}$,

$$\frac{\exists \vec{X}.\ \ v, L \subseteq e_0 \lhd x \rhd e_1 \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v_t, L \subseteq e_0 \quad \wedge \quad v_f, L \subseteq e_1 \quad \wedge \quad \Phi}\ x \text{ not in } v \qquad \text{(if-elimr)}$$

$$\frac{\exists \vec{X}.\ \ v, x{=}\mathtt{tt}, v', L \subseteq e_0 \lhd x \rhd e_1 \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v, x{=}\mathtt{tt}, v', L \subseteq e_0 \quad \wedge \quad \Phi} \qquad \text{(if-betar1)}$$

$$\frac{\exists \vec{X}.\ \ v, x{=}\mathtt{ff}, v', L \subseteq e_0 \lhd x \rhd e_1 \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v, x{=}\mathtt{ff}, v', L \subseteq e_1 \quad \wedge \quad \Phi} \qquad \text{(if-betar2)}$$

Writing $L(L')$ to denote $L''[L'/X]$ for some L'' with exactly one occurrence of X,

$$\frac{\exists \vec{X}.\ \ v, L(L_0 \lhd x \rhd L_1) \subseteq e \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v_t, L(L_0) \subseteq e \quad \wedge \quad v_f, L(L_1) \subseteq e \quad \wedge \quad \Phi}\ x \text{ not in } v \qquad \text{(if-eliml)}$$

$$\frac{\exists \vec{X}.\ \ v, x{=}\mathtt{tt}, v', L(L_0 \lhd x \rhd L_1) \subseteq e \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v, x{=}\mathtt{tt}, v', L(L_0) \subseteq e \quad \wedge \quad \Phi} \qquad \text{(if-betal1)}$$

$$\frac{\exists \vec{X}.\ \ v, x{=}\mathtt{ff}, v', L(L_0 \lhd x \rhd L_1) \subseteq e \quad \wedge \quad \Phi}{\exists \vec{X}.\ \ v, x{=}\mathtt{ff}, v', L(L_1) \subseteq e \quad \wedge \quad \Phi} \qquad \text{(if-betal2)}$$

Table 5.8: Simplification of $- \lhd - \rhd -$ expressions. Here we, may assume that our verification conditions have form $\exists \vec{X}. (\bigwedge_i v'_i, L \subseteq e'_i) \wedge \Psi_0$ where Ψ_0 is first-order formula.

$$\frac{\exists\vec{X}.\quad T \subseteq \sigma X_1 \quad \wedge \quad \Phi}{\exists\vec{X}.\quad (\exists_\sigma T) \subseteq X_1 \quad \wedge \quad \Phi} \qquad \text{(adj-ren)}$$

$$\frac{\exists\vec{X}.\quad T_0 \subseteq X_1 \quad \wedge \quad T_1 \subseteq X_1 \quad \wedge \quad \Phi}{\exists\vec{X}.\quad (T_0 \vee T_1) \subseteq X_1 \quad \wedge \quad \Phi} \qquad \text{(collate)}$$

Table 5.9: Miscellaneous simplification rules. Here we assume that our VCs have form $\exists\vec{X}.\,(\bigwedge_i T_i \subseteq e_i) \wedge \Psi_0$ where Ψ_0 is a first-order formula, and there are no occurrences of $- \lhd - \rhd -$ subexpressions (except for maybe in Ψ_0).

$$\frac{\exists\vec{X}X.\quad T \subseteq X \quad \wedge \quad \Phi}{\exists\vec{X}.\quad \Phi[\mu X.T/X]} \quad X \text{ free in } T \qquad \text{(freeinst)}$$

$$\frac{\exists\vec{X}X.\quad T \subseteq X \quad \wedge \quad \Phi}{\exists\vec{X}.\quad \Phi[T/X]} \quad X \text{ not free in } T \qquad \text{(notfreeinst)}$$

$$\frac{\exists\vec{X}X.\quad \Phi}{\exists\vec{X}.\quad \Phi} \quad X \text{ not free in } \Phi \qquad \text{(falseinst)}$$

Table 5.10: Minimal set instantiations. Here we assume that our VCs have form $\exists\vec{X}.\,(\bigwedge_i T_i \subseteq X_i) \wedge \Psi_0$ where Ψ_0 is a first-order formula, each $X \in \vec{X}$ occurs at most once to the right of a constraint $T \subseteq X$, and there are no occurrences of $- \lhd - \rhd -$ subexpressions (except maybe in Ψ_0).

$$\frac{\exists \vec{X}.\quad v \Rightarrow e = e' \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v \Rightarrow e' = e \quad \wedge \quad \Phi} \qquad \text{(eq-sym)}$$

$$\frac{\exists \vec{X}.\quad v \Rightarrow e_0 = e_1 \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v \Rightarrow e_0' = e_1 \quad \wedge \quad \Phi} \; \{e_0 \rightsquigarrow e_0'\} \qquad \text{(eq-resp)}$$

$$\frac{\exists \vec{X}X.\quad \varepsilon \Rightarrow X = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad \Phi[e/X]} \; X \text{ not free in } e \qquad \text{(eq-inst)}$$

$$\frac{\exists \vec{X}.\quad \varepsilon \Rightarrow X = e \quad \wedge \quad \Phi}{\exists \vec{X}X'.\quad \varepsilon \Rightarrow X = e[X'/X] \quad \wedge \quad \Phi} \; X \text{ free in } e \qquad \text{(eq-idem)}$$

For π_1, π_2 the pullback of σ, τ,

$$\frac{\exists \vec{X}.\quad v \Rightarrow \sigma X_1 = \tau X_2 \quad \wedge \quad \Phi}{\exists \vec{X}Z.\quad v \Rightarrow X_1 = \pi_1 Z \quad \wedge \quad v \Rightarrow X_2 = \pi_2 Z \quad \wedge \quad \Phi} \qquad \text{(eq-pullback)}$$

For ρ the equaliser of σ, τ,

$$\frac{\exists \vec{X}.\quad v \Rightarrow \sigma X_1 = \tau X_1 \quad \wedge \quad \Phi}{\exists \vec{X}Z.\quad v \Rightarrow X_1 = \rho Z \quad \wedge \quad \Phi} \qquad \text{(eq-equaliser)}$$

$$\frac{\exists \vec{X}.\quad v \Rightarrow \sigma X_1 = \tau \phi \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v \Rightarrow X_1 = (\sigma^{-1} \circ \tau)\phi \quad \wedge \quad \Phi} \; \sigma^{-1} \circ \tau \text{ a renaming} \qquad \text{(eq-const)}$$

$$\frac{\exists \vec{X}.\quad v \Rightarrow e_0 \lhd x \rhd e_1 = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v, x{=}\mathtt{tt} \Rightarrow e_0 = e \quad \wedge \quad v, x{=}\mathtt{ff} \Rightarrow e_1 = e \quad \wedge \quad \Phi} \; x \text{ not in } v, \qquad \text{(eq-if-elim)}$$

$$\frac{\exists \vec{X}.\quad v, x{=}\mathtt{tt}, v' \Rightarrow e_0 \lhd x \rhd e_1 = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v, x{=}\mathtt{tt}, v' \Rightarrow e_0 = e \quad \wedge \quad \Phi} \qquad \text{(eq-beta1)}$$

$$\frac{\exists \vec{X}.\quad v, x{=}\mathtt{ff}, v' \Rightarrow e_0 \lhd x \rhd e_1 = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v, x{=}\mathtt{ff}, v' \Rightarrow e_1 = e \quad \wedge \quad \Phi} \qquad \text{(eq-beta2)}$$

$$\frac{\exists \vec{X}.\quad v, x{=}\mathtt{tt} \Rightarrow X_1 = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v \Rightarrow X_1 = e \lhd x \rhd X_1 \quad \wedge \quad \Phi} \qquad \text{(eq-if-intro1)}$$

$$\frac{\exists \vec{X}.\quad v, x{=}\mathtt{ff} \Rightarrow X_1 = e \quad \wedge \quad \Phi}{\exists \vec{X}.\quad v \Rightarrow X_1 = X_1 \lhd x \rhd e \quad \wedge \quad \Phi} \qquad \text{(eq-if-intro2)}$$

Table 5.11: Equality elimination rules.

Chapter 6

A prototype VCG implementation and examples

The VCG algorithm described in Chapter 5 (and its component type inference algorithm described in Chapter 4) has been partly implemented and used to verify some examples. Overall, the experience gained from these has reaffirmed our thesis that such tools make verification more feasible. However, the implementation has also highlighted some areas in which further progress can be made.

6.1 Prototype implementation

For proof of concept, the VCG algorithm has been implemented in prototype form. In order to minimise effort, we made the following implementation decisions (amongst others): (1) we used Objective Caml [LDG+01] as our choice of programming language since it provides functional programming features, such as algebraic data types, as well as parser generator tools in the spirit of lex and yacc; (2) more significantly from the functional point of view, we chose not to implement renaming of bound variables in programs and instead insist that, in the input programs, all bound variables are pairwise distinct; and (3) we incorporated definitions into the input language by way of macros, but instead of supporting definitions directly in our implementation, we delegate this task to the C prepro-

cessor. As a consequence of these last two decisions, the input scripts look more complicated than necessary.

In several places, performance has been compromised in favour of correctness; code has been implemented to reflect, as closely as possible, the pseudocode presented in previous chapters. In the case of type inference (Chapter 4) we still maintain asymptotically cubic-time performance, however, in practice, it is poorer than one would expect.

In the type inference component, careful analysis of the code suggests a limitation in the data structure used. Specifically, for the closure algorithm, the data structure **HG** has been implemented as an adjacency matrix (which suffices to ensure asymptotic performance). Recall that in the closure algorithm, we require to repeatedly iterate along columns. However, in practice, the matrix is typically sparse, and so there is an obvious optimisation which can improve performance.

Surprisingly (at least from the author's point of view) the instantiation part of the VCG algorithm shows good performance whereas the (transition relation) constraints generating component shows poor performance. Analysis of the program again suggests a shortcoming of a data structure implementation decision: we have implemented program occurrences as sequences of integers, but neglected to include information about the syntactic shape of the occurrence; as a consequence, during constraints generation, we must traverse the abstract syntax tree for each occurrence.

It appears the combination of these two performance bottlenecks plus the verbatim (macro) expansion of definitions in the dining philosophers example (giving a large syntax tree of more than 220 occurrences) explains why the time required for the VCG algorithm is of the order of 5 mins on a PIII 500MHz PC.

6.1.1 The back-end

The preceding chapters only describe the VCG algorithm in detail; we do not consider the precise details of how to deal with the resulting VCs. Perhaps naively, the author hoped that the resulting VCs would be succinct to the point where it might be possible to convince oneself of their validity, or at the very

least, it might be feasible to prove them in an interactive theorem prover such as PVS or similar. However, in practice, the VCs are too verbose for that.

Thus we made experiments with feeding the results into an automatic theorem prover, in this case, SPASS [W+99], an automated theorem prover for (sorted) first-order logic with equality.

Though in [AL97, AL98], transition relations are presented as first-order formulae and their validity in terms of first-order derivability, this exact formulation is not so convenient for our purposes. Specifically, in loc. cit., stores are introduced as a pair of (formal) function symbols $\grave{\sigma}, \acute{\sigma}$ and a pair of (formal) predicate symbols $a\grave{l}loc$, $a\acute{l}loc$; thus it is not possible to quantify over them. To allow us to quantify over stores, we consider stores amongst our first-order objects and introduce a (formal) function symbol `update` and a formal predicate symbol `lookup` (since stores are partial we choose this to be a predicate as opposed to a function) with the following intended interpretations: for stores `s`, location `l`, field `f` and value `v`, the term `update(s,l,f,v)` is a store which takes value `v` at `(l,f)` and otherwise is the same as `s` and the formula `lookup(s,l,f,v)` is true whenever `(l,f)` is in the domain of `s` and takes value `v` at the point. (We note that Leino, in his logic with recursive object types [Lei98], also considers stores as first-order objects, but therein `lookup` is a function symbol and partiality is introduced, in essence, by a predicate.)

In fact, we take advantage of sorts which are available to us in SPASS, and the outline of a typical SPASS input file (namely that for the gcd example) can be found in Section B.1.

Having settled on this formulation of transition relations in sorted first-order logic, we implemented a pretty-printer that can output concrete SPASS syntax, provided there are no fix-point expressions. At the time of writing, SPASS does not have primitive support for abbreviations[1]. The author first attempted to introduce formal predicate and function symbols for the abbreviations and axioms to provide their definitions. However, limited experiments seemed to cause SPASS

[1]Through personal communications, the implementors of SPASS assure the author that version 2.0d does have primitive support for abbreviations, but unfortunately, as yet, the documentation has not been updated.

```
#define Doit if y.f < y.g then                \
                 y.g := y.g - y.f; y.m()      \
             else if y.g < y.f then           \
                 y.f := y.f - y.g; y.m()      \
             else                             \
                 y.f

#define gcd [ f=1, g=1, m(y) = Doit :: [ [],"INV",1 ] ]

#define a let x = gcd in                      \
          x.f := 426;                         \
          x.g := 792;                         \
          x.m()

a :: [ [],"GCDSPEC",0 ]
```

Figure 6.1: Concrete syntax for Euclid's algorithm, as introduced in Chapter 3.

to "run away" whilst trying to prove theorems.

Reluctantly, the author resorted to expanding definitions of predicates which would otherwise be abbreviated (e.g. $Res, T_{\mathsf{fsel}}, T_{\mathsf{fupd}}$). Due, once again, to a limitation of an implementation decision, this was most easily integrated into the pretty-printing code. Alas, the VCs in concrete syntax are somewhat bloated. However, the resulting performance from SPASS seems acceptable.

6.2 Euclid's algorithm(s)

Recall the greatest common divisor (gcd) example introduced in Chapter 3. We display the example in its concrete syntax, suitable for parsing by our prototype implementation, in Figure 6.1. Note that we use the C preprocessor to implement definitions and this is reflected by the syntax `#define`, and line continuation using '\'. The program gcd is the same program as seen previously except we now annotate the method body with an invariant `INV`. In our concrete syntax, we represent annotations, which are partial functions mapping paths to transition relations, as lists of triples `p,T,a`, where `p` is a path (list of field names `f` and

method names `m()`), `T` a string literal and `a` an integer arity. (In the prototype implementation, the arity is explicitly specified purely as an aid for debugging purposes.)

For the comfort of the reader, we resort to the typical typeset presentation, as found in Chapter 3. To emphasise the presence of the annotation, we present the whole example again:

$$a \stackrel{\text{def}}{=} (\ \textit{let } x{=}\mathrm{gcd} \ \textit{ in } x.f{:=}426;\ x.g{:=}792;\ x.m()\ )::(\!|a|\!)$$

where

$$(\!|a|\!) \stackrel{\text{def}}{=} \mathrm{pos}(n_1), \mathrm{pos}(n_2) \implies r = \mathrm{gcd}(n_1, n_2)$$

and where (program) abbreviation gcd is defined

$$\begin{array}{lll} \mathrm{gcd} \stackrel{\text{def}}{=} [& f{=}1, g{=}1, & \\ & m{=}\varsigma(y)\ \textit{if } y.f < y.g \textit{ then} & \\ & \qquad\qquad y.g{:=}y.g - y.f;\ y.m() & \\ & \qquad \textit{else if } y.g < y.f \textit{ then} & \\ & \qquad\qquad y.f{:=}y.f - y.g;\ y.m() & \\ & \qquad \textit{else } y.f & \\ & \qquad ::(\varepsilon \mapsto I) &] \end{array}$$

We now write I for the invariant `INV` and $(\!|a|\!)$ for `GCDSPEC`. The predicate symbol pos is intended to be interpreted as "is positive".

Using our implementation, computing $\mathrm{Cons}(\varepsilon, a)$ gives a second-order VC Ψ_1, with 60 (existentially quantified) higher-order variables, 6 equality constraints and 50 inequality constraints. We can eliminate all higher-order variables by applying our simplification rules to obtain a VC Ψ_2.

As explained earlier, the generated VCs in concrete SPASS syntax can be large and unwieldly: for this example, the two VCs were about 3100 and 1000 characters in length. Therefore, for the purposes of this presentation, we use specific facts about AL transition relations, for example

$$(\mathit{Res}(n);_x U(x)) = U(n) \ ,$$

and simplify Ψ_2 further, to obtain the following more readable formulae. We discuss, in Section 7.3, a generalisation of the simplification displayed above which could allow such simplifications to be applied automatically. Here we write the VC as an (implicit) conjunction of first-order formulae with $\grave{\sigma}$ and $\acute{\sigma}$ denoting the initial and final stores and r for the result value.

$$\left(\begin{array}{l}\grave{\sigma}(y,f) \not< \grave{\sigma}(y,g) \wedge \grave{\sigma}(y,g) \not< \grave{\sigma}(y,f) \wedge \\ T_{\mathrm{fsel}}(y,f)\end{array}\right) \subseteq I$$

$$\left(\begin{array}{l}\grave{\sigma}(y,f) \not< \grave{\sigma}(y,g) \wedge \grave{\sigma}(y,g) < \grave{\sigma}(y,f) \wedge \\ T_{\mathrm{fupd}}(y,f,\grave{\sigma}(y,f) - \grave{\sigma}(y,g));I\end{array}\right) \subseteq I$$

$$\left(\begin{array}{l}\grave{\sigma}(y,f) < \grave{\sigma}(y,g) \wedge \\ T_{\mathrm{fupd}}(y,g,\grave{\sigma}(y,g) - \grave{\sigma}(y,f));I\end{array}\right) \subseteq I$$

$$T_{\mathrm{obj}}(1,1);_x T_{\mathrm{fupd}}(x,f,n_1);T_{\mathrm{fupd}}(x,g,n_1);I[x/y] \subseteq (\!|a|\!) \ .$$

Recall that $T \subseteq T'$ is an abbreviation for

$$\forall \vec{x},\grave{\sigma},\acute{\sigma},r.T(\vec{x},\grave{\sigma},\acute{\sigma},r) \implies T'(\vec{x},\grave{\sigma},\acute{\sigma},r) \ .$$

When we define I by

$$I \stackrel{\mathrm{def}}{=} \begin{array}{l}\mathrm{pos}(\grave{\sigma}(y,f)) \wedge \\ \mathrm{pos}(\grave{\sigma}(y,g))\end{array} \implies \begin{array}{l} r = \acute{\sigma}(y,f) \wedge r = \acute{\sigma}(y,g) \wedge \\ r = \mathrm{gcd}(\grave{\sigma}(y,f),\grave{\sigma}(y,g)) \ , \end{array}$$

SPASS can successfully prove Ψ_2, if we add the following axioms about gcd and subtraction,

$$\forall_{\mathbb{Z}} x,y. \quad x < y \implies \mathrm{pos}(y-x) \tag{6.1}$$

$$\forall_{\mathbb{Z}} x,y. \quad x \not< y, y \not< x \implies x = y \tag{6.2}$$

$$\forall_{\mathbb{Z}} x,y. \quad x < y, \mathrm{pos}(x), \mathrm{pos}(y) \implies \mathrm{gcd}(x,y) = \mathrm{gcd}(x, y-x) \tag{6.3}$$

$$\forall_{\mathbb{Z}} x,y. \quad y < x, \mathrm{pos}(x), \mathrm{pos}(y) \implies \mathrm{gcd}(x,y) = \mathrm{gcd}(x-y, y) \tag{6.4}$$

$$\forall_{\mathbb{Z}} x. \quad \mathrm{gcd}(x,x) = x \tag{6.5}$$

$$f \neq g \ . \tag{6.6}$$

Axiom 6.6 can be generated by our implementation, though in this particular case, it saves little effort. SPASS required about 15 seconds to find a proof of Ψ_2, on a 500MHz Pentium III.

