Dynamic Logic
for an Intermediate Language

Verification, Interaction and Refinement

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Mattias Ulbrich

aus Ludwigsburg

Tag der mündlichen Prüfung:	17. Juni 2013
Erster Gutachter:	Prof. Dr. Peter H. Schmitt Karlsruher Institut für Technologie
Zweite Gutachterin:	Assoc. Prof. Dr. Marieke Huisman University of Twente

Acknowledgements

I would not have been able to compose this thesis without the help of many who contributed to it in various ways.

First and foremost, I would like to express my special appreciation and gratitude to my advisor Professor Dr. Peter H. Schmitt for giving me the opportunity to undertake this dissertation project, for his continuous scientific advice and support, and for granting me so much freedom in my research. His gentle way of leading the group always made me feel at home.

I am thankful to my second reviewer Assoc. Prof. Dr. Marieke Huisman for the time and energy she invested into fulfilling this role. My thanks go also to Juniorprofessor Dr. Mana Taghdiri and Professor Dr. Jörn Müller-Quade for their service as examiners in the defense committee.

I have always very much enjoyed working together with my colleagues in the formal method groups at KIT. We had plenty of inspiring discussions, an excellent teamwork and lots of fun. My thanks go to my former colleagues Dr. Christian Engel, Dr. Benjamin Weiß and Dr. Frank Werner as well as to my current fellow researchers Thorsten Bormer, Daniel Bruns, Aboubakr Achraf El Ghazi, Dr. Stephan Falke, David Faragó, Dr. Christoph Gladisch, Sarah Grebing, Simon Greiner, Markus Iser, Dr. Vladimir Klebanov, Tianhai Liu, Florian Merz, Christoph Scheben and Dr. Carsten Sinz. I am thankful to all students who worked with me; in particular to Timm Felden who helped me implementing parts of the *ivil* tool. I am grateful to Professor Dr. Bernhard Beckert for giving me the opportunity to continue my research on verification in his group at KIT.

I would also like to express my thanks to the members of the KeY group with whom I was able to have interesting exchanges of thoughts, in particular to Professor Dr. Reiner Hähnle, Dr. Richard Bubel, Dr. Wojciech Mostowski, Martin Hentschel, Dr. Philipp Rümmer and Dr. Wolfgang Ahrend.

I owe my deepest gratitude to my parents and my family for the many years of their ongoing support and love without whom nothing would have been possible. Lastly and most importantly, infinitely many thanks to Martina for her love, vitality, care and constant encouragement from which I took the energy to finish this work. You are great.

Karlsruhe, June 2013 *Mattias Ulbrich*

Dynamische Logik für eine Zwischensprache

Verifikation, Interaktion und Verfeinerung

(Deutsche Zusammenfassung)

Computerisierte Systeme spielen in unserem Alltag eine immer wichtigere Rolle. Viele werden dabei in Bereichen eingesetzt, in denen ihr korrektes Verhalten von höchster Bedeutung ist, z. B. im Bereich von Medizintechnik oder der Flug- oder Automobiltechnologie. Immer häufiger werden dabei sicherheitskritische Entscheidungen automatisch von Softwaresystemen getroffen. Wenn diese fehlerhaft sind, so kann das fatale Auswirkungen haben, finanzieller Natur, oder gar Schaden an Leib und Leben von Menschen bedeuten.

Diese Arbeit beschäftigt sich damit, sicherzustellen, dass Software sich so verhält, wie sie sich verhalten soll. Präziser gesagt beschäftigt sie sich mit der deduktiven funktionalen Verifikation von Software-Implementierungen in Hinblick auf ihre formale Spezifikation.

Formale Methoden sind eine Alternative zum Durchführen von Softwaretests, wenn es darum geht, die funktionale Sicherheit von Systemen sicherzustellen. Die Methoden, die als „formal“ klassifiziert werden können, decken dabei eine große Bandbreite von Ansätzen und Werkzeugen ab. Diese unterscheiden sich im Abstraktionsgrad, mit dem sie ein System beschreiben und auf ihm operieren und in der Stärke der Aussagen, die sie treffen. Die Stärken reichen von leichtgewichtiger Unterstützung bei der Suche nach Fehlern (*Bugs*) bis hin zur Verifikation vollständig funktionaler Spezifikationen. Je stärker die von einer Analyse gemachten Aussagen jedoch sind, desto mehr Expertise, Zeit und Aufwand muss in ihre Verifikation investiert werden.

Diese Arbeit hält sich dabei im Bereich der Verifikation vollständiger funktionaler Spezifikationen auf. Dies ist ein aufwändiger schwergewichtiger formaler Ansatz; dennoch ist das Ziel, eine Verifikation auf der Ebene des implementierten Programmes zu bewerkstelligen und nicht (alleine) auf einer Abstraktion von ihm. Damit vereinigt der untersuchte Verifikationsansatz Schwierigkeiten aus zwei Gebieten: zum

einen die Probleme der Verifikation auf Implementierungsebene und zum anderen die der hohen Genauigkeit der Spezifikation.

Um mit dieser hohen Komplexität besser umgehen zu können, schlägt diese Arbeit vor, die Aufgabe der Verifikation mittels der zwei Werkzeuge *Interaktion* und *Verfeinerung* zu bändigen:

Interaktion Im Gegensatz zu leichtgewichtigen Ansätzen, für die es charakteristisch ist, dass sie in relativ großem Rahmen mit relativ geringem Aufwand anwendbar sind, ist die volle funktionale Verifikation signifikant aufwändiger und benötigt einen erfahrenen Benutzer.

In dieser Arbeit wird ein interaktives Verifikationswerkzeug vorgestellt, mit dem in den Verifikationsprozess Einblick gewonnen werden kann und das dem Benutzer des Systems erlaubt, mit dem System hauptsächlich in Begriffen des ursprünglichen zu verifizierenden Programmes zu interagieren. Mittels interaktiven Eingreifens kann die automatische Komponente des Systems geleitet werden, bis sie schließlich den Bewies selbständig zu Ende führen kann.

Verfeinerung Volle funktionale Verifikation auf der Ebene von Implementierungen vereinigt zwei an sich schwere Aufgaben in einer. Wir schlagen eine Methodik vor, diesen schweren Auftrag in zwei jeweils leichtere zu unterteilen und dann die etablierte Technik der Verfeinerung zu benutzen, um aus der abstrakteren Beschreibung die Implementierung formal abzuleiten.

Es gibt funktionale Korrektheitsaussagen, die am besten direkt auf der Implementierung beschrieben und dort leicht gezeigt werden können, und es gibt Korrektheitsaussagen, die von konzeptioneller Natur sind und besser auf einer Abstraktion des Programmes gezeigt werden.

Um diese beiden Ziele zu erreichen, wird eine neue Programmlogik entworfen, die zwei seit langem im Bereich der Implementierungsverifikation erfolgreiche Paradigmen vereint.

Zum einen ist dies die dynamische Logik, die einen sehr flexiblen Ansatz für die Formalisierung und den Beweis von Eigenschaften von Programmen darstellt. Sie ist eng verwandt mit anderen Programmlogiken wie der „Hoare-Logik“ oder dem „Weakest-Precondition-Kalkül“, hat aber als besondere Eigenschaft, dass Programme freier kombiniert werden können. Programme sind in dynamischer Logik Konstruktoren für Formeln. Da die Menge der Formeln unter Komposition abgeschlossen ist, können auch programminduzierte Formeln geschachtelt oder logisch verknüpft auftreten.

Zum anderen wird die Verifikation auf einer elementaren, reduzierten aber allgemeinen Verifikationszwischensprache in den Ansatz mit eingebracht. Die Idee, Verifikationsbedingungen einer modernen Programmiersprache auf eine solche elementare Sprache zu reduzieren, scheint auf den ersten Blick einen Rückschritt darzustellen. Aber dieser Eindruck täuscht. Eine schlanke und effiziente Kernsprache erlaubt es, den Verifikationsprozess vom Modellierungsprozess zu entkoppeln. Es gibt fortan das Verifikationswerkzeug und den Übersetzer, der die Beweisverpflichtung aus einem in einer Hochsprache geschriebenen Programm und seiner Spezifikation heraus erzeugt. Die Zwischensprache verbindet die beiden Systeme.

Ein prominenter Vertreter der Systeme, die auf auf dynamischer Logik beruhen, ist das KeY-System, das auf einer speziell ausgelegten dynamischen Logik für Java-Quelltext operiert. Ein erfolgreicher prototypischer Vertreter für ein Verifikationssystem, das auf einer Zwischensprache agiert, ist das System Boogie, für das eine Vielzahl von Übersetzern existiert, die Beweisverpflichtungen aus einer Reihe von Quellsprachen und Systemen heraus erlaubt. Diese Arbeit stellt eine dynamische Logik im Stile von KeY vor, die auf einer Zwischensprache definiert ist, die mit der Sprache von Boogie die primitiven Anweisungen gemeinsam hat.

Diese Arbeit beschreibt eine Methodik um Algorithmen in einer rein abstrakten, mathematischen Pseudocode-Sprache zu formulieren, wie sie aus Lehrbüchern bekannt ist. Diese Algorithmen können dann auf eine Implementierung in der Programmiersprache Java verfeinert werden. Interessante funktionale Eigenschaften, die konzeptionell im Algorithmus begründet sind können auf Ebene der mathematischen Modellierung beschrieben, untersucht und verifiziert werden, wobei die Implementierungsdetails bei dieser Betrachtung außen vor gelassen werden können. Ein formaler Verfeinerungsschritt überträgt dann die Verifikationsergebnisse auf die Implementierung, ohne die Beweise auf der detaillierten Ebene erneut führen zu müssen.

Der kombinierte Ansatz erlaubt eine besonders flexible Formulierung von Verifikationsaufgaben. Er ist flexibel in der Formulierung der der Programme, weil die Verifikationsprogrammiersprache besonders grundlegend und allgemein gehalten ist. Er ist flexibel in der Formulierung von Beweisverpflichtungen, wenn diese über mehrere Programme spricht, weil er von der dynamischen Logik übernimmt, dass mehr als ein Programm in einer Verifikationsaufgabe enthalten sein kann. Ein Vertreter dieser Art von Verifikationssystemen ist das Werkzeug Boogie, für das eine Vielzahl von Übersetzern existiert, die Beweisverpflichtungen aus einer Vielzahl von Quellsprachen und System heraus erlaubt.

Die deduktiven Verifikation vollständiger funktionaler Spezifikationen von Implementierungen in der Programmiersprache Java ist eine schwergewichtige formale Methode. Die vorliegende Arbeit setzt sich zum Ziel, diese Methode praktikabler zu machen. Die wesentlichen Beiträge der Arbeit sind:

- Eine Dynamische Logik *UDL* für eine Verifikationszwischensprache wird vorgestellt und formal definiert. Diese Logik vereinigt zwei Ideen, die in anderen Verifikationssystemen bereits erfolgreich eingesetzt worden sind, aber noch nie zusammengeführt wurden. Die dynamische Logik wird gründlich untersucht und mit der klassischen dynamischen Logik vergleichen wobei Gemeinsamkeiten und Unterschiede herausgearbeitet werden.

  Die Logik, auf der *UDL* basiert, ist eine Prädikatenlogik, die um Variablenbindersymbole (*Binder*) und parametrisierte Typen erweitert wurde. Es wird formal gezeigt, dass die Binder die Ausdrucksmächtigkeit der Sprache nicht erhöhen, die parametrisierten Typen dagegen sehr wohl über die Ausdrucksmächtigkeit der Logik erster Stufe hinausgehen. Ein Sequenzenkalkül für die Logik wird entwickelt. Er erlaubt die Verifikation von Formeln, die Programme enthalten. Invariantenregeln werden motiviert, vorgestellt und als korrekt bewiesen. Wir zeigen, dass der Kalkül relativ vollständig ist mit Bezug auf die Entscheidbarkeit von Aussagen über den natürlichen Zahlen ist.

- Eine Disziplin der formalen Algorithmenverfeinerung wird aufgebaut. Ausgehend von dem etablierten Begriff der formalen Verfeinerung wie er in abstrakten formalen System schon seit langem Verwendung findet, wird ein Begriff der formalen Verfeinerung von Algorithmen zu Implementierungen und eine entsprechende Verfeinerungsbedingung mit Hilfe der dynamischen Logik definiert. Es wird gezeigt, dass diese Bedingung mit den etablierten Begriffen der formalen Verfeinerung kongruent ist.

  Eine Menge von Regeln zur effizienten Behandlung von Schleifen in *UDL*-Verfeinerungsbedingungen wird vorgestellt. Da die Programmiersprache, die in *UDL* Verwendung findet, unstrukturiert ist, können existierende Ansätze zur Behandlung von gleichlaufenden Schleifen nicht direkt in den neuen Ansatz übernommen werden. Die neuen Regeln arbeiten mit dem Konzept von Synchronisationspunkten und schaffen es auf diese Weise, im Beweis die Schleife aufzulösen.

  Eine neue Technik, einen formalen Methodenvertrag für eine Java-Methode aus einem erfolgreichen Verfeinerungsbeweis zu extrahieren, ermöglicht es diesem Verfahren, das abstrakte Modell und die konkrete Implementierung miteinander zu verknüpfen.

- Die Logik *UDL*, ihr Sequenzenkalkül und die Generierung von Verfeinerungsbeweisverpflichtungen wurden im neuen interaktiven Verifikationssystem *ivil* implementiert. Obwohl die Beweise in diesem System auf einer technischen Zwischensprache formuliert sind, kann der Benutzer mit dem System in Begriffen der Quelltextsprache interagieren und bekommt Interaktionselemente, die an bekannte Elemente aus statischen Debugger-Systemen erinnern, an die Hand gereicht.

  Die Implementierung einer Übersetzung von Java-Bytecode in die unstrukturierte Zwischensprache, bei der die Interaktionsfähigkeit auf Quelltextebene erhalten wurde, dient als Machbarkeitsstudie für den Ansatz der Verifikation auf einer Zwischensprache.

Contents

List of Figures and Tables

List of Definitions

List of Theorems

Chapter 1

Introduction

1.1 Motivation

Computerised systems have become pervasive companions in our lives. Many of them are installed in places in which their functioning correctly is of utmost importance, for instance, in financially relevant systems or in safety-critical areas such as medical, aeronautical or automotive systems. More and more critical decisions are made by computer programs. If they fail, it may have fatal economical consequences or even cause people to come to harm.

This thesis is about ensuring that software behaves as it is supposed to behave. More precisely, it is concerned with the deductive verification of the compliance of software implementations with their formal specification.

Traditionally, functional safety of software is mostly guaranteed by performing sufficiently many tests of various kinds on a system. The choice and design of the test cases is crucial for the quality of the test process. Nevertheless, tests are seldom exhaustive and may, therefore, fail to uncover faults hidden in the system.

The use of *formal methods* is an alternative way to ensure correct functional system behaviour. Generally speaking, formal methods apply mathematically rigorous techniques to model and analyse systems. In recent years, quality-ensuring safety standards used for the certification of safety-critical systems have included formal methods as admissible instruments to establish functional correctness. The standard for automotive systems (ISO 26262, released in 2011) and the recent version of the standard for aeronautical systems (DO-178C, released in 2012) explicitly[1] mention formal methods as adequate means to ensure functional safety of safety-critical systems in various stages of the system design.

The methods which can be classified as formal cover a very broad area of approaches and tools. They differ considerably in the level of abstraction on which they describe, formalise and operate on the model. Some formal systems are applied

[1] *ISO-26262*: The standard lists formal verification as a "recommended method" for systems with higher safety level (ISO 26262-6, Req. 7.4.18)

DO-178C: "Formal methods [...] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification." (Dross et al., 2011) long before a program is implemented. They model a system on very abstract terms and thus help to identify design flaws in very early stages of the development process. Other methods operate on implemented code but abstract from it and show conceptual properties on the abstracted models. Again other systems model the behaviour of an implementation meticulously and thus examine behaviour on the implementational level.

Formal methods vary also in the impact of the statements they make. Their purposes range from providing a documentation with formal semantics, over the automatic identification of bugs or unexpected behaviour in implementations to *full functional verification* of software with respect to a formal specification. The more significant propositions of the last category come at a price: The closer the asserted property is to a full formal description, the higher are the expertise, time and effort that its verification requires.

This thesis is concerned with the formal verification of software implementations with respect to full functional specifications. The goal is to perform the verification on the level of the implementation, not (solely) by considering an abstraction. This kind of formal methods is amongst the most cost-intensive. It combines the difficulty of full functional verification with that of detailed implementation-level analysis. To better cope with the increased complexity of this *verification* task, the thesis proposes two instruments: *interaction* and *refinement*.

Verification Different people will understand different things under the notion of "verification". For the purposes of this thesis, verification means the application of deductive proving techniques to provide formal evidence that the implementation of a software system behaves according to its specification. This is done by conducting mathematically rigour formal proofs within a system of formal deduction. If a proof succeeds, then a program is ensured to behave as claimed in its specification.

In this thesis, two successful ideas in program verification are integrated into a new approach which combines the advantages of both: Dynamic logic is brought together with verification on an intermediate verification language.

Interaction In contrast to lightweight formal methods that, characteristically, can be applied on a relatively large scale with relatively little effort, full functional verification is inherently more difficult and requires an experienced specifier.

In this thesis, an interactive verification tool is presented which allows the user to interact with the system mostly in terms of the original program description. For verification tasks which require no interactive inspection, an automated version exists as well.

Refinement Full functional verification of implementations combines two difficult verification tasks in one. We propose a methodology to decompose this difficult task into two easier tasks using the well-established technique of refinement.

The refinement method allows a separation of concerns between the algorithmic, conceptual problem and the issues of the implementation.

1.2 Dynamic Logic and Intermediate Language

This thesis brings together the benefits of two verification paradigms which have proved to be successful in the field of implementation verification. We consequently devise a verification system which combines the ideas of both into a novel approach.

Dynamic Logic On the one hand, there is dynamic logic (Harel et al., 2000) which is a very flexible approach for the formalisation and proof of properties of a program. It is closely related to other program verification logics, in particular to Floyd-Hoare logic (Floyd, 1967; Hoare, 1969) and to the weakest precondition calculus (Dijkstra, 1975). What sets dynamic logic apart from other program logics is the possibility to freely combine program constructs. Programs are part of dynamic logic formulas, and with the set of formulas being closed under logical composition, programs can be nested and logically composed. This flexibility to formulate expressions makes dynamic logic a suitable vehicle for a variety of verification conditions. Besides functional compliance, for example, non-interference of systems or hybrid systems can be modelled.

A prominent representative of a verification system based on dynamic logic is the KeY system covered in the monograph by Beckert et al. (2007). KeY is a verification system for the interactive and automatic verification of sequential Java code. It is based on a dynamic logic for sequential Java source code tailored to the requirements of the tool.

In the context of KeY, dynamic logic has been used to model and verify a variety of problems: Weiß (2010) describes how modular program specification and verification can be accomplished with dynamic logic. Engel (2009) applies the KeY approach to verify safety-critical Java applications with realtime constraints. There, dynamic logic is used for the formalisation of the realtime memory model. Klebanov (2009) describes how dynamic logic can be adapted to allow the verification of multi-threaded Java-like programs. Scheben and Schmitt (2011) use the flexibility of dynamic logic to efficiently verify information flow properties for sequential Java programs. Beckert and Bruns (2012) present an extension of dynamic logic which combines the expressiveness of dynamic logic with that of temporal logic.

There are other systems besides the KeY project which support dynamic logic. KIV (Reif et al., 1995) operates on dynamic logic over various system description languages, amongst others for Java (Stenzel et al., 2008) or for state charts (Thums et al., 2004). Platzer (2010) presents a verification system for hybrid systems based on a dynamic logic.

Intermediate Language The other incorporated successful concept is the reduction of the syntactical material to a minimalistic but general programming language tailored to the needs of verification.

Such a special-purpose language may serve as an intermediate representation of proof obligations which originally stem from another program in a high level language. At the first glance, it seems like a step backward that one wants to verify programs in a high level programming language using such a rudimentary representation. But this impression is deceptive. The verification language can, due to its flexibility, become the target of a translation from the more powerful language, and the verification process thus be decoupled from the process of producing the proof obligation. This partition of the verification process into translation and verification is helpful: The verification system which operates on an intermediate verification language is independent of the input language and may serve as the base beneath other verification systems. The task of the verification system for a particular programming and specification language is reduced to a translation into the intermediate representation.The translated model in intermediate representation is an artifact in its own right and can be inspected and modified. This allows developers of verification systems to experiment within the design space of the translation.

The Why verification tool (Filliâtre and Marché, 2007) is one representative of such a system with intermediate language. Its input language is general enough to serve as target language for renowned verification systems for different programming languages: Krakatoa (Marché et al., 2004) for Java and Cadeceus/Jessie (Moy, 2009) for C.

The Boogie approach is another success story for intermediate verification languages. There exist verification tools for many programming languages which operate by translation to the Boogie verification language: AutoProof (Eiffel, Tschannen et al., 2011), HAVOC (C, Chatterjee et al., 2007), VCC (C, Dahlweid et al., 2009), Spec# (C#, Barnett et al., 2011) and verification systems for Java (Lehner and Müller, 2007) and Scala (Wüstholz, 2009) to name some. The intermediate language is also well suited for experimental research programming languages like Dafny (Leino, 2010a) or Chalice (Leino and Müller, 2009).

The dynamic logic presented in this thesis is inspired by the dynamic logic employed in the KeY system, but goes also beyond it in some respects. The programs which can be incorporated in formulas of the presented logic share the primitive statement constructors with the programming language of Boogie.

1.3 Implementation and Refinement

It has been mentioned that formal methods cover a wide range of abstraction degrees. There are systems which operate on very abstract system descriptions and systems that consider a very detailed model of the actual implementation.

This thesis builds a bridge between these two worlds of verification by applying the well-established instrument of*formal refinement*. Refinement is a standard technique in model verification to cover an abstraction gap. Many established formal methods rely upon it: Z (Woodcock and Davies, 1996), Abstract State Machines (Börger and Stärk, 2003), B (Abrial, 1996) and its successor Event-B (Abrial, 2010) define closely related theories of refinement. The refinement calculus (Morgan, 1990) allows stepwise refinement of abstract code.

Refinement has, however, seldom been applied to make a formal connection from abstract descriptions to program code that has been written (not generated) to implement them.

The thesis describes a methodology to formulate algorithms in a purely abstract, mathematical pseudocode language, like known from textbooks, which can then be formally refined to an implementation given in the Java programming language. This allows a canonical separation of verification concerns. Interesting functional properties can be defined, examined and verified on the algorithmic level where implementational details can be kept aside. The formal refinement then transfers the verification results into results on the implementation level without proving them again on that level. The intermediate verification language and dynamic logic are fundamental and essential building blocks of this refinement verification.

1.4 Contributions

This thesis contributes to the field of deductive software verification by bringing together well-established approaches in the field exploiting the benefits of each in the other. By combining an intermediate language with dynamic logic, it aims for flexibility in the formulation of verification goals. The reach of the instrument of refinement is extended such that implementational code can also be the goal of a refinement step.

In particular, the main contributions of the thesis are the following:

- A dynamic logic called *UDL* for an intermediate verification language is presented and formally defined. This logic incorporates two ideas that are widely used in other verification systems, but have not yet been combined. The dynamic logic is thoroughly examined and compared to classical dynamic logic, pointing out the commonalities and differences.

  The underlying logic is an extension of predicate logic with binders and parametrised types. Formal evidence is given that the former addition does not extend the expressiveness of the logic and that the latter extends it beyond that of first order logic.

  A sequent calculus for the logic is devised which allows the verification of proof obligation formulas containing intermediate programs. A set of invariants rules to deal with loops in the code is motivated, defined and formally proved sound. A proof is given to show that the calculus is relatively complete with respect to the decidability of sentences over natural numbers.

- A novel refinement technique for *UDL* is introduced which can be applied to establish a formal relation between two programs in the intermediate language of *UDL*. The technique can be used to formally refine the abstract description of an algorithm in pseudocode to an implementation in the Java programming language.

It is shown that the notion of refinement ensured by this technique agrees with the established definitions of refinement. Since the programming language of *UDL* is unstructured, existing approaches dealing with loops in refinement pairs cannot be transferred to *UDL*. A set of new inference rules for the efficient treatment of loops in refinement conditions is presented. The rules are proved sound.

A new technique to extract a formal contract for a Java method from a successful refinement proof allows this approach to connect abstract model verification with implementation verification.

- The calculus for *UDL* and the refinement condition rules have been implemented in the new interactive verification system *ivil*. Despite the intermediate verification language, the verification system allows interaction on the level of the source code of the verified program.

  A proof-of-concept translation from Java bytecode to the intermediate language which retains source code information on the intermediate level shows the feasibility of the interactive approach.

1.5 Outline

The main part of the thesis is comprised of five chapters which build up on one another.

In Chapter 2 the formal foundations are laid by defining the syntax and semantics of a dynamic logic called *unstructured dynamic logic*. The underlying parametrically typed predicate first order logic is defined first (2.2). This logic has constructs which go beyond standard first order logic (2.3). The predicate logic is then extended to a dynamic logic over the intermediate language (2.5).

In Chapter 3, a sequent calculus for unstructured dynamic logic is developed. Rules for symbolic execution (3.2) of unstructured programs are presented as well as a thorough examination of rules to handle unstructured loops using loop invariants and variants (3.3). It is proved that the calculus is relatively complete (3.4).

Chapter 4 reports on the implementation of an interactive verification system which puts the logic and calculus into practice. Special attention is turned to the issue of user interaction (4.2). The verification system uses industry constraint solvers to discharge proof obligations in the background; the translation from the system's logic to their input logic is also covered (4.3). To affirm the thesis that the intermediate language is well suited for interactive verification, a prototypical translation from Java bytecode to the intermediate representation is provided (4.4).

Chapter 5 brings abstract model software verification together with software verification on concrete implementations using the established methodology of *refinement*. After a brief introduction to the concept of refinement (5.2), refinement verification conditions are formulated as proof obligations in dynamic logic (5.3). In particular, the refinement of loops is considered extensively (5.4). We describe how refinement can be used to extract a formal contract for the implementation from a property proved for an abstract algorithm (5.5).

Chapter 6 reports on two case studies which use the refinement approach from Chapter 5 to prove non-trivial properties of non-trivial algorithms using the verification system presented in Chapter 4. First (6.1), selection sort is presented as a less complex algorithm which can be verified automatically. The second case study (6.2) reports on a more complex example, an algorithm for finding the shortest paths using breadth first search.

The concluding Chapter 7 summarises the contents of the thesis and gives an outlook onto conceivable future work extending the lines begun in this work.

The appendices provide additional formal definitions (Appendix A) and list the Java programs which were used as examples (Appendix B).

Chapter 2

Unstructured Dynamic Logic

In this chapter, we present the syntax and semantics of Unstructured Dynamic Logic *(UDL), a program logic to formulate verification conditions for an intermediate verification language. It is a variation of classical dynamic logic in which program code constitutes modal operators which describe state changes. The programs which are considered in UDL are non-deterministic, unstructured and may contain embedded assertions. The underlying logic in which the programs are embedded is a predicate logic with a parametrised type system.*

We first present the typed logic and then introduce programs and program formulas. We compare UDL to classical dynamic logic and examine the expressiveness of the predicate logic.

2.1 Types in Proof Assistants

Any deductive formal program verification system bases on a logic in which properties of programs can be formulated. In most cases, the underlying logic is a variation of predicate logic. First-order logic, Higher-order logic or quantifier-free logics are candidates for underlying logics. In many cases, verification logics add extensions on top of the base logic, like *Hoare-triples*, *separation-logic* or *dynamic logic*, which help to formalise proof obligations or to conduct proofs.

The design of the underlying logic includes the choice of a type system. While a type system is not strictly needed for a predicate logic, it is often a good decision to assign a *type* (in logics often called *sort*) of some shape to every expression in the logic. Types are used to differentiate objects of different kinds on a syntactical level already. There are several type systems which come into consideration for designing a predicate logic:

Untyped It is not technically obligatory to structure the domain of discourse any further. The entirety of all objects can be considered as one set with heterogeneous content. Most mathematically motivated material to show properties of predicate logic uses this basic model for its simplicity in presentation. Most fundamental phenomena of predicate logic (like decidability results, compactness, etc.) are properties of the untyped logic already.

Types are usually added as soon as the logic is no longer itself the object under investigation, but is used to denote and solve a formally described problem.

Some purely first order automatic theorem provers like Otter (Kalman, 2001) operate on untyped problem descriptions, so does the constraint solver Simplify (Detlefs et al., 2005).

Simply typed In the most simple case of typed logic, a finite number of types is introduced with every object of the domain belonging to exactly one type. Function symbols are annotated with argument and result types and the construction of terms and formulas is required to respect these types.

The simple logic with typed symbols is not more expressive than the untyped case. Enderton (1972, Sect. 4.3) shows how predicate logic with simple types can be reduced to untyped first order logic.

Formal systems that use simple typed logic include the first version of the SMT-LIB format, the *de facto* standard input language for predicate logic constraint solvers with theories (Ranise and Tinelli, 2003).

Parametrically typed Here, the set of types can consist of composed types. In such a system, types can be composed using type constructors which are applied to argument types. The application of function symbols needs not be limited to one dedicated type, but symbols can be polymorphic, and properties can be stated parametrically for all instantiations of type variables. This kind of type systems is often accredited to Hindley (1969).

Many modern proof environments like Isabelle/HOL, HOL, Boogie (Leino and Rümmer, 2010) and recent versions of SMT-LIB (Barrett et al., 2010) use variations of such a type system for their logic.

Hierarchically typed The type system of statically typed object oriented programming languages is hierarchical. An expression can belong to more than one type and the set of types forms a directed acyclic graph. In many cases even a (semi-)lattice.

The KeY tool targets at the verification of Java source code. That is why the underlying logic has got a hierarchical type system influenced by the Java type hierarchy.

A type hierarchy does not conflict with type parameters. A hierarchical type system with parameters has been proposed in (Ulbrich, 2007) to model Java generics.

Dependently typed Dependent types allow types to depend on values of the domain. The linear space $\mathbb{R}^n$ is a mathematical example. As a type it depends on n which is not a type but a value. In dependent type systems, type checking is as difficult as deciding satisfiability in the logic. This is why in systems with dependent types, showing well-typedness is already a substantial part of the verification task.

The interactive theorem provers PVS (Shankar and Owre, 1999) and Coq (Bertot, 2008) are renowned representatives of this kind.

The purpose of types in deductive proof systems is two-fold:

1. (*for the machine:*) The process of deduction in predicate logic inherently involves steps in which terms or formulas have to be unified (in a resolution step, for instance, or when instantiating a quantified formula, or when finding a closing unifier). Without annotating terms with types, there can be verification steps which are valid in an untyped setting but are not semantically sensible and will mislead the calculus. In a typed environment, however, typing constraints forbid their unification, and the according steps cannot be taken; types serve, hence, as a guidance to the proof system to choose sensible unification partners.

2. (*for the human:*) Type systems themselves are a (lightweight) formal method. The additional annotations in the declarations document their intentions and, if the well-typedness is decidable, serve as an early sanity check that prior to doing any attempts in a proof system with high complexity, the statement is at least well-typed.

   Additionally, types provide a concise way to notate information which must be encoded in the problem. If this knowledge is not given in types, then it must be expressed semantically (by adding further additional predicates). Hence, type systems help keeping proof obligations concise.

In the next section we will present a parametrically typed logic with variable types which allows quantification over type variables. We opted for this design for the following reasons:

- Well-typedness is decidable and ill-typed terms can be detected early on without exception. We want to limit the difficult tasks to the process of proving propositions, their well-formedness is meant to be a lightweight sanity check which can be performed automatically and which increases confidence that the formulated propositions are the intended.

- It is well suited to model general abstract datatypes in a data agnostic way. Many datatypes abstract away from the "payload" they have got. The behaviour of lists, for instance, can be described generically and later applied to concrete type instances.

- It is also well suited to accommodate verification conditions for more concrete problems stemming from concrete program languages. The type system is non-hierarchical even though the logic is meant to be used to model verification tasks for program languages which themselves do have type hierarchies - like in particular object oriented languages. Experiences in other verification systems (Spec$^{\#}$, Dafny, VCC, Krakatoa) show that handling the dynamic type issue semantically rather than by incorporating the type system works out well.

2.2 First Order Logic with Parametric Types

We begin by defining the type system in which types are composed using type variables and type constructors. The set $\mathrm{TVar} = \{\alpha, \beta, \gamma, \dots\}$ of type variables is fixed and does not depend on the type signature. By convention, we denote type variables by small Greek letters.

Definition 2.1 (Type signature) *A* type signature $\Gamma = (\mathrm{TCon}_\Gamma, \mathrm{ar}_\Gamma)$ *is comprised of a set* TCon_Γ *of type constructors and an arity function* $\mathrm{ar}_\Gamma : \mathrm{TCon}_\Gamma \to \mathbb{N}$ *assigning to each constructor symbol the number of type arguments it takes.*

We will throughout this section have a running example to illustrate the definitions of the section. For this example, let us look at the type signature

$$\Gamma_{ex} = (\{\mathsf{set}, \mathsf{nat}, \mathsf{bool}\}, \{\mathsf{set} \mapsto 1, \mathsf{nat} \mapsto 0, \mathsf{bool} \mapsto 0\})$$

which contains one unary type constructor set and the constructors nat and bool which do not take type arguments.

Definition 2.2 (Types) *For a given type signature* $\Gamma = (\mathrm{TCon}_\Gamma, \mathrm{ar}_\Gamma)$*, the set of* variable-free types *(also called* ground types*)* $\mathcal{T}^0_\Gamma$ *is the smallest set such that*

- $c \in \mathrm{TCon}_\Gamma$ *and* $t_1, \dots, t_{\mathrm{ar}_\Gamma(c)} \in \mathcal{T}^0_\Gamma \implies c(t_1, \dots, t_{\mathrm{ar}_\Gamma(c)}) \in \mathcal{T}^0_\Gamma$.

The set of types $\mathcal{T}_\Gamma$ *is the smallest set such that*

- $\mathrm{TVar} \subseteq \mathcal{T}_\Gamma$,
- $c \in \mathrm{TCon}_\Gamma$ *and* $t_1, \dots, t_{\mathrm{ar}_\Gamma(c)} \in \mathcal{T}_\Gamma \implies c(t_1, \dots, t_{\mathrm{ar}_\Gamma(c)}) \in \mathcal{T}_\Gamma$.

Obviously, any variable-free type is a type, that is $\mathcal{T}^0_\Gamma \subseteq \mathcal{T}_\Gamma$.

In the above example type system Γ_{ex} the sentences $\mathsf{nat}, \mathsf{set}(\mathsf{set}(\mathsf{bool})) \in \mathcal{T}^0_{ex}$ are ground types and $\alpha, \mathsf{set}(\beta) \in \mathcal{T}_{ex}$ non-ground types.

Often in the following we will need to talk about the type variables which appear in a type sentence or a tuple of types and, therefore, define the function $\mathrm{typeVars} : \mathcal{T}^+_\Gamma \to 2^{\mathrm{TVar}}$ as

$$\begin{aligned} \mathrm{typeVars}(\alpha) &:= \{\alpha\} && \alpha \in \mathrm{TVar} \\ \mathrm{typeVars}(C(T_1, \dots, T_{\mathrm{ar}_\Gamma(C)})) &:= \bigcup_{i=1}^{\mathrm{ar}_\Gamma(C)} \mathrm{typeVars}(T_i) && C \in \mathrm{TCon}_\Gamma, T_i \in \mathcal{T}_\Gamma \\ \mathrm{typeVars}(\langle T_1, ..., T_n \rangle) &:= \bigcup_{i=1}^{n} \mathrm{typeVars}(T_i)\ . \end{aligned}$$

Definition 2.3 (Type substitution) *A function* $\tau : A \to \mathcal{T}_\Gamma$ *with* $A \subseteq \mathrm{TVar}$ *is called a* type substitution. *We extend every substitution to a function* $\tau : \mathcal{T}_\Gamma \to \mathcal{T}_\Gamma$ *on all types by setting:* $\tau(c(t_1, \dots, t_{\mathrm{ar}_\Gamma(c)})) := c(\tau(t_1), \dots, \tau(t_{\mathrm{ar}_\Gamma(c)}))$, $\tau(\beta) = \beta$ *for* $\beta \notin A$.

Often we are interested in type substitutions which assign a type to a finite set $A = \{\beta_1, \ldots, \beta_k\}$ of type variables. In this case, we write $\{\beta_1/T_1, \beta_2/T_2, \ldots, \beta_k/T_k\}$ to denote the substitution.

In our running example, the application of the type substitution $\tau = \{\alpha/\mathsf{nat}\}$ on the pair $\langle \alpha, \mathsf{set}(\alpha)\rangle$ of types results in the type pair $\langle \mathsf{nat}, \mathsf{set}(\mathsf{nat})\rangle$.

2.2.1 Simple Terms

Having fixed a type system Γ, we continue by defining terms. The set of variables we can use depends on the type signature. For the given Γ, we assume that the set Var_Γ of variables contains infinitely many variables for every type (including the non-ground types containing type variables). By convention we shall use single letter identifiers from the end of the alphabet to denote variables. We will add the type to the variable symbol as a superscript where necessary.