One may argue that what we have really shown is just a special case of the correctness of the gcd program: namely the case where we compute gcd(426, 792). It would be more satisfying if we were able to prove $\gcd(n_1, n_2)$ where n_1, n_2 are metavariables since, after all, we can prove correctness for gcd(426, 792) simply by evaluation! We argue that morally we have proved the latter: SPASS has no primitive notion of arithmetic and the constants 426, 792 really are formal constants and, other than the fact that they are natural numbers, SPASS can use no other assumption about these constants. The reason why we have shown what appears to be a more specific result is again a limitation of our implementation; the parser parses `426` and `792` as integers whereas `n1` and `n2` are parsed as free variables, whose types cannot be inferred; clearly this is a limitation that can be (and should be) removed.

As a demonstration of the robustness of this VCG approach, consider a modification of the algorithm where the second branch of the nested if-then-else statement performs an (inline) swap, before performing a recursive method call:

$$\mathit{let}\ z{=}y.f\ \mathit{in}\ (y.f{:=}y.g;\ y.g{:=}z;\ y.m())\ .$$

The generated VC cannot be discharged by SPASS using only those axioms as previously displayed. Careful analysis of the new algorithm reveals why this is so; the generated VC can be automatically discharged by SPASS if we also add the symmetry axiom

$$\forall_{\mathbb{Z}} x, y.\quad \gcd(x, y) = \gcd(y, x)\ , \tag{6.7}$$

which, one would agree, is expected. Alternatively, if instead we modify the second branch so that the swap is performed by invoking a sibling method, viz,

$$y.\mathit{swap}();\ y.m()\ ,$$

(and we add a new sibling method *swap*) the resulting VC can also be automatically discharged by SPASS with the symmetry axiom (and conversely, cannot without). For reference, the version with the inline swap (resp. swap method) produced a 2nd-order VC with 64 (resp. 70) higher-order variables, 6 (resp. 9) equality constraints and 54 (resp. 55) inequality constraints.

Clearly, the two versions with swapping are implementations of the same algorithm. We argue that this is experimental evidence of the VCG algorithm being indifferent to syntactically different implementations of the same algorithm. In contrast, from personal experience, such a syntactic change requires a significant amount of work to construct a proof using the embedding presented in Chapter 3 alone.

6.3 Dining philosophers revisited, (almost) push-button-style

Recall the dining philosophers example introduced in Chapter 3. We reproduce the code where it differs. The code Fork for creating fork objects stays the same. We modify philosopher objects by representing its state as an integer field (taking values 0,1,2,3 as opposed to a field representing the number of forks held by the philosopher and a boolean field representing whether the philosopher is hungry or not.) In light of this change, the method body for the "tick" method of a philosopher is defined:

```
phil_tick(s) ≝ if (s.state == 0) {
                  if (s.fork1.try_pick_up()) {s.n_forks = 1; false}
                  else {false}
                } else if (s.state == 1) {
                  if (s.fork2.try_pick_up())
                    {s.n_forks = 2; s.hungry = false; false}
                  else {false}
                } else if (s.state == 2) {
                  s.fork2.put_down(); s.n_forks = 1; false
                } else {
                  s.fork1.put_down(); s.n_forks = 0; s.hungry = true; false
                }
```

We keep the definitions of LRPhil and RLPhil the same, but we must change the definition Phil as follows.

$$\mathsf{Phil}(fork_1, fork_2) \stackrel{\text{def}}{=} [\ \mathsf{state}{=}0, \mathsf{fork1}{=}fork_1, \mathsf{fork2}{=}fork_2,\ \mathsf{tick}{=}\varsigma(s)\mathsf{phil_tick}(s)\ ]$$

Finally we introduce annotations:

$$\begin{aligned}
\mathsf{Table} \stackrel{\text{def}}{=} \text{let}\ \ & fk_1 = \mathsf{Fork}{::}\psi_1,\\
& fk_2 = \mathsf{Fork}{::}\psi_2,\\
& fk_3 = \mathsf{Fork}{::}\psi_3,\\
& ph_1 = \mathsf{LRPhil}(fk_2, fk_3){::}\psi_4,\\
& ph_2 = \mathsf{RLPhil}(fk_3, fk_1){::}\psi_5,\\
& ph_3 = \mathsf{LRPhil}(fk_1, fk_2){::}\psi_6,\\
\text{in}\ \ & [\ \mathsf{f1}{=}fk_1, \mathsf{f2}{=}fk_2, \mathsf{f3}{=}fk_3,\\
& \mathsf{p1}{=}ph_1, \mathsf{p2}{=}ph_2, \mathsf{p3}{=}ph_3,\\
& \mathsf{tick1}{=}\varsigma(s)\ (s.\mathsf{p1.tick}()){::}\psi_t,\\
& \mathsf{tick2}{=}\varsigma(s)\ (s.\mathsf{p2.tick}()){::}\psi_t,\\
& \mathsf{tick3}{=}\varsigma(s)\ (s.\mathsf{p3.tick}()){::}\psi_t\ ]\\
& {::}\psi_7\\
{::}\psi' &\\
{::}\psi &
\end{aligned}$$

Just for clarification: ψ_7 annotates the inner-most object creation, ψ' annotates the outer-most let construct (i.e., all the executable code in the definition of Table) and ψ annotates the code and annotation ψ'.

In this example, since there are no loops, one would expect to require only one annotation, namely the annotation to tell the VCG what specification we would like to prove. We recall from Chapter 3, that we require to prove that each tick method preserves some invariant I, and furthermore, after creation of the table object, the store satisfies the invariant I. Thus we would dispense with all

annotations other than ψ, which would have the form

$$\psi \stackrel{\text{def}}{=} \begin{pmatrix} \varepsilon & \mapsto I(\acute{\sigma}) \\ \mathsf{tick1}, \mathsf{tick2}, \mathsf{tick3} & \mapsto I(\grave{\sigma}) \subseteq I(\acute{\sigma}) \end{pmatrix}$$

and, writing A for $(\!|\mathsf{Table}|\!)$, the resulting VCs would have form

$$A \subseteq I(\acute{\sigma}) \tag{6.8}$$

$$A_{\mathsf{tick}j} \subseteq (I(\grave{\sigma}) \subseteq I(\acute{\sigma})) \tag{6.9}$$

for $j = 1..3$. However, due to the size of A, $A_{\mathsf{tick}j}$, these VCs that are too "hard" to prove: given our interface with SPASS, during the proof search, we run out of memory for (6.8) and even after several days there is no proof for (6.9).

And so, we introduce further annotations, which in this case, behave as hints to SPASS, in the form of lemmata. So we define the annotations as follows.

$$\begin{aligned}
\psi_1 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_1(r, \acute{\sigma})) \\
\psi_2 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_1(fk_1, \grave{\sigma}) \subseteq S_2(fk_1, r, \acute{\sigma})) \\
\psi_3 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_2(fk_1, fk_2, \grave{\sigma}) \subseteq S_3(fk_1, fk_2, r, \acute{\sigma})) \\
\psi_4 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_3(fk_1, fk_2, fk_3, \grave{\sigma}) \subseteq S_4(fk_1, fk_2, fk_3, r, \acute{\sigma})) \\
\psi_5 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_4(fk_1, fk_2, fk_3, ph_1, \grave{\sigma}) \subseteq S_5(fk_1, fk_2, fk_3, ph_1, r, \acute{\sigma})) \\
\psi_6 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_5(fk_1, fk_2, fk_3, ph_1, ph_2, \grave{\sigma}) \\
&\qquad\qquad \subseteq S_6(fk_1, fk_2, fk_3, ph_1, ph_2, r, \acute{\sigma})) \\
\psi_7 &\stackrel{\text{def}}{=} (\varepsilon \mapsto S_6(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, \grave{\sigma}) \\
&\qquad\qquad \subseteq S_6(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, r, \acute{\sigma})) \\
\psi_t &\stackrel{\text{def}}{=} (\varepsilon \mapsto T_t(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, \grave{\sigma}, \acute{\sigma}, r)) \\
\psi' &\stackrel{\text{def}}{=} \begin{pmatrix} \varepsilon & \mapsto \exists fk_1, fk_2, fk_3, ph_1, ph_2, ph_3. \\ & \qquad S_7(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, r, \acute{\sigma}) \\ \mathsf{tick1}, \mathsf{tick2}, \mathsf{tick3} & \mapsto \exists fk_1, fk_2, fk_3, ph_1, ph_2, ph_3. \\ & \qquad T_t(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, \grave{\sigma}, \acute{\sigma}, r) \end{pmatrix} \\
\psi &\stackrel{\text{def}}{=} (\varepsilon \mapsto I_t(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, \acute{\sigma}))
\end{aligned}$$

where the predicates S_j describes the store after creation of the jth object, viz,

$$\begin{aligned} S_j(x_1, \dots x_j, \sigma) \stackrel{\text{def}}{=} & \quad \bigwedge_{1 \le i < k \le j} x_i \ne x_k \\ & \wedge \bigwedge_{1 \le i \le \min(j,3)} x_i.\mathsf{on_table} = \mathtt{tt} \\ & \wedge \bigwedge_{4 \le i \le j} (x_i.\mathsf{state} = 0 \wedge x_i.\mathsf{fork1} = x_{f(i)} \wedge x_i.\mathsf{fork2} = x_{g(i)}) \\ f(i) & \stackrel{\text{def}}{=} \min(\{1,2,3\} - \{i\}) \\ g(i) & \stackrel{\text{def}}{=} \max(\{1,2,3\} - \{i\}) \end{aligned}$$

and where the transition relation T_t describing the state change resulting from executing Table is defined

$$T_t(fk_1, fk_2, fk_3, ph_1, ph_2, ph_3, \grave{\sigma}, \acute{\sigma}, r) \stackrel{\text{def}}{=}$$
$$I_t(fk_1, fk_2, fk_3, ph_1, ph_2, \grave{\sigma}) \subseteq I_t(fk_1, fk_2, fk_3, ph_1, ph_2, \acute{\sigma})$$

and where the invariant I_t which each tick method preserves is defined as InvTable is, in Chapter 3.

Thus annotation $\psi(\varepsilon)$ specifies the desired transition relation for the program, and annotations $\psi(\mathsf{tick}j)$ specify the desired transition relation for the tick methods. In the case of the hints, ψ_t is a lemma for VC (6.9) and ψ_i ($i = 1..7$) and $\psi'(\varepsilon)$ are lemmata for VC (6.8).

After the addition of the lemmata, VC (6.9) becomes the following VCs. For $j = 1..3$,

$$(\!|s.\mathsf{p}j.\mathsf{tick}()|\!) \subseteq \psi_t(\varepsilon) \tag{6.10}$$
$$(\exists fk_1, fk_2, fk_3, ph_1, ph_2, ph_3.\psi_t(\varepsilon)) \subseteq \psi'(\mathsf{tick}j) \ . \tag{6.11}$$

And VC (6.8) becomes the following VCs, writing ψ_j for $\psi_j(\varepsilon)$ ($j = 1..7$):

$$(\!|\mathsf{Fork}|\!) \subseteq \psi_j \qquad \text{for } j = 1..3 \tag{6.12}$$
$$(\!|\mathsf{LRPhil}|\!) \subseteq \psi_j \qquad \text{for } j = 4, 6 \tag{6.13}$$
$$(\!|\mathsf{RLPhil}|\!) \subseteq \psi_5 \tag{6.14}$$
$$(\!|[\mathsf{f1}{=}\cdots]|\!) \subseteq \psi_7 \tag{6.15}$$
$$(\psi_1; (\psi_2; (\psi_3; (\psi_4; (\psi_5; (\psi_6; \psi_7)))))) \subseteq \psi'(\varepsilon) \tag{6.16}$$
$$\psi'(\varepsilon) \subseteq \psi(\varepsilon) \ . \tag{6.17}$$

VC	Total time	Memory required
(6.10) ($j = 1, 2, 3$)	9h26m, 10h45m, 66h51m	35MB, 35MB, 62MB
(6.11) ($j = 1, 2, 3$)	0.31s, 0.31s, 0.30s	819KB, 819KB, 819KB
(6.12) ($j = 1, 2, 3$)	0.33s, 0.44s, 0.52s	744KB, 754KB, 765KB
(6.13) ($j = 4, 6$)	14s, 27s	3MB, 6.5MB
(6.14)	14s	4MB
(6.15)	28s	7.4MB
(6.16)	0.35s	792KB
(6.17)	2.5s	1MB

Table 6.1: Performance summary of dining philosophers VC proof search using SPASS v2.0d. These results are as reported by SPASS, run on a Dell PowerEdge 1400 (Intel Pentium III 933MHz, 896MB RAM) running Linux 2.2.19.

Each of these VCs can be proved by SPASS, with only the addition of two axioms (as is required by AL) stating that (1) the field and (2) the method name constants are distinct. These axioms correspond to Axiom 6.6 of the previous example and, again, can be automatically generated by our prototype implementation. The results are summarised in Table 6.1; however, the reader is warned against taking these results on face value. These performance figures were obtained after tweaking the SPASS input. In certain cases, SPASS can prove a harder, more general theorem using fewer resources. For example, for VCs (6.12, 6.13, 6.14), performance is significantly improved if we force SPASS not to expand the definitions of S_j. (This can be done by introducing each S_j as a formal predicate symbol.) As an extreme example, the performance reported in Table 6.1 for VC (6.17), is possible only because, in the definitions of predicates S_7 and I_t, we can factor out $\bigwedge_{1 \leq i < k \leq 7} x_i \neq x_k$ as a formal predicate symbol. Initial attempts to find a proof without this factoring resulted in SPASS running for over 4000mins and still not finding a proof.

Admittedly, it is disappointing that performance is so poor, especially considering it takes 66 hours to find a proof for a VC derived from (6.9). However,

we note that this is an unusual development using the VCG. Since we know in advance (from our experiments detailed in Chapter 3) that the program is correct and, furthermore, provable using the chosen specifications, we were willing to let the computer continue searching even after several days. In contrast, VC (6.8) simply cannot be proved using our interface with SPASS, on available computing resources and so we were forced to introduce hints in the form of more annotations. We believe a similar exercise is possible for (6.9).

But we should point out that we have found a push-button solution to proving correctness of the dining philosophers example. Though it takes less time to check a proof script constructed using theorem provers such as LEGO and PVS and the embedding of Chapter 3, the interactive nature of its construction means this is certainly more expensive than the VCG approach.

Finally, concerning the tweaks required in the SPASS input script, it may be possible to eliminate this by using a back-end theorem prover with primitive support for abbreviations and also better interfacing with our prototype implementation.

For reference, the same example in concrete syntax as parsed by our prototype implementation can be found in Section B.2. Similarly, the input to SPASS for VC (6.10) can be found in Section B.3.

6.4 Conclusions

From personal experience, the prototype implementation of the VCG algorithm was certainly worthwhile. It allowed us to attempt two examples (Euclid's algorithms and the dining philosophers) that would otherwise be infeasible using a pencil-and-paper work-out, since the VCG algorithm generates such a large number of constraints. In the case of Euclid's algorithms, we obtained compelling evidence that this approach to verification of a concrete implementation of an algorithm is, at least to a small extent, independent of the implementation.

However, the implementation also reveals where further work is required. In the case of the dining philosophers, the large and cumbersome verification condi-

tions create genuine performance issues with regards to automatic verification. It appears we only simplify the structural aspects of finding an AL proof; further, non-structural simplifications are still possible. On the other hand, this example also allowed us to showcase the flexibility of annotations: it was possible to provide lemmata as stepping stones for SPASS via extra annotations.

Chapter 7

Conclusions and further work

This thesis proposed that feasible object-oriented program verification requires tools to: (1) automatically check proofs; and (2) automatically infer large parts of proofs. We demonstrated how such tools can be implemented in the case of a specific program logic, AL.

Specifically, we presented a generic method of embedding AL into a theorem prover; from this exercise, we automatically inherited machine-checkable "proofs" (at the very least proof scripts). The style of embedding is notable for its use of both higher-order abstract syntax (HOAS) and a direct embedding of the assertion logic into the metalogic itself. These decisions allowed us to inherit functionality such as variable-management and proof tactics from the theorem prover.

To justify the correctness of this implementation, we used a proof technique proposed in [Hof99] by constructing a categorical model of the metalanguage and arguing semantically.

We then presented, in detail, a type inference algorithm, for AL, which infers the structural aspects of an AL proof. The remaining aspect, namely that relating to dynamic behaviour of programs, was addressed by developing a verification condition generator (VCG). Because in AL method bodies are verified *exactly once* (since we have an internalisation of lemmata via assumptions and a cut—let—rule), it appears that the traditional definition by recursion over syntax is not possible. Instead, our original definition proceeded by simplification

of a second-order verification condition obtained by computing constraints on unknown assertions.

We gave experimental evidence that:

1. this approach is independent of concrete implementations of the same algorithm;

2. it is possible to reduce verification of some programs, including those with loops, to providing invariants; and

3. if necessary, such as in the case of the dining philosophers, providing lemmata.

It was possible to provide both invariants and lemmata using the unified facility of annotations, which become part of the program. A particular consequence is that all the difficult aspects, i.e. aspects requiring human ingenuity, of verification are localised in succinct, extended program source; the invariants and lemmata are not hidden within some large proof script, as is likely the case if using an embedding as described in Chapter 3 with an interactive theorem prover, such as LEGO and PVS, for building proofs.

However, in no way does the author claim that we have finally reached the goal of feasible verification of object-oriented programs. There is certainly much further work possible: especially with regards to (1) supporting more language features, (2) changing the underlying program logic (AL), and (3) improving back-end support (i.e. support for discharging the resulting VCs).

7.1 Supporting more language features

The underlying language of AL is, as previously mentioned, limited. It is believed that features such as explicit parameters for methods and object cloning are straightforward to add (see locus classicus). A more challenging feature is that of recursive object types.

7.1.1 Recursive object types

Consider the standard implementation of linked lists: nodes containing a datum and a pointer (reference) to the next node. Supposing A is the type of such a node and field n stores the next-pointer, then A must satisfy

$$A <: [n{:}A] \ ,$$

if we were to write useful programs with such linked lists. (Given certain programs, it is possible to determine an upper bound on the length of linked lists and thus, in AL, choose A satisfying a finite approximation of the above recursive equation. However such programs defeat the purpose of linked lists!) There is no such A in AL that satisfies the above equation.

This problem has already been considered by Leino who presents a logic (LRO) in [Lei98]. In loc. cit. the subtype relation, which in AL is defined in a *structural type matching* style in the sense of [AC93], has been substituted in favour of a *name matching* style definition. Specifically, the subtype relation is defined as the reflexive, transitive closure of a binary relation $\leq_0$ over types, which is provided explicitly by the programmer. Furthermore, subtyping is no longer covariant along methods.

From the pragmatic point of view, e.g. using the logic as a foundation for verifying programs of a class-based language such as Java, these restrictions are of little consequence, but from the modular verification point of view, it introduces further problems since we must know the whole typing environment (i.e. the type names, the ground subtype relation and the fields and methods of the types). We suggest that this logic is less modular by presenting the following example.

Let T_1 be a type, and suppose we can prove a statement about a program c of type T_1, provided we assume it has a method m, satisfying the following specification

$$m : T_1 \to U_1 :: R_1 \ ,$$

i.e. the return type—static specification—of m is U_1 and its transition relation is R_1. Suppose we later compose this program c, into a larger program $P(c)$, and

we want to use c where a program of type T_0 is expected. As it stands, unless T_0 exists in c and T_1 is a subtype of T_0, this is not possible, even if the shapes of T_0, T_1 suggest one can be a subtype of the other (because their names do not match). A possible work-around is to further insist in the enlarged typing environment that $T_1 \leq T_0$ (i.e. T_1 is a subtype of T_0). This is still not enough since program $P(x)$ might use method m of program x, and the proof (of correctness of $P(x)$) requires the assumption that

$$m : T_0 \rightarrow U_0 :: R_0 \ ,$$

for specifications U_0, R_0, which we may assume without loss of generality to be weaker than U_1, R_1 (for otherwise, in general, c cannot be considered to be of type T_0). We cannot directly combine the two different typing environments, since there will be a clash between the two specifications of m. We cannot strengthen T_0, T_1 since in $P(x)$, we may create an object of type T_0, and thus break the proof. Neither can we weaken U_1, R_1 since we would have a weaker specification of m, and so the proof for c may no longer "go through" (since the method body of m, may recursively invoke m, and now the assumption for m is weaker).

Leino addresses this point in the Section 7 (Limitations of the logic) of loc. cit. He states that if the programmer who authored T_1 is "willing to let his clients rely on the stronger properties" of his implementation, then he can rename the method m as n, say, and then create a stub method m that invokes n, and method n has the stronger specification. In fact, the programmer might have to do this anyway, regardless of whether or not he is willing to provide a stronger specification to the client. Since the method body of m may recursively invoke m and the assumptions are now weaker, the programmer may no longer be able to prove that his implementation satisfies the weaker specification of m.

Of course, it should be possible to formulate a logic with recursive object types and subtyping defined by structural type matching and which is covariant along methods. However, it is unclear exactly how the subtyping rules should be formulated. We believe that the subtyping rules found in [AC93, AC96] may be a good place to start.

In [AC93], the syntactic rules required to define recursive subtyping by struc-

tural type matching appear to be quite complicated. In contrast, subtyping of types defined as finite state automata (like in Chapter 4 and [PWO97, Pal95]) degenerates to inclusion of languages recognised by automata, which is decidable. However automata inclusion does not directly help us in the case of the VCG algorithm: recall the second-order VC is obtainable because we can rewrite $A <: A'$ as an equivalent collection of constraints on the component transition relations; in the case where A, A' are recursive, the equivalence as found in Chapter 5 gives us an infinite number of constraints! And so, it appears a syntactic characterisation of subtyping is still required.

Initial experiments suggest that it might be very easy to apply our VCG algorithm to LRO: the type inference algorithm degenerates and can be implemented simply as a function defined by recursion over program syntax, since now all types are named; and we get far fewer constraints, since there is no covariant subtyping.

7.1.2 Further language features

To be able to verify programs written in current object-oriented programming languages, we must consider further extensions such as classes, inheritance and access control (e.g. private, protected and public access modifiers in C++ classes). With the addition of extra features, one must consider the feasibility of accurately working out the underlying theory of a verification logic. AL as it stands, even with its comparatively limited set of features, is already quite a handful when it comes to working with metatheory; when constructing and checking proofs (of metatheory such as soundness and completeness of the logic), there are already enough cases to make the process tedious, and potentially prone to error[1]. Adding extra cases to provide further features in the language only adds further weight to these concerns.

Not so much as to lighten the effort, but more towards increasing reliability of metatheory, embedding the program logic in a theorem prover, for the purposes of proving metatheorems, in the style of Kleymann [Kle98] and von Oheimb [Ohe01]

[1]The metatheory in this thesis relating to properties of AL is a mild reworking of what has appeared in locus classicus, and thus the author hopes this instils further confidence in the reader.

has much to be commended. A potentially useful approach is to define a simple language which is expressive enough for the more advanced features to be encoded simply as syntactic sugar. Leino's language, Ecstatic [Lei97], is designed as such a language, that is, as a target for translations of languages with more advanced features. In this case, rules for the sugared syntax can either be derived from the logic rules for the basic syntax (which, if possible, also provides concrete evidence towards how complete the logic is), or, at a lower-level, directly from the operational semantics. The author believes features such as while loops, classes and inheritance could possibly be introduced in this way.

In the case of access control, the author believes this should be supported in the basic language and basic verification rules, since this is morally a strengthening of the type system in the sense that, by annotating certain fields and methods, we have fewer legal programs.

7.2 Changing the underlying program logic

As promised in Chapter 2, we now discuss some limitations of AL, and possible approaches to addressing them. As with extensions to the underlying programming language, it may be possible that some of these extra features of the verification logic could be implemented as syntactic sugar.

7.2.1 Incompleteness

As mentioned in Chapter 2, there are some well-known limitations of AL. For example, the logic is incomplete. However, it appears the particular incompleteness example of locus classicus,

$$\begin{aligned} a &\stackrel{\text{def}}{=} \textit{let } y{=}\textit{true in } [m = \varsigma(z)y] \\ b_1 &\stackrel{\text{def}}{=} \textit{let } x{=}a \textit{ in } y.m() \ , \end{aligned}$$

can be easily fixed.

It is suggested that there is "insufficient interaction" between A and T in judgements $E \vdash a : A :: T$, "particularly in the rule for let".

An attempt to prove $\vdash b_1 : Bool :: r = \texttt{tt}$ suggests we need to prove subterm $[m = \varsigma(z)y]$ has specification $[m : \varsigma(z) : Bool :: r = \texttt{tt}]$. However, when building the subproof for $[m = \varsigma(z) = y]$, the only assumption we have about y is that it has specification *Bool*; we do not have as an assumption, $y = \texttt{tt}$.

Consider the following, semantic-style explanation. Suppose we have some denotational semantics for AL. Then the denotation $[\![A]\!]$ of a specification A should be a "set" of values. In a judgement

$$x{:}A \vdash b : B :: U \ ,$$

we have, to the left of the turnstile, our assumptions. In AL, assumptions are only of the form $x{:}A$. We suggest that the language of specifications is not expressive enough: the collection of value-sets $[\![A]\!]$ for all specifications A is too coarse. For if we could find a specification A such that $[\![A]\!] = \{\texttt{tt}\}$ then the incompleteness example should no longer be a problem.

The difficulty is finding a suitable way to extend the syntax for specifications that gives us better granularity. As a first step, we propose a specification should be a pair $A \wedge P$ where A is a specification defined along the lines as for AL, and P is a predicate (set of values), and in particular should not refer to the store. Intuitively, the soundness proof should require minimal alterations, since these new specifications still only describe static properties of values. (Whereas the case may be different if P can mention stores, since the store changes during execution, whereas values do not.) We propose the following rules to allow interaction between dynamic and static specifications:

$$\frac{E \vdash a : A \wedge P :: T}{E \vdash a : A \wedge P :: T \wedge P(r)}$$
$$\frac{E \vdash a : A \wedge P :: T}{E \vdash a : A \wedge Q :: T} \quad \{T \subseteq Q(r)\} \ .$$

Soundness and other properties of AL, extended in this manner, have not been checked.

7.2.2 Data abstraction

Recall the dining philosophers examples found in Chapters 3 and 6. Since we (1) formulate an invariant I for the table object, (2) prove the methods preserve this invariant, and (3) prove, after creation the invariant holds, it may appear that I is an invariant of the object itself. Alas this is not the case since AL does not have data abstraction: there is no way to prevent the fields of the table object being changed in an arbitrary fashion and thus violating the invariant.

The lack of data abstraction is compounded by the following limitation of the specification language; transition relations can only be expressed in terms of the fields of an object. So, for example, suppose we want to specify a mean calculator, i.e. an object that can compute the mean of a sequence of numbers. A typical, naïve concrete implementation stores the cumulative total and the number of elements entered so far, and then divides the former by the latter to compute the mean. Ideally we would like a more abstract specification, for example, in terms of a list which stores all the elements entered so far.

Some specification formalisms allow abstract fields which have no computational content, but allow for specification. Modelling our example after this, we can implement a mean calculator as follows:

$$
\begin{aligned}
mc \stackrel{\text{def}}{=} [\;\; & aelts = \varepsilon, \\
& ctot = 0, \\
& cnum = 0, \\
& arg = 0, \\
& add = \varsigma(y)\; y.aelts := \mathrm{cons}(y.arg, y.aelts); \\
& \qquad\qquad\;\; y.ctot := y.arg + y.ctot; \\
& \qquad\qquad\;\; y.cnum := y.cnum + 1 \\
& mean = \varsigma(y)\; y.ctot/y.cnum \qquad\qquad]\;.
\end{aligned}
$$

Intuitively, $aelts$ is an abstract field, which in this case is a list of elements, $ctot$, $cnum$ are concrete fields storing the cumulative total and the element count, and add, $mean$ are methods that respectively add an element and compute the mean. Since there is no syntactic facility in AL to distinguish abstract and concrete fields, here we employ programming discipline to separate code that manip-

ulates abstract fields so that it may easily be omitted, for example, during compile time. There is an intended coupling invariant for this object, namely, $I(\sigma, y) \stackrel{\text{def}}{=} \sigma(y, ctot) = \text{sum}(\sigma(y, aelts)) \wedge \sigma(y, cnum) = \text{length}(\sigma(y, aelts))$. Clearly, *add* preserves this coupling invariant, and in particular adds an element into the list *aelts*; also, providing the coupling invariant holds, method *mean* computes the mean of the elements in *aelt*; and the object initially satisfies the coupling invariant. Thus it is possible to prove

$$\vdash mc : A :: I(\acute{\sigma}, r).$$

where

$$A \stackrel{\text{def}}{=} [\ add : \varsigma(y)[] :: I(\grave{\sigma}, y) \implies I(\acute{\sigma}, y) \wedge \acute{\sigma}(y, aelts) = \text{cons}(\grave{\sigma}(y, arg), \grave{\sigma}(y, aelts))\ ,\quad mean : \varsigma(y)\text{num} :: I(\grave{\sigma}, y) \implies r = \text{mean}(\grave{\sigma}(y, aelts))\ ]\ .$$

But, this is not a subspecification of A' defined by

$$A' \stackrel{\text{def}}{=} [\ add : \varsigma(y)[] :: \acute{\sigma}(y, aelts) = \text{cons}(\grave{\sigma}(y, arg), \grave{\sigma}(y, aelts))\ ,\quad mean : \varsigma(y)\text{num} :: r = \text{mean}(\grave{\sigma}(y, aelts))\ ]\ ,$$

an abstract specification of a mean calculator, since the latter has a stronger specification for method *mean*.

We argue that the specification of *mean* is too weak in A. The problem is that it would be unsound to simply assume the coupling invariant holds, precisely because we do not have data abstraction.

Thus we suggest, by adding data abstraction into the programming language with support in the logic, we already increase the expressivity of the specifications.

7.2.3 Support for abstract fields

Notice in the previous example, we applied programmer discipline to make a distinction between abstract and concrete fields. Furthermore, we provided explicit code to update the abstract field in order to be able to satisfy the abstract specification. It would be better if we didn't need to provide *code* to satisfy

the abstract specification: for example, it would be interesting to consider specifications as code (i.e. postulate that specification T, when considered as code, satisfies specification T).

7.3 Improving back-end support

In Chapter 6, we applied, by hand, further simplifications to the generated VC to obtain readable formulae. Clearly such simplifications, if done automatically, can improve performance, maybe significantly, for post-VCG verification (e.g. verification using SPASS). As it is now, when pretty printing to SPASS, the prototype implementation expands the definitions of abbreviations such as $T;U$, $Res(x)$, etc., thus losing information. Possible implementations include further simplification as another stage of the VCG, or, preferably, propagating the structure of the VCs (i.e. output the VCs without expanding the abbreviations) to a back-end prover which can use sensible axioms or lemmata to perform the simplifications.

We suggest these simplifications can make use of the following observation. The transition relation $T = Res(x)$ (and also for $T = T_{\text{fsel}}(x, f), T_{\text{fupd}}(x, f, y)$) is total and functional in the sense that for all $\grave{\sigma}$: there are $\acute{\sigma}, r$ (*total*) such that $T(\grave{\sigma}, \acute{\sigma}, r)$; and furthermore $\acute{\sigma}, r$ are unique (*functional*). Moreover, if $T, U(x)$ are total and functional transition relations, then so is $T;U$. More generally, we note that if T is total and functional, for arbitrary U, since

$$(T;U)(\grave{\sigma}, \acute{\sigma}, r) \stackrel{\text{def}}{=} \exists \breve{\sigma}, y. T(\grave{\sigma}, \breve{\sigma}, y) \wedge U(y)(\breve{\sigma}, \acute{\sigma}, r)$$

and $\breve{\sigma}, y$ are uniquely determined by T, this is simply

$$(T;U)(\grave{\sigma}, \acute{\sigma}, r) \equiv U(y)(\breve{\sigma}, \acute{\sigma}, r) \ .$$

As a particular example, consider $T_{\text{fupd}}(x, f_1, y_1); \lambda z. T_{\text{fupd}}(z, f_2, y_2)$:

$$\begin{aligned} &(T_{\text{fupd}}(x, f_1, y_1); \lambda z. T_{\text{fupd}}(z, f_2, y_2))(\grave{\sigma}, \acute{\sigma}, r) \\ \equiv\ & T_{\text{fupd}}(x, f_2, y_2)(\grave{\sigma}[(x, f_1) \mapsto y_1], \acute{\sigma}, r) \\ \equiv\ & \acute{\sigma} = \grave{\sigma}[(x, f_1) \mapsto y_1, (x, f_2) \mapsto y_2] \wedge r = x \ , \end{aligned}$$

since $T_{\mathrm{fupd}}(x, f, y)(\grave{\sigma}, \acute{\sigma}, r) \stackrel{\mathrm{def}}{=} \acute{\sigma} = \grave{\sigma}[(x, f) \mapsto y] \wedge r = x$. In contrast, by only expanding definitions:

$$\begin{aligned}&(T_{\mathrm{fupd}}(x, f_1, y_1); \lambda z. T_{\mathrm{fupd}}(z, f_2, y_2))(\grave{\sigma}, \acute{\sigma}, r)\\ \equiv\ &\exists \breve{\sigma}, z. \breve{\sigma} = \grave{\sigma}[(x, f_1) \mapsto y_1] \wedge z = x \wedge \acute{\sigma} = \breve{\sigma}[(z, f_2) \mapsto y_2] \wedge r = z\ .\end{aligned}$$

Though the two expressions are logically equivalent, for the purposes of automatic checking, the former is preferred.

Appendix A

Proofs

A.1 Proofs for Chapter 3

A.1.1 Proof of Lemma 1 on page 58

Proof We define

$$\begin{aligned} X''' \stackrel{\text{def}}{=} \; & \{(x,0) \mid x \in X\} \cup \\ & \{(x,1) \mid x \in X' \setminus \mathsf{ran}(f)\} \cup \\ & \{(x,2) \mid x \in X'' \setminus \mathsf{ran}(g)\} \end{aligned}$$

$$S''' \stackrel{\text{def}}{=} \begin{cases} (x,0) \mapsto S(x) \\ (x,1) \mapsto S'(x) \\ (x,2) \mapsto S''(x) \end{cases}$$

We define E''' similarly to S'''.

$$u(x) \stackrel{\text{def}}{=} \begin{cases} (y,0) & \text{if there is } y \text{ such that } f(y) = x \\ (x,1) & \text{otherwise} \end{cases}$$

$$v(x) \stackrel{\text{def}}{=} \begin{cases} (y,0) & \text{if there is } y \text{ such that } g(y) = x \\ (x,2) & \text{otherwise} \end{cases}$$

Certainly, we know $X'''S'''E''' \in \mathbb{X}$. It remains to show that $u : X'S'E' \to X'''S'''E'''$, and similarly $v : X''S''E'' \to X'''S'''E'''$. First, we note that u is a

function since f is injective, and u is injective since f is a function. Recalling our definition of morphism in $\mathbb{X}$, this now amounts to showing that for all $x \in X'$, $S'''(u(x)) = S'(x)$. Case $x = f(y)$ for some y: since $f : XSE \to X'S'E'$, we know that $S(y) = S'(f(y)) = S'(x)$, and so $S'''(u(x)) = S'''(y, 0) = S(y) = S'(x)$. Otherwise: $S'''(u(x)) = S'''(x, 1) = S'(x)$. A similar argument shows $E'''(u(x)) = E'(x)$. And for $x \in X''$, a similar argument shows $S'''(v(x)) = S''(x)$ and $E'''(v(x)) = E''(x)$.

To show that $uf = vg$, suppose $x \in X$. We calculate $uf(x) = (x, 0)$ since f is injective. Similarly, $vg(x) = (x, 0)$ since g is injective. □

A.1.2 Proof of Theorem 2 on page 64

Proof We prove each property in turn.

- We show that for $U, V \in \mathbf{Pred}(F)$, the pointwise inclusion ordering forms a boolean algebra. That is, we show that there exist finite meets and joins. Explicitly, we define, pointwise,

$$\begin{aligned} (U \vee V)_{XSE} &\stackrel{\text{def}}{=} U_{XSE} \cup V_{XSE} \\ (U \wedge V)_{XSE} &\stackrel{\text{def}}{=} U_{XSE} \cap V_{XSE} \\ (\overline{U})_{XSE} &\stackrel{\text{def}}{=} \{x \in F^{(01)}{}_{XSE} \mid x \notin U_{XSE}\} \ . \end{aligned}$$

  It is straightforward to show that $U \vee V, U \wedge V, \overline{U}$ are predicates.

  Given $m : F \leftarrow G$, it is also straightforward to show that

$$\begin{aligned} \mathbf{Pred}(m)(U \vee V) &= \mathbf{Pred}(m)(U) \vee \mathbf{Pred}(m)(V) \\ \mathbf{Pred}(m)(U \wedge V) &= \mathbf{Pred}(m)(U) \wedge \mathbf{Pred}(m)(V) \\ \mathbf{Pred}(m)(\overline{U}) &= \overline{\mathbf{Pred}(m)(U)} \ . \end{aligned}$$

- Define

$$Prop \stackrel{\text{def}}{=} (1, \nabla\{\mathtt{ff}, \mathtt{tt}\}) \ .$$

Thus, for $F \in \mathcal{D}$,

$$\begin{aligned}
&\mathcal{D}(F, Prop) \\
&\cong \hat{\mathbb{I}}(F^{(0)}, Prop^{(0)}) \times \hat{\mathbb{X}}(F^{(01)}, Prop^{(1)}) && \text{by definition of homset in } \mathcal{D} \\
&\cong \hat{\mathbb{X}}(F^{(01)}, \nabla\{\mathtt{ff}, \mathtt{tt}\}) && \text{since } Prop^{(0)} = \mathbf{1} \ .
\end{aligned}$$

And so, to show that **Pred** is representable, it suffices to show $\mathbf{Pred}(F) \cong \hat{\mathbb{X}}(F^{(01)}, \nabla\{\mathtt{ff}, \mathtt{tt}\})$.

Suppose $U \in \mathbf{Pred}(F)$, $XSE \in \mathbb{X}$ and $x \in F^{(01)}{}_{XSE}$. Let $A \subseteq F^{(01)}{}_{XSE}$, let $c_A : F^{(01)}{}_{XSE} \to \{\mathtt{ff}, \mathtt{tt}\}$ be the characteristic function of A, viz:

$$c_A(x) = \begin{cases} \mathtt{tt} & \text{if } x \in A \\ \mathtt{ff} & \text{otherwise} \ . \end{cases}$$

Now define a family of functions $(m_{XSE} : F^{(01)}{}_{XSE} \to \{\mathtt{ff}, \mathtt{tt}\})_{XSE}$, pointwise:

$$m_{XSE} \stackrel{\text{def}}{=} c_{U_{XSE}} \ .$$

We now show that $m \in \hat{\mathbb{X}}(F^{(01)}, \nabla\{\mathtt{ff}, \mathtt{tt}\})$. Thus we require to show m is a natural transformation, that is: for all $f : XSE \to X'S'E'$,

$$\begin{array}{ccc}
F^{(01)}{}_{XSE} & \xrightarrow{m_{XSE}} & \{\mathtt{ff}, \mathtt{tt}\} \\
{\scriptstyle F^{(01)}{}_f}\downarrow & \nearrow{\scriptstyle m_{X'S'E'}} & \\
F^{(01)}{}_{X'S'E'} & &
\end{array} \qquad (A.1)$$

commutes. Commutativity is an immediate consequence of the fact that U is a predicate.