Definition 2.4 (Signature) *Given a type signature* Γ, *a* (term) signature $\Sigma = (\mathrm{Fct}_\Sigma, \mathrm{Bnd}_\Sigma, \mathrm{ty}_\Sigma)$ *is comprised of*

- *a set of function symbols* Fct_Σ,
- *a set of binder symbols* Bnd_Σ *(disjoint from* Fct_Σ*)*
- *and a typing function* $\mathrm{ty}_\Sigma : \mathrm{Fct}_\Sigma \cup \mathrm{Bnd}_\Sigma \to \mathcal{T}_\Gamma^+$.

We will deal first with function symbols in the following and postpone the treatment of binder symbols until Section 2.3.1.

Conventions

- A function symbol $f \in \mathrm{Fct}_\Sigma$ with $\mathrm{ty}_\Sigma(f) = \langle T_1, \ldots, T_n, T\rangle$ is called n-ary with *parameter types* $T_1, \ldots, T_n$ and *result type* T.
- A function symbol $f \in \mathrm{Fct}_\Sigma$ with $\mathrm{typeVars}(\mathrm{ty}_\Sigma(f)) = \emptyset$ is called *monomorphic* and
- A function symbol $f \in \mathrm{Fct}_\Sigma$ with $\mathrm{typeVars}(\mathrm{ty}_\Sigma(f)) \neq \emptyset$ is called *polymorphic*.
- The sequence of types in $\mathrm{ty}_\Sigma(f)$ describes the "signature" of the function symbol. Therefore we write $f : T_1 \times \ldots \times T_n \to T$ for a function symbol with $\mathrm{ty}_\Sigma(f) = \langle T_1, \ldots, T_n, T\rangle$.
- Unary function symbols with $\mathrm{ty}_\Sigma(c) = \langle T\rangle$ are called *constants*, and we write $c : T$.

Extending our example from above and its type signature Γ_{ex}, we define an example signature $\Sigma_{ex} = (\mathrm{Fct}_{ex}, \mathrm{Bnd}_{ex}, \mathrm{ty}_{ex})$ with $\mathrm{Fct}_{ex} = \{\mathsf{empty}, \mathsf{singleton}, \mathsf{union}, \mathsf{in}, \mathsf{zero}, \mathsf{suc}\}$ and

- $\mathsf{ty}_{ex}(\mathsf{empty}) = \langle \mathsf{set}(\beta) \rangle$
- $\mathsf{ty}_{ex}(\mathsf{singleton}) = \langle \beta, \mathsf{set}(\beta) \rangle$
- $\mathsf{ty}_{ex}(\mathsf{union}) = \langle \mathsf{set}(\beta), \mathsf{set}(\beta), \mathsf{set}(\beta) \rangle$
- $\mathsf{ty}_{ex}(\mathsf{in}) = \langle \beta, \mathsf{set}(\beta), \mathsf{bool} \rangle$
- $\mathsf{ty}_{ex}(\mathsf{zero}) = \langle \mathsf{nat} \rangle$
- $\mathsf{ty}_{ex}(\mathsf{suc}) = \langle \mathsf{nat}, \mathsf{nat} \rangle$

Using the conventions from above, we could have instead of the third constraint have written $\mathsf{union} : \mathsf{set}(\beta) \times \mathsf{set}(\beta) \to \mathsf{set}(\beta)$. The intention of the introduced signature is obvious: three constructors for sets (for the empty set, singleton sets and the union), the set-membership predicate in and two constructors for natural numbers (zero and successor) are given. The first four are polymorphic function symbols (with type parameter β), while the last two are monomorphic symbols.

Variables and function symbols can be used to construct terms. The following Definition 2.5 will later be subject to extensions when additional types of terms are introduced (Def. 2.10 for binders, Def. 2.12 for type quantifiers, and Def. 2.15 for program formulas and updates). Appendix A.1.1 lists a comprehensive version including the cases of all definitions.

Definition 2.5 (Terms) *Let the type signature* Γ *and the signature* Σ *be given. For every type* $T \in \mathcal{T}_\Gamma$*, the set* Trm_Σ^T *of* terms of type T *is defined as the smallest set such that the following inductive conditions hold:*

1. $$x^T \in \mathrm{Var}_\Gamma \\ \implies x^T \in \mathrm{Trm}_\Sigma^T$$

2. $$\begin{aligned} & f \in \mathrm{Fct}_\Sigma,\ \tau : \mathrm{typeVars}(\mathrm{ty}_\Sigma(f)) \to \mathcal{T}_\Gamma \\ & \langle T_1, \dots, T_n, T \rangle = \tau(\mathrm{ty}_\Sigma(f)), \\ & t_i \in \mathrm{Trm}_\Sigma^{T_i}\ (1 \le i \le n) \\ \implies & f^{[\tau]}(t_1, \dots, t_n) \in \mathrm{Trm}_\Sigma^T \end{aligned}$$

Like in untyped first order logic, terms are constructed by applying a function symbol to a fixed number of terms (the arguments). In addition to the argument terms, however, here we need to specify the instantiations of the type variables of the symbol's signature as well. A polymorphic function symbol can be seen as a "template" symbol standing for an entire family of function symbols and the parametrisation τ in the definition specifies one particular member of the family. Please note that terms may also have a type containing type variables, that is, they can be polymorphic as well. Nonetheless, every term belongs to exactly one type: $t \in \mathrm{Trm}_\Sigma^T \cap \mathrm{Trm}_\Sigma^U \implies T = U$.

Let us look at some correctly constructed terms in our sample signature Σ_{ex}:

$$\begin{aligned}
x^{\mathsf{nat}} &\in \mathrm{Trm}_{ex}^{\mathsf{nat}} && (2.1)\\
y^{\mathsf{set}(\beta)} &\in \mathrm{Trm}_{ex}^{\mathsf{set}(\beta)} && (2.2)\\
\mathsf{zero}^{[\emptyset]} &\in \mathrm{Trm}_{ex}^{\mathsf{nat}} && (2.3)\\
\mathsf{suc}^{[\emptyset]}(\mathsf{suc}^{[\emptyset]}(\mathsf{zero}^{[\emptyset]})) &\in \mathrm{Trm}_{ex}^{\mathsf{nat}} && (2.4)\\
\mathsf{empty}^{[\mathsf{id}]} &\in \mathrm{Trm}_{ex}^{\mathsf{set}(\beta)} && (2.5)\\
\mathsf{empty}^{[\{\beta/\mathsf{nat}\}]}, \mathsf{singleton}^{[\{\beta/\mathsf{nat}\}]}(\mathsf{zero}^{[\emptyset]}) &\in \mathrm{Trm}_{ex}^{\mathsf{set}(\mathsf{nat})} && (2.6)
\end{aligned}$$

The first two examples (2.1) and (2.2) show variables which are exactly of the type to which they are appointed. (2.3) and (2.4) are examples of monomorphic function applications for which no type substitution (hence $\emptyset$) is needed. As shown in (2.5), the type substitution needs not assign variable-free types to the type variables. Here $\mathsf{id} = \{\beta/\beta\}$ is used as instantiation, the resulting term is polymorphic. Finally, in (2.6) the type variable β is substituted for nat to construct the empty set and the singleton set $\{0\}$.

On the other hand, the text $\mathsf{singleton}^{[\emptyset]}(\mathsf{zero}^{[\emptyset]})$ is *not* a valid term of any type since the instantiated function symbol $\mathsf{singleton}^{[\emptyset]}$ can only be applied to a term of type β, but $\mathsf{zero}^{[\emptyset]}$ is of type nat and does, hence, not fulfil the requirement.

Type inference It is obvious that this notation which explicitly requires the specification of type variable instantiations under any circumstances is very verbose. The good news is that it is not always required to notate them but that they can be calculated. A *type inference* algorithm can be used to compute type substitutions for a text without substitutions such that the result gives a valid term. Without going into details here, this is a simple special case of the type inference situation presented by Hindley (1969) and Damas and Milner (1982) which is efficiently decidable; the most general type substitution can be computed in polynomial time. If there is no type substitution to make the sentence a well-formed term, this will also be detected.

We will, hence, in the remainder of this thesis drop the type substitution entirely if the context is unambiguous and assume that the most general type – as inferred by the algorithm – is used. We may also choose to drop the type variable to substitute and notate only the substituted type if the context is clear. For instance, the three sentences

$$\mathsf{singleton}^{[\{\beta/\mathsf{nat}\}]}(\mathsf{zero}^{[\emptyset]}), \quad \mathsf{singleton}^{[\mathsf{nat}]}(\mathsf{zero}^{[]}), \quad \mathsf{singleton}(\mathsf{zero})$$

denote the same term.

The sentence $\mathsf{union}(\mathsf{singleton}(\mathsf{zero}), \mathsf{singleton}(\mathsf{empty}))$, on the other hand, cannot be a valid term expression for any type variable instantiation. The datatype set we have created here is strictly typed unlike in fundamental set theory. A set of naturals cannot be combined with a set over any other type.

2.2.2 Semantics

In the following, we assume a type signature Γ and a signature Σ for Γ to be given. For the sake of readability, we drop the indices Γ and Σ in this section if the context is clear.

Terms are evaluated in semantic structures. In untyped predicate logic the structure contains a set of *object of discourse* upon which terms evaluate. In a multi-sorted environment that set is more structured. The domain is partitioned into subsets over the ground types. Every element belongs to exactly one variable-free type. There are no elements for non-ground types, and no element belongs to more than one type.

The interpretation maps function symbols to functions on the object level. In case of parametrised logic this implies that the lifted function also depends (in its range, domain and actual valuation) on the instantiation of the type variables. As for the domains, only ground-types are taken into consideration here.

Definition 2.6 (Semantic Structure) *A* semantic structure $D = ((\mathcal{D}^T)_{T \in \mathcal{T}^0}, I)$ *is comprised of*

- *a family* $(\mathcal{D}^T)_{T \in \mathcal{T}^0}$ *of pairwise disjoint, non-empty sets (called* domains*) and*

- *an* interpretation I *which assigns to every function symbol* $f : T_1 \times \ldots \times T_n \to T \in \mathrm{Fct}$ *and every ground type substitution* $\tau : \mathrm{typeVars}(\mathrm{ty}(f)) \to \mathcal{T}^0$ *a function*

  $$I(f^{[\tau]}) : \mathcal{D}^{\tau(T_1)} \times \ldots \times \mathcal{D}^{\tau(T_n)} \to \mathcal{D}^{\tau(T)}$$

  and to every binder symbol $b \in \mathrm{Bnd}$ *with* $\mathrm{ty}(b) = \langle T_v, T_1, \ldots, T_n \rangle$ *and every ground type substitution* τ *a function*

  $$I(b^{[\tau]}) : (\mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T_1)}) \times \ldots \times (\mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T_1)}) \to \mathcal{D}^{\tau(T)}$$

Again, the treatment of binder symbols will be postponed until Section 2.3.1. If $f \in \mathrm{Fct}$ is a polymorphic function symbol, its valuation depends on the "typing context" as given by the ground type substitution τ. The evaluation function of function symbol f is $I(f^{[\tau]})$ and depends also on the typing context. Different type parameters make differently typed evaluation functions for the same symbol. For a monomorphic function symbol, there is only one ground substitution (the empty function) and the definition coincides with the situation in simply typed first order logic: There is only one semantic function $I(f)$ for the symbol f.

In our running example, we can define one semantic structure as follows (this is not the only valid choice, but it is the intended standard definition):

$$\begin{array}{rcl}
\mathcal{D}_{ex}^{\mathsf{nat}} & = & \mathbb{N} \\
\mathcal{D}_{ex}^{\mathsf{bool}} & = & \{\mathit{ff}, \mathit{tt}\} \\
\mathcal{D}_{ex}^{\mathsf{set}(T)} & = & 2^{\mathcal{D}_{ex}^{T}} \\
I_{ex}(\mathsf{zero}) & = & 0 \\
I_{ex}(\mathsf{suc})(n) & = & n+1, \quad n \in \mathbb{N} \\
I_{ex}(\mathsf{empty}^{[T]}) & = & \emptyset_T \\
I_{ex}(\mathsf{singleton}^{[T]})(x) & = & \{x\}, \quad x \in \mathcal{D}_{ex}^{T} \\
I_{ex}(\mathsf{union}^{[T]})(s,u) & = & s \cup u, \quad s,u \subseteq \mathcal{D}_{ex}^{\mathsf{set}(T)} \\
I_{ex}(\mathsf{in}^{[T]})(x,s) = \mathit{tt} & \Leftrightarrow & x \in s, \quad x \in \mathcal{D}_{ex}^{T},\, s \in \mathcal{D}_{ex}^{\mathsf{set}(T)}
\end{array}$$

where $T \in \mathcal{T}^0$ ranges over all ground types.

A word about empty sets in this example: By requirement of Definition 2.6, all domains must be pairwise disjoint. This implies that $\mathcal{D}^{\mathsf{set}(\mathsf{nat})}$ and $\mathcal{D}^{\mathsf{set}(\mathsf{set}(\mathsf{nat}))}$ *must not* have any element in common. However, both should contain the empty set and would therefore not be disjoint. We overcome this dilemma by having a different empty set object $\emptyset_T$ for every $T \in \mathcal{T}^0$. These behave identically under the membership predicate $\in$ (always resulting in ff), but are in different domains.

To talk about the set of objects of more than one domain, we use the abbreviations

$$\mathcal{D} = \bigcup_{T \in \mathcal{T}^0} \mathcal{D}^T \qquad \text{and} \qquad \mathcal{D}^c = \bigcup_{\bigwedge_{i \le \mathrm{ar}(c)} T_i \in \mathcal{T}^0} \mathcal{D}^{c(T_1,\ldots,T_{\mathrm{ar}(c)})},\ c \in \mathrm{TCon}$$

to denote the entirety $\mathcal{D}$ of all objects and the set $\mathcal{D}^c$ of all objects belonging to some instantiation of a polymorphic type constructor c. In the above example this means that $\mathcal{D}_{ex}^{\mathsf{set}} = \bigcup_{T \in \mathcal{T}_{ex}^0} \mathcal{D}_{ex}^{\mathsf{set}(T)}$ is the set of all homogeneous sets within $\mathcal{D}$ in which all elements have a common type $T \in \mathcal{T}^0$.

Definition 2.7 ((Type) variable assignment)

A type variable assignment *is a function* $\tau : \mathrm{TVar} \to \mathcal{T}^0$*.*

A variable assignment *is a function* $\beta : \mathrm{Var} \to \mathcal{D}$*.*

A variable assignment is called compatible *with a type variable assignment* τ *if for any* $x^T \in \mathrm{Var}$*, the condition* $\beta(x) \in \mathcal{D}^{\tau(T)}$ *holds.*

The concept of compatibility is essential for the evaluation of expressions ensuring that a term evaluates to the domain corresponding to its type. For instance, the evaluation of the term $\mathsf{singleton}^{[\alpha]}(y^\alpha)$ depends on the choice of the variable elements α and y^α, and, hence, on the used type variable assignment τ and variable assignment β. We lose compatibility if we chose $\tau(\alpha) = \mathsf{bool}$ and $\beta(y^\alpha) = 1$ since being of type α, the variable y^α should be evaluated to an element of the domain to which α points.

Definition 2.8 (Evaluation of terms) *For a semantic structure $D = (\mathcal{D}, I)$, a type variable assignment τ and a variable assignment β compatible with τ, the evaluation function* $\mathrm{val}_{I,\tau,\beta} : \mathrm{Trm} \to \mathcal{D}$ *is defined inductively as follows:*

$$\mathrm{val}_{I,\tau,\beta}(x) = \beta(x)$$
$$\mathrm{val}_{I,\tau,\beta}(f^{[\sigma]}(t_1,\ldots,t_n)) = I(f^{[\tau\circ\sigma]})(\mathrm{val}_{I,\tau,\beta}(t_1),\ldots,\mathrm{val}_{I,\tau,\beta}(t_n))$$

for $x \in \mathrm{Var}, f \in \mathrm{Fct}, \sigma : \mathrm{typeVars}(\mathrm{ty}(f)) \to \mathcal{T}$.

This definition will be amended in Definitions 2.11 for binder symbols, 2.13 for type quantifiers, and in 2.16 and 2.18 for program formulas and updates. Appendix A.1.2 lists a comprehensive version including the cases of all definitions. The definition here resembles the definition of text book predicate logic in that the evaluation of a compound term $f(t_1,\ldots,t_n)$ is broken down to the evaluation of the subterms $t_1,\ldots,t_n$ whose results are then applied to the semantic function $I(f)$. However, to deal with types containing type variables, Definition 2.8 employs a type variable assignment τ to map all occurrences of type variables to ground types prior to evaluation.

Let us return to the running example of this chapter and evaluate the two terms $\mathsf{singleton}^{[\mathsf{nat}]}(\mathsf{zero}^{[\emptyset]})$ and $\mathsf{singleton}^{[\{\beta/\alpha\}]}(y^\alpha)$. The second example is evaluated with $\tau = \{\alpha/\mathsf{nat}\}$, $\beta = \{y^\alpha/5\}$.

$$\begin{array}{ll}
\quad\mathrm{val}_{I_{ex},\tau,\beta}(\mathsf{singleton}^{[\mathsf{nat}]}(\mathsf{zero}^{[\emptyset]})) & \quad\mathrm{val}_{I_{ex},\{\alpha/\mathsf{nat}\},\{y^\alpha/5\}}(\mathsf{singleton}^{[\{\beta/\alpha\}]}(y^\alpha)) \\
= I_{ex}(\mathsf{singleton}^{[\mathsf{nat}]})\big(I_{ex}(\mathsf{zero}^{[\emptyset]})\big) & = I_{ex}(\mathsf{singleton}^{[\{\alpha/\mathsf{nat}\}\circ\{\beta/\alpha\}]})(\beta(y^\alpha)) \\
= \{0\} & = I_{ex}(\mathsf{singleton}^{[\{\beta/\mathsf{nat}\}]})(5) \\
 & = \{5\}
\end{array}$$

Theorem 2.1 (Compatibility of evaluation) *Given a semantic structure D, a type variable assignment τ, and a variable assignment β compatible with τ, the evaluation of terms is well-defined and compatible with τ, that is:*

$$t \in \mathrm{Trm}^T \implies \mathrm{val}_{I,\tau,\beta}(t) \in \mathcal{D}^{\tau(T)}$$

PROOF The statement is true for variables since the variable assignment β is compatible with τ by assumption. Let $f : T_1 \times \ldots \times T_n \to T$ be a function symbol and $\sigma : \mathrm{typeVars}(\mathrm{ty}(f)) \to \mathcal{T}$, and $t_i \in \mathrm{Trm}^{\sigma(T_1)}$. The semantic function $I(f^{[\tau\circ\sigma]})$ has the range $\mathcal{D}^{(\tau\circ\sigma)T}$ according to Definition 2.6, and, hence, for $f^{[\sigma]}(t_1,\ldots,t_n) \in \mathrm{Trm}^{\sigma(T)}$ we get $\mathrm{val}_{I,\tau,\beta}(f^{[\sigma]}(t_1,\ldots,t_n)) = I(f^{[\tau\circ\sigma]})(\mathrm{val}_{I,\tau,\beta}(t_1,\ldots,t_n)) \in \mathcal{D}^{\tau(\sigma(T))}$ which is compatible with respect to τ.

By induction hypothesis we know $t_i \in \mathrm{Trm}^{T_i}$ entails that $\mathrm{val}_{I,\tau,\beta}(t_i) \in \mathcal{D}^{(\tau\circ\sigma)T_i}$. It is this fact that ensures the arguments to the function application on the right hand side in Definition 2.8 are in the respective domains: The definition is well-defined.□

2.2.3 Formulas

Up to this point, we only mentioned terms and never spoke of formulas even though the latter are the fundamental building blocks for sentences in a logic. The reason is that in this framework every formula is also a term; a term of the distinguished type bool. This unifying view to terms and formulas simplifies upcoming definitions and observations; predicate symbols are only boolean function symbols, and need not be treated any specially.

In the following, we assume that the type signature contains a nullary type constructor bool and that the evaluation is limited to structures with $\mathcal{D}^{\mathsf{bool}} = \{tt, f\!f\}$, in which tt is the semantic value for "true" and $f\!f$ for "false".

Definition 2.9 (Formulas) *The set* $\mathrm{Trm}^{\mathsf{bool}}$ *of terms of type* bool *is called the* set of formulas.

We write $I,\tau,\beta \models \varphi$ *for a formula* $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$ *to denote that* $val_{I,\tau,\beta}(\varphi) = tt$.

We write $I \models \varphi$ *if* $I,\beta,\tau \models \varphi$ *for every type variable assignment* τ *and every variable assignment* β *compatible with* τ.

We write $\models \varphi$ *if* $I \models \varphi$ *for every semantic structure* $(\mathcal{D}, I)$*, the formula is then called* (universally) valid.

To complete the embedding of formulas, we assume from now on that the signature Σ contains the unary function symbol $\neg : \mathsf{bool} \to \mathsf{bool}$, and the binary symbols $\wedge, \vee, \to, \leftrightarrow : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$, $\doteq : \alpha \times \alpha \to \mathsf{bool}$ and $\dot{\approx} : \alpha \times \beta \to \mathsf{bool}$. We will write these symbols in their usual prefix or infix notation. The symbols bind their arguments with different strengths, $\doteq$, $\dot{\approx}$ bind the most, followed by $\neg$, $\wedge$ binds more than $\vee$ and $\to$, and $\leftrightarrow$ binds least. We constrain the semantic structures which interpret formulas to those in which the propositional junctors have their obvious intended meaning:

$$\begin{aligned}
I,\tau,\beta \models \neg a &\iff \mathrm{val}_{I,\tau,\beta}(a) = f\!f \\
I,\tau,\beta \models a \wedge b &\iff I,\tau,\beta \models a \text{ and } I,\tau,\beta \models b \\
I,\tau,\beta \models a \vee b &\iff I,\tau,\beta \models a \text{ or } I,\tau,\beta \models b \\
I,\tau,\beta \models a \to b &\iff I,\tau,\beta \models a \text{ implies } I,\tau,\beta \models b \\
I,\tau,\beta \models a \leftrightarrow b &\iff I,\tau,\beta \models a \text{ if and only if } I,\tau,\beta \models b \\
I,\tau,\beta \models t_1 \doteq t_2 &\iff \mathrm{val}_{I,\tau,\beta}(t_1) = \mathrm{val}_{I,\tau,\beta}(t_2) \\
I,\tau,\beta \models t_1 \dot{\approx} t_2 &\iff \mathrm{val}_{I,\tau,\beta}(t_1) = \mathrm{val}_{I,\tau,\beta}(t_2)
\end{aligned}$$

Two different equality symbols $\doteq : \alpha \times \alpha \to \mathsf{bool}$ and $\dot{\approx} : \alpha \times \beta \to \mathsf{bool}$ are introduced. In multi-sorted logic, the equality is usually restricted to elements of the same type only. To express this kind of equality, the symbol $\doteq$ is used. As the logic has type variable types which stand for other types, it is possible that two terms with different types still have the same value. The two variables x^α and x^β may evaluate to the same value if $\tau(\alpha) = \tau(\beta)$. The *weakly typed equality* $x^\alpha \dot{\approx} x^\beta$ can be used in these cases. In general, it is better to use strongly typed equality if possible as it comprises equality both on the values and on the types.

We say that two terms $t_1, t_2 \in \mathrm{Trm}^T$ are semantically equal (and write $t_1 \equiv t_2$) if $\mathrm{val}_{I,\tau,\beta}(t_1) = \mathrm{val}_{I,\tau,\beta}(t_2)$ for all structures and variable assignments. This is the same as saying that $t_1 \doteq t_2$ is universally valid.

2.3 Extensions to Standard Logic

In the previous section we have presented a first order predicate logic with a parametric type system. Apart from the type system, the logic did not go beyond what usually is expected in any first order introduction. In this chapter we will now add two logical extensions which are not new, but less common for logic definitions. We will also discuss the implications on the expressive power of the resulting logic.

Historically and mostly in the context of mathematical logic, definitions of terms and formulas are kept with as few constructors as possible to facilitate their theoretical examination. The additions we make in the following enrich the syntax of the logic. While this makes theoretical examinations more laborious, this is sensible if the logic is used as a vehicle to describe problems rather than a subject of investigation itself.

2.3.1 Binder Symbols

In mathematically motivated contexts, symbols often bind a variable ranging over a set of values. Prominent instances of such binding symbol are the summation symbol Σ or the logical quantifiers $\forall, \exists$. Table 2.1 shows some typical binder symbols with a typical use case and the corresponding translation into an extended predicate logic with binder symbols.

Formally, binders are function symbols which bind a variable that can then appear in its argument terms as a bound variable. The notion of a term signature (Definition 2.4) already contains a set Bnd_Σ of binder symbols which have been ignored up to this point. Like for a non-binding function symbol, $\mathrm{ty}(b) = \langle T_v, T_1, \ldots, T_n, T \rangle$ specifies the type signature of a binder symbol. The first element T_v of the sequence is the type of the bound variable, followed by the types of the arguments of the binder. The last element describes the type of the resulting term.

Syntactically, binder terms are very similar to function applications (see second case in Definition 2.5). Binder symbols may be polymorphic as well, and when used to compose terms, the instantiation of the type variables in the signature must be also explicitly stated in the same way as for function symbols.

Definition 2.10 (Binder terms, Amendment to Definition 2.5) *In addition to the cases in Definition 2.5, the following construction creates terms:*

Quantification:	$\forall x.\ x * x \geq 0$	$(\mathsf{forall}\ x^{\mathsf{nat}}.\ x^{\mathsf{nat}} * x^{\mathsf{nat}} \geq 0)$
Minimum of an expression:	$\min\ x.\ x + x * x$	$(\mathsf{min}\ x^{\mathsf{nat}}.\ x^{\mathsf{nat}} + x^{\mathsf{nat}} * x^{\mathsf{nat}})$
Summation:	$\sum_{x=a}^{b} x * x$	$(\mathsf{sum}\ x^{\mathsf{nat}}.\ a, b, x^{\mathsf{nat}} * x^{\mathsf{nat}})$
Set Comprehension:	$\{x \mid x * x > 2\}$	$(\mathsf{setComp}\ x^{\mathsf{nat}}.\ x^{\mathsf{nat}} * x^{\mathsf{nat}} > 2)$

Declarations used: $+, * : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ denote the arithmetic operations written in infix notation.

Table 2.1: Some binder symbols with typical application and the corresponding term in extended predicate logic syntax

3.

$$\begin{array}{ll}
 & b \in \mathrm{Bnd}_\Sigma,\ \tau : \mathrm{typeVars}(\mathrm{ty}_\Sigma(b)) \to \mathcal{T}_\Gamma \\
 & \langle T_v, T_1, \ldots, T_n, T \rangle = \tau(\mathrm{ty}_\Sigma(b)) \\
 & x^{T_v} \in \mathrm{Var}_\Gamma,\ t_1 \in \mathrm{Trm}^{T_1}, \ldots, t_n \in \mathrm{Trm}^{T_n} \\
\Longrightarrow & (b^{[\tau]}\ x^{T_v}.\ t_1, \ldots, t_n) \in \mathrm{Trm}^T
\end{array}$$

The parentheses around the expression are mandatory and fix the scope of the bound variable of the binder. For the universal and existential quantifiers, we will usually use the well-established symbols $\forall$ and $\exists$ and take the liberty to drop the type information and the parentheses if the context is unambiguous. Quantifiers bind less strongly than all other operators. The formula $(\forall x.\varphi \land \psi)$, for instance, stands for $(\forall x.(\varphi \land \psi))$ and not for $((\forall x.\varphi) \land \psi)$.

Table 2.1 lists a number of sample terms and uses some symbols of $\mathrm{Bnd}_{ex} = \{\mathsf{forall}, \mathsf{exists}, \mathsf{min}, \mathsf{sum}, \mathsf{setComp}\}$ with

- $\mathsf{ty}_{ex}(\mathsf{forall}) = \mathsf{ty}_{ex}(\mathsf{exists}) = \langle \alpha, \mathsf{bool}, \mathsf{bool} \rangle$
- $\mathsf{ty}_{ex}(\mathsf{min}) = \langle \alpha, \mathsf{nat}, \mathsf{nat} \rangle$
- $\mathsf{ty}_{ex}(\mathsf{sum}) = \langle \mathsf{nat}, \mathsf{nat}, \mathsf{nat}, \mathsf{nat} \rangle$
- $\mathsf{ty}_{ex}(\mathsf{setComp}) = \langle \alpha, \mathsf{bool}, \mathsf{set}(\alpha) \rangle$

Most of the binder symbols are polymorphic, only the bounded summation is monomorphic. Type inference can be applied for terms involving binder symbols in precisely the same manner as for other terms. And we allow ourselves the same liberal attitude when it comes to dropping type annotations for binder symbols as for function symbols. Because of that, the following sentences denote the same term

$$(\mathsf{setComp}^{[\alpha/\beta]}\ x^\beta.\ x^\beta \in \mathsf{empty}^{[\alpha/\beta]}), (\mathsf{setComp}^{[\beta]}\ x^\beta.\ x^\beta \in \mathsf{empty}^{[\beta]}),$$
$$(\mathsf{setComp}\ x^\beta.\ x^\beta \in \mathsf{empty})\ .$$

We sometimes distinguish between *bound* variables occurrences which are in arguments to a binder symbol binding the same variable and *free* variable occurrences which are not in the scope of a corresponding binder. The set of freely occurring variables $\text{freeVars}(t) \subseteq \text{Var}$ for a term $t \in \text{Trm}$ is defined as

$$\begin{aligned}
\text{freeVars}(x^T) &= \{x^T\} \qquad x^t \in \text{Var} \\
\text{freeVars}(f^{[\sigma]}(t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \text{freeVars}(t_i) \\
\text{freeVars}((b^{[\sigma]}\ v.\ t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \text{freeVars}(t_i) \setminus \{v\}\ .
\end{aligned}$$

The semantic value of a function symbol is a function (more precisely, a family of functions) on the domains of the semantic structure. What is then the semantic value of a binder symbol b? The value of function symbol application may depend on the value of its arguments, and so does the value of a binder application, *but* the binder can consider the argument values for all possible instantiations of the bound variable. A binder symbol is hence represented in the semantic domain by a function from functions to a function (see Definition 2.6):

$$I(b^{[\tau]}) : (\mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T_1)}) \times \ldots \times (\mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T_1)}) \to \mathcal{D}^{\tau(T)}$$

The notion of "all possible instantiations" is formally captured as a function going from the domain of the variable type into the type of the corresponding argument domain. Having the semantic equivalent of a binder symbol, we can define the semantic value of a binder application by extending Definition 2.8.

Definition 2.11 (Evaluation of binder symbols, Amendment to Definition 2.8) *Let $D = (\mathcal{D}, I)$ be a semantic structure, τ a type variable assignment and β a variable assignment compatible with τ. For the binder symbol $b \in \text{Bnd}$ with $\text{ty}(b) = \langle T_v, T_1, \ldots, T_n\rangle$ applied to $\sigma : \text{typeVars}(\text{ty}(b)) \to \mathcal{T}$, a variable $v : T_v$ and the terms $t_1 \in \text{Trm}^{T_1}, \ldots, t_n \in \text{Trm}^{T_n}$ let the functions*

$$\begin{aligned}
eval_i &: \mathcal{D}^{\tau(\sigma(T_v))} \to \mathcal{D}^{\tau(\sigma(T_i))} \\
eval_i(d) &:= \text{val}_{I,\tau,\beta[v \mapsto d]}(t_i), \quad d \in \mathcal{D}^{\tau(\sigma(T_v))}
\end{aligned}$$

for $1 \le i \le n$ be the argument evaluations. The evaluation $\text{val}_{I,\tau,\beta} : \text{Trm} \to \mathcal{D}$ from Def. 2.8 is then extended for binder applications as

$$\text{val}_{I,\tau,\beta}((b^{[\sigma]}\ v.\ t_1,\ldots,t_n)) = I(b^{[\tau\circ\sigma]})(eval_1,\ldots,eval_n)\ .$$

The following simple example also clarifies this definition: A unary monomorphic binder symbol b with $\text{ty}(b) = \langle A, B, C\rangle$ applied to variable v^A and argument $t \in \text{Trm}^B$ is evaluated as

$$\text{val}_{I,\tau,\beta}((b\ v^A.t)) = I(b)(\{d \mapsto \text{val}_{I,\tau,\beta[v^A \mapsto d]}(t) \mid d \in \mathcal{D}^A\})\ .$$

The most prominent representatives of binder symbols are the universal and existential quantifier, and we can conclude the embedding of predicate logic into our definitions by defining their semantics as

$$I(\mathsf{forall}^{[\alpha]}) : (\mathcal{D}^{\alpha} \to \mathcal{D}^{\mathsf{bool}}) \to \mathcal{D}^{\mathsf{bool}} \quad I(\mathsf{forall}^{[\alpha]})(f) := \begin{cases} \mathit{tt} & \text{if } \{\mathit{tt}\} = \{f(d) \mid d \in \mathcal{D}^{\alpha}\} \\ \mathit{ff} & \text{otherwise} \end{cases}$$

$$I(\mathsf{exists}^{[\alpha]}) : (\mathcal{D}^{\alpha} \to \mathcal{D}^{\mathsf{bool}}) \to \mathcal{D}^{\mathsf{bool}} \quad I(\mathsf{exists}^{[\alpha]})(f) := \begin{cases} \mathit{tt} & \text{if } \mathit{tt} \in \{f(d) \mid d \in \mathcal{D}^{\alpha}\} \\ \mathit{ff} & \text{otherwise} \end{cases} .$$

In the following, we constrain ourselves to semantic structures which have this interpretation of quantifiers and consider them 'built-in' with the expected semantics

$$\begin{aligned} I,\tau,\beta \models (\forall x^T.\varphi) &\iff I,\tau,\beta[x^T \mapsto d] \models \varphi \text{ for all } d \in \mathcal{D}^{\tau(T)} \\ I,\tau,\beta \models (\exists x^T.\varphi) &\iff I,\tau,\beta[x^T \mapsto d] \models \varphi \text{ for some } d \in \mathcal{D}^{\tau(T)} . \end{aligned} \tag{2.7}$$

When the logic is reduced to first order logic without binders in the following, this does not affect the quantifiers which are still considered present unless they are explicitly excluded as well.

We can also give the other examples in Table 2.1 a sensible interpretation by setting:

$$\begin{aligned} I_{ex}(\mathsf{min}^{[T]})(f) &:= \min\{f(d) \mid d \in \mathcal{D}^T\} \\ I_{ex}(\mathsf{sum})(a,b,c) &:= \sum_{d=a(0)}^{b(0)} c(d) \\ I_{ex}(\mathsf{setComp}^{[T]})(f) &:= \{d \in \mathcal{D}^T \mid f(d) = \mathit{tt}\} \end{aligned}$$

While the addition of binder terms does not increase the expressiveness of the logic (see Section 2.3.3 below), it facilitates both tasks of problem statement and solving considerably. Binders often allow the concise specification of operations which would otherwise need the introduction of extra function symbols. Binders do not add to the expressiveness of the logic (as we will see in Section 2.3.3), and it is always possible to find a description without binder symbols, but the formula sizes are generally considerably smaller when the more sophisticated symbols are employed. They increase readability and comprehensibility of specifications.

Usually, there is a straightforward intuition about the meaning of a binder symbol, but their more complex nature inherently bears some pitfalls. Some function or binder symbols only describe partial functions. The division x/y, for instance, is only defined if the second argument y is semantically different from 0. A similar situation may arise if the semantics of the maximum operator is to be defined as analogon to the minimum operator as

$$\mathsf{max}^{[T]}(f) := \max\{f(d) \mid d \in \mathcal{D}^T\} .$$

While every non-empty set of naturals has a minimum, there is no guarantee that it has a maximum. What would be the value of $(\mathsf{max}^{[\mathsf{nat}]}\ x.\ x)$? There is no maximum for the arguments, the result should not exist. But since every function and binder symbol in the logic is total, this term must evaluate to a value in $\mathcal{D}^{\mathsf{nat}}$. The problem can be solved by *underspecification*. The value of a binder (or function) application without defined value for some arguments is left open. In the axioms which give the symbols their meaning, guarding conditions are added. For the max symbol, the existence of a maximum value amongst all argument values must be guaranteed prior to exploiting that the maximum over a term f is greater than any application of f.

$$(\exists y.\forall x.\ y \geq f(x)) \rightarrow (\forall y.(\mathsf{max}\ x.f(x)) \geq f(y))$$

Hähnle (2005) compares various possibilities to handle partial functions in predicate logic (without binders) and Schmitt (2011) suggests to prove well-definedness conditions for formulas with partial function symbols which could also be extended to binder applications.

The proposed binder setComp to notate set comprehension allows the formulation of arbitrary class terms $\{x \mid \varphi(x)\}$. From Russell's famous antinomy, we know that such class terms are dangerous in general, as they allow the formulation of inconsistent sentences like $\{x \mid x \notin x\}$. Fortunately, due to our type system, such antinomies cannot be formulated, and we can give a superset for every set comprehension: $\mathrm{val}_{I,\tau,\beta}((\mathsf{setComp}^{[T]}\ x.\varphi)) \subseteq \mathcal{D}^{\tau(T)}$. The sentence $\neg\mathsf{in}(t,t)$ cannot be typed since type T of the first argument influences $\mathsf{set}(T)$ the type of the second. But the typing constraint $T = \mathsf{set}(T)$ can never be resolved and the sentence cannot denote a formula.