Conversely, suppose $m \in \hat{\mathbb{X}}(F^{(01)}, \nabla\{\mathtt{ff}, \mathtt{tt}\})$. Define a family of subsets U, pointwise by

$$U_{XSE} \stackrel{\text{def}}{=} \{x \in F^{(01)}{}_{XSE} \mid m_{XSE} = \mathtt{tt}\} \ .$$

Since m is a natural transformation, Diagram A.1 commutes. Immediately, we conclude $x \in U_{XSE}$ iff $F^{(01)}{}_f(x) \in U_{X'S'E'}$, that is, $U \in \mathbf{Pred}(F)$.

- Let $m : G \to F$ be a morphism. We have a function $\forall_m : \mathbf{Pred}(G) \to \mathbf{Pred}(F)$ defined as before and it remains to show that it is right adjoint to $\mathbf{Pred}(m)$.

  Let $U \in \mathbf{Pred}(F)$ and $V \in \mathbf{Pred}(G)$.

  Suppose $U \leq (\forall_m V)$, that is, for all XSE, $U_{XSE} \subseteq (\forall_m V)_{XSE}$. Let $XSE \in \mathbb{X}$ and $x \in (\mathbf{Pred}(m)(U))_{XSE}$. Thus by definition of $\mathbf{Pred}(m)$, we know that $m_{XSE}(x) \in U_{XSE} \subseteq (\forall_m V)_{XSE}$ and since (i) $id : XSE \to XSE$, (ii) $x \in G^{(01)}{}_{XSE}$, and (iii) $m_{XSE}(x) = m_{XSE}(x)$, by definition of $\forall_m V$, we know that $x \in V_{XSE}$. Thus for all XSE, $(\mathbf{Pred}(m)(U))_{XSE} \subseteq V_{XSE}$, i.e., $\mathbf{Pred}(m)(U) \leq V$.

  Conversely, suppose $\mathbf{Pred}(m)(U) \leq V$, that is, for all XSE, we have $(\mathbf{Pred}(m)(U))_{XSE} \subseteq V_{XSE}$. Let $XSE \in \mathbb{X}$ and assume $x \in U_{XSE}$. Let $f : XSE \to X'S'E'$, $a \in G^{(01)}{}_{X'S'E'}$ and suppose $m_{X'S'E'}(a) = F^{(01)}{}_f(x) \in U_{X'S'E'}$, since U is a predicate. Thus $a \in (\mathbf{Pred}(m)(U))_{X'S'E'} \subseteq V_{X'S'E'}$, that is, $x \in (\forall_m V)_{XSE}$. And so we conclude for all XSE, $U_{XSE} \subseteq (\forall_m V)_{XSE}$, i.e., $U \leq \forall_m V$.

- Let XSE be a world, $A, B, C \in \mathcal{D}$, and $\pi_A : A \times C \to A$, $\pi_B : B \times C \to B$ and $\pi_C : B \times C \to C$ be projections. By the fact that $\mathbf{Pred}$ and $\forall$ form an adjunction (previous property), we know

$$U \leq \forall_{f\times\mathrm{id}}(\mathbf{Pred}(f \times \mathrm{id})(U))$$
$$\mathbf{Pred}(\pi_B)(\forall_{\pi_B}(U)) \leq U \ .$$

  So by transitivity,

$$\mathbf{Pred}(\pi_B)(\forall_{\pi_B}(U)) \leq \forall_{f\times\mathrm{id}}(\mathbf{Pred}(f \times \mathrm{id})(U))$$

  by adjunction and functor ppty of $\mathbf{Pred}$,

$$\mathbf{Pred}(\pi_B \circ (f \times \mathrm{id}))(\forall_{\pi_B} U) \leq \mathbf{Pred}(f \times \mathrm{id})(U)$$

  by ppty of products,

$$\mathbf{Pred}(f \circ \pi_A)(\forall_{\pi_B} U) \leq \mathbf{Pred}(f \times \mathrm{id})(U)$$

by adjunction and functor ppty of **Pred**,

$$\mathbf{Pred}(f)(\forall_{\pi_B} U) \leq \forall_{\pi_A}(\mathbf{Pred}(f \times \mathrm{id})(U)) \ .$$

Thus, in particular, $(\mathbf{Pred}(f)(\forall_{\pi_B} U))_{XSE} \subseteq (\forall_{\pi_A}(\mathit{pred}(f \times \mathrm{id})(U)))_{XSE}$.

To show the converse, $(\forall_{\pi_A}(\mathbf{Pred}(f \times \mathrm{id})(U)))_{XSE} \subseteq (\mathbf{Pred}(f)(\forall_{\pi_B} U))_{XSE}$, suppose $x \in (\forall_{\pi_A}(\mathbf{Pred}(f \times \mathrm{id})(U)))_{XSE} \subseteq A^{(01)}{}_{XSE}$. Let $X'S'E' \in \mathbb{X}$, $u : XSE \to X'S'E'$, $b \in (B \times C)^{(01)}{}_{X'S'E'}$ and suppose $\pi_B(b) = B^{(01)}{}_u(f_{XSE}(x))$. Now define $a \stackrel{\text{def}}{=} \langle A^{(01)}{}_u(x), \pi_C(b)\rangle \in (A \times C)_{X'S'E'}$. Since $x \in (\forall_{\pi_A}(\mathbf{Pred}(f \times \mathrm{id})(U)))_{XSE}$, and (i) $u : XSE \to X'S'E'$, (ii) $a \in (A \times C)_{X'S'E'}$, and (iii) $\pi_A(a) = x$, we know $a \in (\mathbf{Pred}(f \times \mathrm{id})(U))_{X'S'E'}$, i.e., $(f \times \mathrm{id})^{(01)}{}_{X'S'E'}(a) \in U_{X'S'E'}$. And thus

$$\begin{aligned}
b &= \langle \pi_B b, \pi_C b\rangle && \text{by definition of projection}\\
&= \langle B^{(01)}{}_u(f_{XSE}(x)), \pi_C(b)\rangle && \text{by supposition}\\
&= \langle f^{(01)}{}_{X'S'E'}(A^{(01)}{}_u(x)), \pi_C(b)\rangle && \text{by naturality}\\
&= (f \times \mathrm{id})^{(01)}{}_{X'S'E'}(a) && \text{by definition of } a\\
&\in U_{X'S'E'} && \text{by previous observation.}
\end{aligned}$$

Thus we have shown $f^{(01)}{}_{XSE}(x) \in (\forall_{\pi_B} U)_{XSE}$. Expanding the definition of $\mathbf{Pred}(f)$, that is precisely $x \in (\mathbf{Pred}(f)(\forall_{\pi_B}(U)))_{XSE}$.

□

A.2 Proofs for Chapter 4

A.2.1 Theorem 5 on page 108

A.2.1.1 Part 1

Proof We need to show that $\leq$ is reflexive, antisymmetric and transitive. It is clear that $\leq$ is reflexive. Furthermore, since $t \leq t'$ implies $t \supseteq t'$, we immediately deduce that $\leq$ is antisymmetric.

Finally, we consider t, t', t'' such that $t \leq t'$, $t' \leq t''$ and $\alpha f \in t''$. Thus $\varepsilon t''{\downarrow}\alpha f = t'{\downarrow}\alpha f$, and so $\alpha f \in t'$. Since $t \leq t'$, we have $t'{\downarrow}\alpha f = t{\downarrow}\alpha f$. Therefore

we conclude $t{\downarrow}\alpha f = t''{\downarrow}\alpha f$. It remains to show that $t \supseteq t''$ and this is immediate since $t \supseteq t'$, $t' \supseteq t''$ and $\supseteq$ is transitive. □

A.2.1.2 Part 3

Proof Assume $t \leq t'$.

We proceed by induction over the size of t'.

Suppose $b \in t'$ for some $b \in \mathcal{B}$. Since $t \supseteq t''$, we conclude that $b \in t$ also. Thus by properties of pretypes, we conclude that $t = b = t'$.

Suppose $\star \in t'$. By the characterisation of types, we can assume that $t' = [f_i{:}A_i{}^{i=1..k'}, m_j{:}B_j'{}^{j=1..\ell'}]$, where $A_i = t'{\downarrow}{\star}f_i$ and $B_j' = t'{\downarrow}{\star}m_j$. In particular $\star f_i \in t'$ for $1 \leq i \leq k'$, and since $t \supseteq t'$, it must be that $t{\downarrow}{\star}f_i = t'{\downarrow}{\star}f_i$. Also, since $t \supseteq t'$ and $\star m_j \in t'$, it follows that $\star m_j \in t$. That is to say $t = [f_i{:}A_i{}^{i=1..k}, m_j{:}B_j{}^{j=1..\ell}]$ for some $k \geq k'$ and $\ell \geq \ell'$, with $A_i = t{\downarrow}{\star}f_i$ and $B_j = t{\downarrow}{\star}m_j$.

It remains to show that $t{\downarrow}{\star}m_j = B_j \leq B_j' = t'{\downarrow}{\star}m_j$ for $1 \leq j \leq \ell'$. That is,

$$\alpha f \in t'{\downarrow}{\star}m_j \implies t'{\downarrow}{\star}m_j\alpha f = t{\downarrow}{\star}m_j\alpha f$$

and

$$t{\downarrow}{\star}m_j \supseteq t{\downarrow}{\star}m_j \ .$$

Suppose $\alpha f \in t'{\downarrow}{\star}m_j$. From this we deduce $\star m_j\alpha f \in t'$. And so

$$t'{\downarrow}{\star}m_j\alpha f = t{\downarrow}{\star}m_j\alpha f$$

since $t \leq t'$. By simply rewriting, we have

$$(t'{\downarrow}{\star}m_j){\downarrow}\alpha f = (t{\downarrow}{\star}m_j){\downarrow}\alpha f \ .$$

As required. Finally, since ${\downarrow}\alpha$ preserves $\subseteq$ and $t \supseteq t'$ we conclude $t{\downarrow}{\star}m_j \supseteq t'{\downarrow}{\star}m_j$

□

A.2.2 Theorem 6 on page 115

Proof Let H be a set of presolutions.

Define $h : G \to \Sigma^*$ pointwise by $h(u) = \bigcap_{k \in H} k(u)$.

Since P is closed under intersections, we know, in fact, $h : G \to P$.

We now proceed to check that h satisfies the remaining properties of presolution as defined in Section 4.5.1 on page 114.

Clearly, for b in B, for any $k \in H$, we have $k(b) = b$ and so $h(b) = b$.

Also, for u in C, we have $\star \in h(u)$ since for any $k \in H$ we have $\star \in k(u)$.

Now suppose $u \xrightarrow{\ell} v$. We have

$$\begin{aligned} h(v) &= \bigcap_{k \in H} k(v) && \text{by definition} \\ &= \bigcap_{k \in H} (k(u){\downarrow}\star\ell) && \text{since each } k \in H \text{ is a presolution} \\ &= (\bigcap_{k \in H} k(u)){\downarrow}\star\ell && \text{since } {\downarrow}\alpha \text{ distributes over intersection} \\ &= h(u){\downarrow}\star\ell && \text{by definition.} \end{aligned}$$

Conversely, suppose $u \in C$ and $\star\ell \in h(u)$. Since $h(u) = \bigcap_{k \in H} k(u)$, we certainly have $\star\ell \in k(u)$ for all $k \in H$. Any such k gives us some v such that $u \xrightarrow{\ell} v$, since k is a presolution.

Now suppose $u \leq v$. Since each $k \in H$ is a presolution, we know that $k(u) \leq k(v)$.

Consider $\alpha f \in h(u) = \bigcap_{k \in H} k(u)$. Thus, for any $k \in H$, we have $k(u){\downarrow}\alpha f = k(v){\downarrow}\alpha f$. And so we trivially conclude that $h(u){\downarrow}\alpha f = h(v){\downarrow}\alpha f$.

For any $k \in H$, since $k(u) \supseteq k(v)$, we have $\bigcap_{k \in H} k(u) \supseteq \bigcap_{k \in H} k(v)$. □

A.2.3 Theorem 7 on page 115

To prove this theorem, we need the following lemma.

Lemma 20 *Given a sequence α of length n in Σ_0^*, define $\alpha^\star$ in Σ^* to be a sequence of length $2n$ whose $2i$-th letter is the i-th letter of α and whose odd letters are all $\star$. If h is a presolution and $u \stackrel{\alpha}{\Rightarrow} v$ then*

$$h(u){\downarrow}\alpha^\star \leq h(v) \ ,$$

where $\alpha^\star$ is the sequence as defined above.

Proof We proceed by induction over the length of α.

Case ε. That is $u \overset{\leq}{\Rightarrow} v$ and so $h(u){\downarrow}\varepsilon = h(u) \leq h(v)$, by property 4 of presolutions.

Case $\ell\alpha$. So there are u_0, u_1 and u_2 such that

$$u \overset{\leq}{\Rightarrow} u_0 \overset{\ell}{\rightarrow} u_1 \overset{\leq}{\Rightarrow} u_2 \overset{\alpha}{\Rightarrow} v \ .$$

Since h is a presolution, by property 4 $h(u) \leq h(u_0)$, by property 3 $h(u){\downarrow}{\star}\ell = h(u_1)$, and by property 4 $h(u_1) \leq h(u_2)$. By our induction hypothesis, we also have $h(u_2){\downarrow}\alpha^\star \leq h(v)$. By Lemma 8 on page 108 and the previous observations, we have

$$h(u){\downarrow}{\star}\ell \leq h(u_0){\downarrow}{\star}\ell \leq h(u_2) \ .$$

And since ${\downarrow}\alpha$ preserves $\leq$ (Lemma 8 on page 108),

$$h(u){\downarrow}{\star}\ell\alpha^\star \leq h(u_2){\downarrow}\alpha^\star \leq h(v) \ .$$

Which is what we require, since $(\ell\alpha)^\star$ is precisely $\star\ell\alpha^\star$. □

Now we can prove Theorem 7.

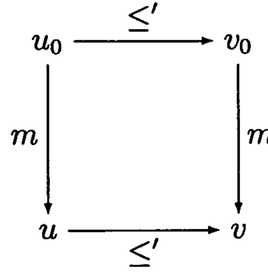
Proof Suppose h is a presolution of G and G' is the closure of G. We proceed by checking the four properties of presolution, with respect to G'.

Properties 1, 2 and 3 follow immediately from the fact that h is a presolution of G and its closure only makes $\leq$ bigger. It remains to check that property 4 holds, namely if $u \leq' v$ then $h(u) \leq h(v)$.

We will use the following fact. If $u \leq' v$ then either $u \leq v$ or $u \leq' v$ is obtained from one of the closure operations. Since G' can be obtained by applying these closure rules iteratively starting from G, we can proceed by induction and consider each one of these closure operations in turn.

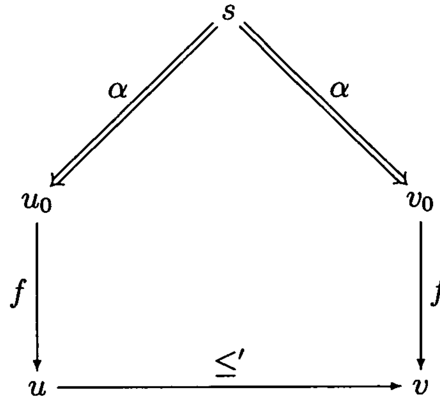
Since $(P, \leq)$ forms a partial order, the cases where $u \leq' v$ is obtained from the reflexive and transitive closure operations is immediate.

Suppose



From our induction hypothesis, we conclude $h(u_0) \leq h(v_0)$. And since h is a presolution, by property 3 we know that $h(u_0){\downarrow}{\star}m = h(u)$ and $h(v_0){\downarrow}{\star}m = h(v)$. And since ${\downarrow}\alpha$ preserves $\leq$ (Lemma 8 on page 108) we conclude that $h(u) \leq h(v)$.

Suppose



By our induction hypothesis and Lemma 20 on page 217, we have

$$h(s){\downarrow}\alpha^{\star} \leq h(u_0) \tag{A.2}$$

$$h(s){\downarrow}\alpha^{\star} \leq h(v_0) \ . \tag{A.3}$$

Since h is a presolution, by property 3, we know that ${\star}f \in h(u_0)$ and from equation A.2, we conclude that $h(s){\downarrow}\alpha^{\star}{\star}f = h(u_0){\downarrow}{\star}f$. Similarly, we conclude that $h(s){\downarrow}\alpha^{\star}{\star}f = h(v_0){\downarrow}{\star}f$. Thus $h(u_0){\downarrow}{\star}f = h(v_0){\downarrow}{\star}f$, and in particular, $h(u_0){\downarrow}{\star}f \leq h(v_0){\downarrow}{\star}f$. Therefore

$$h(u) = h(u_0){\downarrow}{\star}f \leq h(v_0){\downarrow}{\star}f = h(v) \ .$$

□

A.2.4 Theorem 9 on page 119

We have the following lemma.

Lemma 21 *Assuming that h is any presolution of G, if*

$$(0, s) \xrightarrow{\alpha} (0, u)$$

then

$$h(s){\downarrow}\alpha \supseteq h(u) \ .$$

Proof We recall the invariant property of our automaton and note that α must be of even length.

We proceed by induction over the number of transitions in $\mathcal{A}$. We only need to consider the following two cases.

Case $u_0 \in C$, $u_0 \leq u_1$ and $u_1 \xrightarrow{\ell} u$,

$$(0, s) \xrightarrow{\alpha} (0, u_0) \xrightarrow{\star} (1, u_0) \xrightarrow{\varepsilon} (1, u_1) \xrightarrow{\ell} (0, u)$$

Our induction hypothesis gives us $h(s){\downarrow}\alpha \supseteq h(u_0)$. Also, since $u_0 \leq u_1$, by property 4 of presolution, we have $h(u_0) \leq h(u_1)$ and in particular $h(u_0) \supseteq h(u_1)$. Thus we conclude

$$h(s){\downarrow}\alpha \supseteq h(u_1)$$

and since ${\downarrow}{\star}\ell$ preserves $\supseteq$, we have

$$h(s){\downarrow}\alpha{\star}\ell \supseteq h(u_1){\downarrow}{\star}\ell \ .$$

But since $u_1 \xrightarrow{\ell} u$ and h is a presolution, by property 3,

$$h(u) = h(u_1){\downarrow}{\star}\ell \subseteq h(s){\downarrow}\alpha{\star}\ell \ .$$

Case $u_0 \leq u$,

$$(0, s) \xrightarrow{\alpha} (0, u_0) \xrightarrow{\varepsilon} (0, u)$$

From the fact that $u_0 \leq u$ and h is a presolution, by property 4, we conclude that $h(u_0) \leq h(u)$, and in particular $h(u_0) \supseteq h(u)$. Our induction hypothesis gives us $h(s){\downarrow}\alpha \supseteq h(u_0)$ and the result follows by transitivity of $\supseteq$. □

Now we can prove Theorem 9 on page 119.

Proof Consider some α in $\mathcal{L}_s$, that is, starting from s, our automaton accepts α. We now show that $\alpha \in h(s)$.

We proceed by induction on the number of transition steps.

Supposing α is of even length, from Lemma 21 on the facing page, we deduce that $h(s){\downarrow}\alpha \supseteq h(u)$. Since h is a presolution, we know that ε is in $h(u)$, and hence is in $h(s){\downarrow}\alpha$. Thus $\alpha \in h(s)$.

Now suppose α is of odd length. We consider cases where the last transition is ε, $\star$ and $d \in B$ respectively.

Case last transition is ε. That is

$$(0, s) \xrightarrow{\alpha} (1, u_0) \xrightarrow{\varepsilon} (1, u) \ .$$

So by our induction hypothesis, $\alpha \in h(s)$.