Dowek et al. (2002) propose a more general extension of predicate logic with binder symbols. They relax extensional requirements of the behaviour of binder symbols, that is, they do not require that the formula

$$(\forall x.t \doteq u) \rightarrow (b\ x.t) \doteq (b\ x.u) \tag{2.8}$$

holds for a unary binder b. In our setting, (2.8) is implied by the semantical evaluation $\mathrm{val}_{I,\tau,\beta}((b\ x.t)) = I(b)(\{d \mapsto \mathrm{val}_{I,\tau,\beta[x\mapsto d]}(t)\})$ which interprets $(b\ x.t)$ and $(b\ x.u)$ equivalently (under I) if the terms t and u are semantically indistinguishable. The binder symbols we have introduced and will introduce in this document depend only on the semantic value of their bound terms and, hence, the less liberal semantic Definition 2.11 fits better our needs.

2.3.2 Type Quantification

In the last section, binder symbols have been introduced to bind (object) variables, but we have no means to bind a type variable, yet. In congruence with universal and existential quantifiers on domain variables, we extend Def. 2.5 by the quantification over type variables.

Definition 2.12 (Type quantification, Amendment to Definition 2.5) *In addition to the cases in Definitions 2.5 and 2.10, the following constructions create terms:*

4. $\alpha \in \mathrm{TVar},\ \varphi \in \mathrm{Trm}^{\mathsf{bool}}$
$\Longrightarrow\ (\mathbb{V}\alpha.\ \varphi),\ (\mathbb{E}\alpha.\ \varphi) \in \mathrm{Trm}^{\mathsf{bool}}$

The ability to quantify over types gives the possibility to formulate properties for all possible type instantiations at a time. Instead of enumerating infinitely many sentences

$$(\forall x^{\mathsf{nat}}.\neg\mathsf{in}(x^{\mathsf{nat}},\mathsf{empty})),(\forall x^{\mathsf{set(nat)}}.\neg\mathsf{in}(x^{\mathsf{set(nat)}},\mathsf{empty})),$$
$$(\forall x^{\mathsf{set(set(nat))}}.\neg\mathsf{in}(x^{\mathsf{set(set(nat))}},\mathsf{empty})),\ldots$$

for all possible variable-free types, we can now write

$$(\mathbb{V}\alpha.\ \forall x^{\alpha}.\ \neg\mathsf{in}(x^{\alpha},\mathsf{empty}))\ . \tag{2.9}$$

The addition of the existential type quantifier is redundant since if the logic contains an universal type quantifier, the existential quantifier can be defined as its dual

$$\mathbb{E}\alpha.\ \varphi := \neg\mathbb{V}\alpha.\ \neg\varphi\ .$$

One might be tempted to adapt (2.7) to type quantified formula which would yield

$$I,\tau,\beta \models (\mathbb{V}\alpha.\ \varphi) \iff I,\tau[\alpha\mapsto T],\beta \models \varphi \text{ for all variable-free types } T \in \mathcal{T}^0\ .$$

However, for $T \neq \tau(\alpha)$ the assignments $\tau[\alpha\mapsto T]$ and β are *not* compatible: $\beta(x^{\alpha}) \in \mathcal{D}^{\tau(\alpha)}$ by assumption that τ and β are compatible and therefore $\beta(x^{\alpha}) \notin \mathcal{D}^{T}$.

To mend this situation, we need two helper notions which put β back into compatibility if the type variable assignment has been modified. Firstly, we assume that there is a *default value* $\delta_{D,\Gamma}(T)$ for every ground-type $T \in \mathcal{T}^0$. Since the domains of a semantic structure D are defined non-empty, the following definition is feasible for any D and any type signature Γ:

$$\delta_{D,\Gamma} : \mathcal{T}^0 \to \mathcal{D} \text{ with } \delta_{D,\Gamma}(T) \in \mathcal{D}^T$$

Secondly, a function is needed which renders a variable assignment β incompatible with τ to one which is compatible. This coercion is performed by mapping all incompatible values to the default value of the according type:

$$\mathrm{coerce}(\beta,\tau)(x^T) := \begin{cases} \beta(x) & \text{if } \beta(x^T) \in \mathcal{D}^{\tau(T)} \\ \delta_{D,\Gamma}(\tau(T)) & \text{otherwise} \end{cases}$$

for a variable $x^T \in \mathrm{Var}$ of type $T \in \mathcal{T}$.

Observation 2.2 (Coercion)

1. *For any variable assignment and any type variable assignment τ, the coerced value* $\mathrm{coerce}(\beta,\tau)$ *is compatible with τ.*
2. *For a variable assignment β compatible with τ, coercion has no effect:* $\mathrm{coerce}(\beta,\tau) = \beta$

PROOF Obvious. □

Using the notion of coercion, it is now possible to define the semantics of type-quantified formulas.

Definition 2.13 (Evaluation of type quantification, Amendment to Definitions 2.8) *For a given semantic structure $(\mathcal{D}, I)$, a type variable assignment τ and a variable assignment β compatible with τ, the evaluation for universal type quantification $(\mathbb{V}\alpha.\varphi)$ is defined as*

$$I,\tau,\beta \models (\mathbb{V}\alpha.\varphi) \quad :\Longleftrightarrow \quad I,\tau[\alpha \mapsto T],\mathrm{coerce}(\beta,\tau[\alpha \mapsto T]) \models \varphi \quad \textit{for all } T \in \mathcal{T}^0$$

This definition is similar to the definition of the universal quantification over elements in (2.7). One might at first expect that the condition would be $I,\tau[\alpha \mapsto T],\beta \models \varphi$. But there would no reason to assume that $\tau[\alpha \mapsto T]$ and β are compatible even if τ and β were. One implication of this would be that Theorem 2.1 would no longer hold. The definitions of the semantics in this section build on this theorem since their definitions would otherwise be ill-defined (functions would be used outside their defined domain).

Coercion is not a problem in reality, however. It has an effect on the semantics only if the formula on which the quantifier operates contains a *free* variable whose type contains the quantified type variable. Let $p : \alpha \to \mathsf{bool}$ for instance be a polymorphic predicate symbol. The formula $(\mathbb{V}\alpha.\forall x^\alpha.\ p^{[\alpha]}(x^\alpha))$ is a formula in which the quantified type variable α does not occur in a free variable in its scope, its evaluation is therefore straightforward (see Proof of Observation 2.3 for details):

$$I,\tau,\beta \models (\mathbb{V}\alpha.\forall x^\alpha.\ p^{[\alpha]}(x^\alpha)) \quad \Longleftrightarrow \quad I(p^{[T]})(d) = \mathit{tt} \text{ for all } T \in \mathcal{T}^0, d \in \mathcal{D}^T \qquad (2.10)$$

which means that the formula holds if and only if p is the universal predicate resulting in tt on every argument. In the formula $(\forall x^\alpha.\mathbb{V}\alpha.\ p^{[\alpha]}(x^\alpha))$, on the other hand, the two quantifiers have swapped places. Now the type quantifier binds α even if it appears in the type of the variable x^α which occurs free in its scope.

$$\begin{aligned} I,\tau,\beta \models (\forall x^\alpha.\mathbb{V}\alpha.\ p^{[\alpha]}(x^\alpha)) \quad \Longleftrightarrow \quad & I(p^{[T]})(\delta(T)) = \mathit{tt} \text{ for all } T \in \mathcal{T}^0, \qquad (2.11)\\ & \text{and } I(p^{[\tau(\alpha)]})(d) = \mathit{tt} \text{ for all } d \in \mathcal{D}^{\tau(\alpha)} \end{aligned}$$

which means that this formula may hold even if p is not universal. It suffices for p to be true on all values in $\mathcal{D}^{\tau(\alpha)}$ and on the default values. The coercion did not vanish from the evaluation and it depends on the choice of default values $\delta(T)$.

Moreover, we have found a counter example that shows that the implication $(\forall x^\alpha.\mathbb{\forall}\alpha.\varphi) \to (\mathbb{\forall}\alpha.\forall x^\alpha.\varphi)$ is not valid in general. The opposite direction is valid, however:

Observation 2.3 (Permuting object and type quantifiers) *Let* $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$ *be a formula,* $\alpha \in \mathrm{TVar}$ *a type variable and* $x^\alpha \in \mathrm{Var}$ *a variable. Then the implication* $(\mathbb{\forall}\alpha.\forall x^\alpha.\varphi) \to (\forall x^\alpha.\mathbb{\forall}\alpha.\varphi)$ *is valid. The converse is not valid in general.*

PROOF Using the abbreviation $\tau' = \tau[\alpha \mapsto T]$ we show the implication on the semantics:

$$\begin{array}{cc}
 & I,\tau,\beta \models \mathbb{\forall}\alpha.\forall x^\alpha.\varphi \\
\Longleftrightarrow & I,\tau',\mathrm{coerce}(\beta,\tau') \models \forall x^\alpha.\varphi \text{ for all } T \in \mathcal{T}^0 \\
\Longleftrightarrow & I,\tau',\mathrm{coerce}(\beta,\tau')[x^\alpha \mapsto d] \models \varphi \text{ for all } T \in \mathcal{T}^0,\ d \in \mathcal{D}^T \\
\overset{(a)}{\Longleftrightarrow} & I,\tau',\mathrm{coerce}(\beta[x^\alpha \mapsto d],\tau') \models \varphi \text{ for all } T \in \mathcal{T}^0,\ d \in \mathcal{D}^T \\
\overset{(b)}{\Longrightarrow} & I,\tau',\mathrm{coerce}(\beta[x^\alpha \mapsto d],\tau') \models \varphi \text{ for all } T \in \mathcal{T}^0,\ d \in \mathcal{D}^{\tau(\alpha)} \\
\Longleftrightarrow & I,\tau,\beta[x^\alpha \mapsto d] \models \mathbb{\forall}\alpha.\varphi \text{ for all } d \in \mathcal{D}^{\tau(\alpha)} \\
\Longleftrightarrow & I,\tau,\beta \models (\forall x^\alpha.\mathbb{\forall}\alpha.\varphi)
\end{array}$$

(a) The two variable assignments are equal to $\mathrm{coerce}(\beta,\tau)(v)$ on any variable $v \in \mathrm{Var}$ other than x^α. Since $d \in \mathcal{D}^{\tau'(\alpha)}$, it easy to see that both variable assignments result in d for x^α.

(b) The consequence is trivially true for $T = \tau(\alpha)$. If $T \neq \tau(\alpha)$, then the update to the variable assignment is incompatible with τ' and by definition we get the equality $\mathrm{coerce}(\beta[x \mapsto d],\tau)(x^\alpha) = \delta(T) = \mathrm{coerce}(\beta[x \mapsto \delta(T)])(x^\alpha)$. The value $\delta(T) \in \mathcal{D}^T$ is covered by the premiss: The implication holds because the general case automatically covers the default value as well. It is here that the opposite direction is not necessarily the case.

The formula $(\mathbb{\forall}\alpha.\forall x^\alpha.\ p^{[\alpha]}(x^\alpha))$ for $p : \alpha \to \mathsf{bool}$ holds if the predicate p is true on all arguments. This has already been mentioned in (2.10), we provide the formal evidence here.

$$\begin{array}{cc}
 & I,\tau,\beta \models (\mathbb{\forall}\alpha.\forall x^\alpha.\ p^{[\alpha]}(x^\alpha)) \\
\Longleftrightarrow & I,\tau',\mathrm{coerce}(\beta,\tau') \models \forall x^\alpha.\ p^{[\alpha]}(x^\alpha) \text{ for all } T \in \mathcal{T}^0 \\
\Longleftrightarrow & I,\tau',\mathrm{coerce}(\beta,\tau')[x^\alpha \mapsto d] \models p^{[\alpha]}(x^\alpha) \text{ for all } T \in \mathcal{T}^0, d \in \mathcal{D}^T \\
\Longleftrightarrow & I(p^{[T]})\big((\mathrm{coerce}(\beta,\tau')[x^\alpha \mapsto d])(x^\alpha)\big) = \mathit{tt} \text{ for all } T \in \mathcal{T}^0, d \in \mathcal{D}^T \\
\Longleftrightarrow & I(p^{[T]})(d) = \mathit{tt} \text{ for all } T \in \mathcal{T}^0, d \in \mathcal{D}^T
\end{array}$$

With the additional abbreviation $\beta' := \beta[x^\alpha \mapsto d]$, the converse $(\forall x^\alpha . \mathbb{V}\alpha .\ p^{[\alpha]}(x^\alpha))$ then holds under different conditions:

$$\begin{aligned}
& I, \tau, \beta \models (\forall x^\alpha . \mathbb{V}\alpha .\ p^{[\alpha]}(x^\alpha)) \\
\iff\quad & I, \tau, \beta' \models (\mathbb{V}\alpha .\ p^{[\alpha]}(x^\alpha)) \text{ for all } d \in \mathcal{D}^{\tau(\alpha)} \\
\iff\quad & I, \tau', \mathrm{coerce}(\beta', \tau') \models p^{[\alpha]}(x^\alpha) \text{ for all } d \in \mathcal{D}^{\tau(\alpha)}, \text{ all } T \in \mathcal{T}^0 \\
\iff\quad & I(p^{[T]})(\mathrm{coerce}(\beta', \tau')(x^\alpha)) = \mathit{tt} \text{ for all } d \in \mathcal{D}^{\tau(\alpha)}, \text{ all } T \in \mathcal{T}^0 \\
\iff\quad & I(p^{[T]})(d) = \mathit{tt} \text{ if } T = \tau(\alpha) \text{ or } d = \delta(T), T \in \mathcal{T}^0, d \in \mathcal{D}^T \\
\iff\quad & I(p^{[T]})(\delta(T)) = \mathit{tt} \text{ for all } T \in \mathcal{T}^0, \text{ and } I(p^{[\tau(\alpha)]})(d) = \mathit{tt} \text{ for all } d \in \mathcal{D}^{\tau(\alpha)}
\end{aligned}$$

which means that this formula may hold even if p is not universal, see also (2.11). A concrete interpretation can easily be found in which the second formula holds but the first does not. □

The resemblance between this definition and the valuation of object quantifiers (2.7) might lead to the conclusion that type quantification is an application of object quantification. But this is not the case. Types are not reified into a "type of types" which could then be subject to quantification. Instead, the type quantifiers are a new syntactic category. The context of the quantifier influences the type variable assignment τ and leaves the object variable assignment β untouched. In example (2.9), the set over which the quantification $(\forall x^\alpha . \ldots)$ ranges is influenced by the type quantifier. Such an evaluation context can never be created using a variable assignment, it must be the type variable context which is modified.

While parametric type systems, polymorphic function symbols and type variables are common for the definition of predicate logics for proof assistants, type quantification is not. There is one suggestion by Melham (1993) to add it to a higher order logic framework, but it has not been incorporated into the system. To our knowledge, the Boogie 2 system described by Leino and Rümmer (2010) is the only logic system which supports type quantification. The reason for this reluctance may be that type quantifiers increase expressiveness of the logic (see next section).

2.3.3 Expressiveness

In the last sections we have introduced a predicate logic with some deviations from textbook first order logic. The question which naturally arises is whether the additional constructs enrich the syntax of the logic without increasing its expressiveness.

Usually, a logic is more expressive than first order logic if there is a set of semantic structures which can be characterised by the enriched logic but not in first order logic. Of course, this notion is only sensible if the semantic structures coincide between the logics under examination.

Since some of the defined extensions change the notion of semantical structures, we will use the less strict notion of *reducibility*. A logic can be reduced to first order predicate logic if for every recursive set of formulas M in the richer logic an equisatisfiable[1] set M' of first order formulas can be computed.

Binders

The addition of binders as new syntactical concept into the logic does not increase its expressiveness. It is possible to reduce logic with binders to traditional predicate logic by replacing the binder symbols with ordinary function symbols. A binder symbol can this be seen as an abbreviation for an entire family of function symbols.

Theorem 2.4 (Reduction of binder logic) *For any recursive set $M \subseteq \mathrm{Trm}^{\mathsf{bool}}$ of formulas containing binder symbols, an equisatisfiable set $M' \subseteq \mathrm{Trm}^{\mathsf{bool}}$ of formulas without binder symbols can be computed.*

In M' only binder terms have been replaced by first order terms.

The idea behind the proof is to introduce a new function symbol $f_{(b\,x.t_1,\ldots,t_n)}$ for every binder term $(b\ x.t_1,\ldots,t_n)$. These new function symbols must then be constrained such that they are consistent with the semantics of binder symbols. However, while these additional constraints are straightforward, there will be infinitely many of them.

PROOF We present the proof for the untyped case, that is, there are only boolean terms and terms of one other sort. The aspect of types is orthogonal to the presented idea and omitting type annotations clarifies the proof. For the same purpose, the presentation is limited to unary binder symbols. The extension to higher arities is canonical.

Let $M^{(0)} = M$ be the set of formulas containing binder symbols. The set $M^{(1)}$ is constructed from $M^{(0)}$ by replacing every binder term $(b\ x.t)$ in which no binder occurs in t by the term $f_{(b\,x.t)}(\bar{v})$ in which $f_{(b\,x.t)} \in \mathrm{Fct}$ is a fresh replacement function symbol and $\bar{v}$ denotes the free variables (without the variable x bound to b) occurring in t. Let, in general, the set $M^{(n+1)}$ emerge from $M^{(n)}$ by the same construction. Let ultimately M° denote the set of terms in which all binders have been thus replaced:

$$M^{\circ} := \big(\bigcup_{n\in\mathbb{N}} M^{(n)}\big) \cap \{t \in \mathrm{Trm} \mid t \text{ is binder-free}\}$$

The auxiliary set $M^{\dagger} \subset \mathrm{Trm}^{\mathsf{bool}}$ captures the constraints which ensure that the fresh function symbols behave consistently. It is defined as

$$M^{\dagger} = \{(\forall \bar{v}.\forall \bar{w}.(\forall z.t^{z} \doteq u^{z}) \to f_{(b\,x.t)}(\bar{v}) \doteq f_{(b\,y.u)}(\bar{w})) \mid t,u \in \mathrm{Trm}, \text{binder-free}\} \tag{2.12}$$

in which $\bar{w}$ are the free variables in u (again, without the bound variable y). Since the bound variable differs between t and u (variable x versus y), the comparison is

[1] This would be "equivalent" instead of "equisatisfiable" to show a logic was not more expressive than first order logic.

performed on the terms t^z and u^z in which all occurrences of the bound variables have been replaced by a common variable z not occurring in t or u.

Two binder applications in which the argument terms are semantically (but not syntactically) equal need to result in the same evaluation (by Def. 2.11). The constraints in $M^\dagger$ ensure that any two binder terms with semantically equivalent arguments t and u have equivalent replacement functions $f_{(b\,x.t)}$ and $f_{(b\,y.u)}$.

The union $M' = M^\circ \cup M^\dagger$ has a model if and only if M has: Let $(\mathcal{D}, I)$ be a model of M (with binders), we construct the model $(\mathcal{D}, I')$ (without binders) by setting the interpretation I congruent with I' on all function symbols in M and extend it for the new symbols by setting:

$$I'(f_{(b\,x.t)})(\bar{d}) := \mathrm{val}_{I,\tau,\beta[\bar{v}\mapsto\bar{d}]}(b\;x.t) = I(b)(e \mapsto \mathrm{val}_{I,\tau,\beta[x\mapsto e][\bar{v}\mapsto\bar{d}]}(t)) \tag{2.13}$$

The evaluation of the replacement function symbols is chosen to match the value of the replaced binder terms. Hence, by construction, the formulas in M° are true in I'. Due to the equality in (2.13), it is obvious that a term u which behaves identically to t under I for all instantiations of x will have the same value if applied as arguments to the binder b. The equalities in $M^\dagger$ are thus rendered true as well.

For the other direction, let now $(\mathcal{D}, J')$ be a model of M', from which we construct a model $(\mathcal{D}, J)$ for M. On function symbols, J behaves identically as J' and for any binder symbol b, we set

$$J(b)(e \mapsto \mathrm{val}_{J',\tau,\beta[x\mapsto e]}(t)) = \mathrm{val}_{J',\tau,\beta}(f_{(b\,x.t)}(\bar{v})) \tag{2.14}$$

for all applicable terms t and variable assignments β. This does not fix J entirely. If $g : \mathcal{D} \to \mathcal{D}$ is a function not induced by a term[2], the valuation $J(g)$ is not fixed by (2.14). For the evaluation of any term, however, J will only be applied to functions which *are* induced by terms. We can therefore choose the value of J on the remaining places arbitrarily. It is still fixed "tightly enough" for our purposes. The binder terms evaluate equal to their replacement terms and since $J' \models M^\circ$, we have $J \models M$.

For the choice of (2.14) to be well-defined, the replacement functions $f_{(b\,x.t)}$ and $f_{(b\,x.u)}$ for two equivalent terms $t \equiv u$ inducing the same function, must result in the same value. This is the case since J' satisfies $M^\dagger$ which implies exactly this condition.□

This result considers all binder (and function) symbols as *uninterpreted* symbols which have no fixed meaning. The valid formulas in first order predicate logic are recursively enumerable if semantics of the function symbols is not fixed. If their meaning can be fixed by means of axioms in the logic itself, the decidability state does not change.

Often function symbols are given their fixed meaning using a finite number of axioms. The function symbol for the empty set has its meaning fixed by the single

[2] If the domain is infinite, there must be one such function on the domain which cannot be described by a term. This is since the set of terms is countable while the set of functions on the domain is not.

axiom $(\forall\!\!\!\forall\alpha.\forall x^{\alpha}.\neg\mathsf{in}(x,\mathsf{empty}))$. Binder symbols are of a different nature: They can usually not be adequately axiomatised by a single axiom but need an infinite set of formulas. The binder setComp for the set comprehension (mentioned in Table 2.1), for instance, is axiomatised by the infinite set $\{(\forall T.\forall x^{T}.\ \mathsf{in}(x,(\mathsf{setComp}\ x.\varphi)) \leftrightarrow \varphi) \mid T \in \mathcal{T}^{0}, \varphi \in \mathrm{Trm}^{\mathsf{bool}}$ with $\mathrm{freeVars}(\varphi) \subseteq \{x^{T}\}\}$. The set is schematic and as such recursive, the validity problem remains semi-decidable for predicate logic with binders which have there semantics fixed by such schematic axiomatisations.

In Section 4.3.2 we will come back to the issue of translating logic with binder symbols to first order logic, but then with the intention to efficiently feed *UDL* proof obligations to a constraint solver. The translation presented there reifies functions using a first order theory of functions. It is theoretically incomplete but better suited for practical usage.

Dowek et al. (2002) provide a sound and complete calculus for a more general extension of predicate logic with binder symbols. In their setting, the extensional equation (2.8) needs not hold. If the structures are restricted to such structures with finite domain, binders *do* increase the expressiveness of the logic. Otto (2000) shows that Hilbert's ε choice binder allows the formulation of properties over finite structures which are not expressible in first order logic itself.

Parametrised Types and Type Quantification

The situation is different if type parameters and quantification over types are admitted. Using type quantification, it is possible to axiomatise that the domain of a type is isomorphic to the natural numbers, a fact which cannot be specified in first order predicate logic as a result of Gödel's incompleteness theorem.

We will in the following construct a formula over the type signature $\Gamma_{arith} = \{S, n, \mathsf{bool}\}$ and signature $\Sigma_{arith} = \{c : \alpha, f : S(\alpha) \to n, succ : n \to n, zero : n\}$ such that for every model $(\mathcal{D}^{n}, I(zero), I(succ)) \cong (\mathbb{N}, 0, +1)$

The formula is as follows:

$$\forall\!\!\!\forall\alpha.\forall x^{S(\alpha)}.\ x^{S(\alpha)} \doteq c^{[S(\alpha)]} \tag{2.15}$$

$$\wedge \quad \forall x^{n}.\exists\!\!\!\exists\alpha.\ f(c^{[S(\alpha)]}) \doteq x^{n} \tag{2.16}$$

$$\wedge \quad \forall\!\!\!\forall\alpha.\forall\!\!\!\forall\beta.\ f(c^{[S(\alpha)]}) \doteq f(c^{[S(\beta)]}) \to c^{[S(\alpha)]} \mathrel{\dot{\approx}} c^{[S(\beta)]} \tag{2.17}$$

$$\wedge \quad zero \doteq f(c^{[S(n)]}) \tag{2.18}$$

$$\wedge \quad \forall\!\!\!\forall\alpha.\ succ(f(c^{[S(\alpha)]})) \doteq f(c^{[S(S(\alpha))]}) \tag{2.19}$$

Equality (2.15) ensures that the domains $\mathcal{D}^{S(S(\ldots(n)\ldots))}$ are singletons. The following formulas achieve that the chain $I(c^{[S(n)]}), I(c^{[S(S(n))]}), \ldots$ is isomorphic to the natural numbers. Using (2.16) for surjectivity and (2.17) for injectivity[3] we define a bijection between that chain and the domain $\mathcal{D}^{n}$ via the polymorphic function symbol $f : S(\alpha) \to n$. The last two conjuncts (2.18) and (2.19) fix the semantics of zero and succ.

[3] We must use weak equality here since the types of $c^{[S(\alpha)]}$ and $c^{[S(\beta)]}$ make them incomparable using $\doteq$.

It is in this case the existential quantifier in (2.16), which is used to formulate the surjectivity of f, that is responsible for the increased expressiveness in this formula. However, even without explicit type quantifiers (all type variables are implicitly universally quantified), there would be a way to encode structures (for instance, non-linear arithmetic) which cannot otherwise be encoded in first order logic.

The set of ground types is the freely generated algebra over the type constructors and it is this structure (which cannot generally be expressed in first order logic) inherited to the domains which causes the logic to theoretically go beyond first order logic here. Only if the type system is relaxed to contain *at least* the generated types (but possibly more types) can the logic be first order again. Schmitt et al. (2009) describe a similar criterion for the reification of a hierarchical type system.

This result is a motivation for us to leave first order definability behind us and to semantically fix certain domains. The resulting logic will not have a complete calculus, but practice has shown that the theoretical completeness gap and the incompleteness of theorem proof systems have little in common.

2.4 Structured Dynamic Logic

To formally prove properties about programs in a deductive manner, we need a formalism which combines programs and properties in one language. *Dynamic Logic* (DL) is such a formalism embedding pieces of code into logical formulas. It was introduced by Pratt (1976), refined by Harel (1979), and is extensively covered in the monograph by Harel et al. (2000).

DL is a modal logic, that is, the evaluation of a formula spans over more than one semantic structure, modal operators switch between the semantic structures in the evaluation. The modalities of DL are induced by programs: A program P can be embedded into modal operators $[\cdot]$ (called *box*) or $\langle\cdot\rangle$ (called *diamond*) and is followed by a formula φ. The formula $[P]\varphi$ is true if φ is true in *all* possible terminal states of the execution of P and $\langle P\rangle\varphi$ is true if there is one terminal state such that φ holds.

Dynamic Logic has strong connections with the weakest precondition calculus by Dijkstra (1975) and Floyd-Hoare logic (Floyd, 1967; Hoare, 1969). The DL-formula $\psi \to [P]\varphi$ is valid if the metaformula $\psi \to wlp(P;\varphi)$ of the weakest precondition calculus is, or – equivalently – if the Hoare triple $\{\psi\}P\{\varphi\}$ is valid in Floyd-Hoare calculus. Unlike Hoare triples and Dijkstra's weakest precondition meta-operators *wp* and *wlp*, however, the DL-modalities are first class formula constructors and the logic is closed under composition. We shall in later sections, for instance, make use of that fact by constructing formulas like $[P]\langle Q\rangle\varphi$ which have no counterpart as Hoare triple.

An example of a formula in dynamic logic is the following stating that the addition of two natural numbers a, b can be performed by iteratively incrementing variable a and decrementing b until $b = 0$.

$$a \doteq a_0 \land b \doteq b_0 \to [\mathbf{while}\ \neg b \doteq 0\ \mathbf{do}\ a := a + 1;\ b := b - 1\ \mathbf{end}]a \doteq a_0 + b_0 \qquad (2.20)$$

The language employed in the program of this tiny example supports the **while** loop statement. In the original presentation of DL, however, programs in modalities were composed using more fundamental combinators influenced by Kleene Algebra (Kozen, 1997). The thus constructed programs are called *regular programs* due to their resemblance with regular expressions; Table 2.2 gives an overview of the constructors of regular programs. Please note that regular programs are *indeterministic*, in the sense that the execution started in one state can result in more than one (or no) terminal state.

Construct	Semantics when used in regular programs
$P_1 \cup P_2$	*(Indeterministic choice)* The execution of this program indeterministically chooses either P_1 or P_2 to continue the execution.
$P_1 \mathbin{;} P_2$	*(Sequential composition)* After the execution of P_1, the program P_2 is executed. This operator is deterministic.
P^*	*(Kleene-star repetition)* The execution of the program repeats the program P an indeterministically chosen number of times.
$\varphi\mathbf{?}$	*(Test)* The execution of this program proceeds if the formula φ holds, and blocks execution (i.e., does not result in a terminal state) if not.
$x := t$	*(Assignment)* The value of the expression t is assigned to a program variable x.
$x := ?$	*(Random assignment)* An extension to DL defined by Harel et al. (2000, §11.2). An arbitrary, indeterministically chosen value is assigned to the program variable x

P, P_1, P_2 are regular programs, x is a program variable, t a term of the same type as x and φ a formula.

Table 2.2: Statement constructors in structured DL

The more convenient program constructors like "while" or "if-then-else" can be defined as macros using the regular program junctors. The DL formula (2.20) from above using the high-level **while** construct can, for instance, be seen as an abbreviation of the spelled out formula

$$a \doteq a_0 \land b \doteq b_0 \to [(\neg b \doteq 0\mathbf{?}; a := a + 1; b := b - 1)^*; b \doteq 0\mathbf{?}]a \doteq a_0 + b_0$$

which uses a regular program. The loop with a condition has been replaced by a indeterministic repetition. The repeated program starts with a test ensuring that the loop condition holds in every iteration of the loop. The actual assignments in the body remain untouched. Only after the repeated block, another test ensures that the loop condition does not hold any longer after the execution of the loop. Despite the fact that the repetition and the choice operation individually are indeterministic, this particular combination ensures that the program seen as a whole is deterministic.

Dynamic logic is used as underlying logic in the deductive program verifiers KIV (Reif et al., 1995) and KeY (Beckert et al., 2007) to formulate program verification conditions. Inference rules and deduction are then used to reduce verification conditions in DL to formulas of the underlying logic.

2.5 Unstructured Dynamic Logic

In this section we devise a dynamic logic for a special indeterministic verification language and embed it into the presented predicate logic.

2.5.1 Unstructured Programming Language

"Go to statements considered harmful" is the title of a famous letter by Dijkstra (1968) criticising that excessive use of unstructured jump commands decreases the quality of the resulting code: "The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program." In the decades following 1968, when Dijkstra expressed his concerns, programming in unstructured languages has been overcome. Why, then, do modern verification approaches revert to an unstructured goto language again? Is this not a step backward?

The reason is that intermediate verification languages are not intended as input languages for a human programmer (or specifier) but as platforms for intermediate translation representations. They must, therefore, not be compared to high-level programming languages (such as Java, for instance) but rather to intermediate representations within a compilation process (such as Java bytecode, for instance). Such low-level, machine-oriented languages are still based on the goto statement due to the nature of target processors. An intermediate verification language is intended as a vehicle to formulate proof obligations which originally stem from a (possibly already compiled) program in a higher programming language. For a machine-targeted language, generality and efficiency are higher design goals than human comprehensibility. A structured program containing loops can canonically be transformed into a goto-program, the converse is not as easy to achieve, see Section 2.5.5 for a detailed analysis.

When choosing the set of control flow operations, we align the unstructured language with the structured case presented in Table 2.2 and transfer the statements into a goto-oriented setting. The language primitives essentially remain the same, only the control-flow operators $\cup$ and * are replaced by the general goto statement to control flow more flexibly.

The resulting programming language bases on the same primitive building blocks as the programming language of Boogie defined by Leino and Rümmer (2010). Although the Boogie proof system uses weakest-precondition-techniques rather than dynamic logic, the likeness cannot be overseen and is intentional. Boogie has a long tradition going back to its predecessor intermediate representations of ESC/Modula and ESC/Java(2). These systems used a language similar to Dijkstra's guarded commands, indeterministic but structured. It was discovered that for a translation from already compiled code to an intermediate language it helps if the verification language allows for more flexible control flow.

Unlike in Boogie, we do not structure the program into basic blocks which are sequences of statements such that no path through the program enters or leaves a basic block other than through the first or last statement. The finer granularity of indexing statement-wise allows us to point to each statement individually and to symbolically execute the code statement by statement.

We will together with the primitives also adopt the terminology introduced by Boogie and name our statements equally. Table 2.3 states a full comparison between the constructors of regular programs and the ones of *UDL*.

Traditionally, program verification systems consider pairs of states: One state before the program execution and one state which can be reached by the execution. Depending on the termination and determinism status of the program, there may be none, one, or many pairs for a given start-state. Pre- and postcondition refer to the start- and end-state, the states which are traversed during the execution are of no interest for the semantic meaning.

However, it proved more flexible to disseminate the verification condition onto various places within the program. The benefits include:

1. During the course of a program, constraints on the values of the program variables must hold and these refer to the values of the intermediate state not to those of the end state. While it would be possible to remember the values and to do the checks in the end-state, it seems natural to resolve this *in situ*.

2. If a constraint does not hold or cannot be proved valid, it would be nice to be able to give an adequate feedback on what and where the problem occurred. If the constraints are kept close to the place where the action happens, the localisation of errors is a lot easier.

Therefore, we will allow for embedded assertions within the program in addition to the other program constructs. This is a significant extension to the traditional DL paradigm which does not consider intermediate states.

2.5.2 Formal Definition of *UDL*

It is in the nature of a program to change the state of the environment it is executed in. In the context of dynamic logic, this means that the execution of a program may change the valuation of some symbols. In the following, we limit the effects of a program execution to some logical symbols which we call the *program variables*. The set PVar_Σ of program variables for a given signature Σ is a subset of the nullary monomorphic function symbols, formally

$$\mathrm{PVar}_\Sigma \subseteq \{f \in \mathrm{Fct}_\Sigma \mid \mathrm{ty}(f) = \langle T \rangle, \mathrm{typeVars}(T) = \emptyset\} \; .$$

Construct	**Struct. DL**	***UDL***
(Indeterministic choice)	$P_1 \cup P_2$	goto idx_1, idx_2
(Kleene-star repetition)	P^*	goto idx
(Test)	$\varphi\mathbf{?}$	assume φ
(Assignment)	$x := t$	$x := t$
(Random assignment)	$x :=?$	havoc x
(Embedded assertions)	n/a	assert φ

idx, idx_1, idx_2 are natural number literals, P_1, P_2 regular programs, x is a program variable, t a term of the same type as x and φ a formula.

Table 2.3: Comparison between constructors in structured and unstructured DL

Definition 2.14 (Statement, Unstructured program) *For a given signature Σ, the set of* statements Stm_Σ *is defined as*

$$\begin{array}{rcl}
\mathrm{Stm}_\Sigma & = & \{\mathsf{skip}, \mathsf{end}\} \\
& \cup & \{\mathsf{assert}\ \varphi \mid \varphi \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}\} \\
& \cup & \{\mathsf{assume}\ \varphi \mid \varphi \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}\} \\
& \cup & \{\mathsf{goto}\ n_1, \ldots, n_k \mid n_1, \ldots, n_k \in \mathbb{N}\} \\
& \cup & \{p := t \mid p \in \mathrm{PVar}_\Sigma,\ t \in \mathrm{Trm}_\Sigma^{\mathsf{ty}(p)}\} \\
& \cup & \{\mathsf{havoc}\ p \mid p \in \mathrm{PVar}_\Sigma\}
\end{array}$$

An unstructured program *is a finite sequence of statements. The set of programs is denoted by Π_Σ and $|\pi| \in \mathbb{N}$ denotes the length (that is, the number of statements) of $\pi \in \Pi_\Sigma$. For a natural number $i \in \mathbb{N}$, the selection $\pi[i]$ refers to the i-th statement in π if $i < |\pi|$ and refers to the statement "*end*" if $i \geq |\pi|$.*

A program is called self-contained *if (1) the last statement is an* end*-statement and (2) $n < |\pi|$ for every argument $n \in \mathbb{N}$ to any* goto*-statement in π.*

Unlike programs in structured higher programming languages in which the program text forms a (syntax) tree, unstructured programs have no such recursive building nature, but are a raw sequence of commands. Control flow can be transferred arbitrarily using goto statements.

The program flow may go beyond the range of the defined program, but terminates immediately then. It is obvious that every program is equivalent to a very similar self-contained program.

The upcoming definition will introduce dynamic logic constructions which incorporate unstructured programs, similar to the structured case. But program modalities are not the only means to describe changes of state. In addition we also permit the notation of explicit substitutions for program variables, called *updates*. Updates are used, for instance, to store intermediate states during symbolic execution of programs (see Section 3.2).