Case $\alpha\star$, i.e. last step was $\star$. Thus α is of even length, $u \in C$ and

$$(0, s) \xrightarrow{\alpha} (0, u) \xrightarrow{\star} (1, u) \ .$$

The fact that $h(s){\downarrow}\alpha \supseteq h(u)$ follows from Lemma 21 on the preceding page. And together with the facts that $u \in C$ and h is a presolution, we know, by property 2, that

$$\star \in h(u) \subseteq h(s){\downarrow}\alpha \ .$$

That is, $\alpha\star \in h(s)$.

Case αd for some $d \in B$, i.e. last step was d. Thus

$$(0, s) \xrightarrow{\alpha} (0, d) \xrightarrow{d} (1, d) \ .$$

The Lemma 21 on the facing page allows us to conclude that $h(s){\downarrow}\alpha \supseteq h(d)$ and since h is a presolution, $d = h(d)$, and by our unfortunate abuse of notation, $d \in h(d)$. Thus $\alpha d \in h(s)$ as required. □

A.2.5 Theorem 10 on page 120

Proof Suppose G is well-formed. First of all, we observe that $\psi(s) \stackrel{\text{def}}{=} \mathcal{L}_s$ is a pretype for each s, by hypothesis. Hence $\psi : G \to P$. Now it remains to check that ψ satisfies the remaining properties of a presolution.

Consider b in B. By the definition of $\mathcal{A}$, we clearly have $b \in \psi(b)$. Since $\mathcal{L}_s$ is a pretype, it must be the case that $\psi(b) = b$.

Consider u in C. By the definition of $\mathcal{A}$, we clearly have $\star \in \psi(u)$.

Now consider $u \xrightarrow{\ell} v$. Certainly $\psi(u){\downarrow}\star\ell \supseteq \psi(v)$ since

$$(0,u) \xrightarrow{\star} (1,u) \xrightarrow{\ell} (0,v) \ .$$

Conversely, suppose $\beta \in \psi(u){\downarrow}\star\ell$, i.e., there are u_0, u_1 and v_1 such that

$$(0,u) \xrightarrow{\varepsilon} (0,u_0) \xrightarrow{\star} (1,u_0) \xrightarrow{\varepsilon} (1,u_1) \xrightarrow{\ell} (0,v_1) \xrightarrow{\beta} (i,v_1') \ .$$

That is $u \leq u_0$, $u_0 \leq u_1$ and $u_1 \xrightarrow{\ell} v_1$. Since G is closed we have $u \leq u_1$ since $\leq$ is transitive, and since we have $u \xrightarrow{\ell} v$ by assumption, by the closure rule corresponding to the diagram

$$\begin{array}{ccc} u & \xrightarrow{\leq} & u_1 \\ {\scriptstyle \ell}\downarrow & & \downarrow{\scriptstyle \ell} \\ v & \underset{\leq}{\dashrightarrow} & v_1 \end{array}$$

we know $v \leq v_1$. And so

$$(0,v) \xrightarrow{\varepsilon} (0,v_1) \xrightarrow{\beta} (i,v_1') \ ,$$

that is, $\beta \in \psi(v)$. Thus we conclude $\psi(u){\downarrow}\star\ell \subseteq \psi(v)$.

Now suppose $u \in C$ and $\star\ell \in h(u)$. That is there are u_0, u_1 and v_1 such that

$$(0,u) \xrightarrow{\varepsilon} (0,u_0) \xrightarrow{\star} (1,u_0) \xrightarrow{\varepsilon} (1,u_1) \xrightarrow{\ell} (0,v_1) \ .$$

And so, $u \leq u_0$, $u_0 \leq u_1$ and $u_1 \xrightarrow{\ell} v_1$. Since G is closed (thus, in particular $\leq$ is transitive), $u \leq u_1$ and since it is also well-formed, there must be some v such

that

$$\begin{array}{ccc} u & \xrightarrow{\leq} & u_1 \\ {\scriptstyle \ell}\,\vdots\downarrow & & \downarrow{\scriptstyle \ell} \\ v & & v_1 \end{array}$$

To show that ψ satisfies the last property of presolution, we assume that $u \leq v$ and are required show that $\psi(u) \subseteq \psi(v)$ and $\alpha f \in \psi(v)$ implies $\psi(u){\downarrow}\alpha f = \psi(v){\downarrow}\alpha f$. Since $u \leq v$, we know that the automaton can make the transition step

$$(0, u) \xrightarrow{\varepsilon} (0, v) \ ,$$

and we immediately conclude that $\psi(u) \supseteq \psi(v)$. Suppose $\alpha f \in \psi(v)$. Since ${\downarrow}\alpha f$ preserves $\subseteq$, we certainly know

$$\psi(u){\downarrow}\alpha f \supseteq \psi(v){\downarrow}\alpha f \ .$$

It now remains to show the converse: $\psi(u){\downarrow}\alpha f \subseteq \psi(v){\downarrow}\alpha f$. Since by supposition $\alpha f \in \psi(v)$, there are v_0 and v' such that

$$(0, v) \xrightarrow{\alpha} (1, v_0) \xrightarrow{f} (0, v') \ .$$

Now assume there are u_0 and u' such that

$$(0, u) \xrightarrow{\alpha} (1, u_0) \xrightarrow{f} (0, u') \ .$$

Thus, writing α' for α omitting all occurrences of $\star$, we have

$$\begin{array}{ccccc} u & \xRightarrow{\alpha'} & u_0 & \xrightarrow{f} & u' \\ {\scriptstyle \leq}\downarrow & & & & \uparrow\vdots{\scriptstyle \leq} \\ v & \xRightarrow[\alpha']{} & v_0 & \xrightarrow[f]{} & v' \end{array}$$

in G. The dashed arrow exists because G is closed. Hence $(0, v') \xrightarrow{\varepsilon} (0, u')$ is a valid transition, that is,

$$\psi(u){\downarrow}\alpha f \subseteq \psi(v){\downarrow}\alpha f \ ,$$

which completes our proof. $\square$

A.2.6 Lemma 10 on page 123

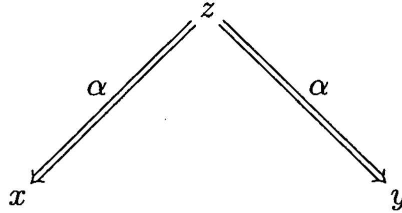
Proof Suppose $x \bigtriangleup y$. We proceed by induction over the closure rules in Table 4.5.

Case hst. So $x \leq y$, and so $x \Rightarrow x$ and $x \Rightarrow y$.

Case htr. So there is y_0 such that $x \bigtriangleup y_0$ and $y_0 \leq y$. And so by our induction hypothesis, there is some z and α such that $z \stackrel{\alpha}{\Rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow} y_0$. So by definition of $\stackrel{\alpha}{\Rightarrow}$, we conclude $z \stackrel{\alpha}{\Rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow} y$.

Case hm. So there are x_0 and y_0 such that $x_0 \bigtriangleup y_0$, $x_0 \stackrel{m}{\rightarrow} x$ and $y_0 \stackrel{m}{\rightarrow} y$. And so by our induction hypothesis, there is z and α such that $z \stackrel{\alpha}{\Rightarrow} x_0$ and $z \stackrel{\alpha}{\Rightarrow} y_0$. By the definition of $\stackrel{\alpha m}{\Rightarrow}$ we conclude $z \stackrel{\alpha m}{\Rightarrow} x$ and $z \stackrel{\alpha m}{\Rightarrow} y$.

And so we conclude that $x \bigtriangleup y$ implies there is z and α such that $z \stackrel{\alpha}{\Rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow}$, or graphically



Conversely, suppose there is z and α such that $z \stackrel{\alpha}{\Rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow} y$. We proceed by list induction over α.

Case ε. So $z \leq x$ and $z \leq y$. By rule hst, we conclude that $z \bigtriangleup y$ and $x \bigtriangleup z$. And by rule htr, we conclude $x \bigtriangleup y$.

Case αm. So $z \stackrel{\alpha}{\Rightarrow} x_0 \stackrel{m}{\rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow} y_0 \stackrel{m}{\rightarrow} y$. By our induction hypothesis $x_0 \bigtriangleup y_0$. By hm, we conclude $x \bigtriangleup y$.

Case αf. So $z \stackrel{\alpha}{\Rightarrow} x_0 \stackrel{f}{\rightarrow} x$ and $z \stackrel{\alpha}{\Rightarrow} y_0 \stackrel{f}{\rightarrow} y$. By our induction hypothesis $x_0 \bigtriangleup y_0$. By stf, we deduce $x \leq y$. And by hst, we conclude $x \bigtriangleup y$. □

A.2.7 Theorem 11 on page 123

Proof Suppose $x \leq y$ in $\overline{G}$. Since $\overline{G}$ is the least graph containing G and closed under the rules in Table 4.5, we consider, separately, cases for: $x \leq y$ in G; and $x \leq y$ is imposed by the closure rules.

Case $x \leq y$ in G. And so $x \leq y$ in G'.

With respect to the remaining rules of Table 4.5, we notice that str and stt correspond to reflexive and transitive closure rules, as used in Definition 5 on page 113. Similarly, rule stm corresponds directly to closure rule 2 in Definition 5 on page 113. It remains to show that rule stf corresponds to rule 1 in Definition 5 on page 113.

Rule stf states that if there are x_0, y_0 and f such that $x_0 \bigtriangleup y_0$, $x_0 \xrightarrow{f} x$ and $y_0 \xrightarrow{f} y$, then $x \leq y$ and $y \leq x$. Using Lemma 10 on page 123, the premise is the same as there are f, z and α such that $z \overset{\alpha}{\Rrightarrow} x_0$ and $z \overset{\alpha}{\Rrightarrow} y_0$, $x_0 \xrightarrow{f} x$ and $y_0 \xrightarrow{f} y$. Thus stf is precisely the same as rule 1 in Definition 5 on page 113. □

A.2.8 Lemma 12 on page 127

Proof Assuming $\mathbf{P}_0$, $\mathbf{Q}_0$, $\mathbf{HG}_0$ and $\mathbf{ITC}_0$ are the sets before execution of the iteration step, after execution the following equations hold.

In the case where $e = (x, y)$,

$$
\begin{aligned}
\mathbf{P} &= \mathbf{P}_0 \cup \{(x,y)\} \\
\mathbf{ITC} &= (\mathbf{ITC}_0 \cup \{(x,y)\})^+ \\
R &= \mathbf{ITC} - \mathbf{ITC}_0 \\
\mathbf{Q} &= \quad \mathbf{Q}_0 - \{(x,y)\} \\
&\cup \ \{[x,y]\}^s - \mathbf{HG}_0 \\
&\cup \ \{[x_0,y] \mid [x_0,x] \in \mathbf{HG}_0\}^s - \mathbf{HG}_0 \\
&\cup \ \{(x',y') \mid (x',y',m) \in \mathbf{PE}[x,y]\} - \mathbf{ITC} \\
&\cup \ R - \{(x,y)\} \\
\mathbf{HG} &= \mathbf{HG}_0 \ .
\end{aligned}
$$

And in the case where $e = [x, y]$,

$$\begin{aligned}
\mathbf{P} &= \mathbf{P}_0 \cup \{[x,y]\} \\
\mathbf{HG} &= \mathbf{HG}_0 \cup \{[x,y]\} \\
\mathbf{Q} &= \quad \mathbf{Q}_0 - \{[x,y]\} \\
&\quad \cup \ \{[x,y_0] \mid (y,y_0) \in \mathbf{ITC}_0 - \mathbf{Q}_0\} - \mathbf{HG} \\
&\quad \cup \ \{[x',y'] \mid (x',y',m) \in \mathbf{PE}[x,y]\}^s - \mathbf{HG} \\
&\quad \cup \ \{(x',y') \mid (x',y',f) \in \mathbf{PE}[x,y]\}^s - \mathbf{ITC}_0 \\
\mathbf{ITC} &= \mathbf{ITC}_0 \ .
\end{aligned}$$

Now we compute, for reflexive X,

$$\begin{aligned}
F(X \cup \{(x,y)\}) = &\quad F(X) \cup \{(x,y)\} \\
&\cup \ \{[x,y]\}^s \\
&\cup \ \{[x_0,y] \mid [x_0,x] \in X\}^s \\
&\cup \ \emptyset \\
&\cup \ \{(x',y') \mid x \xrightarrow{m} x', y \xrightarrow{m} y'\} \\
&\cup \ \emptyset \\
&\cup \ \{(x,y_0) \mid (y,y_0) \in X\} \cup \{(x_0,y) \mid (x_0,x) \in X\}
\end{aligned}$$

and

$$\begin{aligned}
F(X \cup \{[x,y]\}) = &\quad F(X) \cup \{[x,y]\} \\
&\cup \ \emptyset \\
&\cup \ \{[x,y_0] \mid (y,y_0) \in X\}^s \\
&\cup \ \{[x',y'] \mid x \xrightarrow{m} x', y \xrightarrow{m} y'\}^s \\
&\cup \ \emptyset \\
&\cup \ \{(x',y') \mid x \xrightarrow{f} x', y \xrightarrow{f} y'\}^s \\
&\cup \ \emptyset \ .
\end{aligned}$$

Returning to the invariants, we consider separately, cases for $e = (x, y)$ and $e = [x, y]$.

Case $e = (x, y)$. (1) We have as our induction hypothesis

$$\mathbf{P}_0 = \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0) \ .$$

Using the equalities above, we note that

$$\mathbf{HG} + (\mathbf{ITC} - \mathbf{Q}) = \mathbf{HG}_0 + ((\mathbf{ITC}_0 \cup \{(x, y)\})^+ - \mathbf{Q})$$

and since $R - (x, y) \subseteq \mathbf{Q}$ and $(x, y) \notin \mathbf{Q}$,

$$= \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0) \cup \{(x, y)\}$$

and by our induction hypothesis,

$$\begin{aligned} &= \mathbf{P}_0 \cup \{(x, y)\} \\ &= \mathbf{P} \ , \end{aligned}$$

as required.

(2) We have as our induction hypothesis, $\mathbf{P}_0 \cap \mathbf{Q}_0 = \emptyset$. Using invariant (1), we have

$$\mathbf{P} \cap \mathbf{Q} = (\mathbf{HG} + (\mathbf{ITC} - \mathbf{Q})) \cap Q$$

and since intersection distributes over unions,

$$= (\mathbf{HG} \cap \mathbf{Q}) + ((\mathbf{ITC} - \mathbf{Q}) \cap \mathbf{Q})$$

and clearly $(\mathbf{ITC} - \mathbf{Q}) \cap \mathbf{Q} = \emptyset = (\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0$, and using the equalities above,

$$= (\mathbf{HG}_0 \cap \mathbf{Q}) + ((\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0)$$

and since $[x, y] \in \mathbf{Q} - \mathbf{Q}_0$ implies $[x, y] \notin \mathbf{HG}_0$,

$$= (\mathbf{HG}_0 \cap \mathbf{Q}_0) + ((\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0)$$

using $\mathbf{P}_0 = \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0)$ and the fact that intersection distributes over union,

$$= \mathbf{P}_0 \cap \mathbf{Q}_0 \ .$$

(3) First we note that $\mathbf{P}_0 \cup \mathbf{Q}_0 \subseteq \mathbf{P} \cup \mathbf{Q}$, i.e. $\mathbf{P} \cup \mathbf{Q}$ does not get smaller. And so, assuming that $G \subseteq \mathbf{P}_0 \cup \mathbf{Q}_0$ we immediately conclude $G \subseteq \mathbf{P} \cup \mathbf{Q}$.

We have as our induction hypothesis, $F(\mathbf{P}_0) \subseteq \mathbf{P}_0 \cup \mathbf{Q}_0$. Then we let $e' \in F(\mathbf{P} \cup \{(x,y)\})$, and show, by cases, for each component in the union that $e' \in \mathbf{P} \cup \mathbf{Q}$. We must use the fact that $\mathbf{P} = \mathbf{HG} + (\mathbf{ITC} - \mathbf{Q})$ (and similarly for the subscripted version) and Equation 4.5 on page 125.

(4) We have as our induction hypothesis, $\mathbf{P}_0 \cup \mathbf{Q}_0 \subseteq \overline{G}$. To show $\mathbf{P} \cup \mathbf{Q} \subseteq \overline{G}$, it suffices to show $\mathbf{P} \cup \mathbf{Q} \subseteq F^k(\mathbf{P}_0 \cup \mathbf{Q}_0)$ for some k; since F is monotone, together with our induction hypothesis, we know $F^k(\mathbf{P}_0 \cup \mathbf{Q}_0) \subseteq F^k(\overline{G})$ and since $\overline{G}$ is defined as the fix-point of F, we know $F^k(\overline{G}) = \overline{G}$. Thus, we compute $\mathbf{P} \cup \mathbf{Q}$ and consider each of component of the union in turn.

Each of the following inequalities are trivial to show:

$$\begin{aligned}
\mathbf{P}_0 \cup \mathbf{Q}_0 &\subseteq \mathbf{P}_0 \cup \mathbf{Q}_0 \\
\emptyset &\subseteq \{[x',y'] \mid [a,b] \in \mathbf{P}_0 \cup \mathbf{Q}_0,\ a \xrightarrow{m} x',\ b \xrightarrow{m} y' \}^s \\
\emptyset &\subseteq \{(x',y') \mid [a,b] \in \mathbf{P}_0 \cup \mathbf{Q}_0,\ a \xrightarrow{f} x',\ b \xrightarrow{f} y' \}\ .
\end{aligned}$$

The right hand side of each of these inequalities can be shown to be included in $F(\mathbf{P}_0 \cup \mathbf{Q}_0)$. And since F is monotone, the left hand side of each of them is included in $F^k(\mathbf{P}_0 \cup \mathbf{Q}_0)$ for any positive k.

We can show the next three inequalities with a little more work:

$$\begin{aligned}
\{[x,y]\}^s - \mathbf{HG}_0 &\subseteq \{[a,b] \mid (a,b) \in \mathbf{P}_0 \cup \mathbf{Q}_0\}^s \\
\{(x',y') \mid (x',y',m) \in \mathbf{PE}[x,y]\} - \mathbf{ITC} &\subseteq \{(x',y') \mid (a,b) \in \mathbf{P}_0 \cup \mathbf{Q}_0,\ a \xrightarrow{m} x',\ b \xrightarrow{m} y' \} \\
\{[x_0,y] \mid [x_0,x] \in \mathbf{HG}_0\}^s - \mathbf{HG}_0 &\subseteq \{[a,c] \mid [a,b] \in \mathbf{P}_0 \cup \mathbf{Q}_0,\ (b,c) \in \mathbf{P}_0 \cup \mathbf{Q}_0 \}^s\ .
\end{aligned}$$

The first of these follows immediately from our assumption $(x,y) \in \mathbf{Q}_0$ and the definition of $-^s$ as symmetric closure. The second inequality is an immediate consequence of the same assumption and Equation 4.5 on page 125. We argue

the third inequality as follows: suppose $[x_0, x] \in \mathbf{HG}_0$ (that is suppose (x_0, x) and (x, x_0) are in the lhs); since $\mathbf{P}_0 = \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0)$, we know $[x_0, x] \in \mathbf{P}_0$; and since $(x, y) \in \mathbf{P}_0 \cup \mathbf{Q}_0$, we conclude that $[x_0, x]$ (respectively $[x, x_0]$) is in the rhs. Again, the right hand sides of these three inequalities are included in $F(\mathbf{P}_0 \cup \mathbf{Q}_0)$ and using the same argument as before, we conclude that the left hand side of each of these is included in $F^k(\mathbf{P}_0 \cup \mathbf{Q}_0)$ for any positive k.