Definition 2.15 (Update and program formulas, Amendment to Definition 2.5) *In addition to the cases in Definitions 2.5, 2.10, and 2.12, the following two constructions create terms:*

5. $p_1, \ldots, p_n \in \mathrm{PVar}_\Sigma$ *and* $t_1 \in \mathrm{Trm}_\Sigma^{\mathsf{ty}(p_1)}, \ldots, t_n \in \mathrm{Trm}_\Sigma^{\mathsf{ty}(p_n)}$, $t \in \mathrm{Trm}_\Sigma^T$
$\implies \{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t \in \mathrm{Trm}_\Sigma^T$ *(Update term)*

6. $\pi \in \Pi_\Sigma, n \in \mathbb{N}$
$\implies [n;\pi],\ [\![n;\pi]\!] \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}$ *(Program formula)*

The first component n of the pair $n;\pi$ serves as a *program counter* or *instruction pointer* into the program π. It indicates that the next statement to be executed is $\pi[n]$, the n-th statement of π. Since we often consider execution starting at the beginning of a program, we take the liberty to write $[\pi]$ (or, respectively, $[\![\pi]\!]$) to denote the modalities $[0;\pi]$ ($[\![0;\pi]\!]$) starting from the beginning of π.

The definitions of statements and terms need to be read simultaneously as they refer to each other (statements may contain terms and, vice versa, formulas can contain programs and therefore statements). It is possible to nest programs and formulas and the statement $\mathsf{assert}\ [0;\pi]$ is a legal statement. That such a deep embedding is desirable is shown in the approach presented by Barnett and Leino (2010). There, the authors formulate verification conditions for specifications themselves written in code (in this case *Code Contracts*, see Fähndrich et al. (2010)) in Boogie using very similar deep embeddings.

Dynamic logic is a special case of *modal logic*. Traditionally, the semantics of modal logic is defined using the notion of a *Kripke structure*. A Kripke structure consists of a set of states with a binary state relationship. Each state may interpret symbols differently. The evaluation of formulas behind a modal operator switches the evaluation state. In the case of dynamic logic, the modal operators are induced by a piece of code.

In our case, we will deviate from this pure doctrine in so far as not only the begin/end-pair of the state relation is considered but all traversed states in between them as well. However, we will use a Kripke structure as evaluation basis and define our semantics upon them. As set of states we use the set of all possible semantic structures I over the signature Σ.

This is of course an over-relaxation of the state space since programs may only change program variables and must leave every other symbol of the signature semantically untouched. However, it simplifies definitions a lot. If you like, you can also think of the state space as the set of all possible value assignments to the program variables.

Updates are a simple modal operator in which the state change is explicitly described as set of assignments which are executed in parallel and the state change happens without intermediate step. The value of an updated term $\{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t$ is equivalent to the value of t in a modified semantic structure in which the program variables $p_1,\ldots,p_n$ are evaluated as the value of their replacement terms $t_1,\ldots,t_n$:

Definition 2.16 (Evaluation of update terms, Amendment to Definition 2.8) *Let $D = (\mathcal{D}, I)$ be a semantic structure, τ a type variable assignment and β a variable assignment compatible with τ. For program variables $p_1 : T_1,\ldots,p_n : T_n$ and equally typed terms $t_1 \in \mathrm{Trm}^{T_1},\ldots,t_n \in \mathrm{Trm}^{T_n}$ and an arbitrary term $t \in \mathrm{Trm}$ the evaluation function $\mathrm{val}_{I,\tau,\beta}$ as introduced in Def. 2.8 is extended by setting*

$$\mathrm{val}_{I,\tau,\beta}(\{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t) = \mathrm{val}_{I',\tau,\beta}(t)$$

with

$$I' = I[p_1 \mapsto \mathrm{val}_{I,\tau,\beta}(t_1)][p_2 \mapsto \mathrm{val}_{I,\tau,\beta}(t_2)]\ldots[p_n \mapsto \mathrm{val}_{I,\tau,\beta}(t_n)]\ .$$

Note that this definition includes a "last-win"-semantics: In case the same program variable is assigned two or more (possibly contradictory) values, the last occurrence outplays its predecessors. This makes, for instance, the equalities $(\{p := 5 \parallel p := 7\}t) \doteq (\{p := 7\}t)$ and $(\{p := 5 \parallel p := p\}t) \doteq t$ valid for any term t.

Our logic is different to other modal logics because we do not have one formula behind the operator to be evaluated under a different evaluation context but want to consider several asserted formulas in various intermediate states. To achieve this, we define the semantics of the modal operator using *traces*, that is, sequences of states making up runs of the program. The upcoming definition will give an operational semantics for the unstructured intermediate language using a state-successor-function R_π.

The states of a trace have two components: 1) The semantic structure they are evaluated in and 2) the current program counter value. We define the set of all states as

$$\mathcal{S}_{\Sigma,\mathcal{D}} := \{(I,n) \mid I \text{ is the interpretation of a semantic structure } (\mathcal{D},I)\ , n \in \mathbb{N}\}.$$

Traditionally in DL, the interpretation of a semantic structure is split up into a variable part (the evaluation $I\big|_{\mathrm{PVar}}$ of the program variables) and an invariable part (the evaluation $I\big|_{\complement\mathrm{PVar}}$ of the remaining symbols). The set of states is then defined over the variable part of the interpretation. The invariable part makes up the environment. We deviate from that standard in the presentation to make it more concise but could equivalently have split up the interpretations.

Intuitively, a program formula should hold if all assertions in all conceivable successful program runs hold. This can be put formally into

Definition 2.17 (Program execution, Traces) *The* program execution function $R_\pi : \mathcal{S} \to 2^{\mathcal{S}}$ *is a mapping that for a program $\pi \in \Pi$ assigns to every state a set of one-step-successor states. Its result depends on the active statement $\pi[n]$ of π.*

Let $s = (I,n) \in \mathcal{S}$ be a state, τ a type variable assignment and β a compatible variable assignment. Then the value of $R_\pi(s)$ is according to the following table:

If $\pi[n]$ ***matches***	***and***	***then*** $R_\pi(s) =$
skip		$\{(I,n+1)\}$
$p := t$		$\{(I[p \mapsto \mathrm{val}_{I,\tau,\beta}(t)], n+1)\}$
assert φ	$I,\tau,\beta \models \varphi$	$\{(I,n+1)\}$
assert φ	$I,\tau,\beta \not\models \varphi$	$\emptyset$
assume φ	$I,\tau,\beta \models \varphi$	$\{(I,n+1)\}$
assume φ	$I,\tau,\beta \not\models \varphi$	$\emptyset$
end		$\emptyset$
goto $n_1,\ldots,n_k$		$\{(I,n_1),\ldots,(I,n_k)\}$
havoc p		$\{(I[p \mapsto d], n+1) \mid d \in \mathcal{D}^{\mathrm{ty}(p)}\}$

- *An infinite sequence* $(s_0, s_1, \ldots)$ *with* $s_0 \in \mathcal{S}$ *and* $s_{i+1} \in R_\pi(s_i)$ *for* $i \in \mathbb{N}$ *is called an* infinite trace *of* π *starting in* s_0.

- *A finite sequence* $(s_0, s_1, \ldots, s_r)$ *with* $s_0 \in \mathcal{S}$ *and* $s_{i+1} \in R_\pi(s_i)$ *for* $i \in \{0, \ldots, r-1\}$ *is called a* finite trace *of* π *starting in* s_0 *if* $R_\pi(s_r) = \emptyset$.

- *A finite trace* $(s_0, \ldots, s_r)$ *is called* failing *if the statement* $\pi[n_r]$ *in the last state* $s_r = (I_r, n_r)$ *is an* assert *statement.*

- *A trace which is not failing is called* successful.

The table in the definition shows that a trace may possibly also terminate at an assumption. In structured DL, a failing assumption means that this execution branch diverges without post-state. We also do not consider a trace with a missed assumption any more. However, the state sequence is still counted a trace as it may already have passed some checked assertion and thus witness some specified properties of π.

Most of the statements increase the instruction pointer by one stepping into the next statement. Only the goto statement transfers control to other places than the successor statement and may hence have more than one successor state. The interpretation I is modified only by assignments and havoc statements which both change the value for a program variable p. The havoc statement may have more[4] than one successor, depending on the dimension of the domain. It may even have infinitely many.

The state transition function R_π modifies the interpretation function I, but does not touch the domain $\mathcal{D}$. This is known as the "constant-domain assumption" in modal predicate logic. Most program logics assume the set of considered objects to be constant throughout all conceivable states. But sometimes new elements may be created during the course of a program (by object creation or memory allocation). With a constant domain, this is modelled such that every element has always been present, even before its creation, but has been semantically marked as not yet created. The creation changes this mark. This makes the logic simpler and more uniform. However, there needs to be an extra predicate (or similar) to model the creation status of objects. Ahrendt et al. (2009) present a dynamic logic for an object-oriented

[4] It has at least one successor since a domain $\mathcal{D}^{\mathrm{ty}(p)}$ is a non-empty set.

language with a non-constant, growing domain. Its suitability for practical use remains to be examined.

For a finite trace, the type of the last executed statement is of the utmost importance as it decides upon the fate of the trace. If an assert-condition is not met, the program fails to fulfil its specification embedded into the program. If an end- or a failing assume-statement is reached, the trace is successful.

Definition 2.18 (Evaluation of program formulas, Amendment to Definition 2.8) *Let $D = (\mathcal{D}, I)$ be a semantic structure, τ a type variable assignment, β a compatible variable assignment, $\pi \in \Pi$ a program, $n \in \mathbb{N}$ an index and $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$ a formula. The evaluation function $\mathrm{val}_{I,\tau,\beta}$ introduced in Def. 2.8 is then extended to program formulas as follows:*

$$I,\tau,\beta \models [n;\pi] \iff \text{Every trace starting in } (I,n) \text{ is successful.}$$
$$I,\tau,\beta \models [\![n;\pi]\!] \iff I,\tau,\beta \models [n;\pi] \text{ and } \pi \text{ has no infinite trace starting in } (I,n).$$

This distinction is also the explanation of the fact that the table of Definition 2.17 only seemingly suggests the semantics of assert- and assume-statements be the same. This is not the case; while the successor relations are identical, their semantics differ fundamentally.

If $I,\tau,\beta \models [n;\pi]$, we can say that the program fulfils all of its embedded specification elements when started in (I,n). If $I,\tau,\beta \models [\![n;\pi]\!]$, termination of the program started in (I,n) is additionally ensured in all possible cases.

2.5.3 Post-fix Assertions

One point of difference between structured and unstructured DL is evident: DL modalities are followed by a formula following the modality while *UDL* has all assertions embedded in the modality. Note that structured DL has also embedded formulas in the tests (assume statements).

We will overcome this difference and allow formulas after program modalities also in *UDL*. The motivation behind this is that we regain thus the possibility to nest modalities. While it is syntactically possible to do so already (by asserting a program formula within another program), it is more comprehensible and comparable to other approaches if a post-fixed assertion is annotated.

Definition 2.19 (Post-fix assertions) *Let $\pi \in \Pi$ be a self-contained program and $0 \leq n < |\pi|$. The construct $[n;\pi]\varphi$ stands for the program formula $[n;\pi']$ with self-contained program $\pi' \in \Pi$ with $|\pi'| = |\pi| + 2$ and*

$$\pi'[n] = \begin{cases} \mathsf{goto}\ |\pi| & \text{if } \pi[n] = \mathsf{end} \\ \mathsf{assert}\ \varphi & \text{if } n = |\pi| \\ \mathsf{end} & \text{if } n = |\pi| + 1 \\ \pi[n] & \text{otherwise} \end{cases}$$

In comparison to program π, the modified program π' possesses two more statements at the end: $\mathsf{assert}\ \varphi$ followed by end. Every end statement within the range of the original program π is redirected to this appendix. This ensures that at the end of every successful trace, φ holds in addition to the embedded assertions.

It is only now that the end statement gets its right of existence. In the Definitions 2.17 and 2.18, there is no observable difference between the two statements assume false and end, both terminate a trace successfully. The difference becomes evident now that postfixed assertions are added to the modalities. We observe, for instance, that $[\mathsf{end}]\varphi \equiv \varphi$ while $[\mathsf{assume\ false}]\varphi \equiv \mathsf{true}$. Furthermore, we also name successful traces differently for the two cases: A successful finite trace $((I_0,n_0),\ldots,(I_r,n_r))$ with $\pi[n_r] = \mathsf{end}$ is called *convergent* and *divergent* if $\pi[n_r]$ is an assume-statement.

The syntactical alignment opens a common ground with the structured dynamic logic, and it allows the investigation of their relationship.

As has been mentioned earlier, structured dynamic logic has got two types of modality, the box $[\cdot]$ and the diamond $\langle\cdot\rangle$ operator. *UDL*, as we have defined it so far, has got the box $[\cdot]$ and the terminating box $[\![\cdot]\!]$ but no diamond operator. The diamond modality is, as usual for modal logics, defined to be the dual of the box-modality. We could not define a dual operator for the unstructured box operator up to this point as the post-fixed argument formula had been missing. With Definition 2.19 we can now finally define for $n < |\pi|$:

$$\langle n;\pi\rangle\varphi := \neg[n;\pi]\neg\varphi \qquad\qquad \langle\!\langle n;\pi\rangle\!\rangle\varphi := \neg[\![n;\pi]\!]\neg\varphi \tag{2.21}$$

Note that the post-fixed assertion φ plays a special role amongst the assertions checked in the program. It acts as pivotal point for the duality definition and appears negated in the definition of the diamond modality. All assertions embedded in π are not negated.

The semantics of the new constructs is then the following:

$I,\tau,\beta \models [n;\pi]\varphi \iff$ Every finite trace $((I,n),...,(I_r,n_r))$ of π is successful and $I_r,\tau,\beta \models \varphi$ if it is convergent.

$I,\tau,\beta \models [\![n;\pi]\!]\varphi \iff I,\tau,\beta \models [n;\pi]\varphi$ and every trace starting in (I,n) is finite.

$I,\tau,\beta \models \langle n;\pi\rangle\varphi \iff$ There is a finite trace $((I,n),...,(I_r,n_r))$ of π which is failing or convergent with $I_r,\tau,\beta \models \varphi$.

$I,\tau,\beta \models \langle\!\langle n;\pi\rangle\!\rangle\varphi \iff I,\tau,\beta \models \langle n;P\rangle\varphi$ or π has an infinite trace $((I,n),\ldots)$.

Let $\pi_{\mathrm{af}} \in \Pi$ be a program without embedded assertions. The trace semantics we defined due to the embedded assertions coincides with the semantics which would be expected of *UDL* as a modal logic.

$$I,\tau,\beta \models [n;\pi_{\mathrm{af}}]\varphi \iff I_r,\tau,\beta \models \varphi \text{ for } \textit{every} \text{ convergent trace } ((I,n),...,(I_r,n_r)) \text{ of } \pi$$

$$I,\tau,\beta \models \langle n;\pi_{\mathrm{af}}\rangle\varphi \iff I_r,\tau,\beta \models \varphi \text{ for } \textit{some} \text{ convergent trace } ((I,n),...,(I_r,n_r)) \text{ of } \pi$$

The Kripke state transition relationship is, hence, between the starting state (I,n) and the final states (I_r,n_r) of convergent traces.

If the program $\pi_{\text{af,det}}$ is assertion-free and also (semantically) deterministic, that is, the program started in any state has at most one non-diverging trace, then the modalities discriminate the issue of termination: $[\pi_{\text{af,det}}]\varphi$ says that $\pi_{\text{af,det}}$ is partially correct with respect to φ (that is, φ holds *if* $\pi_{\text{af,det}}$ terminates), and $\langle\pi_{\text{af,det}}\rangle\varphi$ says that $\pi_{\text{af,det}}$ is totally correct with respect to φ (that is, φ holds *and* π terminates). In particular the implication

$$\models \langle\pi_{\text{af,det}}\rangle\varphi \to [\pi_{\text{af,det}}]\varphi \tag{2.22}$$

is valid for all assertion-free, deterministic programs $\pi_{\text{af,det}}$. This does not hold for general *UDL* programs, however, which may be indeterministic or contain embedded assertions.

Indeterminism The existence of one successful trace does not ensure that every possible execution path is successful. Consider the program $P_{ind} := \langle \mathsf{goto}\ 1,2; x := x+1; \mathsf{end}\rangle$ for a program variable $x : \mathsf{nat}$. The formula $x \doteq 1 \to \langle 0; P_{ind}\rangle x \doteq 2$ is valid since there is one trace (visiting statements 0, 1, 2) which converges by incrementing x. However there is another trace (visiting 0, 2) which leaves x untouched such that $x \doteq 1 \to [0; P_{ind}]x \doteq 2$ is not valid.

Embedded assertions Embedded assertions have unexpected properties under the dual modality. Let us look at a small program which assigns the value 1 to a program variable $x : \mathsf{nat}$ and checks in an embedded assertion whether x equals 0. The name "assertion" suggests that the according program should fail and not reach a post-state regardless of the postcondition. Expanding Definitions 2.17 and 2.18 yields

$$I,\tau,\beta \models [x := 1; \mathsf{assert}\ x \doteq 0]\mathsf{true} \iff I[x \mapsto 1],\tau,\beta \models x \doteq 0 \text{ and } I[x \mapsto 1],\tau,\beta \models \mathsf{true} \tag{2.23}$$

$$\begin{aligned} I,\tau,\beta \models \langle x := 1; \mathsf{assert}\ x \doteq 0\rangle\mathsf{true} &\iff I,\tau,\beta \models \neg[x := 1; \mathsf{assert}\ x \doteq 0]\neg\mathsf{true} \\ &\iff I[x \mapsto 1],\tau,\beta \not\models x \doteq 0 \ \text{ or } \ I[x \mapsto 1],\tau,\beta \not\models \mathsf{true} \end{aligned} \tag{2.24}$$

The first case (2.23) behaves as expected: Despite the trivial postcondition, the formula does not hold, it fails at the assertion $x \doteq 0$. Formula (2.24) using the diamond modality, on the other hand, behaves differently: Due to the duality of box and diamond, the same program succeeds. It is De Morgan's law that lets the failing assertion $x \doteq 0$ make the diamond modal formula true.

We have thus two reasons why (2.22) does not hold and why the diamond modality does not model total correctness. This observation led to the introduction of the new modality $[\![\cdot]\!]$ which combines indeterminism and termination, but is not part of the standard repertoire of dynamic logic. The fourth modality $\langle\!\langle\cdot\rangle\!\rangle$ has been defined in (2.21) as the dual of $[\![\cdot]\!]$ for the sake of completeness. While the other three modal operators all have their right of existence in one or more typical use cases, this operator was never needed throughout the course of this work. If $\pi_{\text{af,det}} \in \Pi$ is again assertion-free and deterministic, the correspondences

$$[\pi_{\text{af,det}}]\varphi \equiv \langle\!\langle\pi_{\text{af,det}}\rangle\!\rangle\varphi \quad \text{and} \quad [\![\pi_{\text{af,det}}]\!]\varphi \equiv \langle\pi_{\text{af,det}}\rangle\varphi$$

emerge as the notions of “there exists one trace” and “for every trace” fall together.

We had observed that the behaviour of assertions under the diamond modality is not as one would expect intuitively. A very similar phenomenon applies to the assume statement under the two modalities. The relationship of assertions and assumptions is even so tangled that assert and assume are complementary in a sense:

Observation 2.5 (Duality of *UDL* statements) *Let $\varphi, \psi \in \mathrm{Trm}^{\mathsf{bool}}$ be UDL formulas, $p : T \in \mathrm{PVar}$ a program variable, $t \in \mathrm{Trm}^T$. Then the following semantic equivalences hold:*

$$\begin{aligned} [\mathsf{assert}\ \varphi]\psi &\equiv \langle \mathsf{assume}\ \varphi \rangle \psi \\ [\mathsf{assume}\ \varphi]\psi &\equiv \langle \mathsf{assert}\ \varphi \rangle \psi \\ [p := t]\psi &\equiv \langle p := t \rangle \psi \end{aligned}$$

Updates also each have a complementary dual operator:

Observation 2.6 (Updates are self-dual) *The formula $\{U\}\varphi \leftrightarrow \neg\{U\}\neg\varphi$ is valid for any update U and any formula $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$.*

PROOF Let I' like in Def. 2.16 denote the semantic structure which results from I after applying the update U.

$$\begin{aligned} \mathrm{val}_{I,\tau,\beta}(\neg\{U\}\neg\varphi) &= I(\neg)(\mathrm{val}_{I,\tau,\beta}(\{U\}\neg\varphi)) \\ &= I(\neg)(\mathrm{val}_{I',\tau,\beta}(\neg\varphi)) = I(\neg)(I'(\neg)(\mathrm{val}_{I',\tau,\beta}(\varphi))) \\ \text{using } I(\neg) = I'(\neg) \quad &= \underbrace{(I(\neg) \circ I(\neg))}_{=\mathrm{id}}(\mathrm{val}_{I',\tau,\beta}(\varphi)) = \mathrm{val}_{I',\tau,\beta}(\varphi) = \mathrm{val}_{I,\tau,\beta}(\{U\}\varphi) \end{aligned}$$

Assignment statements are closely related to update operators. Their being self-dual is therefore closely related to Observation 2.6. The dual operators to the goto and havoc statement could also be added to the syntax of statements, but no application for them could be found which would justify their introduction.

2.5.4 Removing Embedded Assertions

Embedding specification as assertions into the program is a deviation from the usual definitions in dynamic logic. We show, however, that in a first order logic setting, the checking of embedded assertions can efficiently be postponed until after the end of the program execution, thus resorting to standard notions of DL.

At first glance, this result may be unexpected since an embedded assertion may be called at an unbounded number of times (in infinite traces even infinitely often) while a post-fixed assertion can be examined only once at the end of the execution. Nonetheless, for any *UDL* program, a canonical equivalent program which does not contain embedded assertions can be constructed efficiently. The addition of assertions to the verification language then becomes a mere convenience, neither does it increase expressiveness of the language nor does it significantly reduce the sizes of the program text.

Theorem 2.7 (Removing embedded assertions in UDL) *Let $\pi \in \Pi$ be a self-contained UDL-program, $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$ a formula and $0 \leq n < |\pi|$ an index into π. The signature needs to contain a boolean program variable which does not occur in π and φ. Then there exist assertion-free programs $\pi', \pi'' \in \Pi$ and formulas $\varphi', \varphi'' \in \mathrm{Trm}^{\mathsf{bool}}$ such that*

$$[n;\pi]\varphi \equiv [n;\pi']\varphi' \quad \textit{and} \quad [\![n;\pi]\!]\varphi \equiv [\![n;\pi']\!]\varphi' \quad \textit{and} \quad \langle n;\pi\rangle\varphi \equiv \langle n;\pi''\rangle\varphi'' \ .$$

The sizes of π' and π'' are linear in the sizes of the original program.

PROOF The idea of the construction of π' is that the result of the embedded assertion checks can be "stored" in a common boolean program variable. Checking that program variable then stands in for checking any of the assertions. Let $l = |\pi|$ be the number of statements in π and $a : \mathsf{bool} \in \mathrm{PVar}$ be a program variable of type bool not occurring in π or φ. The program π' with $|\pi'| = 3l$ is then chosen such that for all $n \in \mathbb{N}, n < l$

$$\begin{aligned}
\pi[n] = \mathsf{assert}\ \psi &\implies \begin{cases} \pi'[n] = \mathsf{goto}\ n+1, l+2n & (*) \\ \pi'[l+2n] = a := \psi \end{cases} \\
\pi[n] = \mathsf{end} &\implies \begin{cases} \pi'[n] = \mathsf{goto}\ l+2n \\ \pi'[l+2n] = a := \varphi \end{cases} \\
\pi[n] \text{ is neither } \mathsf{assert} \text{ nor } \mathsf{end} &\implies \begin{cases} \pi'[n] = \pi[n] \\ \pi'[l+2n] = \mathsf{end} \end{cases} \\
\text{for all } 0 \leq n < l &\implies \pi'[l+2n+1] = \mathsf{end}
\end{aligned}$$

The first l statements coincide between programs π and π' except for end- and assert-statements, which are each replaced by a goto-statement to a unique index $l + 2n$ beyond the domain of π. At the target position the program variable a is set to the condition to be checked at $\pi[n]$ (either the assertion ψ or the postcondition φ). Since $\pi[l + 2n + 1] = \mathsf{end}$ for all $n < l$, execution terminates after that assignment. Checking that a is true after the execution, hence, indirectly checks whether the assertion holds in the same state. The position of every assert-statement in π is replaced by an indeterministic goto (at $(*)$) branching indeterministically between the check of the assertion and the next position of the program. This ensures that the program execution continues after a successful assertion check.

The thus constructed program has the desired properties for the modality $[\cdot]$. We show this by examining the semantics when the modality formula does not hold:

$I \not\models [n;\pi]\varphi$

$\iff$ there is a maximal trace $(I,n),\ldots,(I_r,n_r)$ with
 $\pi[n_r] = \text{end}$ and $I_r \not\models \varphi$
 or $\pi[n_r] = \text{assert } \psi$ and $I_r \not\models \psi$

$\iff$ there is a maximal trace $(I,n),\ldots,(I_r,n_r),(I_r,l+2n_r),(I_{r+2},l+2n_r+1)$ for π' with
 $\pi[n_r] = \text{end}$ and $I_{r+2} = I_r[a \mapsto \varphi]$ and $I_{r+2} \not\models a$
 or $\pi[n_r] = \text{assert } \psi$ and $I_{r+2} = I_r[a \mapsto \psi]$ and $I_{r+2} \not\models a$

$\iff$ $I \not\models [n;\pi']a$ (since the only end points of π' are those induced by an assert or end in π)

The set of infinite traces is the same for $[n;\pi]$ and $[n;\pi']$. Together with the aforementioned this implies that $[\![n;\pi]\!]\varphi \leftrightarrow [\![n;\pi']\!]a$ is valid.

Due to the duality of the modalities, the following formulas are equivalent:

$$I \models \langle n;\pi\rangle\varphi \iff I \models \neg[n;\pi]\neg\varphi \iff I \models \neg[n;\pi'']a \iff I \models \langle n;\pi''\rangle\neg a$$

in which program π'' is the same as π' but with $\pi[n] = \text{end}$ implying $\pi[l+2n] = a := \neg\varphi$ (since the postcondition in this place is $\neg\varphi$). To apply the duality of (2.21) again, the evident equivalence $a \equiv \neg\neg a$ is used. □

2.5.5 Propositional *UDL*

To investigate the relationship of the control flow mechanism of *UDL* in comparison to structured DL, we will in this section look at a version of dynamic logic abstracting away from data representation and concrete programs resulting in a logic with a decidable satisfiability problem. This abstracted logic is called *Propositional Dynamic Logic* (PDL) and has been proposed by Fischer and Ladner (1977, 1979).

In PDL, formulas are abstracted from by parameterless propositional variables, and data modifying statements (that is assignments and havoc statements) are seen as *atomic programs* whose semantics is not specified any further. The regular program constructors ?, ∪ and * responsible for control flow are kept unchanged, however.

Unstructured PDL (UPDL) now applies the same abstraction to *UDL*: Propositional variables replace formulas and atomic programs replace assignments and havoc-statements. The control-flow statement constructors goto, assume, assert, end of *UDL* are, again, kept in UPDL. The difference between PDL and UPDL is that the control flow is transferred using goto and end statements rather than by structured blocks.

The embedded assertions in UPDL lack a corresponding counterpart in structured PDL. It is now interesting to see if the assertions increase expressiveness in this abstracted setting. In Theorem 2.7 we have seen that using a fresh boolean program variable, embedded assertions can be removed. But now propositional abstraction took away from us the possibility to store the assertion result in a program variable. Instead we state the slightly weaker

Theorem 2.8 (Removing embedded assertions in UPDL) *For every UPDL formula φ containing embedded assertions, there is an equivalent UPDL formula ψ without assertions statements in programs.*

PROOF by structural induction. We show the interesting case: Every formula $[n;P]\varphi$ is equivalent to a formula ψ such that the top-level program formulas in ψ do not contain assertions. Let $\{\mathsf{assert}\ \sigma_1, \ldots, \mathsf{assert}\ \sigma_k\}$ enumerate all assert-statements in P. Define then for $1 \le i \le k$:

$$\begin{aligned}
\psi &:= \bigwedge_{i=0}^{k} [n;P_i]\varphi_i \\
P_0 &:= P \text{ with all assertions replaced by } \mathsf{skip} \\
\varphi_0 &:= \varphi \quad \text{(the postfixed assertion)} \\
P_i &:= P \text{ with all } \mathsf{end} \text{ statements replaced by } \mathsf{assume\ false} \text{ and all assertions} \\
&\ \text{replaced by } \mathsf{skip}\text{, but } i\text{-th one which becomes } \mathsf{end} \\
\varphi_i &:= \sigma_i \quad \text{(the } i\text{-th assertion)}
\end{aligned}$$

Let I be an interpretation with $I \not\models [n;P]\varphi$. There is then a trace starting in (I,n) which fails at an assertion assert σ_j in P. The same trace is also a failing trace for $[n;P_j]\varphi_j$, therefore $I \not\models \psi$.

Let $I \not\models \psi$ which means that there is an index $1 \le j \le k$ such that $[n;P_j]\varphi_j$ has a failing trace starting in (I,n). This state sequence induces a failing trace for $[n;P]$ as well. Either an assertion prior to assert σ_j (one of those replaced by skip) fails or the execution reaches assert σ_j. Since the trace fails the asserted condition σ_j of P_i for this state, the embedded assertion assert σ_j is falsified as well.

All formulas which occur within the programs P_i can be replaced by an assertion-free equivalent formula by induction hypothesis. □

The predicate logic Theorem 2.7 gave us that any program modality term with embedded assertions can canonically and efficiently be transformed into one without. Here every program formula must be replaced by a conjunction of several terms. No equivalent single program can be given. But the theorem allows us to concentrate on UPDL programs without embedded assertions.

Harel and Sherman (1985) examine the expressiveness and complexity of UPDL without embedded assertions (they call it APDL for Automaton PDL). Any UPDL program can be seen as a non-deterministic finite automaton (NFA) over atomic programs and tests in which goto statements model the state transitions. Any regular PDL program, on the other hand, can be seen as a regular expression over atomic programs and tests. Finite traces of programs can then be seen as words over tests and atomic programs.

The semantics of a PDL program α can be described by the set of all possible sequences of tests and atomic programs. These sequences are called *computation sequences* of α. The set of computation sequences generated by an unstructured or a structured PDL program is always regular. It is only the notation formalism (regular expressions versus NFA) that distinguishes them. Figure 2.4 draws a picture of the related natures of automata (unstructured programs) and regular programs.

It is comparatively straightforward to construct an accepting NFA from a regular expression with n literals such that the automaton has $O(n)$ states (see Chang and Paige (1992) for details). The reverse transformation is harder. There exist various algorithms to extract an equivalent regular expression from a finite automaton: for instance the stepwise state-removal as described by Linz (1997), or the algebraic approach of Brzozowski (1964) constructing a solution of a system of linear equations on regular expressions. The resulting regular expressions may still be exponential in the number of states of the NFA (Harel and Sherman, 1985).

This is a strong argument in favour of the more flexible formalisation of algorithms as unstructured programs (that is, as NFA) rather than as regular programs: It is straight-forward to convert structured input into unstructured, but highly difficult to do the other way, the result will be larger and less intuitive to understand.

```
while $P$ do
   if $Q$ then $\alpha$
      else $\beta$ end;
$\gamma$
```

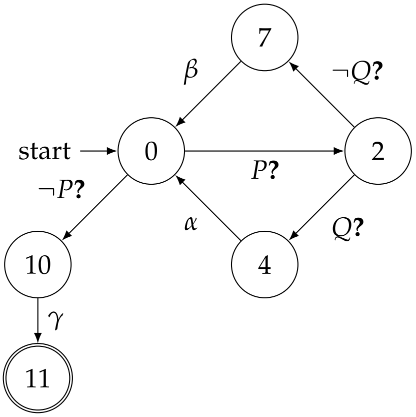
(a) Pseudocode

$$(P?;((Q?;\alpha)\cup(\neg Q?;\beta)))^*;\neg P?;\gamma$$

(b) Regular program

```
0: goto 1,9
1: assume P
2: goto 3,6
3: assume Q
4: α
5: goto 8
6: assume ¬Q
7: β
8: goto 0
9: assume ¬P
10: γ
11: end
```

(c) Unstructured program



(d) Finite Automaton

Figure 2.4: Example for different representations of the same regular set of execution paths

In one respect, UPDL is strictly more expressive than PDL. For a deterministic program α, the formula $\varphi \rightarrow \langle\alpha\rangle\psi$ describes the verification condition for total correctness of α with precondition φ and postcondition ψ. For an indeterministic program, no PDL formula can express a total correctness condition, the UPDL formula $[\![\alpha; \mathsf{assume}\ P; \mathsf{goto}\ 0]\!]$ for an atomic program α and a propositional variable P has no equivalent in PDL. See Harel et al. (2000) for details.

2.6 Chapter Summary

This chapter has laid the foundations for the thesis by formally defining the syntax and semantics of Unstructured Dynamic Logic (*UDL*), the logic in which verification problems will be formulated and solved in the remaining chapters of the thesis. The logic has been outlined also in Ulbrich (2011).

First, the underlying predicate logic which does not contain programs to verify has been defined and examined. The type system of the logic is parametric, types are themselves expressions built from type constructors and type variables. But every term belongs to precisely one variable-free type, there is no hierarchy (subtyping relationship) in the type system. Although the logic is first order, it possesses syntactical constructions to quantify over type variables. This allows the formulation of type-agnostic statements which hold for all instantiations of a type variable. The logic shares with many other typed logics, that the notion of formulas falls together with that of boolean terms.

In addition to predicate logic function symbols, *UDL* allows the declaration of *binder* symbols which bind a variable in their arguments and are, hence, not evaluations depending on argument values but on argument functions. We have shown that first order predicate logic with binder symbols can be reduced to first order logic without them. Although this is a straightforward result, we could not find it in the literature.

With the underlying logic outlined, we have turned to the dynamic logic aspect of *UDL*. Dynamic logic allows the embedding of programs in modal constructors for formulation of program verification conditions. The programming language which has been chosen for the embedded programs shares its primitive statement constructors with the intermediate verification language Boogie (Leino and Rümmer, 2010), the choice is intentional as the language of Boogie has been successfully used in many verification projects. The indeterministic programming language is intentionally limited in the number of constructors. They are: Assignment, assertion, assumption, anonymisation (called havoc) and branching. The logic has common ground with standard structured dynamic logic (Harel et al., 2000) but they also have mentionable differences.

Dynamic logic is defined in terms of a state relation between the beginning and the terminal state of a program. We have seen a formal definition of the semantics of *UDL* in terms of *program traces*. A trace is a sequence of program states which is backed up by a run of the program. This different program semantics is required since *UDL* programs may contain *embedded assertions* (defined by their assertion statements). As an important result we have shown that every program with embedded assertions can be canonically reduced to a program with a single assertion checked after the program execution, the standard semantics of modal logic. Nevertheless, when applying the logic to verification tasks (in Chapter 4) , we shall see that having several checks which are localised helps the comprehensibility of programs and proofs.

Classical structured dynamic logic has got a variant which is called propositional dynamic logic (PDL) in which programs and formulas are abstracted. We have applied the same kind of abstraction to *UDL* and obtained the unstructured version UPDL of PDL. Also in the abstracted logic, embedded assertions are not essential. In general, there is, as we have shown, not a canonical program with the same semantics but every formula with programs with embedded assertions can be reduced to an equivalent formula in which all programs are assertion-free.

CHAPTER 3

A Sequent Calculus for *UDL*

In this chapter, we present a sound sequent calculus for the Unstructured Dynamic Logic *(UDL) presented in Chapter 2.*

UDL formulas may contain programs in their formulas. Rules will be presented that perform symbolic execution *on programs in formulas to reduce proof obligations with programs to formulas without programs in a stepwise manner. Programs which contain loops must be treated differently. Several rules are proposed that allow the resolution of programs with loops.*

UDL is more expressive than first order logic; hence, no complete calculus exists. However, we show that the presented calculus is relatively complete*, that is, that the provided set of rules allow the reduction of proof obligations with programs to obligations on natural numbers.*

3.1 Sequent Calculus

The calculus to reason about the validity of *UDL* is an extension of the *Sequent calculus* originally proposed by Gentzen (1935) for first-order predicate logic.

Definition 3.1 (Sequent, Inference rule) *A* sequent *is a pair of finite sets* $\Gamma, \Delta \subseteq \mathrm{Trm}^{\mathsf{bool}}$ *of formulas without free variables written as* $\Gamma \vdash \Delta$ *in which the set* $\Gamma = \{\gamma_1, ..., \gamma_g\}$ *is called the* antecedent *and* $\Delta = \{\delta_1, ..., \delta_d\}$ *the* succedent. *The sequent* $\Gamma \vdash \Delta$ *has the same semantic value as the formula* $(\gamma_1 \wedge \ldots \wedge \gamma_g) \to (\delta_1 \vee \ldots \vee \delta_d)$.

A sequent calculus inference rule *is a sequence* $\langle P_1, ..., P_n, C\rangle$ *of sequents usually written as*

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

in which the sequents P_i *above the line are called the* premisses *of the rule and the sequent* C *below the line the* conclusion. *An inference rule is called* sound *if the validity of all premisses implies the validity of the conclusion.*

The rules close, andLeft and andRight are examples of rules that have no, one or two premisses and one conclusion:

$$\frac{}{\Gamma, A \vdash A, \Delta}\text{close} \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\text{andLeft}$$
$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\text{andRight} \tag{3.1}$$

Rules are usually stated in a schematic form not only containing terms but also placeholders for syntactical entities (formulas, terms, updates, modalities, ...). In the schematic rules (3.1), A and B are schematic placeholders for formulas, and Γ and Δ for sets of formulas. A schematic rule represents the set of all rules that can be derived by instantiating its schematic placeholders. When notating rules, we will take the liberty of relaxing the set notation and write Γ, A, B (and similar) instead of $\Gamma \cup \{A, B\}$.

Proofs in sequent calculus are conducted by applying inference rules to sequents yielding new sequents which can then again be subjected to rule applications. The iterative application of rules results in a tree:

Definition 3.2 (Proof tree) *A sequent calculus* proof tree *is a finitely branching tree such that*

1. *Every node is labelled with a sequent.*
2. *Every inner node is labelled with an inference rule. Every leaf may be labelled with an inference rule.*
3. *An inner node labelled with sequence C whose children are labelled with sequents $P_1, \ldots, P_k$, is labelled with the inference rule* $\frac{P_1 \quad \ldots \quad P_k}{C}$.
4. *If a leaf is labelled with sequent C and an inference rule, then the inference rule label is* $\frac{}{C}$.

A proof tree is called closed *if every node is labelled with a rule. A leaf without inference rule label is called an* open goal.

During a proof search, a proof tree is constructed bottom up starting with a single node containing the sequent to be proved valid. A rule with conclusion C can be applied to an open goal with sequence C. The premisses of the rule are then added to the proof tree as new children to the leaf node. A rule schema can be applied by matching the schematic conclusion against an open goal. The premisses are instantiated with the unifying substitution before they are added as new children. A rule without premisses (close in (3.1) for instance) is called an *axiom*. Applying an axiom to a leaf of the tree, closes the branch.

Observation 3.1 (Soundness of sequent calculus) *The sequent S is universally valid if there exists a closed proof tree such that*

1. *the root is labelled with S and*
2. *all inference rule labels are sound rules.*

PROOF Since the tree is closed, the sequents in the leaves must be valid formulas (since they result from the application of sound axioms). The fact that every inference rule label is a sound rule inductively ensures that every sequent (including the root S) in the tree is valid. □

It is very important that every rule of the calculus be sound as any unsound rule will compromise the *correctness* of the approach. Programs with faults would potentially be proved correct which is not acceptable. *Completeness*, on the other hand, cannot be achieved for *UDL* both theoretically and practically. Theoretically not since *UDL* comprises logics for which no complete calculus exists (for instance program formulas or type quantifiers). But the extend of the search space for many programs renders their verification technically infeasible and this practical boundary is well below the theoretical frontier. But it is still desirable to have a calculus which is "as complete as possible". An inference rule is called *complete* if there exists a closed proof tree after its application to a leaf if there has been one before[1]. Semantically, this means that the relationship between the premisses and conclusion of the rule is not only an implication but an equivalence. Applying incomplete rules to a proof tree may lead into a dead end, hence, they should be applied with caution, especially if their application is triggered automatically during a mechanised proof search. Most inference rules that will be presented in this chapter will be complete with exception of the loop invariant rules defined in Section 3.3.

There is another category of rules called *rewrite rules*: For two equally typed terms $s, t \in \mathrm{Trm}^T$, we write $s \rightsquigarrow t$ to denote the rule schema which replaces one occurrence of term s anywhere in a sequent by the term t. If terms s and t are semantically equivalent (that is, if $s \equiv t$), then the replacement does not change the truth value of the sequent. The respective inference rules are correct. Rewrite rules can be, like inference rules, schematic. The conditional schematic rewrite rule $C \implies s \rightsquigarrow t$ possibly containing schematic variables is an abbreviation for the set $\{s \rightsquigarrow t \mid C\}$ of all instances for which the condition C holds. An example of a conditional schematic rewrite rule is the rule $x \notin \mathrm{freeVars}(\varphi) \implies (\forall x.\ \varphi) \rightsquigarrow \varphi$ which permits the removal of unneeded quantifiers. The rule is not essential but is a convenient extension to the calculus.

The soundness condition for an inference rule can be formalised in predicate logic itself; Bubel et al. (2008) show how derived inference rules can be proved correct using more fundamental rules. This allows us to enrich the calculus with lemma inference rules which are not strictly needed but may improve the efficiency of the calculus without compromising its correctness.

[1] This is sometimes also referred to as a *confluent* rule.

3.1.1 Rules of the Calculus

Tables 3.1 and 3.2 summarise the sequent calculus inference rules for propositional logic, first-order logic and the extensions introduced in Section 2.3. The part of *UDL* dealing with program modalities will be subject of the next sections.

Propositional Logic

$$\frac{\Gamma, \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\text{notLeft} \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\text{andLeft} \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\text{orLeft}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\text{notRight} \qquad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\,\text{andRight} \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}\,\text{orRight}$$

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\,\text{impLeft} \qquad \frac{\Gamma, A \to B, B \to A \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}\,\text{equivLeft} \qquad \frac{\Gamma, C \vdash \Delta \quad \Gamma \vdash \Delta, C}{\Gamma \vdash \Delta}\,\text{cut}$$

$$\frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B}\,\text{impRight} \qquad \frac{\Gamma \vdash \Delta, A \wedge B, \neg A \wedge \neg B}{\Gamma \vdash \Delta, A \leftrightarrow B}\,\text{equivRight} \qquad \frac{}{\Gamma, A \vdash A, \Delta}\,\text{axiom}$$

$$\frac{}{\Gamma, \mathsf{false} \vdash \Delta}\,\text{falseLeft} \qquad \frac{}{\Gamma \vdash \mathsf{true}, \Delta}\,\text{trueRight} \qquad \begin{array}{c}(A \doteq B) \rightsquigarrow (A \leftrightarrow B)\\ A, B \in \mathrm{Trm}^{\mathsf{bool}} \quad \text{eqToEquiv}\end{array}$$

Object Quantifiers

$$\frac{\Gamma, (\forall x^T.t), t[x^T/t'] \vdash \Delta}{\Gamma, (\forall x^T.t) \vdash \Delta}\,\text{forallLeft} \qquad \frac{\Gamma \vdash t[x^T/c], \Delta}{\Gamma \vdash (\forall x^T.t), \Delta}\,\text{forallRight}$$

with $T \in \mathcal{T}$, $t' \in \mathrm{Trm}^T$ without free variables — with $c : T$ a fresh constant symbol

$$\frac{\Gamma, t[x^T/c] \vdash \Delta}{\Gamma, (\exists x^T.t) \vdash \Delta}\,\text{existsLeft} \qquad \frac{\Gamma, \vdash (\exists x^T.t), t[x^T/t'] \vdash \Delta}{\Gamma, \vdash (\exists x^T.t) \vdash \Delta}\,\text{existsRight}$$

with $c : T$ a fresh constant symbol — with $T \in \mathcal{T}$, $t' \in \mathrm{Trm}^T$ without free variables

Strong and weak equality

$$\frac{\Gamma', t_1 \doteq t_2 \vdash \Delta'}{\Gamma, t_1 \doteq t_2 \vdash \Delta}\,\text{applyEq} \qquad \frac{\sigma(\Gamma), \sigma(t_1 \doteq t_2) \vdash \sigma(\Delta)}{\Gamma, t_1 \dot{\approx} t_2 \vdash \Delta}\,\text{leftTypeEq}$$

where Γ', Δ' emerge from Γ, Δ by replacing one or more occurrences of t_1 by t_2 which are not within a program formula or behind an update.

where σ is a most general unifier of T_1 and T_2 with $t_i \in \mathrm{Trm}^{T_i}$

$$\begin{array}{rcccll}
 & & t \doteq t & \rightsquigarrow & \mathsf{true} & \text{eqRefl}\\
\text{the types of } t_1 \text{ and } t_2 \text{ cannot be unified} & \Longrightarrow & t_1 \dot{\approx} t_2 & \rightsquigarrow & \mathsf{false} & \text{typeDiff}\\
t_1, t_2 \in \mathrm{Trm}^T & \Longrightarrow & t_1 \dot{\approx} t_2 & \rightsquigarrow & t_1 \doteq t_2 & \text{typeEq}
\end{array}$$

Table 3.1: Sequent calculus I

General binder symbols

$$y^T \text{ does not occur in } t \implies (b\; x^T.t) \rightsquigarrow (b\; y^T.t[x^T/y^T]) \quad \text{binderAlpha}$$

$$\frac{\Gamma \vdash (\forall x^T.t_1 \doteq u_1 \wedge \ldots \wedge t_n \doteq u_n),\Delta}{\Gamma \vdash (b\; x^T.t_1,\ldots,t_n) \doteq (b\; x^T.u_1,\ldots,u_n),\Delta}\,\text{binderExt}$$

Type quantification

$$\frac{\Gamma,(\mathbb{V}\alpha.t),\sigma(t) \vdash \Delta}{\Gamma,(\mathbb{V}\alpha.t) \vdash \Delta}\,\text{alltypesLeft}$$
with $\sigma = \{\alpha \mapsto T\}$ for some type $T \in \mathcal{T}$

$$\frac{\Gamma \vdash \sigma(t)\Delta}{\Gamma \vdash (\mathbb{V}\alpha.t),\Delta}\,\text{alltypesRight}$$
with $\sigma = \{\alpha \mapsto \beta\}$ for $\beta \in$ TVar not occurring in the conclusion

$$\frac{\Gamma,\sigma(t) \vdash \Delta}{\Gamma,(\reflectbox{$\mathbb{E}$}\alpha.t) \vdash \Delta}\,\text{extypeLeft}$$
with $\sigma = \{\alpha \mapsto \beta\}$ for $\beta \in$ TVar not occurring in the conclusion

$$\frac{\Gamma \vdash (\reflectbox{$\mathbb{E}$}\alpha.t),\sigma(t),\Delta}{\Gamma \vdash (\reflectbox{$\mathbb{E}$}\alpha.t),\Delta}\,\text{extypeRight}$$
with $\sigma = \{\alpha \mapsto T\}$ for some type $T \in \mathcal{T}$

$$\frac{\begin{array}{l}\Gamma \vdash (\mathbb{V}\alpha_1.\ \ldots \mathbb{V}\alpha_{\mathrm{ar}(C_1)}.\ \{\alpha/C_1(\alpha_1,\ldots,\alpha_{\mathrm{ar}(C_1)})\}\varphi),\Delta\\ \ldots\\ \Gamma \vdash (\mathbb{V}\alpha_1.\ \ldots \mathbb{V}\alpha_{\mathrm{ar}(C_{|\Gamma|})}.\ \{\alpha/C_{|\Gamma|}(\alpha_1,\ldots,\alpha_{\mathrm{ar}(C_{|\Gamma|})})\}\varphi),\Delta\end{array}}{\Gamma \vdash (\mathbb{V}\alpha.\ \varphi),\Delta}\,\text{typeInduction}$$
where $\mathrm{TCon}_\Gamma = \{C_1,\ldots,C_{|\Gamma|}\}$ and α_i are fresh type variables

Table 3.2: Sequent calculus II

The propositional rules are the rules of the traditional sequent calculus. They include also the rule 'cut' which allows the interactive application of case distinctions over a formula C. There are two general approaches to deal with quantifiers in sequent calculus. Either universal quantifiers can be instantiated with arbitrary ground terms or they are instantiated with free variables which become instantiated at a later point during the proof. We opted for the ground version of the calculus. This has the advantage that only Skolem *constants* need to be instantiated for existential quantifieres whereas free variables would require that Skolem *functions* (with the free variables as arguments) be introduced making the formulas more complex. The rules for treatment of quantifiers are thus taken directly from the classical sequent calculus but have been adapted to the parametric type system.

UDL has not one but *two* equality symbols $\doteq$: $\alpha \times \alpha \to \texttt{bool}$ (the strongly typed equality) and $\dot{\approx}$: $\alpha \times \beta \to \texttt{bool}$ (the weakly typed equality). The rules eqRefl and applyEq are known to be sound (and complete) for the strong equality (see also Degtyarev and Voronkov, 2001). Semantically, weak and strong equality are equivalent but $\dot{\approx}$ has a less strict type signature. If the types of the arguments of the weak equality are equal, it can equivalently be rewritten using the strong equality (rule typeEq). If a weak equality occurs toplevel in the antecedent of a sequent, equality of its arguments is assumed, including equality of their types. Rule leftTypeEq implements this by applying a unifying type substitution to the entire sequent. If the argument types of a weak equality cannot be unified, its value is known to be false (rule typeDiff).

For the general treatment of binder symbols, a rewriting rule for renaming the bound variable (binderAlpha) is provided. Also an extensionality rule for binder symbols is introduced which allows the reduction of an equality of binder terms to the equality of their arguments. The rule binderExt is correct since binder symbols must, by definition, yield the same result if applied to semantically equal (for all values of the bound variable) parameter terms.

Type quantifications possess instantiation rules which resemble the rules for object quantification. They do not instantiate variables of the argument φ, but apply a type variable assignment σ to the type variables which appear in the type of φ. Additionally, an induction schema typeInduction is provided with which universal statements for all types can be inductively conducted for all type constructors. It is sound since the type system is the freely generated structure over the type constructors.

3.1.2 Why Sequent Calculus?

Calculi need a representation of their intermediate proof state. Some calculi have representations which are little comprehensible for the reader. For example, *resolution*-based systems transform formulas into sets of clauses and *tableau* calculi store proof states as trees.

Interactive proof assistants (like HOL, Isabelle or Event-B) often employ a *Natural Deduction Calculus*. Natural deduction can be regarded as the special case of sequent calculus in which all judgements $\Gamma \vdash \Delta$ are constrained by $|\Delta| = 1$. The succedent in such calculi holds the formula under inspection while the antecedent contains the available prerequisites.

Originally, the sequent calculus was introduced to establish Gentzen's Hauptsatz stating that every proof in sequent calculus can be conducted without resorting to the Cut-Rule (also known as Cut-elimination). This is particularly in contrast to systems based on natural deduction. For an automated proof system the prospect of cut-freedom is very appealing since it greatly reduces the search space: Not all conceivable formulas have to be considered for instantiation of the Cut rule at all times, a more goal-driven reasoning can be performed (like in tableaux or resolution-based calculi).

The other side of the coin is that a representation which is well-suited for a machine, may be less intuitive to a human user. A survey conducted by Grebing (2012) revealed that the equal distribution of formulas over the right and left hand sides tends to render proof obligations less conceivable for a human. In particular the rules notLeft[2], notRight (and similar) can cause confusion on the sequent by transferring a statement from the antecedent to the succedent or vice versa.

[2] see Table 3.1

Beside the logic and system presented here, the interactive theorem provers KeY (Beckert et al., 2007) and PVS (Shankar and Owre, 1999) employ sequent calculus: It appears to be a good compromise for the combination of interactive and automated reasoning.

3.2 Symbolic Execution in *UDL*

Traditionally, the calculation of weakest preconditions is done in a backward fashion starting at the last statement and modifying the postcondition in a stepwise manner until the program has been fully removed. In Dynamic Logic, *symbolic forward execution* (King, 1976), which preserves the natural execution direction, can be used to compute weakest preconditions. In an interactive proof environment, the execution direction plays a role when the comprehension of the human user is an issue.

Symbolic forward execution keeps a record of the symbolic state which is constantly updated while stepping over the statements of a program. This engulfs an *execution tree* of all possible execution paths through the program. Despite the fact that the execution is done from front to end, symbolic execution computes the weakest precondition rather than the strongest postcondition of the program.

The collected effects of assignments are accumulated in updates which are used as representation of the intermediate execution state. After the program has been fully symbolically executed, the updates can syntactically be applied to the postcondition, resulting in the update-free weakest precondition of the according branch.

Updates can always be removed from formulas which do not contain program formulas. There may be variable naming clashes hindering the application of updates to terms with bound variable, but these can always be avoided by renaming the bound variable[3].

Theorem 3.2 (Update resolution) *Let $f \in \mathrm{Fct} \setminus \mathrm{PVar}$ be a function symbol which is not a program variable, $p,q \in \mathrm{PVar}$ program variables, $b \in \mathrm{Bnd}$ a binder symbol. Moreover, $a_1,\dots,a_n \in \mathrm{Trm}$ are terms of appropriate type, $v \in \mathrm{Var}$ a variable, and $Q \in \{\forall, \exists\}$. The following rules are sound rewrite rules for the resolution of updates:*

$$\begin{aligned}
& \{U\}v \rightsquigarrow v \\
& \{U \parallel p := t\}p \rightsquigarrow t \\
q \neq p \implies & \{U \parallel q := t\}p \rightsquigarrow \{U\}p \\
& \{U\}f(a_1,\dots,a_n) \rightsquigarrow f(\{U\}a_1,\dots,\{U\}a_n) \\
\textit{v does not occur free in U} \implies & \{U\}(b\, v.a_1,\dots,a_n) \rightsquigarrow (b\, v.\{U\}a_1,\dots,\{U\}a_n) \qquad (3.2) \\
\alpha \textit{ does not occur in U} \implies & \{U\}(Q\alpha.a_1) \rightsquigarrow (Q\, \alpha.\{U\}a_1) \qquad (3.3)
\end{aligned}$$

[3] Using rule binderAlpha from Table 3.2

PROOF According to Definition 2.16, updates modify the interpretation function I' under which the argument term is evaluated. Due to the last-win-semantics of the successive modification of I, updates to program variables must be coiled up from the back. If the considered program variable p is the last updated program variable, the result is $I'(p) = \mathrm{val}_{I,\tau,\beta}(t)$. An update assignment to a program variable q different from p, does not touch the value of $I(p)$ and can be discarded; not however the other modifications made to I. Since updates only touch the values of program variables, $I(f)$ and $I(b)$ and the type quantifier cannot be the affected, and the updates are distributed over their arguments. □

The restrictions in (3.2) and (3.3) are necessary. Moving an update containing a free variable into the arguments of a binder symbol binding the very same variable results in an illegal change of scope of the bound variable:

$$(\exists x^S.\{p := x^S\}(\forall x^S.p \doteq x^S)) \not\equiv (\exists x^S.\forall x^S.\{p := x^S\}p \doteq x^S) \tag{3.4}$$

$$\exists\!\!\exists \alpha.\{p := q(c^{[\alpha]})\}\forall\!\!\forall \alpha.p \to q(c^{[\alpha]}) \not\equiv \exists\!\!\exists \alpha.\forall\!\!\forall \alpha.\{p := q(c^{[\alpha]})\}p \to q(c^{[\alpha]}) \tag{3.5}$$

Example (3.4) shows that the distribution of an update into a conflicting binding is not sound. After renaming the first bound variable to y^S, the semantical difference becomes apparent. The left-hand side is then equivalent to $(\exists y^S.\forall x^S.y^S \doteq x^S)$, which is not a valid formula in general (unless $|\mathcal{D}^S| = 1$), while the right-hand side is equivalent to $(\exists y^S.\forall x^S.x^S \doteq x^S)$, which is valid. A similar restriction exists for the distribution of updates over a type quantification and (3.5) shows that it is necessary as well.

Updates can be seen as a form of explicitly notated substitutions. These substitutions can be syntactically applied to program variables by substituting them in terms everywhere but in program formulas and behind other updates. A term is said to be in *update normal form* if every update stands in front of a program formula.

Let us now turn towards program formulas, the last remaining construct without calculus rules. First, a set of rewrite rules which formally capture the effects of a single step of symbolic execution is presented. These rules already suffice to resolve linear[4] program formulas to first-order formulas.

[4] that is, programs without loops

Theorem 3.3 (Symbolic execution) *The following rules are sound rewrite rules for the forward symbolic execution of unstructured programs. Let $x^{\mathrm{ty}(c)} \in \mathrm{Var}$ be a variable that does not occur in π.*

$$\begin{aligned}
\pi[n] = \mathsf{skip} &\implies [n;\pi] \rightsquigarrow [n+1;\pi] \\
\pi[n] = \mathsf{end} &\implies [n;\pi] \rightsquigarrow \mathsf{true} \\
\pi[n] = c := v &\implies [n;\pi] \rightsquigarrow \{c := v\}[n+1;\pi] && (3.6)\\
\pi[n] = \mathsf{havoc}\ c &\implies [n;\pi] \rightsquigarrow (\forall x^{\mathrm{ty}(c)}.\{c := x\}[n+1;\pi]) && (3.7)\\
\pi[n] = \mathsf{goto}\ g_1,\ldots,g_k &\implies [n;\pi] \rightsquigarrow [g_1;\pi] \land \ldots \land [g_k;\pi] \\
\pi[n] = \mathsf{assume}\ \varphi &\implies [n;\pi] \rightsquigarrow \varphi \to [n+1;\pi] \\
\pi[n] = \mathsf{assert}\ \varphi &\implies [n;\pi] \rightsquigarrow \varphi \land [n+1;\pi]
\end{aligned}$$

Replacing $[\cdot]$ by $[\![\cdot]\!]$ in the rewrite rules yields a set of sound rewrite rules including termination. The introduced logical variables $x^{\mathrm{ty}(c)}$ are chosen such that they do not occur in π.

PROOF To prove the correctness of the rewrite rules, equivalence of the replaced term and its replacement must be shown. By Definition 2.18, the formula $[n;\pi]$ holds in I if every trace beginning in (I,n) is successful.

The basic argument is the same for all cases: We reduce the case that all finite traces starting in (I,n) must be successful to the case that all finite traces from $(I',n') \in R_\pi(I,n)$ are successful and encode the knowledge on I' either into an update, an implication or conjunction. The state successor relation R_π is identical for assert and assume statements (as defined in Definition 2.17), but their semantics differ due to the definition of successful traces.

We exemplarily show havoc, assume and assert and leave the remainder as an easy exercise.

havoc: For $\pi[n] = \mathsf{havoc}\ c$ we have $(*)$ $R_\pi(n) = \{(I[c \mapsto d], n+1) \mid d \in \mathcal{D}^{\mathrm{ty}(c)}\}$ and can write

$$\begin{aligned}
& I,\tau,\beta \models [n;\pi] \\
\iff & \text{every trace beginning in } (I,n) \text{ is successful} \\
\overset{(*)}{\iff} & \text{for any } d \in \mathcal{D}^{\mathrm{ty}(c)} \text{ every trace beginning in } (I[c \mapsto d], n+1) \text{ is successful} \\
\iff & \text{for any } d \in \mathcal{D}^{\mathrm{ty}(c)}\text{: } I[c \mapsto d],\tau,\beta \models [n+1,\pi] \\
\overset{(\dagger)}{\iff} & \text{for any } d \in \mathcal{D}^{\mathrm{ty}(c)}\text{: } I[c \mapsto d],\tau,\beta[x \mapsto d] \models [n+1,\pi] \\
\overset{\text{(Def. 2.16)}}{\iff} & \text{for any } d \in \mathcal{D}^{\mathrm{ty}(c)}\text{: } I,\tau,\beta[x \mapsto d] \models \{c := x\}[n+1;\pi] \\
\iff & I,\tau,\beta \models (\forall x.\{c := x\}[n+1;\pi])
\end{aligned}$$

$(\dagger)$ Variable x does not occur in π.

assume: Let $\pi[n] = \mathsf{assume}\ \varphi$. If $I,\tau,\beta \not\models \varphi$, then $R_\pi(I,n) = \emptyset$ and the only trace beginning in (I,n) ends in this assume statement and *is* successful. If $I,\tau,\beta \models \varphi$, the truth value depends entirely on the traces starting in $(I,n+1)$, therefore, on $[n+1;\pi]$.

$$\begin{aligned}
&I,\tau,\beta \models [n;\pi]\\
\iff &\text{every trace beginning in } (I,n) \text{ is successful}\\
\iff &I,\tau,\beta \not\models \varphi \text{ (assumption does not hold) or}\\
&I,\tau,\beta \models \varphi \text{ and every trace beginning in } (I,n{+}1) \text{ is successful (continues)}\\
\iff &I,\tau,\beta \not\models \varphi \quad\text{or}\quad \text{every trace beginning in } (I,n{+}1) \text{ is successful}\\
\iff &I,\tau,\beta \models \varphi \to [n+1;\pi]
\end{aligned}$$

assert: Let now $\pi[n] = \mathsf{assert}\ \psi$. If $I,\tau,\beta \not\models \psi$, the only trace beginning in (I,n) ends in an assert statement and, hence, fails. The other case depends again on the traces from $(I,n+1)$:

$$\begin{aligned}
&I,\tau,\beta \models [n;\pi]\\
\iff &\text{every trace beginning in } (I,n) \text{ is successful}\\
\iff &I,\tau,\beta \models \varphi \quad\text{and}\quad \text{every trace beginning in } (I,n+1) \text{ is successful}\\
\iff &I,\tau,\beta \models \varphi \land [n+1;\pi]
\end{aligned}$$

A trace $((I,n),(I',n'),\ldots)$ is finite if and only if the one starting in $((I',n'),\ldots)$ is. The above arguments can, hence, be adapted to the $[\![\cdot]\!]$-modality by replacing "is successful" by "is successful and finite". □

These rules are the basic blocks for the symbolic execution. For the application in a verification system, there are cases in which alternative inference rules are more efficient. Applying an alternative can always be simulated by a series of basic rules, but shorter proofs with fewer branches can be produced if they are considered rules in their own right.

For a havoc statement in the succedent, for instance, the introduction of a fresh variable and its Skolemisation $sk : \mathrm{ty}(p)$ can be immediately resolved by the rule

$$\pi[n] = \mathsf{havoc}\ p \implies \frac{\Gamma \vdash \{U \parallel p := sk\}[n+1;\pi],\Delta}{\Gamma \vdash \{U\}[n;\pi],\Delta}\,\text{havocAndSkolem}\ .$$

An assertion within the program which has been proved valid may still be helpful later during the proof. It is therefore wise to provide the asserted condition as an additional assumption to the use case sequent.

$$\pi[n]{=}\mathsf{assert}\ \varphi \implies \frac{\Gamma \vdash \{U\}\varphi,\Delta \qquad \Gamma,\{U\}\varphi \vdash \{U\}[n{+}1;\pi],\Delta}{\Gamma \vdash \{U\}[n;\pi],\Delta}\,\text{assumeAssertion}$$

This rule is correct due to the propositional tautology $A \land B \equiv A \land (A \to B)$ which is sometimes referred to as the *local lemma* property since A may be used to show B. In practice, it showed that although it enlarges the sequent, the additional knowledge is often helpful to establish the remaining assertions of a program.

Another important special case rule is the treatment of deterministic case distinctions in the antecedent. Assume the program

$$\pi = (\mathsf{goto}\ 1{,}3 + |\alpha|; \mathsf{assume}\ \varphi; \alpha; \mathsf{end}; \mathsf{assume}\ \neg\varphi; \beta) \mathrel{\widehat{=}} \langle \mathbf{if}\ \varphi\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta \rangle$$

which performs a deterministic case distinction and either executes program α if φ holds, or program β otherwise. If $[0;\pi]$ appears on the succedent side of a proof obligation, the symbolic execution with the basic rules will decompose it as follows:

$$\frac{\dfrac{\varphi \vdash [\alpha] \qquad \vdash \varphi, [\beta]}{\vdash (\varphi \to [\alpha]) \land (\neg\varphi \to [\beta])}}{\vdash [0;\pi]} \tag{3.8}$$

in which $[\alpha]$ and $[\beta]$ abbreviate $[2;\pi]$ and $[4 + |\alpha|;\pi]$. If $[0;\pi]$ appears on the left hand side of the sequent, the resulting symbolic execution proof tree

$$\frac{\dfrac{\dfrac{}{\varphi \vdash \varphi} \qquad [\beta] \vdash \varphi \qquad \varphi, [\alpha] \vdash \qquad [\alpha], [\beta] \vdash}{(\varphi \to [\alpha]),\ (\neg\varphi \to [\beta]) \vdash} \cdots}{[0;\pi] \vdash}$$

has four instead of only two branches. One of the branches can be closed more trivially; on it, φ occurs both in the antecedent and succedent. The two branches $[\beta] \vdash \varphi$ and $\varphi, [\alpha] \vdash$ are the two branches, one would have expected from the execution of an if-statement. The fourth unexpected branch now makes no reference to the branching condition at all and only contains the two program formulas.

The symbolic execution opens more proof goals in this case than needed. This situation is not satisfactory for the symbolic execution of programs in the antecedent and, therefore, the rule

$$\left.\begin{array}{rcl} \pi[n] & = & \mathsf{goto}\ k_1{,}k_2 \\ \pi[k_1] & = & \mathsf{assume}\ \varphi \\ \pi[k_2] & = & \mathsf{assume}\ \neg\varphi \end{array}\right\} \Longrightarrow \frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}\varphi, \{\mathcal{U}\}[k_1 + 1;\pi] \vdash \Delta \\ \Gamma, \{\mathcal{U}\}[k_2 + 1;\pi] \vdash \{\mathcal{U}\}\varphi, \Delta\end{array}}{\Gamma, \{\mathcal{U}\}[n;\pi] \vdash \Delta}\text{BranchLeft}$$

is introduced which allows a branching behaviour in the antecedent congruent to the case in (3.8). This rule is sound as it exploits the following propositional equivalences: $[n;\pi] \equiv (\varphi \to [k_1 + 1;\pi]) \land (\neg\varphi \to [k_2 + 1;\pi]) \equiv (\varphi \land [k_1 + 1;\pi]) \lor (\neg\varphi \land [k_2 + 1;\pi])$. A direct consequence of this equivalence is that the premisses of the rule imply its conclusion (and vice versa).

3.3 Loop Invariants in *UDL*

The rewrite rules in Theorem 3.3 allow the symbolic execution of an unstructured program in a stepwise manner. If a program contains no loops, symbolic execution eventually results in a formula free of program formulas. However, as soon as the program flow allows a statement to be executed more than once during the run of a program, these rules can no longer remove program formulas entirely. A calculus for symbolic execution requires rules using loop invariants to resolve programs with loops. Such rules will, naturally, closely resemble invariant rules which are used to resolve loops in structured programs.

As mentioned earlier, the rules presented in this section will not be complete: If they are applied with too weak a loop invariant, the proof may be stuck in a dead end. We consider this as an acceptable gap since in deductive approaches the loop invariant is considered a kind of "user input" to the proof which we can confide in.

3.3.1 Informal Introduction

Dijkstra (1968) wrote in his seminal paper "Go to statement considered harmful":

> The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress.

This is precisely the problem that we have to face when devising inference rules for the treatment of loops in the unstructured programming language of *UDL*. This will lead to a relaxed notion of loop invariant which can be applied to any statement in an unstructured program, not only to a loop.

The invariant inference rules for *UDL* can be motivated by a comparison to the situation in classical structured dynamic logic. In structured DL, the sound inference rule

$$\frac{\varphi \vdash [\alpha]\varphi}{\varphi \vdash [\alpha^*]\varphi}$$

called the *loop invariant rule* or *induction axiom* is the base behind the treatment of the Kleene-star program construct. If the formula φ is preserved under the execution of α in all states, it is also preserved under finitely many repetitions α^* since each of the successive executions of α preserves φ. In the context of a while-language, this rule becomes the while-invariant rule

$$\frac{\psi, \varphi \vdash [\alpha]\varphi}{\varphi \vdash [\textbf{while } \psi \textbf{ do } \alpha \textbf{ end}](\neg\psi \wedge \varphi)}$$

in which φ plays the role of the loop invariant which, if initially valid, holds after the loop, if the loop body preserves it. The loop condition plays a special role as it may be assumed true before the loop body α and is ensured false after the loop.

In a sequent calculus proof obligating, the context sets Γ, Δ seldom contain the loop invariant. This is why typically the rule variant

$$\frac{\Gamma \vdash \varphi, \Delta \qquad \psi, \varphi \vdash [\alpha]\varphi \qquad \neg\psi, \varphi \vdash \sigma}{\Gamma \vdash [\mathbf{while}\ \psi\ \mathbf{do}\ \alpha\ \mathbf{end}]\sigma, \Delta} \tag{3.9}$$

with three premisses is used. The loop invariant φ is a parameter to this rule and can be chosen freely. It is introduced into the new proof obligations. The first premiss is called the *base case*, the second the *induction step* and the third the *use case*. This is very closely related to conducting an induction proof.

These rules have in common that we can extract the loop body α from a composed program (α^* or **while** ψ **do** α **end**). In unstructured programs of *UDL*, we lack the capability of such structural decomposition. In a goto-program the statements which are to be repeated may be spread all over the program text and cannot be extracted.

As discussed in Section 2.5.5, every unstructured program can be transformed into a structured one. But this transformation would be complex, and result in a larger and probably unintuitive result. The necessary loop invariants would be difficult to find and formulate.

Instead, we propose a rule operating on the statements of an unstructured program π. The idea behind it is to use a modified program π' into which additional statements have been inserted:

$$\frac{\Gamma \vdash \varphi, \Delta \qquad \varphi \vdash [n', \pi']}{\Gamma \vdash [n, \pi], \Delta}$$

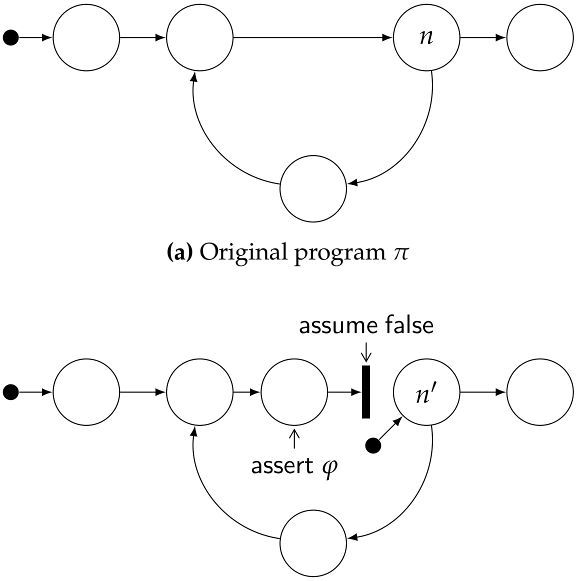
Program π' has its program flow "cut" at index n within a loop of statements, using an invariant φ to abstract from the program state at this point. Figure 3.3 informally shows how π' is derived from π. Nodes in this schematic control flow graph represent statements and edges represent the statement successor relationship (by walking into a statement or by explicit goto statements). In the original program (Fig. 3.3a), the program control flow has got a cycle and n is the index of a statement within this loop.

In the modified program (Fig. 3.3b), the general structure has remained, yet the cycle has been broken up. The "gap" introduced by this removal is "bridged" by the invariant $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$. We use φ as assumption to the modified program formulas $[n', \pi']$ and have to ensure that, whenever we reach this point in the program again, φ holds again. This check is encoded as an inserted statement assert φ. After the insertion, the abstraction has been proved correct and the symbolic execution may stop. This realises the "cut" in the program flow and is implemented by inserting the statement assume false quitting unconditionally.

This procedure does not require that the control flow of the program corresponds to one achieved by a structured loop. Also goto programs with arbitrary control-flow transfer or indeterministic forking can be subject to this rule, Figure 3.4 shows how the cycle can be broken in a general goto program.

There are even cases in which it may be advisable to utilise the splitting invariant rules even if there is *no* loop at all. The rules support the abstraction of the program at certain statements, and it can simplify the proof of a second part of a program significantly if the first part is not given by all its formulas but only by a user-defined abstraction.

In the remainder of the section, we will present three versions of the invariant rule extending one another. First, we give the simple version of an invariant rule. Then, a rule involving termination is defined and, finally, a rule which preserves more context information. The latter two can canonically be combined to a rule with termination and context preservation. Section 3.3.6 completes the collection of invariant rules by stating inference rules for the case that program formulas occur in a sequent's antecedent.



(a) Original program π

(b) Modified program π'

Figure 3.3: Informal description of the modification of *UDL* programs

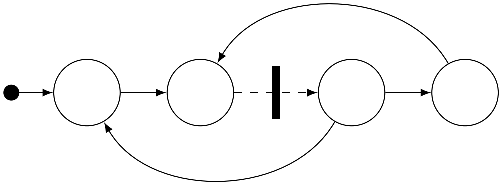


Figure 3.4: General goto programs (not induced by loops) can be treated as well

3.3.2 Program Modifications

In structured dynamic logic, the invariant rule introduces new proof goals on the loop body, that is, on a strict subprogram of the original code. We are not able to decompose a program's code to a subset of statements in *UDL* since no restriction is imposed on the targets of goto statements and any statement, also outside the loop body, may be addressed.

We need, however, a means to reduce the number of loop body iterations to one. This is achieved by inserting new statements into the program under inspection. There is a technical issue, however, since index changes may make goto statements point to wrong targets afterwards. To compensate for this effect, we introduce an offset correction function off^k_m which increments the target indices by k if they lie after the insertion point m.

$$\mathit{off}^k_m(a) = \begin{cases} a & \text{if } a \leq m \\ a + k & \text{otherwise} \end{cases}$$

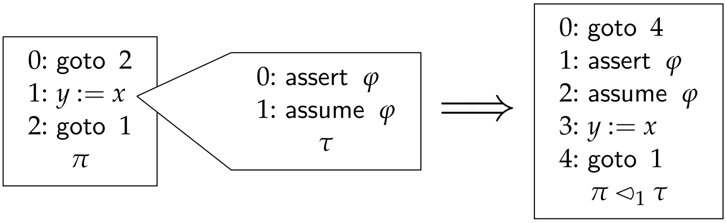
We also apply off^k_m to statements. Here, it operates only on the target indices of goto statements and behaves like the identity function on all other statements.