Finally we argue that there is some k such that $R - \{(x, y)\} \subseteq F^k(\mathbf{P}_0 \cup \mathbf{Q}_0)$. We note: given X, its transitive closure X^+ can be defined as the least fix-point, including X, of H that is defined

$$H(X) \stackrel{\text{def}}{=} X \cup \{(a, c) \mid (a, b) \in X, (b, c) \in X\} \ .$$

We note also that by definition of F, we have $H(X) \subseteq F(X)$; and by iterating this and using the fact that H is order-preserving, we know that for any k, we also have $H^k(X) \subseteq F^k(X)$. And so we argue

$$\begin{array}{ll}
R - \{(x, y)\} & \\
= ((\mathbf{ITC}_0 \cup \{(x, y)\})^+ - \mathbf{ITC}_0) - \{(x, y)\} & \text{by definition of } R \\
\subseteq (\mathbf{ITC}_0 \cup \{(x, y)\})^+ & \\
= H^k(\mathbf{ITC}_0 \cup \{(x, y)\}) & \text{for some } k \\
\subseteq H^k(\mathbf{P}_0 \cup \mathbf{Q}_0) & \text{since } \mathbf{ITC}_0 \cup \{(x, y)\} \subseteq \mathbf{P}_0 \cup \mathbf{Q}_0 \\
\subseteq F^k(\mathbf{P}_0 \cup \mathbf{Q}_0) & \text{by earlier observation.}
\end{array}$$

Thus we have shown

$$R - \{(x, y)\} \subseteq F^k(\mathbf{P}_0 \cup \mathbf{Q}_0) \ .$$

We note that the union of the left hand sides of all the previous inequalities is precisely $\mathbf{P} \cup \mathbf{Q}$ and thus we have shown $\mathbf{P} \cup \mathbf{Q} \subseteq F^k(\mathbf{P}_0 \cup \mathbf{Q}_0)$ for some k.

Case $e = [x, y]$. (1) We have as our induction hypothesis,

$$\mathbf{P}_0 = \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0) \ .$$

Using the equalities above, we have

$$\mathbf{HG} + (\mathbf{ITC} - \mathbf{Q}) = (\mathbf{HG}_0 \cup \{[x, y]\}) + (\mathbf{ITC}_0 - \mathbf{Q}) \ ,$$

and since all new (x', y') edges in $\mathbf{Q}$ are not in $\mathbf{ITC}_0$,

$$= (\mathbf{HG}_0 \cup \{[x, y]\}) + (\mathbf{ITC}_0 - \mathbf{Q}_0) \ ,$$

and since unions commute,

$$= \mathbf{P}_0 \cup \{[x, y]\} = \mathbf{P} \ ,$$

as required.

(2) We have as our induction hypothesis, $\mathbf{P}_0 \cap \mathbf{Q}_0 = \emptyset$. Using invariant (1), we have

$$\mathbf{P} \cap \mathbf{Q} = (\mathbf{HG} + (\mathbf{ITC} - \mathbf{Q})) \cap \mathbf{Q} \ ,$$

and since intersection distributes over unions,

$$\mathbf{P} \cap \mathbf{Q} = (\mathbf{HG} \cap \mathbf{Q}) + ((\mathbf{ITC} - \mathbf{Q}) \cap \mathbf{Q}) \ ,$$

and since $(\mathbf{ITC} - \mathbf{Q}) \cap \mathbf{Q} = \emptyset (\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0$, and using the equation for $\mathbf{Q}$ above,

$$= (\mathbf{HG} \cap (\mathbf{Q}_0 - [x, y])) + ((\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0) \ ,$$

and since $[x, y] \notin \mathbf{P}_0$, therefore $[x, y] \notin \mathbf{HG}_0$,

$$\begin{aligned} &= (\mathbf{HG}_0 \cap \mathbf{Q}_0) + (\mathbf{ITC}_0 - \mathbf{Q}_0) \cap \mathbf{Q}_0 \\ &= \mathbf{P}_0 \cap \mathbf{Q}_0 = \emptyset \ , \end{aligned}$$

as required.

(3) We use exactly the same argument as in the case for $e = (x, y)$ to prove that $G \subseteq \mathbf{P} \cup \mathbf{Q}$. Similarly, we assume as our induction hypothesis $F(\mathbf{P}_0) \subseteq \mathbf{P}_0 \cup \mathbf{Q}_0$ and compute $F(\mathbf{P}_0 \cup \{[x, y]\})$. We let $e' \in F(\mathbf{P}_0 \cup \{[x, y]\})$ and show, by cases for each component in the union, that $e' \in \mathbf{P} \cup \mathbf{Q}$. We must use the facts that: since $\mathbf{P}_0 = \mathbf{HG}_0 + (\mathbf{ITC}_0 - \mathbf{Q}_0)$, the set of all (x', y') edges in $\mathbf{P}_0$ is $\mathbf{ITC}_0 - \mathbf{Q}_0$; and Equation 4.5 on page 125.

(4) We prove that $\mathbf{P} \cup \mathbf{Q} \subseteq F(\mathbf{P}_0) \subseteq \overline{G}$ as for the case $e = (x, y)$, except here it is slightly simpler. We check, in turn, assuming $e = [x, y] \in \mathbf{Q}_0$, the following inclusions:

$$\begin{aligned}
\mathbf{P}_0 \cup \mathbf{Q}_0 &\subseteq \mathbf{P}_0 \cup \mathbf{Q}_0 \\
\emptyset &\subseteq \{[a, b] \mid (a, b) \in \mathbf{P}_0 \cup \mathbf{Q}_0\}^s \\
\{[x, y_0] \mid (y, y_0) \in \mathbf{ITC}_0 - \mathbf{Q}_0\}^s - \mathbf{HG} &\subseteq \{[a, c] \mid \ [a, b] \in \mathbf{P}_0 \cup \mathbf{Q}_0, \\
&\qquad\qquad\quad (b, c) \in \mathbf{P}_0 \cup \mathbf{Q}_0 \ \}^s \\
\{[x', y'] \mid (x', y', m) \in \mathbf{PE}[x, y]\}^s - \mathbf{HG} &\subseteq \{[x', y'] \mid \ [a, b] \in \mathbf{P}_0 \cup \mathbf{Q}_0, \\
&\qquad\qquad\quad a \xrightarrow{m} x',\ b \xrightarrow{m} y' \ \}^s \\
\emptyset &\subseteq \{(x', y') \mid \ (a, b) \in \mathbf{P}_0 \cup \mathbf{Q}_0, \\
&\qquad\qquad\quad a \xrightarrow{m} x',\ b \xrightarrow{m} y' \ \} \\
\{(x', y') \mid (x', y', f) \in \mathbf{PE}[x, y]\}^s - \mathbf{ITC}_0 &\subseteq \{(x', y') \mid \ [a, b] \in \mathbf{P}_0 \cup \mathbf{Q}_0, \\
&\qquad\qquad\quad a \xrightarrow{f} x',\ b \xrightarrow{f} y' \ \} \\
\emptyset &\subseteq \{(a, c) \mid \ (a, b) \in \mathbf{P}_0 \cup \mathbf{Q}_0, \\
&\qquad\qquad\quad (b, c) \in \mathbf{P}_0 \cup \mathbf{Q}_0 \ \} .
\end{aligned}$$

The first, second, fifth and seventh inequalities are trivial. To show the third, we use the facts: $\mathbf{HG}$ contains only $[a, b]$ edges; $\mathbf{ITC}$ contains only (a, b) edges; and $\mathbf{P} = \mathbf{HG} + (\mathbf{ITC} - \mathbf{Q})$. So, if $[x, y_0]$ is in the lhs, then there must be some $(y, y_0) \in \mathbf{ITC}_0 - \mathbf{Q}_0$, and since we know $[x, y] \in \mathbf{Q}_0$, we conclude that $[x, y_0]$ is in the rhs. We use a similar argument in the case $[y_0, x]$ in rhs, using the fact that $-^s$ is symmetric closure. To show the fourth and sixth inequalities, we use Equation 4.5 on page 125 and the assumption $[x, y] \in \mathbf{Q}_0$.

Now we note that the union of the left hand sides of the above inequalities is precisely $\mathbf{P} \cup \mathbf{Q}$ and that of the right hand sides is precisely $F(\mathbf{P}_0 \cup \mathbf{Q}_0)$. So we argue

$$\begin{aligned}
\mathbf{P} \cup \mathbf{Q} &\subseteq F(\mathbf{P}_0 \cup \mathbf{Q}_0) && \text{since union preserves inclusion} \\
&\subseteq F(\overline{G}) && \text{since } \mathbf{P}_0 \cup \mathbf{Q}_0 \subseteq \overline{G} \text{ and } F \text{ is monotone} \\
&\subseteq \overline{G} && \text{since } \overline{G} \text{ is defined to be a fix-point of } F.
\end{aligned}$$

□

A.3 Proofs for Chapter 5

A.3.1 Proof of Proposition 3

Proof We now consider each rule in turn. In each case, we make the following notational assumptions.

- We assume Ψ, Ψ' are respectively, the premise and conclusion of the rule.
- We assume that $(N_i, (\overset{B_i}{\sim}_s), (\overset{B_i}{\rightarrow}_s)) \overset{\text{def}}{=} B_i$ for $i = 0, 1$, and similarly for B_i and D_i.
- We assume that $B_0 \overset{\text{def}}{=} G_0(\Psi)$, $B_1 \overset{\text{def}}{=} G_0(\Psi')$.
- We assume that $C_i \overset{\text{def}}{=} (N_i, (\overset{C_i}{\sim}_s), (\overset{C_i}{\rightarrow}_s)) \overset{\text{def}}{=} B_i^{\text{dg}}$, that is $\overset{C_i}{\sim}_s$ is the reflexive, symmetric and transitive closure of $\overset{B_i}{\sim}_s$, and $\overset{B_i}{\rightarrow}_s \overset{\text{def}}{=} \overset{C_i}{\sim}_s \overset{B_i}{\rightarrow}_s \overset{C_i}{\sim}_s$.
- We assume that $D_i \overset{\text{def}}{=} C_i^{\text{tdg}}$, that is, for each s, $\overset{D_i}{\sim}_s \overset{\text{def}}{=} \overset{C_i}{\sim}_s$ and $\overset{D_i}{\rightarrow}_s \overset{\text{def}}{=} \overset{C_i}{\rightarrow}_s^{+}$.

We note that $(N_0, (\overset{D_0}{\sim}_s), (\overset{D_0}{\rightarrow}_s)) = \mathsf{DG}(\Psi)$ and $(N_1, (\overset{D_1}{\sim}_s), (\overset{D_1}{\rightarrow}_s)) = \mathsf{DG}(\Psi')$.

For cases (freeinst) and (notfreeinst), we exhibit a homomorphism $f : B_1 \rightarrow D_0$, that is, a function $f : N_1 \rightarrow N_0$ such that: (1) $X \overset{B_1}{\sim}_s Y$ implies $X \overset{D_0}{\sim}_s Y$; and (2) $X \overset{B_1}{\rightarrow}_s Y$ implies $X \overset{D_0}{\rightarrow}_s Y$. These two conditions ensure that f is a homomorphism $\mathsf{DG}(\Psi') \rightarrow \mathsf{DG}(\Psi)$.

- Case (freeinst). Thus $\Psi \equiv \exists\vec{X}X.T \subseteq X \wedge \Phi$ and $\Psi' \equiv \exists\vec{X}.\Phi[\mu X.T/X]$, and we may assume X is free in T. So we calculate

  $$B_0 = \{X \overset{B_0}{\rightarrow}_s Y \mid \text{switching } s,\ T' \subseteq e' \in \Psi,\ X \text{ free in } T',\ Y \text{ free in } e'\}$$
  $$\begin{aligned} B_1 = &\ \{X \overset{B_1}{\rightarrow}_s X \mid \text{switching } s\} \\ \cup &\ \{X \overset{B_1}{\rightarrow}_s Y \mid \text{switching } s,\ T' \subseteq e' \in \Psi',\ X \text{ free in } T',\ Y \text{ free in } e'\}\end{aligned}$$

  Now we argue that $f : Y \mapsto Y$ is a homomorphism $B_1 \rightarrow D_0$.

  Suppose $Y \overset{B_1}{\rightarrow}_s Z$. Either $Y = X = Z$, or there is T' with Y free in T' and $T' \subseteq Z \in \Psi'$. In the former case, since we know that X is free in T, then certainly $X \overset{B_0}{\rightarrow}_s X$, and therefore $f(X) = X \overset{D_0}{\rightarrow}_s X = f(X)$.

In the latter case, there must be some T'' such that $T' \equiv T''[\mu X.T/X]$ and $T'' \subseteq Z \in \Psi$. Thus either (1) Y is free in T'', or (2) X is free in T'' and Y is free in T. So either (1) $Y \xrightarrow{B_0}_s Z$ and so $f(Y) = Y \xrightarrow{D_0}_s Z = f(Z)$, or (2) $Y \xrightarrow{B_0}_s X$ and $X \xrightarrow{B_0}_s Z$, and so $f(Y) = Y \xrightarrow{D_0}_s Z = f(Z)$ since $\xrightarrow{D_0}_s$ includes the transitive closure of $\xrightarrow{B_0}_s$.

- Case (notfreeinst). $\Psi \equiv \exists \vec{X} X.T \subseteq X \wedge \Phi$ and $\Psi' \equiv \exists \vec{X}.\Phi[T/X]$, and we may assume X is not free in T. So we calculate

$$
\begin{aligned}
B_0 = \{X \xrightarrow{B_0}_s Y \mid\ & \text{switching } s, \\
& T' \subseteq e' \in \Psi, \\
& X \text{ free in } T', \\
& Y \text{ free in } e'\} \\
B_1 = \{X \xrightarrow{B_1}_s Y \mid\ & \text{switching } s, \\
& T' \subseteq e' \in \Psi', \\
& X \text{ free in } T', \\
& Y \text{ free in } e'\}
\end{aligned}
$$

  Now we show that $f : Y \mapsto Y$ is a homomorphism $B_1 \to D_0$.

  Suppose $Y \xrightarrow{B_1}_s Z$. So there must be $T' \subseteq Z \in \Psi'$ with Y free in T'. From the definition of Ψ', we know that there must be $T'' \subseteq Z \in \Psi$ with $T' \equiv T''[T/X]$. Either (1) Y is free in T'', or (2) X is free in T'' and Y is free in T. Therefore (1) $Y \xrightarrow{B_0}_s Z$ and so $f(Y) = Y \xrightarrow{D_0}_s Z = f(Z)$, or (2) $Y \xrightarrow{B_0}_s X$ and $X \xrightarrow{B_0}_s Z$, and so $f(Y) = Y \xrightarrow{D_0}_s Z = f(Z)$ since $\xrightarrow{D_0}_s$ includes the transitive closure of $\xrightarrow{B_0}_s$.

For the following rules, we show a stronger result by exhibiting a homomorphism $f : B_1 \to C_0$, i.e. f is a function $N_1 \to N_0$ such that (1) $X \overset{B_1}{\sim}_s Y$ implies $f(X) \overset{C_0}{\sim}_s f(Y)$ and (2) $X \xrightarrow{B_1}_s Y$ implies $f(X) \xrightarrow{C_0}_s f(Y)$. These two conditions ensure that f is a homomorphism $\mathsf{DG}(\Psi') \to \mathsf{DG}(\Psi)$.

- Case (eq-inst). Thus $\Psi \equiv \exists \vec{X}.\varepsilon \Rightarrow X = e \wedge \Phi$ and $\Psi' \equiv \exists \vec{X}.\Phi[e/X]$ and we

may assume that X is not free in e. We calculate

$$
\begin{array}{rcl}
B_0 & = & \{X \overset{B_0}{\sim}_s Y \mid \text{switching } s,\ Y \text{ free in } e{\downarrow}s\} \cup G_0(\Phi) \\
B_1 & = & G_0(\Phi[e/X]) \\
 & = & \{Y \overset{B_1}{\sim}_s Z \mid \text{switching } s, \\
 & & \quad e_0 = e_1 \in (\Phi[e/X]){\downarrow}s, \\
 & & \quad Y \text{ free in } e_0, \\
 & & \quad Z \text{ free in } e_1\} \\
 & \cup & \{Y \overset{B_1}{\rightarrow}_s Z \mid \text{switching } s, \\
 & & \quad L \subseteq e' \in (\Phi[e/X]){\downarrow}s, \\
 & & \quad Y \text{ free in } L, \\
 & & \quad Z \text{ free in } e'\}
\end{array}
$$

and from our technical lemma, we know that $(\Phi[e/X]){\downarrow}s = (\Phi{\downarrow}s)[e{\downarrow}s/X]$.

We now show that $f : Y \mapsto Y$ is a homomorphism $B_1 \longrightarrow C_0$. We consider cases $Y \overset{B_1}{\sim}_s Z$, $Y \overset{B_1}{\rightarrow}_s Z$ and respectively show that $Y \overset{C_0}{\sim}_s Z$, $Y \overset{C_0}{\rightarrow}_s Z$.

Suppose $Y \overset{B_1}{\sim}_s Z$. So there must be

$$e_0 = e_1 \in (\Phi{\downarrow}s)[e{\downarrow}s/X] \ ,$$

with Y (resp. Z), free in e_0 (resp. e_1). So there must be

$$e'_0 = e'_1 \in \Phi{\downarrow}s \ ,$$

with $e_0 \equiv e'_0[e{\downarrow}s/X]$ and $e_1 \equiv e'_1[e{\downarrow}s/X]$.

Since Y is free in e_0, then by the definition of substitution, either Y is free in e'_0, or X is free in e'_0 and Y is free in $e{\downarrow}s$.

Similarly, either Z is free in e'_1, or X is free in e'_1 and Z is free in $e{\downarrow}s$.

Thus we have four subcases, and we conclude: (1) $Y \overset{B_0}{\sim}_s Z$; (2) $Y \overset{B_0}{\sim}_s X$ and $X \overset{B_0}{\sim}_s Z$; (3) $Y \overset{B_0}{\sim}_s X$ and $X \overset{B_0}{\sim}_s Z$; and (4) $Y \overset{B_0}{\sim}_s X$, $X \overset{B_0}{\sim}_s X$ and $X \overset{B_0}{\sim}_s Z$. In all cases, since $\overset{C_0}{\sim}_s$ includes the transitive closure of $\overset{B_0}{\sim}_s$, we can conclude $f(Y) = Y \overset{0}{\sim}_s Z = f(Z)$.

Now suppose $Y \xrightarrow{B_1}_s Z$. So there must be

$$L \subseteq e_0 \in (\Phi{\downarrow}s)[e{\downarrow}s/X] \ ,$$

with Y free in L, and Z free in e_0. So there must be

$$L' \subseteq e'_0 \in \Phi{\downarrow}s \ ,$$

with $L \equiv L'[e{\downarrow}s/X]$ and $e_0 \equiv e'_0[e{\downarrow}s/X]$.

Again using the definition of substitution, either Y is free in L', or X is free in L' and Y is free in $e{\downarrow}s$. Similarly, either Z is free in e'_0, or X is free in e'_0 and Z is free in $e{\downarrow}s$. Again by considering four cases, we conclude: (1) $Y \xrightarrow{B_0}_s Z$; (2) $Y \xrightarrow{B_0}_s X$, $X \overset{B_0}{\sim}_s Z$; (3) $X \overset{B_0}{\sim}_s Y$ and $X \xrightarrow{B_0}_s Z$; and (4) $X \overset{B_0}{\sim}_s Y$, $X \overset{B_0}{\sim}_s X$ and $X \xrightarrow{B_0}_s Z$. In all cases, we can conclude $f(Y) = Y \xrightarrow{D_0}_s Z = f(Z)$, since $\xrightarrow{C_0}_s = \overset{C_0}{\sim}_s\xrightarrow{B_0}_s\overset{C_0}{\sim}_s$ and $\overset{C_0}{\sim}_s$ is the reflexive, symmetric and transitive closure of $\overset{B_0}{\sim}_s$.

- Case (eq-idem). Thus $\Psi \equiv \exists\vec{X}.X = e\wedge\Phi$ and $\Psi' \equiv \exists\vec{X}X'.X = e[X'/X]\wedge\Phi$, and we may assume X is free in e. We calculate

$$\begin{aligned} B_0 = & \quad \{X \overset{B_0}{\sim}_s Y \mid \text{switching } s,\ Y \text{ free in } e{\downarrow}s\} \\ & \cup\ G_0(\Phi) \\ B_1 = & \quad \{X \overset{B_1}{\sim}_s Y \mid \ \text{switching } s, \\ & \qquad\qquad\qquad Y \text{ free in } (e[X'/X]){\downarrow}s\} \\ & \cup\ G_0(\Phi)\ , \end{aligned}$$

and from our technical lemma, we know that $(e[X'/X]){\downarrow}s \equiv (e{\downarrow}s)[X'/X]$.

We now show that f defined

$$f(Y) \overset{\text{def}}{=} \begin{cases} X & \text{if } Y = X' \\ Y & \text{otherwise} \end{cases}$$

is a homomorphism $B_1 \rightarrow C_0$.

Consider $Y \overset{B_1}{\sim}_s Z$. If $Z \neq X'$, then certainly $Y \overset{B_0}{\sim}_s Z$, and so $f(Y) = Y \overset{C_0}{\sim}_s= Z = f(Z)$. Now if $Z = X'$, then $Y = X$ is the only possibility. Since $\overset{C_0}{\sim}_s$ is reflexive, we conclude $f(X) = X \overset{C_0}{\sim}_s X = f(X')$.

Consider $Y \xrightarrow{B_1}_s Z$. Thus $Y \xrightarrow{B_1}_s Z \in G_0(\Phi)$ and so $Y \xrightarrow{B_0}_s Z$ also. Thus we conclude $Y \xrightarrow{C_0}_s Z$.

- Case (eq-pullback). Thus $\Psi \equiv \exists\vec{X}.v \Rightarrow \sigma X_1 = \tau X_2 \wedge \Phi$ and $\Psi' \equiv \exists\vec{X}Z.v \Rightarrow X_1 = \pi_1 Z \wedge v \Rightarrow X_2 = \pi_w Z \wedge \Phi$. We calculate

$$\begin{aligned} B_0 = & \quad \{X_1 \overset{B_0}{\sim}_s X_2 \mid s \models v\} \\ & \cup \ G_0(\Phi) \\ B_1 = & \quad \{X_1 \overset{B_1}{\sim}_s Z \mid s \models v\} \\ & \cup \ \{X_2 \overset{B_1}{\sim}_s Z \mid s \models v\} \\ & \cup \ G_0(\Phi) \ . \end{aligned}$$

We now show that f defined as

$$f(Y) \stackrel{\text{def}}{=} \begin{cases} X_2 & \text{if } Y = Z \\ Y & \text{otherwise} \end{cases}$$

is a homomorphism $B_1 \to C_0$.

Consider $X \overset{B_1}{\sim}_s Y$. If $Y \neq Z$ then $X \overset{B_1}{\sim}_s Y \in G_0(\Phi)$ and so $X \overset{B_0}{\sim}_s Y$ also, and therefore, $X \overset{C_0}{\sim}_s Y$. Now if $Y = Z$, then either $X = X_1$ or $X = X_2$. In the former case, we conclude $f(X_1) = X_1 \overset{C_0}{\sim}_s X_2 = f(Z)$. In the latter case, since $\overset{C_0}{\sim}_s$ is reflexive, we conclude $f(X_2) = X_2 \overset{C_0}{\sim}_s X_2 = f(Z)$.

Consider $X \xrightarrow{B_1}_s Y$. Certainly $X \xrightarrow{B_1}_s Y \in G_0(\Phi)$, and so $X \xrightarrow{B_0}_s Y$, and therefore, $X \xrightarrow{C_0}_s Y$.

- Case (eq-sym). Thus $\Psi \equiv \exists\vec{X}.v \Rightarrow e = e' \wedge \Phi$, $\Psi' \equiv \exists\vec{X}.v \Rightarrow e' = e \wedge \Phi$. It is easy to check that $f : X \mapsto X$ is a homomorphism $B_1 \to C_0$.

- Case (eq-if-introl), (eq-if-intro2). We consider only (eq-if-introl) since (eq-if-intro2) is similar. Thus

$$\begin{aligned} \Psi &\equiv \exists\vec{X}.v, x{=}\mathtt{tt} \Rightarrow X_1 = e \wedge \Phi \\ \Psi' &\equiv \exists\vec{X}.v \Rightarrow X_1 = e \triangleleft x \triangleright X_1 \wedge \Phi \ . \end{aligned}$$

We calculate

$$\begin{aligned} B_0 = \quad & \{X_1 \overset{B_0}{\rightsquigarrow}_s Y \mid \; s \models v, x{=}\mathtt{tt}, \\ & \qquad\qquad Y \text{ is free in } e{\downarrow}s\} \\ \cup \; & G_0(\Phi) \\ B_1 = \quad & \{X_1 \overset{B_1}{\rightsquigarrow}_s Y \mid \; s \models v, \\ & \qquad\qquad Y \text{ is free in } (e \triangleleft x \triangleright X_1){\downarrow}s\} \\ \cup \; & G_0(\Phi) \ . \end{aligned}$$

We now show that $f : X \mapsto X$ is a homomorphism $B_1 \to C_0$.

Consider $X \overset{B_1}{\rightsquigarrow}_s Y$. If $X \overset{B_1}{\rightsquigarrow}_s Y \in G_0(\Phi)$, then $X \overset{B_0}{\rightsquigarrow}_s Y$ also, and therefore $X \overset{C_0}{\rightsquigarrow}_s Y$. Otherwise, certainly $s \models v$ and $X \equiv X_1$. We consider cases where $s(x) = \mathtt{tt}, \mathtt{ff}$. If $s(x) = \mathtt{tt}$, then $s \models v, x{=}\mathtt{tt}$ and also $(e \triangleleft x \triangleright X_1){\downarrow}s \equiv e{\downarrow}s$, therefore Y is free in $e{\downarrow}s$ and so $X_1 \overset{B_0}{\rightsquigarrow}_s Y$, thus certainly $f(X) = X_1 \overset{C_0}{\rightsquigarrow}_s Y = f(Y)$. However, if $s(x) = \mathtt{ff}$, then $(e \triangleleft x \triangleright X_1){\downarrow}s \equiv X_1$, hence $Y \equiv X_1$, and since $\overset{C_0}{\rightsquigarrow}_s$ is reflexive, we conclude $f(X) = X_1 \overset{C_0}{\rightsquigarrow}_s X_1 = f(Y)$.

Consider $X \overset{B_1}{\rightarrow}_s Y$. So $X \overset{B_1}{\rightarrow}_s Y \in G_0(\Phi)$, therefore $X \overset{B_0}{\rightarrow}_s Y$, so certainly $f(X) = X \overset{C_0}{\rightarrow}_s Y = f(Y)$.

For the remaining rules, we can prove a yet stronger result by exhibiting a homomorphism $B_1 \to B_0$.

- Case (eq-resp). Thus $\Psi \equiv \exists \vec{X}.v \to e_0 = e_1$, $\Psi' \equiv \exists \vec{X}.v \to e_0' = e_1$ and we may assume that $e_0 \rightsquigarrow e_0'$. From our technical lemma, we know that $e_0{\downarrow}s, e_0'{\downarrow}s$ have the same free variables for all switchings s. It is easy to check $B_0 = B_1$, i.e. $f : X \mapsto X$ is a suitable homomorphism.

- Case (eq-equaliser). Thus $\Psi \equiv \exists \vec{X}.v \Rightarrow \sigma X_1 = \tau X_1 \wedge \Phi$ and $\Psi \equiv \exists \vec{X} Z.v \Rightarrow X_1 = \rho Z \wedge \Phi$. We calculate

$$\begin{aligned} B_0 &= \{X_1 \overset{B_0}{\rightsquigarrow}_s X_1 \mid s \models v\} \cup G_0(\Phi) \\ B_1 &= \{X_1 \overset{B_1}{\rightsquigarrow}_s Z \mid s \models v\} \cup G_0(\Phi) \ . \end{aligned}$$

We now show that f defined by

$$f(X) \stackrel{\text{def}}{=} \begin{cases} X_1 & \text{if } X = Z \\ X & \text{otherwise} \end{cases}$$

is a homomorphism $B_1 \to B_0$.

Consider $X \stackrel{B_1}{\sim}_s Y$. If $X \stackrel{B_1}{\sim}_s Y \in G_0(\Phi)$ then also $f(X) = X \stackrel{B_0}{\sim}_s Y = f(Y)$. Otherwise, $s \models v$, $X \equiv X_1$ and $Y \equiv Z$, and $f(X) = X_1 \stackrel{B_0}{\sim}_s X_1 = f(Y)$.

Consider $X \stackrel{B_1}{\to}_s Y$. We must have $X \stackrel{B_1}{\to}_s Y \in G_0(\Phi)$, and so $f(X) = X \stackrel{B_0}{\to}_s Y = f(Y)$.

- Case (eq-const). Thus $\Psi \equiv \exists \vec{X}.v \Rightarrow \sigma X_1 = \tau\phi \wedge \Phi$ and $\Psi \equiv \exists \vec{X}.v \Rightarrow X_1 = (\sigma^{-1} \circ \tau)\phi \wedge \Phi$. It is easy to check that $B_0 = B_1$, that is, $f : X \mapsto X$ is a suitable homomorphism.

- Case (if-eliml), (if-betal1), (if-betal2). For these rules, we use the fact that $L(L') \equiv L''[L'/X]$ for some L'', and so $(L(L'))\downarrow s \equiv (L''\downarrow s)[L'\downarrow s/X]$.

- Case (falseinst). It is easy to show that the function $f : X \mapsto X$ is a homomorphism.

For rules (eq-if-elim), (eq-betal), (eq-beta2), (if-elimr), (if-betar1), (if-betar2), (adj-ren) and (collate), it is simple to check that $B_0 = B_1$, and so $f : X \mapsto X$ is a suitable homomorphism. □

Appendix B

Input scripts for VCG examples

B.1 Example SPASS input script

```
#define INV(y,s0,s1,r) \
      forall([int(yf0),int(yg0)],                                    \
      implies(and(lookup(s0,y,f,yf0),                                \
                  lookup(s0,y,g,yg0),                                \
                  pos(yf0),                                          \
                  pos(yg0)),                                         \
              exists([int(yf1),int(yg1)],                            \
              and(lookup(s1,y,f,yf1),                                \
                  lookup(s1,y,g,yg1),                                \
                  equal(r,yf1),                                      \
                  equal(r,yg1),                                      \
                  equal(r,gcd(yf0,yg0))))))

#define GCDSPEC(s0,s1,r) \
      implies(and(pos(426),pos(792)),equal(r,gcd(426,792)))


begin_problem(Gcd1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_descriptions.
name({* gcd_vc1 *}).
author({* fhlt *}).
status(satisfiable).
description({* vc1 for gcd example *}).
```

```
end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_symbols.
functions[
% update for stores
(update,4),
% boolean constants
(boolt,0),(boolf,0),
% functions for gcd example
(1,0),(426,0),(792,0),
(minus,2),(gcd,2),(f,0),(g,0)
].

predicates[
% lookup for stores
(lookup,4),
% predicates for gcd program
(pos,1),(lt,2)
].

sorts[store,fname,locn,bool,int,value].

end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_declarations.
subsort(locn,value).
subsort(bool,value).
subsort(int,value).
forall([store(s),locn(l),fname(ff),value(v)],
       store(update(s,l,ff,v))).
fname(f).
fname(g).
forall([int(x),int(y)],int(minus(x,y))).
forall([int(x),int(y)],int(gcd(x,y))).
bool(boolt).
bool(boolf).
int(426). int(792).

predicate(lookup,store,locn,fname,value).
```

```
predicate(pos,int).
predicate(lt,int,int).
end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_formulae(axioms).

% Stores

% definition of update
formula(forall([store(s),
                fname(ff),fname(fp),
                locn(l),locn(lp),
                value(v),value(vp)],
equiv(lookup(update(s,l,ff,v),lp,fp,vp),
      or(and(equal(l,lp),equal(ff,fp),equal(v,vp)),
         and(not(and(equal(l,lp),equal(ff,fp))),
             lookup(s,lp,fp,vp))))
),axdefupd).

% extensionality axiom
%formula(forall([store(s0),store(s1)],
%implies(forall([locn(l),fname(ff),value(v)],
%        equiv(lookup(s0,l,f,v),lookup(s1,l,f,v))),
%        equal(s0,s1))
%),axext).

% stores are partial functions
formula(forall([store(s),locn(l),fname(ff),value(v0),value(v1)],
implies(and(lookup(s,l,ff,v0),lookup(s,l,ff,v1)),
        equal(v0,v1))
),axfunstore).

% false is not true
formula(not(equal(boolt,boolf))).

% false and true are the only booleans
%formula(forall([bool(x)],or(equal(x,boolt),equal(x,boolf)))).



% adhoc axioms about pos, lt, minus, gcd
```

```
formula(forall([int(x),int(y)],
implies(lt(x,y),pos(minus(y,x))))
).

formula(forall([int(x),int(y)],
implies(and(not(lt(x,y)),not(lt(y,x))),equal(x,y)))).

formula(forall([int(x),int(y)],
implies(and(lt(x,y),pos(x),pos(y)),
        equal(gcd(x,y),gcd(x,minus(y,x)))))
).

formula(forall([int(x),int(y)],
implies(and(lt(y,x),pos(x),pos(y)),
        equal(gcd(x,y),gcd(minus(x,y),y))))
).

formula(forall([int(x)],equal(gcd(x,x),x))).

% Next axiom is needed for swapping versions of Euclid's algorithm
%
% GCDSYM axiom
formula(forall([int(x),int(y)],equal(gcd(x,y),gcd(y,x)))).

formula(not(or(equal(g,f),equal(f,g)))).


end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_formulae(conjectures).
formula(and(forall([value(x0_1), ...
end_of_list.

end_problem.
```

B.2 Dining philosophers example input script

```
#define Fork(s1,s2)                                              \
                   [ ontable = tt,                               \
                     trypickup(s1) = if s1.ontable then          \
```

```
                                        s1.ontable := ff;         \
                                        tt                        \
                                    else                          \
                                        ff                  ,     \
                        putdown(s2) = s2.ontable := tt          ]


#define Philtick(s,tp0,tp1)                                        \
                            if s.state = 0 then {                 \
                                let tp0 = s.fork1.trypickup() in  \
                                if tp0 then s.state := 1          \
                                else s }                          \
                            else if s.state = 1 then {            \
                                let tp1 = s.fork2.trypickup() in  \
                                if tp1 then s.state := 2          \
                                else s }                          \
                            else if s.state = 2 then              \
                                s.fork2.putdown();                \
                                s.state := 3                      \
                            else {                                \
                               s.fork1.putdown();                 \
                                s.state := 0        }

#define LRPhil(fl,fr,s,tp0,tp1) Phil(fl, fr, s, tp0, tp1)

#define RLPhil(fl,fr,s,tp0,tp1) Phil(fr, fl, s, tp0, tp1)

#define Phil(f1,f2,s,tp0,tp1)                                      \
                              [ state = 0, fork1 = f1, fork2 = f2, \
                                tick(s) = Philtick(s,tp0,tp1) ]

#define Table                                                      \
           {                                                       \
           {                                                       \
           let fk1 = { Fork(sf11, sf12)                            \
                       :: [ [],"MKFK1",0 ] } in                    \
           let fk2 = { Fork(sf21, sf22)                            \
                       :: [ [],"MKFK2",1 ] } in                    \
           let fk3 = { Fork(sf31, sf32)                            \
                       :: [ [],"MKFK3",2 ] } in                    \
           let ph1 = { LRPhil(fk2, fk3, sp1, tp10, tp11)           \
                       :: [ [],"MKPH1",3 ] } in                    \
```

```
        let ph2 = { RLPhil(fk3, fk1, sp2, tp20, tp21)        \
                    :: [ [],"MKPH2",4 ] } in                  \
        let ph3 = { LRPhil(fk1, fk2, sp3, tp30, tp31)        \
                    :: [ [],"MKPH3",5 ] } in                  \
        {                                                     \
        [ f1 = fk1, f2 = fk2, f3 = fk3,                       \
          p1 = ph1, p2 = ph2, p3 = ph3,                       \
          tick1(st1) = st1.p1.tick() :: [ [],"ITICK1",7 ],    \
          tick2(st2) = st2.p2.tick() :: [ [],"ITICK2",7 ],    \
          tick3(st3) = st3.p3.tick() :: [ [],"ITICK3",7 ]     \
        ]                                                     \
          :: [ [],"MKTBL",6 ]                                 \
        }                                                     \
        :: [ [],"TABLE0",0 ;                                  \
             [tick1()],"TICK1",1 ;                            \
             [tick2()],"TICK2",1 ;                            \
             [tick3()],"TICK3",1                              \
           ] }                                                \
        :: [ [],"TABLE",0 ] }

let t = Table in
{t.tick1(); t.tick2(); t.tick3()}
```