Definition 3.3 (Statement injection) *For programs $\pi, \tau \in \Pi$ and an arbitrary index $m \in \mathbb{N}$, the insertion $\pi \lhd_m \tau \in \Pi$ of τ into π at position m has length $|\pi| + |\tau|$ and is defined as*

$$(\pi \lhd_m \tau)[i] = \begin{cases} \mathit{off}^{|\tau|}_m(\pi[i]) & \text{for } i < m \\ \tau[i-m] & \text{for } m \leq i < m + |\tau| \\ \mathit{off}^{|\tau|}_m(\pi[i-|\tau|]) & \text{for } m + |\tau| \leq i \end{cases} .$$

The infix operator is left-associative.

τ is not subject to an offset correction since the programs we use for insertion here will not contain goto-statements.



```
0: goto 2
1: y := x
2: goto 1
π
```

```
0: assert φ
1: assume φ
τ
```

```
0: goto 4
1: assert φ
2: assume φ
3: y := x
4: goto 1
π ◁1 τ
```

Figure 3.5: Example of the program insertion $\pi \lhd_1 \tau$

Figure 3.5 shows a sample program insertion. The program $\tau =$(assert φ; assume φ) is inserted into the program $\pi =$ (goto 2; $y := x$; goto 1) at position 1. Please note that in statement 4 : goto 1 of the resulting program, the target has *not* been incremented and still refers to the insertion point even though the statement to which it points has been changed.

Due to the index adaption off^k_m, a trace for π which does not pass through the insertion point m induces a trace for the program after the insertion (of course with possibly adapted statement indices). The only way to enter the inserted statement sequence is to reach statement m, either as a goto target or by "stepping" into it. The sequence may start off in m, however.

The upcoming observations will not talk about traces but about *partial traces* which are state sequences like traces, but without the requirement that the last state has no successor. Every subsequence of a trace is a partial trace.

Observation 3.4 (Trace correspondence)
For any partial trace $(I_0,m),(I_1,k_1),\ldots,(I_r,k_r)$ for π with $r \in \mathbb{N}$, $k_i \neq m$ for $0 < i < r$, there is a partial trace $(I_0,m+|\tau|),(I_1,k'_1),\ldots,(I_r,k'_r)$ for $\pi \lhd_m \tau$. In particular, if $k_r = m$, then $k'_r = m$ can be chosen.

For any infinite trace $(I_0,m),(I_1,k_1),\ldots$ for π with $k_i \neq m$ for $i > 0$, there is an infinite trace $(I_0,m+|\tau|),(I_1,k'_1),\ldots$ for $\pi \lhd_m \tau$.

Conversely, a trace in a modified program which does not step into the inserted statements via the entry point m also induces a trace in the unmodified program:

Observation 3.5 (Trace correspondence)
For any finite partial trace $(I_0,m+|\tau|),(I_1,k_1),\ldots,(I_r,k_r)$ for $\pi \lhd_m \tau$ with $k_i \neq m$ for $0 < i < r$, there is a state sequence $(I_0,m),(I_1,k'_1),\ldots,(I_r,k'_r)$ for π. In particular, if $k_r = m$, then $k'_r = m$ can be chosen.

For any infinite trace $(I_0,m+|\tau|),(I_1,k_1),\ldots$ for $\pi \lhd_m \tau$ with $k_i \neq m$ for $0 < i$, there is an infinite trace $(I_0,m),(I_1,k'_1),\ldots$ for π.

Updates can be resolved completely from *UDL* formulas without program modalities using the rewrite rules of Theorem 3.2. However, an update which precedes a program modality cannot in general be resolved. Symbolic execution may also introduce new updates before the program formula through applications of (3.6), (3.7). This is why all invariant rules match against a formula $\{\mathcal{U}\}[n;\pi]$.

One problem that is not present in structured dynamic logic but with which we have to cope here, is the detection of loops. In classic dynamic logic, a loop can be identified syntactically as a statement initiated with the keyword "while". We do not have such landmarks in an unstructured program. A loop becomes a loop because of a goto statement targeting backward. Not every such statement, however, is necessarily an indicator for a loop. Therefore, we formulate our invariant rules in such a manner that they can be applied to *every* statement. Of course, the application is not equally expedient for all execution states, and it is the task of either a static analysis or the translation mechanism to identify (and to mark) the points at which an invariant rule should be applied.

3.3.3 Simple Invariant Rule

The general idea in the upcoming invariant rules is to change a program in such a way that a loop becomes dissected. At the beginning of the loop, an invariant is assumed which has to be asserted whenever the initial statement is reached again by symbolic execution. For that purpose we insert the statements $(\mathsf{assert}\ \psi; \mathsf{assume\ \ false})$ at the current position.

Theorem 3.6 (Simple invariant rule) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash [n+2; \rho_1]}{\Gamma \vdash \{\mathcal{U}\}[n; \pi], \Delta}\text{Invariant}$$

with $\rho_1 = \pi \lhd_n (\mathsf{assert}\ \psi; \mathsf{assume\ false})$ *is a sound rule for any formula* ψ*.*

This rule has two premisses: The first provides evidence that the invariant ψ holds initially when arriving in the current state. The second premiss requires that in a state in which the invariant holds, the execution of the changed program is successful. Please note that the antecedent and succedent contexts Γ and Δ are not present in the second premiss. We will address this issue in Theorem 3.8.

This rule is similar to the invariant rule (3.9) for a dynamic logic for a simple 'while'-language. One difference is that, here, we have *two* rather than *three* premisses to establish. This is due to the fact that multiple assertions are embedded into the program ρ_1 and the second premiss $[n+2; \rho_1]$ plays two roles: It proves the absence of assertion violations after the loop (the 'use case' of ψ), and it ensures that the loop body preserves ψ establishing it as an invariant.

PROOF We can without loss of generality assume that $\Delta = \emptyset$. For an arbitrary interpretation[5] I, we need to show that $I \models \bigwedge\Gamma \to \{\mathcal{U}\}[n; \pi]$. If $I \not\models \bigwedge\Gamma$, the proof is completed. Thus, let $I \models \bigwedge\Gamma$. It remains to be shown that $I \models \{\mathcal{U}\}[n; \pi]$. Setting $I_0 := I^{\mathcal{U}}$ (the result of applying the assignments in U to I) yields the obligation $I_0 \models [n; \pi]$.

Let us first look at an arbitrary finite trace of π starting in (I_0, n). We can divide this trace into "loops from n to n", that is, split the trace into r subsequences such that every occurrence of n starts a new partial trace. For any $0 \le i < r$, the state (I_{k_i}, n) initiates a partial trace. The last trace ends in state (I_{k_r}, s_{k_r}) with s_{k_r} (the final state of the trace) not necessarily equal to n. See Fig. 3.6 for an illustration.

We claim that for every first state (I_{k_i}, n) of a partial trace, $I_{k_i} \models \psi$ holds and show this by induction on $0 \le i < r$. For $I_{k_0} (= I_0 = I^{\mathcal{U}})$, this is a consequence of the validity of the first premiss. Let us assume then that $I_{k_i} \models \psi$ for some $0 \le i < r-1$. For the partial trace $(I_{k_i}, n), \ldots, (I_{k_{i+1}}, s_{k_{i+1}})$, only the first and the last state visit statement n: The sequence hence matches the requirements of Observation 3.4, and, thus, we know

[5] For the sake of readability, we leave variable assignments aside in this section. Since formulas on the sequent are assumed to have no free variables, this does not influence the evaluation.

that $(I_{k_i}, n+2), \dots, (I_{k_{i+1}}, n)$ is a trace for program ρ_1. Furthermore, $\rho_1[n] = \mathsf{assert}\ \psi$ and no trace for ρ_1 fails by the second premiss. This implies that (1) $I_{k_{i+1}} \models \psi$ and (2) that the partial trace does not fail.

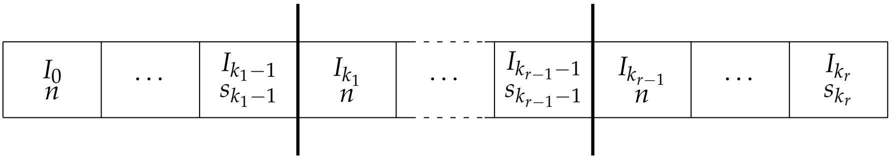


Figure 3.6: Subdividing a trace into partial traces

We have seen that every partial trace begins in an interpretation in which ψ holds. In particular, we have $I_{k_{r-1}} \models \psi$. The last partial trace $(I_{k_{r-1}}, n), \dots, (I_{k_r}, s_{k_r})$ is a trace and statement n does not appear after the first state of this trace. We can therefore apply Observation 3.4 again and obtain a trace $(I_{k_{r-1}}, n+2), \dots, (I_{k_r}, s'_{k_r})$. Again due to the second premiss, this trace must be successful, implying that the entire trace is successful.

The argument is the same for an arbitrary infinite trace for π starting from (I_0, n). If there are finitely many loop iterations from n to n, the last partial trace is infinite, but cannot fail due to the second premiss. If there are infinitely many loop iterations from n to n, the inductive argument from above can be conducted for $\mathbb{N}$ rather than only up to r. □

3.3.4 Invariant Rule with Termination

Theorem 3.6 is not sufficient if we want to incorporate the question of termination into the verification process. The rule for the total modality $[\![\cdot]\!]$ introduces a variant term whose value must strictly decrease from iteration to iteration. We therefore assume there is a binary predicate symbol $\prec : \alpha \times \alpha \to \mathsf{bool} \in \mathsf{Fct}$ whose interpretation is a well-founded relation. With the aid of this predicate symbol, we can formulate an invariant rule which includes termination.

Theoretically, it would suffice to use natural numbers as type for variants, but it proves very valuable, in practice, to allow for more general notions of variants. Typically supported variant predicates are comparison by $<$ on natural numbers, comparison by $\subseteq$ on finite sets and lexicographic ordering on finite sequences or tuples.

Theorem 3.7 (Invariant rule with termination) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi \vdash \{nc := v\}[\![n+2; \rho_2]\!]}{\Gamma \vdash \{\mathcal{U}\}[\![n; \pi]\!], \Delta}\,\text{InvariantTermination}$$

with $\rho_2 = \pi \lhd_n (\textsf{assert } \psi \wedge \nu \prec nc; \textsf{assume false})$ *is a sound rule for a program variable* $nc \in \mathrm{PVar}$ *which does not occur in the conclusion, any formula* $\psi \in \mathrm{Trm}^{\mathsf{bool}}$ *and any term* $\nu \in \mathrm{Trm}^{\mathrm{ty}(nc)}$.

Proof Partial correctness $[n; \pi]$ is a direct consequence of Theorem 3.6 since we made the program modification *stronger* requiring $\psi \wedge var \prec nc$ to hold instead of only ψ.

Like in the proof above, we fix an interpretation I with $I \models \bigwedge\Gamma$ and set $I_0 := I^{\mathcal{U}}$. It remains to be shown that there is no infinite trace for π starting in (I_0, n). Assuming there is such an infinite trace, we could subdivide it into partial traces such that every occurrence of the statement n initiates a new partial trace like in the previous proof. We can use the induction from the proof of Theorem 3.6 to establish that for every first state (I_{k_i}, n) of a partial trace we have $I_{k_i} \models \psi$.

In case there are finitely subtraces, the last partial trace $\big((I_{k_{r-1}}, n), \ldots\big)$ must be infinitely long and not passing through n. Since $I_{k_{r-1}} \models \psi$, there is a contradiction with the second premiss which forbids an infinite trace for ρ_2 starting in $(I_{k_{r-1}}, n+2)$ (because it uses the total modality operator).

In case of infinitely many subtraces, every partial trace is finite. For the first states of the partial traces, we define $v_i := \mathrm{val}_{I_{k_i}}(\nu)$. In a state (I_{k_i}, n) with $i > 0$, we know that (*) $I_{k_i} \models \nu \prec nc$ since this formula is part of the assertion. As nc does not occur elsewhere on the sequent and because of the semantics of the update $nc := \nu$, we get that nc holds the value of ν of the previous iteration, i.e. $I_{k_i}(nc) = v_{i-1}$. This and (*) imply that $(v_{i-1}, v_i) \in I(\prec)$. The sequence $(v_1, v_2, \ldots)$ would therefore be an infinitely descending chain for $I(\prec)$ which cannot be since $\prec$ is a well-founded relation. □

3.3.5 Improved Invariant Rule

The major disadvantage of the rules in Theorems 3.6 and 3.7 is that the information contained in Γ and Δ of the conclusion is not available in the second premiss. There invariant ψ is the only formula in the antecedent of the sequent. If any of the information encoded in $\Gamma \cup \Delta$ was needed to close the proof, it would have to be implied by ψ. This would usually mean that the description of ψ gets longer and that a proof that this information holds initially would be needed, though it holds by construction.

We will provide an invariant rule which keeps the context Γ and Δ but subjects those program variables which are touched during a loop iteration to a generalisation. We can use the havoc statement to do this generalisation because of (3.7) in Theorem 3.3.

The rule follows the ideas of Beckert et al. (2005) where a context preserving invariant rule is defined for a structured dynamic logic. The advantage is that more information on the sequent remains available and does not need to be encoded in the invariant.

Definition 3.4 (loop-reachable) *A statement index* m *is called* loop-reachable *from* n *within a program* π *if there is a trace* $(I_0, k_0), (I_1, k_1), \ldots$ *such that*

1. *$k_o = n$,*
2. *there is an index $r \geq 1$ with $k_r = m$, and*
3. *there is an index $s > r$ with $k_s = n$.*

We write $reach(n,m,\pi)$ in this case.

We use the notion of reachability to define the set of possibly modified program variables as

$$mod(n,\pi) := \left\{ c \;\middle|\; \begin{array}{l} \text{there are } m,c \text{ and } t \text{ s.t. } \ reach(n,m,\pi) \text{ and} \\ (\pi[m] = \mathsf{havoc}\ \ c \text{ or } \pi[m] = c := t) \end{array} \right\} \subseteq \mathrm{PVar} \ .$$

Loop reachability can, in general, not be computed. The reachability of a statement may depend on the satisfiability of an assumption statement earlier in the execution path and this is undecidable. However, a static analysis can be used to over-approximate $mod(n,\pi)$ if it assumes that every assumption might succeed.

The modification of the program for the third invariant rule is also more complex. In addition to the statement injected in Theorem 3.6, statements need to be added to anonymise the values of those program variables that are possibly changed by the execution of the loop body. This anonymisation is performed using a sequence of havoc statements. The second premiss can thus be relaxed and contains the contextual information of Γ, Δ and $\mathcal{U}$.

Theorem 3.8 (Context-preserving invariant rule) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi,\Delta \qquad \Gamma \vdash \{\mathcal{U}\}[n+2;\rho_3],\Delta}{\Gamma \vdash \{\mathcal{U}\}[n;\pi],\Delta}\,\mathrm{InvariantContext}$$

with $\rho_3 = \pi \lhd_n (\mathsf{assert}\ \psi; \mathsf{assume\ false}; \mathsf{havoc}\ r_1; \ldots; \mathsf{havoc}\ r_b; \mathsf{assume}\ \psi)$ is a sound rule for any formula ψ and any finite set $\{r_1,\ldots,r_b\}$ with $mod(n;\pi) \subseteq \{r_1,\ldots,r_b\} \subseteq \mathrm{PVar}$.

The assumption that ψ holds must also be moved from the antecedent of the second premiss to an assumption in the injected program since it is assumed to hold *after* the anonymisation of the program variables.

PROOF Again, let $\Delta = \emptyset$. We observe that the second premiss is (after a number of steps of symbolic execution and simplification) equivalent to

$$\Gamma \vdash \forall x_1.\ldots\forall x_b.\{\mathcal{U} \parallel r_1 := x_1 \parallel \ldots \parallel r_b := x_b\}(\psi \to [n+2+b+1;\rho_3]) \ . \tag{3.10}$$

The injected havoc and following assume statements are not reachable from $n+2+b+1$ in ρ_3 since they follow the statement assume false which ends every trace. Sequent (3.10) is hence equivalent to

$$\Gamma \vdash \forall x_1.\ldots\forall x_b.\{\mathcal{U} \parallel r_1 := x_1 \parallel \ldots \parallel r_b := x_b\}(\psi \to [n+2;\rho_1]) \ . \tag{3.11}$$

For an interpretation I with $I \models \bigwedge\Gamma$, we know, because of the validity of the premiss, that I makes the formula in (3.11) true. If an interpretation I' differs from $I^{\mathcal{U}}$ at most on the values of the program variables $r_1,\ldots,r_b$, then we have due to the semantics of the quantifier and the updates that also

$$I' \models (\psi \to [n+2;\rho_1]) \ .$$

For a trace for $[n;\pi]$ (cf. Fig. 3.6) started in $(I^{\mathcal{U}},n)$, we observe that every statement before $(I_{k_{r-1}},n)$ is loop-reachable from n. The program variables which are changed over this trace are, hence, in $mod(n,\pi)$ and, therefore, also among the $\{r_1,\ldots,r_b\}$. This implies that for all $0 \leq i < r$, the interpretation I_{k_i} coincides with $I^{\mathcal{U}}$ outside the program variables $r_1,\ldots,r_b$, and we obtain $I_{k_i} \models (\psi \to [n+2;\rho_1])$ and, hence, $I_{k_i} \models [n+2;\rho_1]$ by induction from the proof of Theorem 3.6.

In particular we have $I_{k_{r-1}} \models [n+2;\rho_1]$ for which we saw in the proof of Theorem 3.6 that it implies that the entire trace is successful. □

Bringing together the results of Theorem 3.7 and 3.8 yields

Theorem 3.9 (Context-preserving invariant rule with termination) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi,\Delta \qquad \Gamma \vdash \{\mathcal{U}\}[\![n+2;\rho_4]\!],\Delta}{\Gamma \vdash \{\mathcal{U}\}[\![n;\pi]\!],\Delta}\,\text{InvariantTerminationContext}$$

with

$$\tau = (\mathsf{assert}\ \psi \wedge var \prec nc;\ \mathsf{assume\ false};\ \mathsf{havoc}\ r_1;\ldots;\ \mathsf{havoc}\ r_b;\ \mathsf{assume}\ \psi;\ nc := var)$$
$$\rho_4 = \pi \lhd_n \tau$$

is a sound rule for any formula ψ and any finite set $\{r_1,\ldots,r_b\}$ of program variables with $mod(n;\pi) \subseteq \{r_1,\ldots,r_b\} \subseteq$ PVar.

There is one difference to the rule of Theorem 3.7 regarding the second premiss. While rule InvariantTermination uses an update term to store the value of the variant into program variable *nc*, rule InvariantTerminationContext uses an assignment in the inserted statements of ρ_4. The reason for this is that the assignment must happen *after* forgetting the values of the modified program variables, after the havoc statements. As an update cannot be placed within statements, it has been replaced by a semantically equal assignment statement.

3.3.6 Antecedent Invariant Rules

In the last sections we have considered four inference rules to deal with program formulas in the succedent of a sequent. This is by far the most common use of invariant rules in program verification. However, to have a complete calculus, program formulas must also be considered if they appear in the antecedent of a sequent.

The necessity for the rules becomes also more evident, if postfixed assertions (see Section 2.5.3) are taken into consideration. The formula $\langle\pi\rangle\varphi$ is equivalent to $\neg[\pi]\neg\varphi$ due to the duality of (2.21) and would hence, if reduced to the non-postfix version of the logic, not appear in the succedent but in the antecedent of a proof obligation.

Theorem 3.10 (Antecedent invariant rule) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi,\ [\![n+2;\rho_5]\!] \vdash}{\Gamma, \{\mathcal{U}\}[\![n;\pi]\!] \vdash \Delta}\text{AntecedentInvariant}$$

with $\rho_5 = \pi \lhd_n (\mathsf{assert}\ \neg\psi; \mathsf{assume\ false})$ *is a sound inference rule for any formula* ψ.

PROOF As before, let $\Delta = \emptyset$ and an interpretation I with $I \models \bigwedge\Gamma$ be given. The conclusion is true if and only if $I^{\mathcal{U}} \models \neg[\![n;\pi]\!]$, that is, if there is a failing or infinite trace for π starting in $(I^{\mathcal{U}}, n)$.

We iteratively construct such a trace by augmenting an intermediate partial trace $(I^{\mathcal{U}}, n), \ldots, (J_i, n)$ for π. As an invariant it is ensured that $J_i \models \psi$ for its last state. For the start, we use the singleton state sequence (J_0, n) with $J_0 := I^{\mathcal{U}}$ of which we know that $I^{\mathcal{U}} \models \psi$ by the first premiss.

Let (J_i, n) with $J_i \models \psi$ now be the last state of the intermediate partial trace. By the second premiss we know that there is a trace of ρ_5 starting in $(J_1, n+2)$ which either diverges or fails:

1. If it fails in $\rho_5[n]$, the trace $(J_i, n+2), \ldots, (J_{i+1}, n)$ of ρ_5 meets the requirements of Obs. 3.5 and there is a state sequence of π from (J_i, n) ending in (J_{i+1}, n). Due to the inserted statement $\mathsf{assert}\ \neg\psi$ at which the trace failed, we have that $J_{i+1} \not\models \neg\psi$, that is, $J_{i+1} \models \psi$ and we append this trace to the intermediate partial result.

2. If it fails elsewhere, there is an according trace of π from (J, n) (again by Obs. 3.5) failing at an corresponding index. Appending this failing trace to the intermediate partial trace gives a failing trace for π starting in $(I^{\mathcal{U}}, n)$.

3. If it is infinite, it never visits the inserted program (which would always terminate) and Obs. 3.5 can be applied. The corresponding infinite trace of π can be appended to the intermediate result which is then an infinite trace for π starting in $(I^{\mathcal{U}}, n)$.

This iteration is repeated either infinitely often or until either the second or third option is applicable. In either case, a failing or infinite trace of π starting in $(I^{\mathcal{U}}, n)$ has been constructed. □

When occurring in the antecedent, the modalities swap their roles with respect to the issue of termination. While in the rules for the succedent side in the last section, total program formulas $[\![n;\pi]\!]$ required a termination argument using a variant expression, this is the case for program formulas $[n;\pi]$ in the antecedent.

Theorem 3.11 (Antecedent invariant rule with termination) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \psi, \{nc := v\}[n+2; \rho_6] \vdash}{\Gamma, \{\mathcal{U}\}[n; \pi] \vdash \Delta}\ \text{AntecedentInvariantTermination}$$

with $\rho_6 = \pi \lhd_n (\mathsf{assert}\ \neg(\psi \wedge v \prec nc); \mathsf{assume\ false})$ *is a sound inference rule for a program variable nc which does not yet appear elsewhere in the conclusion, any formula* $\psi \in \mathrm{Trm}^{\mathsf{bool}}$ *and any term* $v \in \mathrm{Trm}^{\mathsf{ty}(nc)}$.

PROOF As before, let $\Delta = \emptyset$ and I with $I \models \bigwedge\Gamma$ be given. The conclusion holds then if and only if $I^{\mathcal{U}} \models \neg[n; \pi]$, that is if there is a failing trace. Note that the presence of an infinite trace no longer makes this sequent true.

The argument is essentially the same as for the last theorem. However, the case that the resulting witnessing trace is infinite must be excluded. Since the second premiss requires the non-terminating program formula $[n+2; \rho_6]$ unlike $[\![n+2; \rho_5]\!]$ in Theorem 3.10, the third option (an infinite trace for ρ_6) of the last proof cannot arise here.

It remains to be shown that the iterative construction of the result eventually leads to a failing state. This is done using a variant expression like in the proof for Theorem 3.7.

The finite partial trace is composed of loop iterations from (J_i, n) to (J_{i+1}, n). As we have seen in the last proof, it is an invariant that all J_i fail the assertion $\neg(\psi \wedge v < nc)$. Hence, we have that $J_{i+1} \models v \prec nc$, that is, $\mathrm{val}_{J_{i+1}}(v), \mathrm{val}_{J_i}(v) \in I(\prec)$. The intermediate evaluations $v_i := \mathrm{val}_{J_i}(v)$ of the variant expression hence form a strictly descending chain under $\prec$. Since $\prec$ is well-founded, there cannot be an infinite chain. The resulting trace must be finite. □

For the succedent side of the sequent, we have defined rules which preserve the context Γ, Δ and $\mathcal{U}$ of the original sequent. To parallel this for the other side of the sequent, we state the corresponding rules here. The translation of the rules is not as canonical as one might think, however. In Theorem 3.8, havoc statements were introduced to anonymise the relevant program variables. Since the modality appears now with the complementary polarity on the sequent, the universal quantification implied by havoc has not the indented semantics. There is no statement which possesses the semantics dual to that of the havoc statement. Instead, an anonymising update $\mathcal{V}$ is introduced which assigns fresh Skolem constants to the program variables.

Theorem 3.12 (Context-preserving antecedent invariant rules) *The rule*

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \Gamma,\ \{\mathcal{U}\}\{\mathcal{V}\}(\psi \wedge [\![n+2; \rho_5]\!]) \vdash \Delta}{\Gamma, \{\mathcal{U}\}[\![n; \pi]\!] \vdash \Delta}\ \text{AntecedentInvariantContext}$$

and the rule AntecedentInvariantContextTermination

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi, \Delta \qquad \Gamma,\ \{\mathcal{U}\}\{\mathcal{V}\}(\psi \wedge \{nc := v\}[n+2; \rho_6]) \vdash \Delta}{\Gamma, \{\mathcal{U}\}[n; \pi] \vdash \Delta}$$

with ρ_5 like in Theorem 3.10, ρ_6, v and nc like in Theorem 3.11, are sound inference rules for any formula ψ and any finite set $\{r_1, \ldots, r_b\}$ with $mod(n;\pi) \subseteq \{r_1, \ldots, r_n\} \subseteq \mathrm{PVar}$. The anonymising update is $\{\mathcal{V}\} = \{r_1 := r'_1 \parallel \ldots \parallel r_b := r'_b\}$ for fresh constant symbols $r'_i : \mathrm{ty}(r_i)$ not occurring in the conclusion ($1 \leq i \leq b$).

We omit the proofs for the soundness of these rules since they are simple adaptions of the proofs for Theorem 3.10 and 3.11 with the idea illustrated for Theorem 3.8. The fresh Skolem constants introduced for the modified program variables $r_1, \ldots, r_b$ ensure that the second premiss holds for all possible values for these variables. Since intermediate states differ from the starting state at most in these program variables, the second premiss covers hence all necessary cases.

3.4 Completeness

Harel et al. (2000, Theorem 75) state that there is a sound and relatively[6] complete calculus for structured first order logic if the underlying logic is sufficiently expressive. Since dynamic logic is more expressive than first order logic, the completeness notion employed in this theorem can only be of a weaker kind, that is, modulo the natural numbers: Given an oracle to decide the validity of any formula over the naturals, the validity of every valid dynamic logic formula can be proved.

Since DL and *UDL* are equally expressive (see Section 2.5.5), it is clear that a relatively complete calculus exists also for *UDL*. We show that the rules presented in this chapter constitute a relatively complete calculus if we leave type quantifiers aside. This means that the presented calculus allows the reduction of any proof obligation containing program formulas to one without such terms. Given an oracle for the underlying logic, any formula could be decided, hence.

3.4.1 Completeness for First Order Formulas

The correctness and completeness of the first order sequent calculus with equality is well-known; it has, for instance, been shown by Degtyarev and Voronkov (2001). The rules presented in Tables 3.1 and 3.2 summarise such a complete calculus. However, the first order logic we employ has some idiosyncrasies which deserve being looked at in the light of completeness of the calculus.

Binder symbols We can use the reduction to first order logic without binder symbols used in the proof of Theorem 2.4 to show that the calculus is complete for binder symbols.

[6] Harel uses the notion of arithmetical completeness

In this reduction, every[7] binder term $(b\ v.t)$ is replaced using a fresh function symbol $f_{(b v.t)}$. To reflect the semantic properties of binder symbols, the replacement function symbols are axiomatised by the set (see also (2.12))

$$M^\dagger = \{(\forall \bar{v}.\forall \bar{w}.(\forall z.t^z \doteq u^z) \rightarrow f_{(b x.t)}(\bar{v}) \doteq f_{(b y.u)}(\bar{w})) \mid t, u \in \text{Trm, binder-free}\} .$$

Let S be a valid sequent with binder symbols and $S^\dagger$ denote the equivalid translation without binder symbols. As the calculus without binders is complete, a closed proof tree can be constructed for $S^\dagger$ if it is valid. In the closed proof tree, the reduction can afterwards be undone again yielding a closed proof tree for the sequent S with binder symbols.

However, during the proof of the reduction, formulas from $M^\dagger$ could have been incorporated onto the sequent. Undoing the reduction yields a sequent proof in which the schematic inference rule

$$\frac{\Gamma, (\forall \bar{v}.(\forall z.t^z \doteq u^z) \rightarrow (b x.t) \doteq (b y.u)) \vdash \Delta}{\Gamma \vdash \Delta} \tag{3.12}$$

might have been used. The vector $\bar{v}$ denotes the free variables in t and u. In this rule, t^z and u^z emerge from t and u by replacing the bound variable (x and y respectively) by a common unused variable z. It is obvious that a cut over the introduced formula can mimic an application of (3.12). Yet, this opens a second branch with the cut formula in the succedent. Such remaining open goals can be discharged using predicate logic rules (allRight, impRight), alpha renaming (binderAlpha) and binderExt. Thus a closed sequent proof tree can be found for every valid formula with binders.

This completeness result is only valid if all function and binder symbols are uninterpreted. As soon as symbols have a fixed semantics and new rules are introduced to reflect their meaning, the completeness must be reconsidered.

Boolean terms Traditionally, formulas and terms are different syntactical entities and the calculus is defined accordingly. In *UDL* the distinction between formulas and terms of type bool has been deliberately dropped for the sake of a greater freedom in expressions.

This has as an effect that boolean operators need not appear as toplevel function symbols on the sequent but may also occur embedded into an argument to a function symbol. But the presented propositional rules of the sequent calculus are only applicable if formulas are not embedded into function applications.

However, this relaxation does not increase the expressiveness of the logic as the following short argument shows. Every atomic formula $\varphi(\psi)$ in which formula ψ occurs as parameter to a function symbol which is not a boolean junctor can be equivalently rewritten as

$$\varphi(\psi) \equiv \text{if } \psi \text{ then } \varphi(\mathsf{true}) \text{ else } \varphi(\mathsf{false}) \equiv (\psi \rightarrow \varphi(\mathsf{true})) \wedge (\neg\psi \rightarrow \varphi(\mathsf{false})) .$$

[7] We will, again, restrict ourselves to unary binder symbols here. The generalisation to symbols of higher arity is obvious.

By such rewriting, every formula can be rephrased as one in which every boolean argument to a function is one of the constants true and false. Their being different ($\mathsf{true} \not\doteq \mathsf{false}$) and tertium non datur ($\forall b^{\mathsf{bool}}.b \doteq \mathsf{true} \lor b \doteq \mathsf{false}$) can be shown in the calculus.

These steps can be applied within the calculus, too. Let $\Gamma \vdash p(\psi), \Delta$ be a sequent (without binder symbols) in which the formula γ appears as argument to the function $p : \mathsf{bool} \to \mathsf{bool}$. The available rules can be used to split the above sequent equivalently into the two sequents $\Gamma, \psi \vdash p(\mathsf{true}), \Delta$ and $\Gamma \vdash \psi, p(\mathsf{false}), \Delta$.

This leaves the question of quantifiers. The above direct case distinction approach can only be applied for toplevel ground formulas $p(\psi)$. Fortunately, this is already sufficient. A proof in sequent calculus can always be conducted on a set of ground formulas derived from the formulas on the sequent. The reason for this is Herbrand's theorem which says: For any valid sequent S there is a valid sequent S_H which emerges from S by applying quantifier rules followed by removal of the quantified formulas. If S is in Skolem normal form, then S_H consists of instantiations of the formulas on the sequent of S. Sequent S_H is called a *Herbrand sequent* (Hetzl et al., 2008) of S. Case distinctions on the ground terms of S_H can remove all formulas from embedded applications.

Weak equality and type quantification No completeness can be achieved in the presence of parametrised types and type quantification. It has been pointed out in Section 2.3.3 that these features extend the expressiveness beyond that of first order logic. The first order induction scheme for the type system has been added to provide as much completeness as possible.

The weakly typed equality suffers a similar fate. The validity of a weak equality depends in the end also on the extension of the type system. For instance, if there is only one single non-parametric type, the weak equality $t \mathrel{\dot{\approx}} u$ is equivalent to $\sigma(t) \doteq \sigma(u)$ for any unifying type substitution σ. Rules which resolve the weakly typed equality to strong equality have been added for the cases in which it is possible.

Updates The rewriting rules for updates presented in Theorem 3.2 are complete in the sense that every formula can be brought into update normal form. Naming conflicts impeding update resolution can be averted by alpha renaming of bound variables using rule binderAlpha.

3.4.2 Completeness for Program Formulas

The completeness of the program calculus can only be shown for structures which contain a faithful copy of the natural numbers. For this section we therefore assume that the type system is $\Gamma = \{\mathsf{bool}, \mathsf{nat}\}$ and that we only regard interpretations with $\mathcal{D}^{\mathsf{nat}} = \mathbb{N}, \mathcal{D}^{\mathsf{bool}} = \{\mathit{ff}, \mathit{tt}\}$. We write $\models_{\mathbb{N}}$ to denote this special evaluation. Let us for this section also (without loss of generality) assume that the terms occurring in programs have no free variables.

The completeness proof for the calculus will consist of two results. First, we will adapt the result of Cook (1978) to *UDL* to show that the arithmetic structure is expressive for *UDL*. Then, we will use this result to specify the most specific loop invariant and variant for the rules of Section 3.3. In total we show that the presented calculus can be used to reduce any sequent containing program formulas to a program-free equivalid sequent of the underlying logic.

We begin with the observation that one step of symbolic execution of a program π can be expressed in a before-after-after-predicate χ_π. This predicate formalises the program execution function $R_\pi : \mathcal{S}_{\Sigma,\mathcal{D}} \to 2^{\mathcal{S}_{\Sigma,\mathcal{D}}}$ used in Definition 2.17 to define the semantics of the programming language. Often in the following, an enumeration of all variables standing for the program variables will be needed. We will use vector notation like $\bar{x}$ to denote the finite list of variables $x_{p_1}, \ldots, x_{p_{|PVar|}}$ enumerating the program variables $p_i \in \mathrm{PVar}$ occurring in π. The interpretation $I_{\bar{x}} := I[p_1 \mapsto \beta(x_i)] \ldots [p_P \mapsto \beta(x_{|PVar|})]$ then denotes the interpretation in which the program variables hold the values of the variables $\bar{x}$.

Lemma 3.13 (Encoding symbolic execution steps) *Let $\pi \in \Pi$ be a self-contained UDL program. There exists a formula $\chi_\pi \in \mathrm{Trm}^{\mathrm{bool}}$ with $2|\mathrm{PVar}| + 2$ free variables such that:*

$$I, \tau, \beta \models \chi_\pi(l_1, \bar{x}_1, l_2, \bar{x}_2) \textit{ for } n, m, l : \mathbb{N} \iff (I_{\bar{x}_2}, l_2) \in R_\pi(I_{\bar{x}_1}, l_1).$$

All program formulas occurring in χ_π already occur within π.

PROOF It is possible to encode the state relationship R_π for a single statement $\pi[k]$ with $k < |\pi|$ into a formula $\chi_{\pi,k}$. It is easy to see that the formulas defined in Table 3.7 faithfully formalise the state relationship R_π in the sense that $(I_{\bar{x}_2}, l_2) \in R_\pi(I_{\bar{x}_1}, l_1) \Leftrightarrow I \models \chi_{\pi,l_1}(\bar{x}_1, l_2, \bar{x}_2)$. Note that if $\pi[k] = \mathsf{assume}\ \varphi$ is a false assumption under $I_{\bar{x}_1}$, then $\chi_{\pi,k}$ is also false: An assumption which does not hold ends the trace. Assertions have the same semantics. The formula χ_π can then be defined as a combination of all predicates $\chi_{\pi,k}$ within π:

$$\chi_\pi(l_1, \bar{x}_1, l_2, \bar{x}_2) := \bigwedge_{k<|\pi|} (k \doteq l_1 \to \chi_{\pi,k}(\bar{x}_1, l_2, \bar{x}_2))$$

Any program formula in χ_π has already occurred within an assertion, assumption or assignment in π. □

Since the natural numbers are fixed in the considered structures, we can use standard encoding techniques (see, for instance, Monk, 1976, Ch. 3) to encode a finite sequence of natural numbers $\sigma \in \mathbb{N}^*$ into a single natural number, denoted by $\ulcorner\sigma\urcorner \in \mathbb{N}$. The reverse operation can also be encoded. The current state of the (finitely many) natural[8] program variables can be seen as a sequence of natural numbers. It is, hence, possible to encode a finite sequence of states into a single natural number. Using this expressiveness, we can find a formula χ_π^+ for the n-fold composition of the one-step before-after-predicate χ_π.