B.3 Example SPASS input script for dining philosophers example

```
#define ISONEOF(x1,x2,x3,x)                                   \
                  or(equal(x,x1),equal(x,x2),equal(x,x3))


#define ISHOLDING(pp,ff,s)                                    \
          or(and(lookup(s,pp,state,1),lookup(s,pp,fork1,ff)), \
             and(lookup(s,pp,state,2),lookup(s,pp,fork1,ff)), \
             and(lookup(s,pp,state,2),lookup(s,pp,fork2,ff)), \
             and(lookup(s,pp,state,3),lookup(s,pp,fork1,ff)))

#define INVPHIL(pp,s)                                         \
                    exists([locn(ff1),locn(ff2)],             \
                    and(lookup(s,pp,fork1,ff1),               \
                        lookup(s,pp,fork2,ff2),               \
```

```
                                  orel(ff1,ff2),                                \
                                  or(lookup(s,pp,state,0),                      \
                                     lookup(s,pp,state,1),                      \
                                     lookup(s,pp,state,2),                      \
                                     lookup(s,pp,state,3))))


#define FKPHTA_UNIQ(ff1,ff2,ff3,pp1,pp2,pp3,t,s)                                \
                             not(or(equal(ff1,ff2),                             \
                                    equal(ff1,ff3),                             \
                                    equal(ff1,pp1),                             \
                                    equal(ff1,pp2),                             \
                                    equal(ff1,pp3),                             \
                                    equal(ff1,t),                               \
                                    equal(ff2,ff3),                             \
                                    equal(ff2,pp1),                             \
                                    equal(ff2,pp2),                             \
                                    equal(ff2,pp3),                             \
                                    equal(ff2,t),                               \
                                    equal(ff3,pp1),                             \
                                    equal(ff3,pp2),                             \
                                    equal(ff3,pp3),                             \
                                    equal(ff3,t),                               \
                                    equal(pp1,pp2),                             \
                                    equal(pp1,pp3),                             \
                                    equal(pp1,t),                               \
                                    equal(pp2,pp3),                             \
                                    equal(pp2,t),                               \
                                    equal(pp3,t)))

#define TBL_CFG(ff1,ff2,ff3,pp1,pp2,pp3,t,s)                                    \
                             and(lookup(s,pp2,fork1,ff1),                       \
                                 lookup(s,pp3,fork1,ff1),                       \
                                 lookup(s,pp1,fork1,ff2),                       \
                                 lookup(s,pp3,fork2,ff2),                       \
                                 lookup(s,pp1,fork2,ff3),                       \
                                 lookup(s,pp2,fork2,ff3),                       \
                                 lookup(s,t,p1,pp1),                            \
                                 lookup(s,t,p2,pp2),                            \
                                 lookup(s,t,p3,pp3),                            \
                                 lookup(s,t,f1,ff1),                            \
                                 lookup(s,t,f2,ff2),                            \
```

```
                        lookup(s,t,f3,ff3))


#define ITP(fk1,fk2,fk3,pp1,pp2,pp3,t,s,pp) INVPHIL(pp,s)
#define ITF(f1,f2,f3,p1,p2,p3,t,s,ff) \
                    and(implies(lookup(s,ff,ontable,boolf),    \
                                or(ISHOLDING(p1,ff,s),         \
                                   ISHOLDING(p2,ff,s),         \
                                   ISHOLDING(p3,ff,s))),       \
                        or(lookup(s,ff,ontable,boolf),         \
                           lookup(s,ff,ontable,boolt)))


#define INVTABLE(f1,f2,f3,p1,p2,p3,t,s)                        \
                    and(ITF(f1,f2,f3,p1,p2,p3,t,s,f1),         \
                        ITF(f1,f2,f3,p1,p2,p3,t,s,f2),         \
                        ITF(f1,f2,f3,p1,p2,p3,t,s,f3),         \
                        ITP(f1,f2,f3,p1,p2,p3,t,s,p1),         \
                        ITP(f1,f2,f3,p1,p2,p3,t,s,p2),         \
                        ITP(f1,f2,f3,p1,p2,p3,t,s,p3),         \
                        FKPHTA_UNIQ(f1,f2,f3,p1,p2,p3,t,s),    \
                        TBL_CFG(f1,f2,f3,p1,p2,p3,t,s),        \
                        locn(f1),                              \
                        locn(f2),                              \
                        locn(f3),                              \
                        locn(p1),                              \
                        locn(p2),                              \
                        locn(p3),                              \
                        locn(t))

#define ITICK(fk1,fk2,fk3,pp1,pp2,pp3,t,s0,s1,r)               \
          implies(and(orel(fk1,fk2),                           \
                    orel(fk1,fk3),                             \
                    orel(fk2,fk3),                             \
                    INVTABLE(fk1,fk2,fk3,pp1,pp2,pp3,t,s0)),   \
                INVTABLE(fk1,fk2,fk3,pp1,pp2,pp3,t,s1))

#define ITICK1 ITICK
#define ITICK2 ITICK
#define ITICK3 ITICK

#define TICK(t,s0,s1,r) \
```

```
        exists([value(fk1),value(fk2),value(fk3),                 \
               value(pp1),value(pp2),value(pp3)],                 \
              TRTICK(fk1,fk2,fk3,pp1,pp2,pp3,t,s0,s1,r))

#define TICK1 TICK
#define TICK2 TICK
#define TICK3 TICK

#define xxSTORE1(ff1,s) \
                  and(lookup(s,ff1,ontable,boolt),                \
                      locn(ff1))

#define xxSTORE2(ff1,ff2,s)                                       \
                  and(not(equal(ff1,ff2)),                        \
                      lookup(s,ff1,ontable,boolt),                \
                      lookup(s,ff2,ontable,boolt),                \
                      locn(ff1),                                  \
                      locn(ff2))

#define xxSTORE3(ff1,ff2,ff3,s)                                   \
                  and(not(or(equal(ff1,ff2),                      \
                             equal(ff1,ff3),                      \
                             equal(ff2,ff3))),                    \
                      lookup(s,ff1,ontable,boolt),                \
                      lookup(s,ff2,ontable,boolt),                \
                      lookup(s,ff3,ontable,boolt),                \
                      locn(ff1),                                  \
                      locn(ff2),                                  \
                      locn(ff3))

#define xxSTORE4(ff1,ff2,ff3,pp1,s)                               \
                  and(not(or(equal(ff1,ff2),                      \
                             equal(ff1,ff3),                      \
                             equal(ff1,pp1),                      \
                             equal(ff2,ff3),                      \
                             equal(ff2,pp1),                      \
                             equal(ff3,pp1))),                    \
                      lookup(s,pp1,fork1,ff2),                    \
                      lookup(s,pp1,fork2,ff3),                    \
                      lookup(s,ff1,ontable,boolt),                \
                      lookup(s,ff2,ontable,boolt),                \
                      lookup(s,ff3,ontable,boolt),                \
```

```
                      lookup(s,pp1,state,0),                        \
                      locn(ff1),                                    \
                      locn(ff2),                                    \
                      locn(ff3),                                    \
                      locn(pp1))

#define xxSTORE5(ff1,ff2,ff3,pp1,pp2,s) \
                 and(not(or(equal(ff1,ff2),                         \
                            equal(ff1,ff3),                         \
                            equal(ff1,pp1),                         \
                            equal(ff1,pp2),                         \
                            equal(ff2,ff3),                         \
                            equal(ff2,pp1),                         \
                            equal(ff2,pp2),                         \
                            equal(ff3,pp1),                         \
                            equal(ff3,pp2),                         \
                            equal(pp1,pp2))),                       \
                      lookup(s,pp2,fork1,ff1),                      \
                      lookup(s,pp1,fork1,ff2),                      \
                      lookup(s,pp1,fork2,ff3),                      \
                      lookup(s,pp2,fork2,ff3),                      \
                      lookup(s,ff1,ontable,boolt),                  \
                      lookup(s,ff2,ontable,boolt),                  \
                      lookup(s,ff3,ontable,boolt),                  \
                      lookup(s,pp1,state,0),                        \
                      lookup(s,pp2,state,0),                        \
                      locn(ff1),                                    \
                      locn(ff2),                                    \
                      locn(ff3),                                    \
                      locn(pp1),                                    \
                      locn(pp2))

#define xxSTORE6(ff1,ff2,ff3,pp1,pp2,pp3,s)                         \
                 and(not(or(equal(ff1,ff2),                         \
                            equal(ff1,ff3),                         \
                            equal(ff1,pp1),                         \
                            equal(ff1,pp2),                         \
                            equal(ff1,pp3),                         \
                            equal(ff2,ff3),                         \
                            equal(ff2,pp1),                         \
                            equal(ff2,pp2),                         \
                            equal(ff2,pp3),                         \
```

```
                                   equal(ff3,pp1),                    \
                                   equal(ff3,pp2),                    \
                                   equal(ff3,pp3),                    \
                                   equal(pp1,pp2),                    \
                                   equal(pp1,pp3),                    \
                                   equal(pp2,pp3))),                  \
                             lookup(s,pp2,fork1,ff1),                 \
                             lookup(s,pp3,fork1,ff1),                 \
                             lookup(s,pp1,fork1,ff2),                 \
                             lookup(s,pp3,fork2,ff2),                 \
                             lookup(s,pp1,fork2,ff3),                 \
                             lookup(s,pp2,fork2,ff3),                 \
                             lookup(s,ff1,ontable,boolt),             \
                             lookup(s,ff2,ontable,boolt),             \
                             lookup(s,ff3,ontable,boolt),             \
                             lookup(s,pp1,state,0),                   \
                             lookup(s,pp2,state,0),                   \
                             lookup(s,pp3,state,0),                   \
                             locn(ff1),                               \
                             locn(ff2),                               \
                             locn(ff3),                               \
                             locn(pp1),                               \
                             locn(pp2),                               \
                             locn(pp3))

#define STORE7(ff1,ff2,ff3,pp1,pp2,pp3,t,s) \
                         and(FKPHTA_UNIQ(ff1,ff2,ff3,pp1,pp2,pp3,t,s), \
                             TBL_CFG(ff1,ff2,ff3,pp1,pp2,pp3,t,s),    \
                             lookup(s,ff1,ontable,boolt),             \
                             lookup(s,ff2,ontable,boolt),             \
                             lookup(s,ff3,ontable,boolt),             \
                             lookup(s,pp1,state,0),                   \
                             lookup(s,pp2,state,0),                   \
                             lookup(s,pp3,state,0),                   \
                             locn(ff1),                               \
                             locn(ff2),                               \
                             locn(ff3),                               \
                             locn(pp1),                               \
                             locn(pp2),                               \
                             locn(pp3),                               \
                             locn(t))
```

```
#define MKFK1(s0,s1,r)                                             \
               STORE1(r,s1)

#define MKFK2(ff1,s0,s1,r)                                         \
          implies(STORE1(ff1,s0),                                  \
                  STORE2(ff1,r,s1))

#define MKFK3(ff1,ff2,s0,s1,r)                                     \
          implies(STORE2(ff1,ff2,s0),                              \
                  STORE3(ff1,ff2,r,s1))

#define MKPH1(ff1,ff2,ff3,s0,s1,r)                                 \
          implies(STORE3(ff1,ff2,ff3,s0),                          \
                  STORE4(ff1,ff2,ff3,r,s1))

#define MKPH2(ff1,ff2,ff3,pp1,s0,s1,r)                             \
          implies(STORE4(ff1,ff2,ff3,pp1,s0),                      \
                  STORE5(ff1,ff2,ff3,pp1,r,s1))

#define MKPH3(ff1,ff2,ff3,pp1,pp2,s0,s1,r)                         \
          implies(STORE5(ff1,ff2,ff3,pp1,pp2,s0),                  \
                  STORE6(ff1,ff2,ff3,pp1,pp2,r,s1))

#define MKTBL(ff1,ff2,ff3,pp1,pp2,pp3,s0,s1,r)                     \
          implies(STORE6(ff1,ff2,ff3,pp1,pp2,pp3,s0),              \
                  STORE7(ff1,ff2,ff3,pp1,pp2,pp3,r,s1))

#define TABLE0(s0,s1,r) \
           exists([value(fk1),value(fk2),value(fk3),               \
                   value(pp1),value(pp2),value(pp3)],              \
                  implies(and(orel(fk1,fk2),                       \
                              orel(fk2,fk3),                       \
                              orel(fk1,fk3)),                      \
                      and(STORE7(fk1,fk2,fk3,pp1,pp2,pp3,r,s1),    \
                          orel(fk1,fk2),                           \
                          orel(fk2,fk3),                           \
                          orel(fk1,fk3))))

#define TABLE(s0,s1,r) \
           exists([value(fk1),value(fk2),value(fk3),               \
                   value(pp1),value(pp2),value(pp3)],              \
                  implies(and(orel(fk1,fk2),                       \
```

```
                                     orel(fk2,fk3),                  \
                                     orel(fk1,fk3)),                 \
                            INVTABLE(fk1,fk2,fk3,pp1,pp2,pp3,r,s1)))

begin_problem(Gcd1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_descriptions.
name({* dinphil_vc1 *}).
author({* fhlt *}).
status(satisfiable).
description({* vc1 for dinphil example *}).
end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_symbols.
functions[
% update for stores
(update,4),
% boolean constants
(boolt,0),(boolf,0),
% fields, methods, constants
(ontable,0),(state,0),(fork1,0),(fork2,0),
(f1,0),(f2,0),(f3,0),(p1,0),(p2,0),(p3,0),
(trypickup,0),(putdown,0),(tick,0),(tick1,0),(tick2,0),(tick3,0),
(0,0),(1,0),(2,0),(3,0)
].

predicates[
%(STORE1,2),
%(STORE2,3),
%(STORE3,4),
%(STORE4,5),
%(STORE5,6),
%(STORE6,7),
%(STORE7,8),
%(FKPHTA_UNIQ,8),
% lookup for stores
(lookup,4),
% predicates for dinphil program
(orel,2)
```

```
].

sorts[store,fname,mname,locn,bool,int,value,fork].

end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_declarations.
subsort(locn,value).
subsort(bool,value).
subsort(int,value).
forall([store(s),locn(l),fname(ff),value(v)],
       store(update(s,l,ff,v))).

fname(ontable). fname(state).
fname(fork1). fname(fork2).
fname(f1). fname(f2). fname(f3).
fname(p1). fname(p2). fname(p3).
mname(trypickup). mname(putdown). mname(tick).
mname(tick1). mname(tick2). mname(tick3).
int(0). int(1). int(2). int(3).
bool(boolt). bool(boolf).

predicate(lookup,store,locn,fname,value).

predicate(orel,locn,locn).

end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_formulae(axioms).

% Stores

% definition of update
formula(forall([store(s),
                fname(ff),fname(fp),
                locn(l),locn(lp),
                value(v),value(vp)],
equiv(lookup(update(s,l,ff,v),lp,fp,vp),
      or(and(and(equal(l,lp),equal(ff,fp)),equal(v,vp)),
         and(not(and(equal(l,lp),equal(ff,fp))),
```

```
              lookup(s,lp,fp,vp))))
),axdefupd).

% extensionality axiom
%formula(forall([store(s0),store(s1)],
%implies(forall([locn(l),fname(ff),value(v)],
%        equiv(lookup(s0,l,ff,v),lookup(s1,l,ff,v))),
%        equal(s0,s1))
%),axext).

% stores are partial functions
formula(forall([store(s),locn(l),fname(ff),value(v0),value(v1)],
implies(and(lookup(s,l,ff,v0),lookup(s,l,ff,v1)),
        equal(v0,v1))
),axfunstore).

% false is not true
formula(not(equal(boolt,boolf))).

% false and true are the only booleans
%formula(forall([bool(x)],or(equal(x,boolt),equal(x,boolf)))).



% adhoc axioms for dinphil example

formula(not(or(equal(0,1),
               equal(0,2),
               equal(0,3),
               equal(1,2),
               equal(1,3),
               equal(2,3)))).

formula(not(or(
equal(f1,f2),
equal(f1,f3),
equal(f1,p1),
equal(f1,p2),
equal(f1,p3),
equal(f1,state),
equal(f1,fork1),
equal(f1,fork2),
```

```
equal(f1,ontable),
equal(f2,f1),
equal(f2,f3),
equal(f2,p1),
equal(f2,p2),
equal(f2,p3),
equal(f2,state),
equal(f2,fork1),
equal(f2,fork2),
equal(f2,ontable),
equal(f3,f1),
equal(f3,f2),
equal(f3,p1),
equal(f3,p2),
equal(f3,p3),
equal(f3,state),
equal(f3,fork1),
equal(f3,fork2),
equal(f3,ontable),
equal(p1,f1),
equal(p1,f2),
equal(p1,f3),
equal(p1,p2),
equal(p1,p3),
equal(p1,state),
equal(p1,fork1),
equal(p1,fork2),
equal(p1,ontable),
equal(p2,f1),
equal(p2,f2),
equal(p2,f3),
equal(p2,p1),
equal(p2,p3),
equal(p2,state),
equal(p2,fork1),
equal(p2,fork2),
equal(p2,ontable),
equal(p3,f1),
equal(p3,f2),
equal(p3,f3),
equal(p3,p1),
equal(p3,p2),
```

```
equal(p3,state),
equal(p3,fork1),
equal(p3,fork2),
equal(p3,ontable),
equal(state,f1),
equal(state,f2),
equal(state,f3),
equal(state,p1),
equal(state,p2),
equal(state,p3),
equal(state,fork1),
equal(state,fork2),
equal(state,ontable),
equal(fork1,f1),
equal(fork1,f2),
equal(fork1,f3),
equal(fork1,p1),
equal(fork1,p2),
equal(fork1,p3),
equal(fork1,state),
equal(fork1,fork2),
equal(fork1,ontable),
equal(fork2,f1),
equal(fork2,f2),
equal(fork2,f3),
equal(fork2,p1),
equal(fork2,p2),
equal(fork2,p3),
equal(fork2,state),
equal(fork2,fork1),
equal(fork2,ontable),
equal(ontable,f1),equal(ontable,f2),
equal(ontable,f3),
equal(ontable,p1),
equal(ontable,p2),
equal(ontable,p3),
equal(ontable,state),
equal(ontable,fork1),
equal(ontable,fork2)))).

formula(not(or(
equal(tick3,tick2),
```

```
equal(tick3,tick1),
equal(tick3,tick),
equal(tick3,putdown),
equal(tick3,trypickup),
equal(tick2,tick3),
equal(tick2,tick1),
equal(tick2,tick),
equal(tick2,putdown),
equal(tick2,trypickup),
equal(tick1,tick3),
equal(tick1,tick2),
equal(tick1,tick),
equal(tick1,putdown),
equal(tick1,trypickup),
equal(tick,tick3),
equal(tick,tick2),
equal(tick,tick1),
equal(tick,putdown),
equal(tick,trypickup),
equal(putdown,tick3),
equal(putdown,tick2),
equal(putdown,tick1),
equal(putdown,tick),
equal(putdown,trypickup),
equal(trypickup,tick3),
equal(trypickup,tick2),
equal(trypickup,tick1),
equal(trypickup,tick),
equal(trypickup,putdown)))).

end_of_list.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
list_of_formulae(conjectures).

formula(
forall( ... ITICK1(x0_1,x0_2,x0_3,x0_4,x0_5,x0_6,x0_7,s0,s1,r)))
).

end_of_list.

end_problem.
```

Bibliography

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 9 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AG98] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, April 1997.

[AL98] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. SRC Research Reports SRC-161, Compaq SRC, September 1998. Extended version of [AL97].

[Ble79] Woodrow W. Bledsoe. A maximal method for set variables. *Machine Intelligence*, 9:53–100, 1979.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of on Computing*, 7(1):70–90, February 1978.

[dB91] Frank S. de Boer. *Reasoning about dynamically evolving process structures; a proof theory for the parallel object-oriented language POOL*. PhD thesis, The Free University of Amsterdam, 1991.

[dB99] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures, FoSSaCS '99*, volume 1578 of *LNCS*, pages 135–149. Springer-Verlag, 1999.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Reports SRC-159, Compaq SRC, December 1998.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *28th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, 2001.

[Gor88] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[Hen97] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In *FOOL4: 4th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 1997.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 284–303. Springer Verlag, 2000.

[HJP80] Martin Hyland, Peter Johnstone, and Andrew Pitts. Tripos theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88:205–232, 1980.

[HM94] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In T. F. Melham and J. Camilleri, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 859 of *LNCS*, pages 269–284. Springer-Verlag, 1994.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hof99] Martin Hofmann. Semantical Analysis of Higher-Order Abstract Syntax. In *Logic in Computer Science (LICS).* IEEE, Computer Society Press, 1999.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[Hom95] Peter V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures.* PhD thesis, University of California, Los Angeles, 1995.

[HT00] Martin Hofmann and Francis Tang. Implementing a program logic of objects in a higher-order logic theorem prover. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 267–282. Springer-Verlag, 2000.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Hui01] Marieke Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

[Ita86] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[JP01] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussman, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[Kle98] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, University of Edinburgh, 1998.

[Kow77] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7, 1977.

[LDG+01] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.02: Documentation and user's manual.* INRIA, 7 2001.

[Lei97] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages*, 1997.

[Lei98] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, Winter 1998.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Language and Systems*, 16(6):1811–1841, 11 1994.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Feruniversität Hagen, 2001.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 15–17, 1997.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI96*, pages 229–243, Seattle, WA, Oct 1996. USENIX Assoc.

[NvOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[Ohe99] David von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *LNCS*, pages 168–180. Springer-Verlag, 1999.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[ON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited, 2002. Submitted for publication.

[OP00] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 2000.

[Pal95] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[Pit02] Andrew Pitts. Tripos theory in retrospect. *Mathematical Structures in Computer Science*, 12:1–15, 2002.

[PJ97] Jens Palsberg and Trevor Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, Fall 1997.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[PZJ02] Jens Palsberg, Tian Zhao, and Trevor Jim. Automatic discovery of covariant read-only fields. In *FOOL9: 9th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 2002. Full version available from `http://www.cs.purdue.edu/homes/palsberg/publications.html`.

[Rey82] John C. Reynolds. Idealized Algol and its specification logic. In Danielle Néel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[Tan01] Francis Tang. Reducing proof burden in object-oriented verification. In *OOPSLA 2001 Conference Companion.* ACM, October 2001. Accompanying poster available from `http://www.dcs.ed.ac.uk/home/fhlt/`.

[TH02] Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects. In *FOOL9: 9th. Int. Workshop on Foundations of Object-oriented programming Languages*, January 2002. Full version available from `http://www.dcs.ed.ac.uk/home/fhlt/`.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.

[W+99] Christoph Weidenbach et al. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 378–382, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[WSR00] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *9th International Conference on Compiler Construction*, number 1781 in LNCS, pages 1–16. Springer-Verlag, 2000.