[8] boolean program variables can be canonically subsumed by using 0 and 1 for the values true and false.

$\pi[k]$	$\chi_{\pi,k}(\bar{x},m,\bar{y})$
skip	$m \doteq k+1 \land \bar{y} \doteq \bar{x}$
q := e	$m \doteq k+1$ $\land \quad y_q \doteq \{ \underset{p\in\mathrm{PVar}}{\Vert} p := x_p\}e$ $\land \quad \bigwedge_{p\in\mathrm{PVar}\setminus\{q\}} y_p \doteq x_p$
havoc q	$m \doteq k+1$ $\land \quad \bigwedge_{p\in\mathrm{PVar}\setminus\{q\}} y_p \doteq x_p$
assume φ assert φ	$m \doteq k+1 \land \bar{y} \doteq \bar{x}$ $\land \quad \{ \underset{p\in\mathrm{PVar}}{\Vert} p := x_p\}\varphi$
goto $g_1,\ldots,g_n$	$\bar{y} \doteq \bar{x}$ $\land \quad (m \doteq g_1 \lor \ldots \lor m \doteq g_n)$
end	false

Table 3.7: State transition for a program π encoded as formulas

Lemma 3.14 (Encoding symbolic execution sequences) *Let $\pi \in \Pi$ be a self-contained UDL program. There exists a formula $\chi_\pi^+ \in \mathrm{Trm}^{\mathsf{bool}}$ with $2|\mathrm{PVar}|+3$ free variables such that for any $n \in \mathbb{N}$*

$$I,\tau,\beta \models_{\mathbb{N}} \chi_\pi^+(n,l_1,\bar{x}_1,l_2,\bar{x}_2) \iff \text{there is a state sequence } ((I_{\bar{x}_1},l_1),\ldots,(I_{\bar{x}_2},l_2)) \text{ of length } n \text{ for } \pi.$$

All program formulas occurring in χ_π already occur within π.

PROOF Using the encoding of sequences of naturals within natural numbers, a formula $\tilde{\chi}_\pi(N_1,N_2)$ with two free variables $N_1,N_2:\mathsf{nat}$ can be defined such that $\tilde{\chi}_\pi$ encodes χ_π, that is, $\tilde{\chi}_\pi(\ulcorner l_1,\bar{x}_1\urcorner,\ulcorner l_2,\bar{x}_2\urcorner) \equiv_{\mathbb{N}} \chi_\pi(l_1,\bar{x}_1,l_2,\bar{x}_2)$. The first free variable N_1 encodes the start state $l_1,\bar{x}_1$ and the second N_2 the end state $l_2,\bar{x}_2$ of a single execution step.

The n-fold composition of $\tilde{\chi}_\pi$ can also be arithmetically represented using quantification over all (encoded) sequences of length n. The resulting predicate has an additional free variable n and is denoted by $\tilde{\chi}_\pi^+(n,N_1,N_2)$.

The two encoded arguments can be decoded into $2|\mathrm{PVar}|+2$ variables resulting in a formula χ_π^+ with $\chi_\pi^+(n,l_1,x_1,l_2,x_2) \equiv_{\mathbb{N}} \tilde{\chi}_\pi^+(n,\ulcorner l_1,\bar{x}_1\urcorner,\ulcorner l_2,\bar{x}_2\urcorner)$ which has the required property. □

The formula χ_π^+ can be used to quantify over all reachable states. Thus, it is possible to state a formula that is equivalent to a program formula by formulating that all reachable assertions hold. This can be done in a formula that itself does not contain program formulas.

Theorem 3.15 ($\mathbb{N}$ expressive for *UDL*) *Let $\pi \in \Pi$ be a self-contained UDL program and $0 \leq n < |\pi|$. There exist formulas $\zeta_{[n,\pi]}, \zeta_{[\![n,\pi]\!]} \in \mathrm{Trm}^{\mathsf{bool}}$ without program formulas such that*

$$\models_{\mathbb{N}} \zeta_{[n;\pi]} \leftrightarrow [n;\pi] \qquad \textit{and} \qquad \models_{\mathbb{N}} \zeta_{[\![n;\pi]\!]} \leftrightarrow [\![n;\pi]\!] \,.$$

PROOF This is an inductive argument on the nesting level of program terms. A logic without program formula has nesting level 0. A program term in which all terms do not themselves have program formulas inside has level 1, and so on. A formula of nesting level 0 already has the required properties.

Let π be of nesting level $l > 0$. We define

$$\zeta'_{[n;\pi]} := \forall s.\forall \bar{x}. \bigwedge_{\substack{k<|\pi| \\ \pi[k]\doteq\mathsf{assert}\ \varphi}} (\chi^+_\pi(s,n,\bar{p},k,\bar{x}) \to \{ \mathop{\|}_{p\in\mathrm{PVar}} p := x_p\}\varphi) \tag{3.13}$$

$$\zeta'_{[\![n;\pi]\!]} := \zeta'_{[n;\pi]} \wedge \exists s.\forall l.\forall \bar{x}.\neg\chi^+_\pi(s,n,\bar{p},l,\bar{x}) \tag{3.14}$$

Definition (3.13) encodes that $\zeta'_{[\![n;\pi]\!]}$ is true if every assertion $\pi[k] = \mathsf{assert}\ \varphi$ reachable from the initial state holds in its respective state $(I_{\bar{x}},k)$, that is, if $[\![n;\pi]\!]$ holds. This is the case if and only if every trace is successful. Definition (3.14) additionally requires an upper bound s on the length of traces; no state must be reachable with (precisely) s execution steps.

The primed formulas may still contain program formulas. But by Lemma 3.14, these program formulas already occured in π and, hence, have a nesting level strictly lower than l. By induction hypothesis, these can be replaced by equivalent program-free formulas, yielding the unprimed formulas $\zeta_{[n;\pi]}, \zeta_{[\![n;\pi]\!]}$. □

A direct consequence of this theorem is that every *UDL* formula has got a (computable) equivalent *UDL* formula without program formulas: Natural arithmetic is *expressive* for *UDL*. We have concluded the first goal of the section.

To show the completeness of the calculus, we still need to reason that the rules presented in the last sections suffice to remove program formulas from proof obligations. The first observation we can make in this direction is that the ζ-formulas allow us to always state a sufficiently strong loop invariant and a valid invariant. A statement in a program usually has many invariants which are satisfied whenever the statement is visited. The formula true is the weakest possible loop invariant. We are interested in loop invariants which are not only inductive but also imply that the program does not fail, and we are particularly interested in the weakest condition which implies this.

Under the assumption of this loop invariant, a program cannot fail. This implies in particular that the invariant rules instantiated with such loop invariants are not an overapproximation any more. While the invariants may still approximate the program state, they preserve validity when used in the invariant rules.

Lemma 3.16 (Complete invariant rules) *There exist terms for the invariant and variant such that the inference rules presented in Theorems 3.6, 3.7, 3.10 and 3.11 are complete.*

For the definition of the variants, we make use of the fact that binder symbols do not add to the expressiveness of the logic. The binder symbol $\boldsymbol{\tau} : \alpha \to \mathsf{bool} \to \alpha$ is called the *description operator* and denotes the unique value for which a predicate holds. It has the semantics

$$I(\boldsymbol{\tau}^{[T]})(b) = \begin{cases} u & \text{if } \{u\} = \{x \in \mathcal{D}^T \mid b(x) = tt\} \\ \text{underspecified} & \text{otherwise} \end{cases}$$

and can be axiomatised by $(\forall\alpha.(\forall y^\alpha.(\boldsymbol{\tau} x^\alpha.\ x \doteq y) \doteq y))$. The introduction of the binder $\boldsymbol{\tau}$ is a conservative extension of first order predicate logic as, for instance, shown by Monk (1976, Ch. 29).

Specialisations of this operator are $\boldsymbol{\tau}_{min}, \boldsymbol{\tau}_{max} : \mathsf{nat} \times \mathsf{bool} \to \mathsf{nat}$ denoting the minimum (maximum) natural number for which the argument term evaluates to true. They are defined as abbreviations in terms of $\boldsymbol{\tau}$ as

$$(\boldsymbol{\tau}_{\left\{ \substack{max \\ min} \right\}} x^{\mathsf{nat}}.\varphi) := (\boldsymbol{\tau} x^{\mathsf{nat}}.\varphi \land (\forall y^{\mathsf{nat}}.\ \varphi[x/y] \to x \left\{ \substack{\geq \\ \leq} \right\} y))$$

The value of $\boldsymbol{\tau}$ binder expressions is underspecified in cases in which the unique value does not exist. When using the binders to specify the variants in the proof of the following lemma, binder applications will be limited to cases in which a minimum or maximum is ensured to exist.

PROOF To prove that the application of the rules is complete, it must be shown that the *converse rule* is sound. The converse inference rule is the rule which arises from a rule by exchanging premisses and conclusions. The converse rules have two conclusions which must both be implied by the premiss.

Naturally, the rules are not complete for every invariant. Different invariants $\psi_1, \psi_2, \psi_3, \psi_4$ and variants ν_2, ν_4 are needed for the four rules to be complete. Their proofs are presented sequentially.

Invariant We choose $\psi_1 := \zeta_{[n;\pi]}$ as the invariant for this rule. To show that rule Invariant is complete, we have to show that the converse inference

$$\frac{\Gamma \vdash \{\mathcal{U}\}[n;\pi], \Delta}{\Gamma \vdash \{\mathcal{U}\}\psi_1, \Delta \qquad \psi_1 \vdash [n+2;\rho_1]}$$

with $\rho_1 = \pi \lhd_n (\mathsf{assert}\ \psi_1; \mathsf{assume\ false})$ is valid. Since $\models_{\mathbb{N}} \zeta_{[n;\pi]} \leftrightarrow [n;\pi]$ and $\psi_1 = \zeta_{[n;\pi]}$, the first conclusion (base case) is equivalent to the assumption.

Due to the definition of the loop invariant, the step case is equivalent to $[n;\pi] \to [n+2;\rho_1]$. Assume that $(*) I \models [n;\pi]$, that is, there is no failing trace for π starting in (I,n).

Let $(I,n+2),\ldots,(I',k)$ be a partial trace for ρ_1 reaching an assertion. If $k \neq n$, then the original program π has a corresponding assertion and $(*)$ implies that the trace for the modified program ρ_1 will also not fail here. If $k = n$, then the execution reaches the insertion point n. By definition of the loop invariant, $I' \models \zeta_{[n;\pi]}$ or equivalently $I' \models [n;\pi]$ must be established. If there was a failing trace $(I',n),\ldots,(I'',m)$ for π, then together with the initial partial trace, a failing trace $(I,n),\ldots,(I',n),\ldots,(I'',m)$ for π would exist. But this cannot be by assumption $(*)$.

Note that the validity of the second conclusion does not depend on the validity of the premiss for the choice of the strongest loop invariant.

InvariantTermination For the rule with termination, we use the invariant $\psi_2 := \zeta_{[\![n;\pi]\!]}$ and the variant $\nu_2 := (\boldsymbol{\tau}_{max}\ s^{\mathsf{nat}}.\ (\exists l.\ \exists \bar{x}.\ \chi^{+}_{\pi}(s,n,\bar{p},l,\bar{x})))$. It must be shown that the inference

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\![n;\pi]\!],\Delta}{\Gamma \vdash \{\mathcal{U}\}\psi_2,\Delta \qquad \psi \vdash \{nc := \nu_2\}[\![n+2;\rho_2]\!]}$$

with $\rho_2 = \pi \lhd_n (\mathsf{assert}\ \psi_2 \land \nu_2 \prec nc; \mathsf{assume\ false})$ is sound.

The first conclusion is again equivalent to the premiss due the chosen invariant. Also the proof for the invariant part of the second conclusion is the same as above.

It remains to be shown that the variant ν_2 is indeed decreased in every step. The term ν_2 encodes the length of the longest trace $(I,n),\ldots,(I_{\bar{x}},l)$ starting in (I,n). The assignment $I_{\bar{x}}$ or the l do not play a role, it is the length of this trace which counts. If the program terminates, the length of the longest remaining trace strictly decreases from loop iteration to iteration. Invariant ψ_2 is an assumption in the second conclusion and implies that there are no infinite traces in the states in which the variant is evaluated. This guarantees that the operator $\boldsymbol{\tau}_{max}$ is always applied to a predicate which has a maximum.

AntecedentInvariant To show the result for the rule applied to the program formula in the antecedent the invariant $\psi_3 := \neg\zeta_{[\![n;\pi]\!]}$ is used. The converse rule is

$$\frac{\Gamma, \{\mathcal{U}\}[\![n;\pi]\!] \vdash \Delta}{\Gamma \vdash \{\mathcal{U}\}\psi_3,\Delta \qquad \psi_3,\ [\![n+2;\rho_5]\!] \vdash}$$

with $\rho_5 = \pi \lhd_n (\mathsf{assert}\ \neg\psi_3; \mathsf{assume\ false})$. The base case is again equivalent to the premiss. The sequent $\neg\zeta_{[\![n;\pi]\!]}, [\![n+2;\rho_5]\!] \vdash$ in the second conclusion is equivalent to $[\![n+2;\rho_5]\!] \to [\![n;\pi]\!]$.

Assume that $(*) I \models [\![n+2;\rho_5]\!]$ and let $(I,n),\ldots$ be a trace of π. If the trace does not visit $\pi[n]$ again after the initial state, it has an according trace in ρ_5 (by Obs. 3.4) starting in $(I,n+2)$. By $(*)$, the trace (and, hence, also the trace for π) is finite and does not fail.

If the trace visits $\pi[n]$ in a state I' again, the corresponding trace visits statement $\rho_5[n] = \mathsf{assert}\ \neg(\neg\zeta_{[\![n;\pi]\!]})$, that is $I' \models [\![n;\pi]\!]$. Any trace $(I',n),\ldots,(I'',k)$ for π is finite and successful. The concatenated trace $(I,n),\ldots,(I',n),\ldots,(I'',k)$ is also finite and does not fail.

AntecedentInvariantTermination For the last rule, the strongest invariant is $\psi_4 = \neg\zeta_{[n;\pi]}$ and the according variant is

$$\nu_4 = (\boldsymbol{\tau}_{min}\, s^{\mathsf{nat}}. \bigvee_{\substack{k<|\pi| \\ \pi[k] \doteq \mathsf{assert}\ \varphi}} (\exists \bar{x}.\ \chi^{+}_{\pi}(s,n,\bar{p},k,\bar{x}) \wedge \{ \mathop{\|}_{p \in \mathrm{PVar}} p := x_p \} \neg\varphi)$$

encoding the length of the shortest failing path: The value is the smallest number of steps which can be executed resulting in a state $(I_{\bar{x}},k)$ with $\pi[k] = \mathsf{assert}\ \varphi$ such that the assertion is not satisfied. The value ν_4 is only defined, if there is a failing trace and unspecified otherwise.

The converse rule is

$$\frac{\Gamma, \{\mathcal{U}\}[n;\pi] \vdash \Delta}{\Gamma \vdash \{\mathcal{U}\}\psi_4, \Delta \qquad \psi_4, \{nc := \nu_4\}[n+2;\rho_6] \vdash}$$

with $\rho_6 = \pi \lhd_n (\mathsf{assert}\ \neg(\psi_4 \wedge \nu_4 \prec nc); \mathsf{assume\ false})$.

The base case of the converse rule is again equivalent to the premiss. The sequent $\psi_4, \{nc := \nu_4\}[n+2;\rho_6] \vdash$ in the second conclusion is equivalent to $\neg[n;\pi] \to \neg\{nc := \nu_4\}[n+2;\rho_6]$.

Assume that $I \models \neg[n;\pi]$, that is, there is a failing trace $(I,n),\ldots,(I',k)$ for π. We may, without loss of generality, choose this trace to be the *shortest* failing trace. If $\pi[n]$ is not visited after the first state, then the trace has a corresponding failing trace for ρ_6 according to Obs. 3.4. Otherwise, if (I'',n) is a state in the above trace after the first state, then (by Obs. 3.4) $\rho_6[n] = \mathsf{assert}\ \neg(\psi_4 \wedge \nu_4 \prec nc)$ is visited in a corresponding trace of ρ_6. The assertion is equivalent to $\nu_4 \prec nc \to [n;\pi]$.

Since the trace from I has been chosen the shortest failing trace (its length is stored in nc), the shortest failing trace from I'' (length ν_4) must be strictly shorter since at least one step of execution has been performed. That means: $I'' \models \nu_4 \prec nc$. The trace for π from the intermediate state (I'',n) fails, therefore $I'' \not\models [n;\pi]$. These two facts imply that I'' does not satisfy the assertion $\rho_6[n]$. There is a failing trace for ρ_6.

The invariant ψ_4 guarantees that there always exists a failing path; the predicate in $\boldsymbol{\tau}_{min}$ is satisfiable and the operator used within its specified range. □

With these invariants and variants, we have a complete version for every rule in the calculus at hand. The open goals of a proof tree for a valid formula in which only complete inference rules have been applied are all valid themselves. It remains to be seen that we can always apply the rules to get the sequent program-free. First, we observe a special case of this reduction:

Lemma 3.17 (Completeness of the calculus for a modality definitions) *Let $\mathcal{U}$ be an arbitrary update, $\pi \in \Pi$ a program and every term occurring in π be program-free.*

There exist closed proof trees for $\vdash \{\mathcal{U}\}(\zeta_{[n;\pi]} \leftrightarrow [n;\pi])$ and $\vdash \{\mathcal{U}\}(\zeta_{[\![n;\pi]\!]} \leftrightarrow [\![n;\pi]\!])$.

PROOF Let us consider the modality $[n;\pi]$ (the argument is the same for $[\![n;\pi]\!]$). Calculus rules can be applied resulting in two sequents for the form $\{\mathcal{U}\}\zeta_{[n;\pi]} \vdash \{\mathcal{U}\}[n;\pi]$ and $\{\mathcal{U}\}[n;\pi] \vdash \{\mathcal{U}\}\zeta_{[n;\pi]}$. These sequents can be reduced to sequents without modality or update using symbolic execution (Theorem 3.3), update simplification (Theorem 3.2) and loop invariant rules with the strongest invariant and variant terms as found in Lemma 3.16.

The original sequent is valid by construction, and, hence, as every applied rule is complete, also every leaf of the proof tree. The oracle for sentences over natural numbers can be used to close the proof tree. □

These considered equivalences between $\zeta_{[n;\pi]}$ and $[n;\pi]$ have a special significance as they "define" the truth value of the program formula $[n;\pi]$ in terms of a modality-free formula. Replacing $[n;\pi]$ by the program-free $\zeta_{[n;\pi]}$ does not change the semantics but removes this program formula from the sequent. To be able do so, we need a technical lemma which shows us that we can use quantified equalities to replace expressions using the rules of the calculus.

Lemma 3.18 (Application of quantified equalities) *A sequent $(\forall x_1 ... x_n . t \doteq u), \Gamma \vdash \Delta$ in update normal form can be reduced by application of calculus rules to a sequent $\Gamma' \vdash \Delta'$ in which every instantiation of $t[x_1/s_1, \ldots, x_n/s_n]$ has been replaced by the corresponding instantiation $u[x_1/s_1, \ldots, x_n/s_n]$ but in program formulas.*

This is obvious for every ground instance $t[x/g]$. The quantified equality can be instantiated (using allLeft) and the replacement made by applyEq directly. But the replacement is also possible for non-ground terms with free variables but this requires a structural decomposition over binder applications. With help of the rules cut and binderExt, the non-ground problem can eventually be reduced to the variable-free case. The simple proof works by structural induction and is not elaborated on here since it is rather technical and does not elucidate the goal of the section.

These lemmas allow us to finally formulate the completeness theorem for *UDL*. Defining formulas for modalities are discharged using Lemma 3.17, and Lemma 3.18 is used to remove them from the sequent.

Theorem 3.19 (Complete calculus for *UDL*) *Given an oracle for first order sentences over natural numbers, the UDL calculus presented in this chapter is complete:*

$$\models_{\mathbb{N}} \varphi \quad \Longrightarrow \quad \vdash_{\mathbb{N}} \varphi \qquad \textit{for all } \varphi \in \mathrm{Trm}^{\mathsf{bool}}$$

PROOF Proof by induction on the number of program formulas. If φ has no program, the assumed oracle can be relied upon directly. Let otherwise $\Gamma \vdash \Delta$ be a sequent in the process of the proof. We select a program formula $[n;\pi]$ without embedded program formulas and extract it onto the top level of the sequent applying a cut with the formula $(\forall \bar{v}.\{\mathcal{V}\}(\zeta_{[n;\pi]} \doteq [n;\pi]))$ in which the update $\{\mathcal{V}\} = \{p_1 := v_1 \parallel \ldots \parallel p_n := v_n\}$ ensures that the definition captures all valuations of the program variables $\mathrm{PVar} = \{p_1, \ldots, p_n\}$. Using the cut rule we introduce this axiomatisation on the sequent. The proof tree has two children (A) and (B).

$$\frac{\text{(A)}\quad \Gamma \vdash (\forall \bar{v}.\{\mathcal{V}\}(\zeta_{[n;\pi]} \doteq [n;\pi])), \Delta \qquad \Gamma, (\forall \bar{v}.\{\mathcal{V}\}(\zeta_{[n;\pi]} \doteq [n;\pi])) \vdash \Delta \quad \text{(B)}}{\Gamma \vdash \Delta}\,\text{cut}$$

Branch (A) contains the definition of the modality term in the succedent which can be closed[9] by Lemma 3.17. (There may be additional modalities in the context Γ, Δ, but they do not hinder this proof.)

In branch (B), the definition can be used (by Lemma 3.18) to replace every occurrence of $\{\mathcal{U}\}[v;\pi]$ in Γ and Δ such that the resulting node in the proof tree has one program term less and can be closed by induction hypothesis. □

3.5 Chapter Summary

This chapter has presented a calculus for *UDL*. The logic has also, in less detail, been described in (Ulbrich, 2011). The proof methodology is based on the sequent calculus by Gentzen (1935). In addition to the classical logical rules for sequent calculi, inference rules to handle the idiosyncratic properties of the underlying predicate logic of *UDL* have been introduced.

Updates within formulas can be simplified using a set of update simplification rules. With them, every formula can be brought into an *update normal form* in which every update occurs directly in front of a program formula. These updates can be considered as the intermediate representation of the program state during the execution.

Special emphasis has been put on the treatment the program formulas of the dynamic logic. *UDL* programs can be processed by a set of rules performing *stepwise forward symbolic execution* in the calculus. The application of a step of symbolic execution moves the instruction pointer within the program formula and modifies the proof context of the program formula according to the executed statement. Depending on the type of statement, this may be by adding an assumption, by opening a new side proof or by modifying the program variable assignment.

Unlike in the Boogie approach where the code is organised in basic blocks each comprising several statements of the program, in *UDL* every statement can be addressed

[9] Recall that $\doteq$ and $\leftrightarrow$ denote the same operation on boolean values (see rewrite rule eqToEquiv in Figure 3.1

and executed individually. This allows the calculus to perform a very fine-grained symbolic execution in which every intermediate state can be inspected.

Programs containing loops must be treated differently. After reviewing the situation in classical structured dynamic logic, a methodology to break up loops in unstructured programs has been devised. It has been used to define a sequent calculus rule for the application of loop invariants in the unstructured case. This first inference rule has then been joined by a rule considering termination using a well-founded variant expression which is decreased in every loop iteration. The loop invariant usually provides the necessary information of an abstraction of the current proof state to conduct the proof. However, Beckert et al. (2005) show that the part of the proof context which cannot be affected by the program execution may also be used within the proof even if it is not part of the loop invariant. A rule in which the context of the currently undertaken proof is better preserved has been presented.

Usually in program verification, proof obligations are sequents in which a single program formula appears on the right hand side of a sequent. However, there are cases, in which program formulas may also appear on the left hand side of the sequent. Rules which cover this case have also been introduced. They resemble the rules for the execution under normal conditions – with subtle differences.

Since the program modalities make *UDL* more expressive than first order logic, completeness cannot be achieved for this calculus. Instead a proof has been given for the *relative completeness* of *UDL*: Under the hypothetical assumption that there was a complete calculus for the validity problem over natural numbers, there would also be one for *UDL*. A more practical implication of this completeness result is that provided sufficiently strong loop invariants can be stated, every verification condition involving program formulas can be reduced to proof obligations without programs.

CHAPTER 4

Implementation

Interactive Verification on Intermediate Language

This chapter reports on the reference implementation of a verification system for UDL proof obligations which implements the sequent calculus outlined in the last chapter.

The developed theorem prover ivil supports both interactive and automatic verification. For the automation, a translation of UDL to the input logic of industry-standard decision procedures is presented. The interactive system is designed to allow for user interaction on the source-code level despite the presence of an intermediate verification language. To broaden the reach of automatic verification, additional annotations can be added on the source-code level as hints to the verification engine.

The implementation of the verification system for the core logic is amended by a prototypical implementation of a translation from annotated Java bytecode to ivil programs. It proves the feasibility of both the interactive and the automatic aspect of the approach.

4.1 *ivil* - A Theorem Prover for *UDL*

The verification system "*ivil*" (which stands for Interactive Verification on Intermediate Language) is a proof assistant for the discharge of *UDL* formulas combining interactive and automatic proof search. The human interaction in its interactive component is based on and extends interaction philosophies of other systems, in particular of the KeY system (Beckert et al., 2007) and the Event-B prover *Rodin* (Hallerstede, 2008). The automation component implements automated symbolic execution, automatic application of logical rules and supports the connection of decision procedures to solve problems. *ivil* possesses both a command line interface (allowing for embedding it into integrated development environments for instance) and an easy-to-use graphical user interface. It has been implemented in Java and the source code base comprises approximately 50.000 lines of code.

The system operates on the variant of *UDL* in which program formulas are postfixed with an additional assertion (see Section 2.5.3). The implemented calculus is the sequent calculus for *UDL* presented in Chapter 3. The fundamental rules presented there are complemented by numerous additional lemmas introduced to make the system more efficient and powerful. The validity of derived rules can, under certain restrictions, be proved within the system itself. An adaptation of the approach by Bubel et al. (2008) has been used to check the validity of the lemmas (where possible). This check is part of the regression test base of the system.

The input language of *ivil* is designed to allow for extensions of the system. Types, function and binder symbols, axioms, rewriting or inference rules can be declared in *ivil* input files. Common data types often needed in verification contexts (namely finite sequences, sets and maps) together with typical operators on them are specified in a small, but extensible, library of input files.

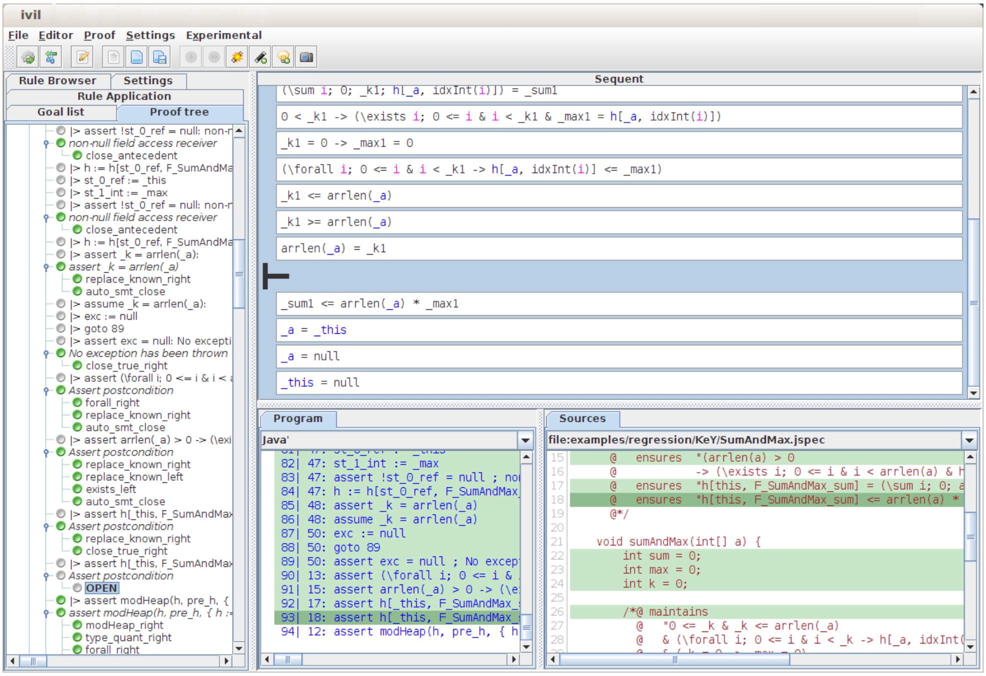
The programming language of *ivil* is intentionally closely related to the language used in the Boogie verification system but there are concepts in *ivil* like the inference rules which are not available in Boogie. The input language of Boogie is syntactically richer though every input can be reduced to a core language which resembles that of *ivil*. Felden (2011) describes the implementation of a front-end which allows reading Boogie files as input into *ivil*.

One design decision for the system was not to emphasise a fully automated calculus serving as a decision procedure for *UDL* but to rely on existing decision procedures for first order logic with theories (*satisfiability modulo theory* solvers; SMT). *ivil* has got an engine to apply rules automatically, buts its purpose is more to pre-process the received input than to prove it correct. In more evident cases, the automation can find a proof without relying on solvers. For the automatic application of rewriting and inference rules, a number of configurable strategies have been implemented to control the process of rule application. Simplification by term rewriting is a simple strategy. It can be restricted to rules not splitting the goal such that the number of open proof branches does not explode. Symbolic execution is the strategy responsible for removing program formulas from the proof obligation. It can be configured to stop execution under certain conditions. The strategies are designed to keep the formulas on the sequent comprehensible for the human reader, leaving transformation to normal forms, skolemisation and other processing entirely to the theory solver.

The verification engines relies on findings that a decision procedure returns. For the communication with the underlying decision procedures, the de-facto standard format SMT-LIB (Barrett et al., 2010) has been chosen. It is widely supported and can be used to drive many different solvers. We aimed in particular for the solver Z3 (de Moura and Bjørner, 2008) which stands out with its wide range of featured theories, its power and active development.

4.2 User Interaction

It has been a major concern that an intermediate verification language would severely abate the comprehensibility of the verification process for the human user. Can the verification state be presented understandably to the user of the verification system if it operates on an internal representation? And can the system provide useful feedback to the user in case of a failed verification attempt? This section will report on the



```
|> assert !st_0_ref = null: non-n
non-null field access receiver
    close_antecedent
|> h := h[st_0_ref, F_SumAndMa
|> st_0_ref := _this
|> st_1_int := _max
|> assert !st_0_ref = null: non-n
non-null field access receiver
    close_antecedent
|> h := h[st_0_ref, F_SumAndMa
|> assert _k = arrlen(_a):
assert _k = arrlen(_a)
    replace_known_right
    auto_smt_close
|> assume _k = arrlen(_a):
|> exc := null
|> goto 89
|> assert exc = null: No excepti
No exception has been thrown
    close_true_right
|> assert (\forall i; 0 <= i & i < a
Assert postcondition
    forall_right
    replace_known_right
    auto_smt_close
|> assert arrlen(_a) > 0 -> (\exi
Assert postcondition
    replace_known_right
    replace_known_left
    exists_left
    auto_smt_close
|> assert h[_this, F_SumAndMax
Assert postcondition
    replace_known_right
    close_true_right
|> assert h[_this, F_SumAndMax
Assert postcondition
    OPEN
|> assert modHeap(h, pre_h, {
assert modHeap(h, pre_h, { h :=
    modHeap_right
    type_quant_right
    forall_right
```

```
(\sum i; 0; _k1; h[_a, idxInt(i)]) = _sum1
0 < _k1 -> (\exists i; 0 <= i & i < _k1 & _max1 = h[_a, idxInt(i)])
_k1 = 0 -> _max1 = 0
(\forall i; 0 <= i & i < _k1 -> h[_a, idxInt(i)] <= _max1)
_k1 <= arrlen(_a)
_k1 >= arrlen(_a)
arrlen(_a) = _k1
⊢
_sum1 <= arrlen(_a) * _max1
_a = _this
_a = null
_this = null
```

```
82| 47: st_1_int := _max
83| 47: assert !st_0_ref = null ; no
84| 47: h := h[st_0_ref, F_SumAndMax
85| 48: assert _k = arrlen(_a)
86| 48: assume _k = arrlen(_a)
87| 50: exc := null
88| 50: goto 89
89| 50: assert exc = null ; No excep
90| 13: assert (\forall i; 0 <= i & 
91| 15: assert arrlen(_a) > 0 -> (\e
92| 17: assert h[_this, F_SumAndMax_
93| 18: assert h[_this, F_SumAndMax_
94| 12: assert modHeap(h, pre_h, { h
```

```
      @   ensures  "(arrlen(a) > 0
      @            -> (\exists i; 0 <= i & i < arrlen(a) & h
      @   ensures  "h[this, F_SumAndMax_sum] = (\sum i; 0; a
      @   ensures  "h[this, F_SumAndMax_sum] <= arrlen(a) *
      @*/

    void sumAndMax(int[] a) {
        int sum = 0;
        int max = 0;
        int k = 0;

        /*@ maintains
          @   "0 <= _k & _k <= arrlen(_a)
          @   & (\forall i; 0 <= i & i < _k -> h[_a, idxInt(
```

Figure 4.1: Screenshot of *ivil* in action

measures which have been taken in the *ivil* system to achieve a powerful tool which is still comprehensible.

Figure 4.1 shows a typical situation during a proof within *ivil*. The main view is the sequent in the upper right part of the window. The formulas on the sequent are kept in individual boxes which separate them more clearly. This presentation is similar to the one in Rodin while KeY separates formulas by commas making it sometimes hard to tell where one formula ends and the next one starts.

On the left hand side of the frame, the sequent calculus proof tree with the applied inference rules is visible. The granularity of the tree can be adjusted. Inference rules can be annotated with a level of importance and the tree can be configured to only show rule applications whose rules have at least the configured level of importance. Thus, the proof tree can be kept clean from "background clatter" (such as, for instance, propositional or update normalisation). The user can concentrate on the relevant rule applications. These include, as the most prominent instances, the rules which have been applied to perform symbolic execution. This allows any intermediate state in the transformation from code into logic to be inspected by examining the associated sequent. A green bullet next to the node label indicates that a node has a closed subtree beneath it, that is, that its sequent has already been proved valid.

Verification goals will usually not be formulated in the cumbersome intermediate language directly but in a higher language and then automatically translated into their *UDL* equivalent. With the deliberately small number of statement constructors in the intermediate programming language, *UDL* programs tend to be significantly longer than the source programs. For the sake of better readability, the programs are not displayed in the sequent view, but instead referred to by name. On the lower left of the sequent view resides an area in which all relevant *UDL* programs are listed. To the right of this area, there is another code area containing the source code which gave rise to the intermediate code.

4.2.1 Application of Rules

Moving the mouse pointer over a subterm of a formula on the sequent highlights that term. A mouse click opens a popup menu in which all applicable rules (and their instantiations) are presented and from which the next rule application can be chosen. Applying a rule changes the sequent accordingly and the next rule application can be selected. This method of interaction for rule selection is (in general) also used in KeY and Rodin, and has already been presented by Bertot et al. (1994). Rules requiring a term parameter, like quantifier instantiation, replacement of equal terms etc. can also be applied conveniently by dragging a subterm and dropping it onto some other term on the sequent.

4.2.2 Background Constraint Solving

During an intricate interactive proof, often situations arise in which a number of open branches remain to be discharged. It is usually the case that the difficulty of a particular proof obligation manifests itself only on a limited number of open proof goals while the majority of them can be discharged automatically without further help from the user.

ivil features a background process running with low priority which constantly invokes the constraint solver on the currently open goals, which automatically closes them if they can be discharged and leaves them untouched if they cannot be closed. This process is, hence, totally transparent to the user.

This mechanism allows an analytic and explorative approach to examine goals in which it is unclear what steps are to be taken. Case distinguishing rules like Cut (for immediate lemma generation), andRight or orLeft can be applied repeatedly to split the proof into individual cases. If the proof obligation has a local problem which is specific to one of these cases, this case will remain open as the only branch, while the others are closed automatically. On the remaining branches, the user has more detailed information which facilitates the problem analysis.

4.2.3 Strict Separation between Interaction and Automation

A feature which separates *ivil* noticeably from KeY is the way in which automatic and interactive rule application are kept separate.

The proof strategies in KeY aim to implement a complete calculus for its logic. Hence, rules are automatically applied eagerly resulting in proof states which are often not comprehensible for the human user: Formulas are brought into negation normal form, arithmetic expressions are normalised to allow for better algorithmic treatment. Grebing (2012) reports in her usability study that this makes it difficult to keep track of the proof. *ivil* does not suffer from this deficiency since the inference rules it applies are meant to help the user understand the proof situation, normalisations are left to the decision procedure. The subsystem of the SMT solver can, in this regard, be considered a black box.

This is mainly achieved by *reducing* the set of rules which are applied by default. Propositional rules, for instance, which decompose larger formulas into their smaller parts usually make the proof situation more understandable and are included in this basic set of rules. Rules which change the way in which the statements are represented are less suited for interactive presentation. Expanding a function symbol by its definition (for instance replacing $S \subseteq T$ by $(\forall x. x \in S \to x \in T)$) often does not improve readability and is, hence, not automatically applied in the interactive environment. It may, however, be necessary to also apply such rules, and the strategies can be reconfigured such that such rules are enclosed in the automatically applied set of rules when necessary.

One separating advantage lies in the logic and calculus themselves. The intermediate proof states can be kept relatively small as all assignments handled during symbolic execution are stored in the update preceding a program term. Other approaches that do not have this kind of intermediate value storage must add an equality to the context describing the assignment. A program becomes thus executed as a large number of equalities. This may be efficient for automatic provers, but is not suited to be shown to the user for interaction. Capturing intermediate states by means of updates condenses this presentation: unneeded assignments are discarded, consecutive value changes combined.

4.2.4 Presentation of Code

It is evident that an intermediate language layer poses additional challenges for an interactive verification of programs in a higher programming language.

The input language of *ivil* allows the specification of correspondences between lines in the source program text and statements in the intermediate program. The user interface then presents to the user both the source code and the intermediate code side by side. This allows the symbolic execution to be tracked not only on the intermediate program but also on the level of the original source language.

A proof is organised as a tree in which every branch corresponds to one path in the intermediate program. When a node in the proof tree is displayed, those statements which have been traversed by symbolic execution in the current proof situation are highlighted. The defined correspondence between source and intermediate code allows that the traversed lines of the original program be highlighted as well. It is hence possible to recognise the currently examined path through the program at a glance on the program text. The statements executed last are additionally highlighted in a more intense colour to indicate the execution state to which the sequent belongs; see the screenshot in Figure 4.1 for an example.

The interactive component can thus be used as a *formal static debugger*. The user is acquainted with the mentioned highlighting from their source level dynamic debugger integrated in modern software development environments where the currently active statement is similarly marked. Other features known from dynamic debuggers have found their way into the interactive prover: The system also allows setting breakpoints both in the source code and the intermediate program. They cause the symbolic execution to run up to the marked statement but no further. This can be used to inspect the system state at defined points. It is also possible to step through a program statement by statement, or line by line in the sources, to observe the evolution of the proof state captured in the sequent.

In the sequent of a node in the proof tree, the current state of the symbolic execution manifests itself in form of the path condition as formulas on the sequent and the update which captures the assignments which have been made in the course of the execution.

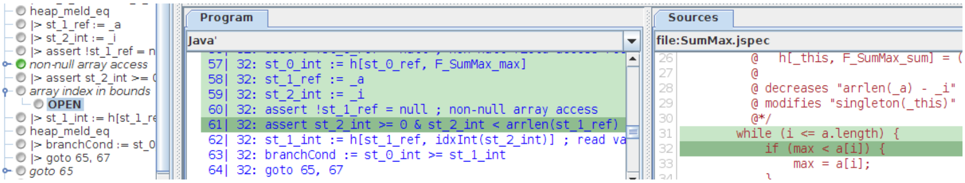
Assertions in *ivil* are embedded within the programs to be verified. This also supports human comprehensibility: By disseminating the proof obligation into many individual assertions distributed over the intermediate program, every unclosable branch points directly to its failing assertion. Any statement within a program declared in an *ivil* input file can optionally be annotated with a text. If assertions are furnished with a description of their intention, every failing branch directly informs the user about the cause. Figure 4.2 depicts a proof situation in which the check "array index in bounds" cannot be closed; the program under investigation has a bug such that the index may go beyond the legal range. The *UDL* program statement and the source code line responsible for the check are highlighted. The user is immediately informed about the nature of the failed check. Also the command line tool can fail with a meaningful error message and a pointer to the failing source code line:

IVIL COMMAND LINE OUTPUT

```
This is ivil - 0.20
SumMax.m(int[])-normal.p#Java_total :
  file:SumMax.jspec:32:
   annotation: array index in bounds
```

IVIL COMMAND LINE OUTPUT – 4.1

In classical dynamic logic without embedded assertions, any verification condition must be checked after the execution of the entire program using the modal operator. Embedding these conditions into the code opens the field for a clearer separation



```
heap_meld_eq
|> st_1_ref := _a
|> st_2_int := _i
|> assert !st_1_ref = n
non-null array access
|> assert st_2_int >= (
array index in bounds
  OPEN
|> st_1_int := h[st_1_re
heap_meld_eq
|> branchCond := st_0
|> goto 65, 67
goto 65
```

```
32: st_0_int := h[st_0_ref, F_SumMax_max]
32: st_1_ref := _a
32: st_2_int := _i
32: assert !st_1_ref = null ; non-null array access
32: assert st_2_int >= 0 & st_2_int < arrlen(st_1_ref)
32: st_1_int := h[st_1_ref, idxInt(st_2_int)] ; read va
32: branchCond := st_0_int >= st_1_int
32: goto 65, 67
```

```
      @   h[_this, F_SumMax_sum] = (
      @
      @ decreases "arrlen(_a) - _i"
      @ modifies "singleton(_this)"
      @*/
    while (i <= a.length) {
      if (max < a[i]) {
        max = a[i];
```

Figure 4.2: A failed assertion giving feedback to its reason

of the different proof goals. If the postcondition is a conjunction of more than one expression, these conditions can be checked in separate assertions, allowing clear indication of which of them failed and which succeeded. Checks for division by zero, index out of bounds, illegal heap accesses and similar need not be postponed to the end of the code but can thus be executed in place.

In Section 4.4, a translation from Java bytecode to *UDL* programs is presented. Java bytecode is the result of a Java compilation process and, as such, another intermediate code representation. Despite the *two* intermediate stages, the information from the sources is preserved such that the verification can be performed in terms of the source code elements.

However, completely hiding the intermediate program proved not so good an idea. At times, during the verification one wonders why specific elements on the sequent show up and where they stem from, or also why expected elements do not show up. It may then be helpful to be able to have a glance under the hood of the verification machine to understand the cause of action. It is a feature of this interactive verification system that the verification can be inspected on both levels of details, depending on the needs of the user.

4.2.5 Autoactive Proof Control

The *ivil* tool has an interactive user interface which allows direct user intervention on the level of the logic. However, this not always the desired level of abstraction:

In the field of program verification, it is a concern that the user of the verification system should get as little as possible in touch with the technical details of the underlying theorem prover engine as using such a system requires detailed knowledge and experience. Intermediate proof state representations tend to become rather large and little comprehensible.

The *specifier* and *programmer* of a piece of code will most probably have an intuition of the program which operates in terms of the programming language of that code. An appropriate specification language is, hence, close to the programming language which is used[1]. In a fully automatic, non-interactive proof system, the specifier then invokes the verification system as a black box without seeing an intermediate level at all. Even though the possibilities of theorem solvers have increased tremendously

[1] like the embedded specification language of Eiffel, the Java Modeling Language for Java or the Spec$^{\#}$ language extending C$^{\#}$.

in recent years, deductive verification will always be a complex task. There will always be specifications which cannot be decided totally automatically (even if this automation-frontier will be constantly pushed further and further).

The adjective *autoactive* (as in the combination of *auto*matic and inter*active*) has been coined by Leino (2010b) to characterise verification tools with a certain limited kind of user interaction: In an autoactive verification environment, the specifier interacts with the system solely by annotation of the sources with additional proof hints interpreted by the verification tool. The most commonly known and widely accepted autoactive annotation is a loop invariant: It is not part of the formal contract of an operation but a mere aid for the verification tool to establish the proof of the contract. A loop invariant is not essential: It can be omitted if an invariant generator is powerful enough to infer it from the surrounding code and specification.

Other annotations which may be added to guide the non-interactive background solver include additional assertions which can be added as statements to give the "stepping stones" that the automatic tool can use to perform the proof.

When a program cannot be verified automatically, either the property might not hold or the theorem prover might not be strong enough to show it. Without interacting with the verification on a more detailed level, autoactive annotations can be modified in an iterative fashion. Either this will eventually lead the automation to success or it helps to pinpoint the potential error in the code or specification.

Autoaction inherently has major advantages over the reduction to interactive theorem proving technologies: The specifier does not need to switch their intuition between two semantic systems. A verification engine has to map notions of the programming and specification language onto their logical counterparts. Usually it is inherent that the notions of the logic differ (subtly or substantially) between these two systems. In a verification system with autoactive interaction, there is only one level of syntax and semantics, that of the programming and specification language, and every input and output happens in terms of that level. If the verification fails, the failing specification part is indicated together with the reason for the failure.

Annotations serve also as additional documentation for the code: A formal loop invariant strong enough to establish the postcondition and formal assertions are (generally) comprehensible annotations that help to make the code more understandable. Another positive effect of autoactive annotations is that they serve as proof scripts to reconstruct the proof at a later date. The annotated sources become *proof carrying (source) code*.

But a proof cannot always be conducted autoactively. The advantage of autoactive systems is, at the same time, their major disadvantage. They do not give any insight on the reasons of a failure as they provide no details in terms of a logical representation of the verification conditions. The feedback an autoactive system can give consists of a pointer to a problematic specification element and the message *that* a corresponding proof obligation could not be closed but not *why*. If there is a flaw in the specification, a counter example might also be presented in addition, but if the verification fails due to an incompleteness of the calculus, that is not an option either. It may require a lot of understanding of the verification system and an experienced intuition to learn about the actual cause of the failure. This implies that the verification effort must be driven by repeated trial-and-error refinement of assertions added to the program. Without adequate insight into the formalisation and the theorem prover, this is often a wild-goose chase.

There are situation when introspection into the prover is needed to learn more about the current proof situation. This may be because the failing of an assertion is justified but hard to understand or because of an incompleteness issue in the theorem solver which could be amended by a little interactive guidance. The user of an *interactive* verification system needs to know about the logical counterparts of a specification if they want to investigate why a proof fails. Since the effort for full interaction is considerably higher than for autoaction, this should be reduced to as few cases as possible.

Interactive additions to the major available autoactive tools provide evidence that automation is not always achievable. The tool additions grant more insight than the push-button process of the tool itself. The Why verification tool family (Filliâtre and Paskevich, 2013) uses Coq (Bertot, 2008) as interactive back-end for the most intricate proof situations. Böhme et al. (2008) describe such an extension for Boogie using Isabelle/HOL (Paulson, 1986) to manually discharge difficult proof obligations. Dahlweid et al. (2009) present a tool suite for the VCC verification tool permitting the analysis of counter examples and proof logs provided by Boogie and Z3.

The *ivil* approach combines the best of both worlds: It provides an interactive proof environment in which proof situations can be investigated on a level of presentation which does not expose all technical details needed by the automation. On the other hand, an efficient connection to decision procedures has been established such that automation can be achieved to a high degree.

We have taken the source level interaction one step further and allow the specification not only of intermediate lemmas but can also give hints on how their proof show be conducted. In purely automatic systems, a verification condition is generated from the source code and handed to an automatic theorem solver which then hopefully discharges it. But no further influence can be exerted on the process of the solver. *ivil* is different in this respect. The course of proof can be influenced after it has been started.

We allow statements to be accompanied by a *proof hint* indicating how the verification engine should continue on a particular proof branch. The calculus possesses rules which are correct but which are not applied automatically as they could lead an automated strategy astray. By adding an annotation, the application of such rules can be triggered. Figure 4.3 lists the proof hints implemented in *ivil*.

As a small instructive example assume that there is a proof situation in which it must be proved that the factorial of $n \geq 0$ is positive. This cannot be deduced automatically by neither the *ivil*-strategy nor by the constraint solver. To autoactively mend the situation we add an assertion.

$$\mathsf{assert}\ (\forall n. n \geq 0 \rightarrow \mathit{fac}(n) \geq 1) \quad \text{“(}\textbf{axiom}\text{ facDef) (}\textbf{rule}\text{ intInduction)”}$$

(**rule** R) Apply the inference rule named R to the asserted condition.

(**drop** φ) Remove the formula φ from the sequent under inspection. There are cases when a (usually quantified) formula can prevent the solver from deciding a otherwise decidable sequent. Dropping them makes life easier for the decision procedure.

(**focus** $\varphi_1\ \varphi_2\ \dots\ \varphi_n$) Remove all but the given formulas φ_i from the sequent under inspection. This follows the same principles as **drop** but allows the selection of the relevant formulas rather than of the irrelevant.

(**axiom** A) Add the axiom named A known to the system to the sequent. This axiom is normally not part of the SMT translation (it might be a recursion) and can thus be explicitly added.

(**expand** $R\ [n]$) Apply the rewriting rule R to the entire sequent. Optionally a bound n can be stated on how often a recursion is to be expanded.

(**cut** φ) Do a case distinction over the formula φ.

(**inst** $\varphi\ t$) Instantiate the quantified formula φ with the term t.

(**decproc** *params*) Call the decision procedure with the given parameters. This is a rather technical directive which can be used to explore the capabilities of the decision procedure.

Figure 4.3: Proof hints in *ivil*

This proof hint annotation first adds the definition of the factorial function onto the sequent (see also (4.22)) and then applies the rule intInduction (see also Appendix A.2) on it which is the induction schema rule for natural numbers. The obligation can then be closed by the SMT solver.

The notation of proof hint annotated assertions bears an inherent resemblance with intermediate proof results in Isabelle/HOL's proof script language Isar (Wenzel, 1999):

ivil : assert φ "(**hint** arg) (**hint** args)"
Isabelle/Isar : **have** φ **by** (tactic args)

Proof hint annotations have, for instance, been used to make the proof of the benchmark "First in Linked List" in Section 4.4.11 run automatically.

4.3 Interaction with the Decision Procedure

Section 3.4 showed that the calculus presented there is complete given an oracle on the natural numbers. When suitably strong loop invariants have been provided for all control flow cycles of a program, the calculus is able to reduce a verification proof obligation containing program formulas to a formula without modalities or updates. This process, which then reduces the problem to verification conditions in the base logic, is deterministic and can be implemented efficiently.

Since an oracle for the natural numbers cannot exist, we instead rely on the results of sophisticated theory solvers which approximate an oracle. Modern SMT solvers are quite mature when it comes to integer arithmetic. They support linear arithmetic well and some systems even support non-linear arithmetic involving multiplication and division. We rely on their abilities, in particular for the theories of integers.

This does not take away the necessity of sensible inference rules which are applied within the calculus either automatically or interactively as not every problem can be solved by a constraint solver. In particular, proofs involving some kind of induction are not within reach for constraint solvers. Additionally, even if a problem could theoretically be decided by a solver, it may be beyond its means due to the resources (time or memory) that would be needed to find the proof. It is then helpful to guide the automation by a few interactive steps, like by instantiating quantified formulas or by removing irrelevant parts from the problem to reduce the search space. Moreover, additionally introduced (and proved) lemmas can be applied as inference rules.

4.3.1 Translation from *UDL* to SMT

Leino and Rümmer (2010) show how parametrically typed first order logic can be reduced to the logic supported by SMT solvers; further details on the translation can also be found there. We extend their translation to obtain a function $\hat{\cdot}$ to translate a *UDL* proof obligation φ over the signatures Γ,Σ to an equisatisfiable formula $\hat{\varphi}$ over $\hat{\Gamma},\hat{\Sigma}$ doing without the advanced features of the logic. In particular, parametric types, binder symbols and type quantification are removed. The translation silently assumes that all program formulas and updates have been resolved by symbolic execution and update simplification. Table 4.4 gives an overview over the translation.

The translation is sound, yet deliberately incomplete. In some cases where more information could theoretically be transferred to the constraint solver, this is not done to not lead the fully automatic tool astray. For instance, the axiom

$$(\forall s^{\mathsf{set}(\mathsf{nat})}.\ \mathsf{in}(0,s) \land (\forall x^{\mathsf{nat}}.\ \mathsf{in}(x,s) \to \mathsf{in}(x+1,s)) \to s \doteq \mathsf{fullset}^{\mathsf{nat}})$$

representing an induction scheme for the type $\mathsf{set}(\mathsf{nat})$ should not be presented to the SMT solver since the prover could be tempted to instantiate s with many matching terms that would not help the proof. During an interactive proof, well-chosen instances of such rules should be used only.

The target type system $\hat{\Gamma} = \{\mathsf{type}, \mathsf{universe}, \mathsf{int}, \mathsf{bool}, \mathsf{array}\}$ is fixed independently of the type system Γ. The types int, bool and array are built into the SMT solver and the theories build upon them. The type '$\mathsf{universe}$' serves as common reservoir for all objects of the original logic, its domain $\mathcal{D}_{\hat{\Gamma}}^{\mathsf{universe}}$ corresponds to the union domain $\mathcal{D}_{\Gamma}$ in the original type system. Although the target logic would support parametrically polymorphic types, the translation does not make use of it. Instead, the original type system is flattened into a single type $\mathsf{universe}$. The SMT logic does not support quantification over types; therefore, the translation is designed to preserve the possibility to translate polymorphic statements which are in the translation no longer polymorphic.

The type 'type' models the type system Γ as a type itself. For every type constructor $c \in \mathrm{TCon}_\Gamma$, there is a function symbol $\hat{c} : \mathsf{type}^{\mathrm{ar}(c)} \to \mathsf{type}$ of corresponding arity. The intuition of this type is that its domain $\mathcal{D}^{\mathsf{type}} = \mathcal{T}_\Gamma^0$ is the set of ground terms over these function symbols. The type of types is not fully specified; in particular, it is not formalised that the types are freely generated by the constructors. However, for the practical applications we encountered, it was sufficient to encode that the functions $\hat{c}$ for $c \in \mathrm{TCon}_\Gamma$ are injections with disjoint images. A predicate $\mathsf{instance} : \mathsf{universe} \times \mathsf{type} \to \mathsf{bool}$ sets objects and their types into relation. Using the type of types, a type quantification $(\mathbb{V}\alpha.\varphi)$ or $(\mathbb{E}\alpha.\varphi)$ can be translated to an ordinary object quantification $(\forall \hat{\alpha}^{\mathsf{type}}.\ \hat{\varphi})$ or $(\exists \hat{\alpha}^{\mathsf{type}}.\ \hat{\varphi})$ respectively. The type variable $\alpha \in \mathrm{TVar}$ becomes the object variable $\hat{\alpha}^{\mathsf{type}} \in \mathrm{Var}_{\hat{\Gamma}}$.

To fully utilise the built-in theories of the solver, their respective types must be used. To bridge between these built-in types and the universe of all objects, mapping functions like $\mathsf{i2u} : \mathsf{int} \to \mathsf{universe}$ and $\mathsf{u2i} : \mathsf{universe} \to \mathsf{int}$ are introduced to switch from one representation to another when needed. Every term $t \in \mathrm{Trm}_\Sigma^T$ for any type $T \in \mathcal{T}_\Gamma$ is translated to a term $\hat{t} \in \mathrm{Trm}_{\hat{\Sigma}}^{\mathsf{universe}}$, but operators like "+" are defined on the type int. The addition $t + u$ of two terms $t, u \in \mathrm{Trm}^{\mathsf{int}}$ is hence translated by a temporary projection to the integer type as $\mathsf{i2u}(\mathsf{u2i}(\hat{t}) + \mathsf{u2i}(\hat{u}))$. If properly axiomatised, these injections do not notably impede the efficiency of the SMT solver. We will in the following drop these mappings from the notation and silently assume them applied where needed. A similar encoding has been done for the boolean type using the functions $\mathsf{b2u} : \mathsf{bool} \to \mathsf{universe}$ and $\mathsf{u2b} : \mathsf{universe} \to \mathsf{bool}$.

For every function symbol $f : T_1 \times \ldots \times T_m \to T \in \mathrm{Fct}_\Sigma$, there is a function symbol $\hat{f} : \mathsf{type}^n \times \mathsf{universe}^m \to \mathsf{universe}$ on the target side of the translation in which n is the number of the distinct type $\alpha_1, \ldots, \alpha_n$ variables occurring in $T_1, \ldots, T_m, T$. The type arguments only implicitly present in the polymorphic input logic are made explicit in the result logic.

For an example, assume there is a polymorphic constant symbol $d : \alpha$ in the signature whose interpretation is partially given by the equalities $d^{[\mathsf{int}]} \doteq 1$ and $d^{[\mathsf{bool}]} \doteq \mathsf{false}$. Clearly, this symbol cannot be translated as constant in the type $\mathsf{universe}$ as its value depends on the type applied to it. Therefore, its translation is the unary function symbol $\hat{d} : \mathsf{type} \to \mathsf{universe}$. It needs the explicit type argument to distinguish between the many values in the set $\{I(d^{[T]}) \mid T \in \mathcal{T}^0\}$. The concrete instance $d^{[\mathsf{int}]}$ is translated as the application $\hat{d}(\widehat{\mathsf{int}})$. The axiom $(\forall t^{\mathsf{type}}.\ \mathsf{instance}(\hat{d}(t), t))$ guarantees type safety.

Unlike the symbol d, the function symbol $\mathsf{singleton} : \alpha \to \mathsf{set}(\alpha)$ for singleton sets has no type variables in the result type not already occurring in one of its argument types. This implies that both return type and return value are fully determined by

***UDL** t*	**SMT** $\hat{t}$
$c \in \Gamma$	$\hat{c} : \mathsf{type}^{\mathrm{ar}(c)} \to \mathsf{type}$
$f \in \mathrm{Fct}$	$\hat{f} : \mathsf{type}^{n} \times \mathsf{universe}^{m} \to \mathsf{universe}$
$b \in \mathrm{Bnd}$	$\hat{b} : \mathsf{type}^{n} \times (\mathsf{array}(\mathsf{universe}, \mathsf{universe}))^{m} \to \mathsf{universe}$
$x^T \in \mathrm{Var}_\Gamma$	$\hat{x}^{\mathsf{universe}} \in \mathrm{Var}_{\hat{\Gamma}}$
$\alpha \in \mathrm{TVar}$	$\hat{\alpha}^{\mathsf{type}} \in \mathrm{Var}_{\hat{\Gamma}}$
$f^{[T_1,\dots,T_n]}(t_1,\dots,t_m)$	$\hat{f}(\hat{T}_1,\dots,\hat{T}_n,\hat{t}_1,\dots,\hat{t}_m)$
$t \circ_{\mathsf{int}} u$	$\mathsf{i2u}(\mathsf{u2i}(\hat{t}) \circ_{\mathsf{int}} \mathsf{u2i}(\hat{u}))$
$\varphi \circ_{\mathsf{bool}} \psi$	$\hat{\varphi} \circ_{\mathsf{bool}} \hat{\psi}$
$t \doteq u,\ t \dot{\approx} u$	$\hat{t} \mathrel{\widehat{=}} \hat{u}$
$(\forall x^T.\ \varphi)$	$(\forall \hat{x}^{\mathsf{universe}}.\ \mathsf{instance}(\hat{x}, \hat{T}) \to \hat{\varphi})$
$(\exists x^T.\ \varphi)$	$(\exists \hat{x}^{\mathsf{universe}}.\ \mathsf{instance}(\hat{x}, \hat{T}) \land \hat{\varphi})$
$(\forall\!\!\!\forall \alpha.\ \varphi)$	$(\forall \hat{\alpha}^{\mathsf{type}}.\ \hat{\varphi})$
$(\exists\!\!\!\exists \alpha.\ \varphi)$	$(\exists \hat{\alpha}^{\mathsf{type}}.\ \hat{\varphi})$
$(b^{[T_1,\dots,T_n]}\ v.\ t_1,\dots,t_m)$	$\hat{b}(\hat{T}_1,\dots,\hat{T}_n, A^v_{t_1}(\bar{x}_1),\dots,A^v_{t_m}(\bar{x}_m))$
S^T_v	$\mathsf{select}(\hat{S}^T_v, \hat{v})$

n denotes the number of type arguments of a symbol, m the number of object arguments, $\circ_{\mathsf{bool}} \in \{\land, \lor, \to, \leftrightarrow\}$, $\circ_{\mathsf{int}} \in \{+, -, *\}$

Table 4.4: Synopsis of the translation from *UDL* to SMT

its argument value already. Therefore, an additional type argument would not be required. $\mathsf{singleton}$ could be translated as a unary function $\widehat{\mathsf{singleton}} : \mathsf{universe} \to \mathsf{universe}$ with the axiom $(\forall t^{\mathsf{type}}.\forall u^{\mathsf{universe}}.\ \mathsf{instance}(u,t) \to \mathsf{instance}(\widehat{\mathsf{singleton}}(u), \widehat{\mathsf{set}}(t)))$ guaranteeing welltypedness.

But the employed SMT solver proved to give better results when $\mathsf{singleton}$ was instead translated with an additional type parameter as $\widehat{\mathsf{singleton}} : \mathsf{type} \times \mathsf{universe} \to \mathsf{universe}$ even though this introduces a redundant encoding of the typing and leads to terms that do not have a counterpart in the original logic. For instance, the term $\widehat{\mathsf{singleton}}(\widehat{\mathsf{bool}}, \widehat{\mathsf{zero}})$ in which the *explicit* typing contradicts the *implicit* typing deducible from the arguments, has no correspondent in the source logic. The translated function can hence be regarded as an underspecified partial function. However, despite the spurious descriptions and the seemingly lengthy encoding, the additional arguments appear to provide the SMT calculus with more material to apply its matching algorithms against. This observation coincides with the ones made by Leino and Rümmer (2010).

In general, a function application $f^{[\{\alpha_1/U_1,\ldots,\alpha_k/U_n\}]}(t_1,\ldots,t_m)$ is for types $U_1,\ldots,U_n \in \mathcal{T}$, hence, translated by distributing the translation $\hat{\cdot}$ over its arguments resulting in the term $\hat{f}(\hat{U}_1,\ldots,\hat{U}_n,\hat{t}_1,\ldots,\hat{t}_n)$.

4.3.2 Translation of Binder Symbols

The last section covered most of the constructs of *UDL*, in particular, type quantifications by reducing them to object quantifications. The translation of formulas with binders remains to be defined now as the decision procedure does not support the concept of uninterpreted variable-binding symbols.

As an example let us look at the binder symbol $\mathsf{setComp} : \alpha \times \mathsf{bool} \to \mathsf{set}(\alpha)$ for the set comprehension over a predicate; it has already been mentioned in Section 2.3.3. The term $(\mathsf{setComp}\ x^{\mathsf{nat}}.(\exists y^{\mathsf{nat}}.x \doteq y * y))$, for instance, describes the set of all (natural) square numbers. The binder is axiomatised in terms of the predicate $\mathsf{in} : \alpha \times \mathsf{set}(\alpha) \to \mathsf{bool}$ as[2]

$$(\forall x^T.\ \mathsf{in}(x, (\mathsf{setComp}\ x^T.\varphi)) \leftrightarrow \varphi) \quad \text{for all } T \in \mathcal{T}, \varphi \in \mathrm{Trm}^{\mathsf{bool}} \text{ with free variable } x^T\ . \tag{4.1}$$

For the sake of comparison, let us consider the defining axiom of the polymorphic function symbol singleton:

$$(\mathbb{V}\alpha.\forall x^\alpha.\forall y^\alpha.\mathsf{in}(x, \mathsf{singleton}(y)) \leftrightarrow x \doteq y) \tag{4.2}$$

Note that while the definition (4.2) of the function symbol can be formulated as a single axiom, the formalisation (4.1) of the binder requires an *axiom scheme* in which the schematic formula φ can be instantiated by any formula over the free variable x^T. The schematic description (4.1) stands thus for a (countably) infinite set of formulas.

This situation is prototypical for defining axioms for binders. Since binders do not receive single *values* as arguments for their evaluation, but *functions*[3], it is natural that their definitions do not refer to a single value but to an evaluation (via a schematic term).

This does not pose a problem for the interactive sequent calculus in which rules can be formulated schematically. In an actual sequent, schematic entities do not occur, and the schematic inference rules are matched against the non-schematic formulas on the sequent.

But how can we translate binders into the SMT logic which has no schematic entities? In Section 2.3.3, we have seen that predicate logic with binder symbols can be reduced to predicate logic without binders. However, this reduction was at the cost of obtaining a potentially infinite set of premisses. If we followed the idea of the

[2] As a side note it should be mentioned that this axiom for the class term constructor setComp is not open for Russell's paradox $\{s \mid s \notin s\} \in \{s \mid s \notin s\}$ since the type system forbids sentences of the form $\mathsf{in}(s,s)$.

[3] compare Definition 2.11 on page 22.

proof of Theorem 2.4, every usage $(b\ v.t)$ of a binder symbol b would give rise to a fresh function symbol $f_{(b\,v.t)}$ representing the binder's value. That would require that every schematic axiom over b be repeated for every relevant instance of b. If such a formalisation happened to be recursive (like, for instance, $(b\ x.t) \doteq (b\ x.t-1)+1$), an infinite number of additional formulas would have to be added since any freshly added formula triggers the addition of the next.

Moreover, it does not suffice to include the instances of the binder occurring in the problem into the translation. There are cases in which instances are needed which are not originally present on the scene. Consider the proof obligation

$$(\exists s^{\mathsf{set(nat)}}.\forall x^{\mathsf{nat}}.\mathsf{in}(x,s) \leftrightarrow (\exists y^{\mathsf{nat}}.x \doteq y * y))$$

in which we need to show the existence of the set of square numbers. This can be discharged if the term $(\mathsf{setComp}\ x.\exists y.x \doteq y * y)$ or rather its translation result $f_{(\mathsf{setComp}\ x.\exists y.x \doteq y*y)}$ is available during reasoning as it serves as the witness for s. But the term $(\mathsf{setComp}\ x.\exists y.x \doteq y * y)$ does not occur in the above proof obligation and therefore would not be translated. In an interactive proof using the sequent calculus, the user would instantiate the quantifier manually with this witness.

We will instead find another translation for binder symbols. To motivate this approach, let us for a moment break free from the constraints of first order logic and look at binders from the perspective of higher order logic. A binder is then a (higher order) function symbol receiving a function as parameter:

$$\mathsf{setComp}^{hol} : (\alpha \to \mathsf{bool}) \to \mathsf{set}(\alpha)$$

Using λ-expressions to construct functions, the question whether 16 is a member of the set of square numbers can then be formulated as the higher order term

$$\mathsf{in}(16, \mathsf{setComp}^{hol}(\lambda x.(\exists y.x \doteq y * y))) \tag{4.3}$$

and the symbol $\mathsf{setComp}^{hol}$ can be axiomatised using the higher order axiom

$$(\forall F^{\alpha\to\mathsf{bool}}.\forall x^{\alpha}.\mathsf{in}(x, \mathsf{setComp}^{hol}(F)) \leftrightarrow F(x))\ . \tag{4.4}$$

This closely resembles the axiomatisation in (4.1) with the difference that the meta-quantification over all formulas φ has been replaced by the higher-order quantifier over $F^{\alpha\to\mathsf{bool}}$. But (4.4) is no longer a schematic representation of a set of formulas but a single formula.

If we approximate[4] functional expressions and λ-terms appropriately in first order logic, a suitable translation for binder symbols emerges. This approximation of functions can be found in the well-known theory of arrays (as introduced by McCarthy, 1962). This theory is very slim and is supported by most SMT solvers. The binary type

[4] It is evident that they cannot be fully specified since this is precisely the difference between first and higher order logic.

constructor $\mathsf{array} \in \hat{\Gamma}$ is used to construct arrays and the polymorphic access function $\mathsf{select} : \mathsf{array}(\alpha,\beta) \times \alpha \to \beta$ reads a value from an array. The first-order approximation of the higher order definition (4.4) reads, hence,

$$(\forall a^{\mathsf{array}(\mathsf{universe},\mathsf{bool})}.\forall x^{\mathsf{universe}}.\ \mathsf{instance}(x,\hat{\alpha}) \to \widehat{\mathsf{in}}(\hat{\alpha},x,\widehat{\mathsf{setComp}}(a)) \leftrightarrow \mathsf{select}(a,x)) \tag{4.5}$$

and is still a single formula.

Translating the λ-expression in (4.3) proves a little more intricate as the target logic does not have the according operator available. However, we can consider the formula

$$(\forall F^{\mathsf{nat}\to\mathsf{bool}}.\ F \doteq (\lambda x.\exists y.x \doteq y * y)\ \to\ \mathsf{in}(16,\mathsf{setComp}^{hol}(F)))$$

equivalent to (4.3) and expand the first equality which is an equality over functions. It can be replaced by an equality over all function applications due to the extensionality axiom of higher order logic.

$$(\forall F^{\mathsf{nat}\to\mathsf{bool}}.(\forall z^{\mathsf{nat}}.F(z) \doteq (\lambda x.\exists y.x \doteq y * y)(z))\ \to\ \mathsf{in}(16,\mathsf{setComp}^{hol}(F))) \tag{4.6}$$

The application $(\lambda x.\exists y.x \doteq y * y)(z)$ in (4.6) can be subject to a β-reduction. Replacing the higher order constructs in this formula by their first order counterparts, we can finally formulate the first-order translation of (4.3) as

$$(\forall a^{\mathsf{array}(\mathsf{universe},\mathsf{bool})}.(\forall z^{\mathsf{nat}}.\mathsf{select}(a,z) \doteq (\exists y.z \doteq y * y))\ \to\ \mathsf{in}(\widehat{\mathsf{int}},16,\widehat{\mathsf{setComp}}(a)))\ . \tag{4.7}$$

To present this proof obligation to the SMT solver, it needs to be reformulated as the satisfiability problem (4.5) $\wedge$ $\neg$(4.7). That allows us to skolemise variable a in (4.7). When Z3 is challenged with this problem, the solver can prove it valid.

This translation served as an instructive example for the general description of the translation of expressions with binders: Let us consider the sets $\mathrm{STrm}^T_{\Gamma,\Sigma} \supseteq \mathrm{Trm}^T_{\Gamma,\Sigma}$ of schematic terms of type $T \in \mathcal{T}^0$ augmented by an additional term constructor: schema variables (SVar). A scheme variable $S^T_v \in \mathrm{SVar}$ is a placeholder for a term of type T with $\mathrm{freeVars}(S^T_v) \subseteq \{v\}$. We augment the evaluation context by a schema variable mapping $\sigma : \mathrm{SVar} \to (\mathcal{D}^{\tau(T_v)} \to \mathcal{D})$ with $\sigma(S^T_v) : \mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T)}$. A schematic formula $\varphi \in \mathrm{STrm}^{\mathsf{bool}}_{\Gamma,\Sigma}$ is true if every evaluation of the schematic term variables with according functions satisfies it:

$$I,\tau,\beta \models \varphi \quad\iff\quad I,\tau,\beta,\sigma \models \varphi \text{ for all schema variable mappings } \sigma$$

In the introductory example, the result type of the binder symbol was bool. As, in general, binders may have any result type, this is now generalised to $\mathsf{universe}$. The projection $\mathsf{u2b} : \mathsf{bool} \to \mathsf{universe}$ can be used if a boolean result value is needed. A binder symbol $b : T_v \times T_1 \times \ldots \times T_m \to T \in \mathrm{Bnd}_\Sigma$ is translated to a *function* symbol $\hat{b} : \mathsf{type}^n \times (\mathsf{array}(\mathsf{universe},\mathsf{universe}))^m \to \mathsf{universe}$ in which n denotes the number of type variables in the signature of b. If a binder term occurs within a term, it is translated as follows:

$$(b^{[\{\alpha_1/U_1,\ldots,\alpha_k/U_k\}]}\ v.\ t_1,\ldots,t_n) \quad \text{becomes} \quad \hat{b}(\hat{U}_1,\ldots,\hat{U}_k,\hat{A}^v_{t_1}(\bar{w}_1),\ldots,\hat{A}^v_{t_1}(\bar{w}_n))$$

in which the $\hat{A}^v_{t_i}$ are fresh function symbols with result type $\mathsf{array}(\mathsf{universe},\mathsf{universe})$. If t_i contains more free variables $\bar{w}_i$ than only v, these variables are added as parameters to the Skolem-function $\hat{A}^v_{t_i}$. Whenever a new symbol $\hat{A}^v_t$ is introduced, it is defined by the axiom

$$(\forall \bar{w}.\forall v.\ \mathsf{select}(\hat{A}^v_t(\bar{w}),v) \doteq \hat{t})\ . \tag{4.8}$$

For translating a schematic axiom over b, the following is the case: Every schematic term variable S (in the scope of the variable v) becomes a universally quantified array variable $\hat{S}^{\mathsf{array}(\mathsf{universe},\mathsf{universe})}$. If S occurs in the schematic axiom, it is replaced by the function application $\mathsf{select}(\hat{S},v)$ in the translation.

An optimisation can be made if S is a direct argument to a binder symbol. According to the rules, an expression $(b\ v.S)$ would be translated to $\hat{b}(\hat{A}^v_S(\hat{S}))$ with the auxiliary definition $(\forall \hat{S}.\forall v.\ \mathsf{select}(\hat{A}^v_S(\hat{S}),v) \doteq \mathsf{select}(\hat{S},v))$ after (4.8). Due to the extensionality of arrays, this is equivalent to $\hat{A}^v_S(\hat{S}) \doteq \hat{S}$ and $(b\ v.S)$ can be translated as $\hat{b}(\hat{S})$ avoiding the introduction of unnecessary additional intermediate symbols.

We conclude the description of the translation by stating[5] the axioms which have been collected during the translation and which characterise the function symbols. The implementation uses more efficient axioms with the same effect (described by Leino and Rümmer, 2010, §2.0).

$$\widehat{Ax} := \quad \{(\forall t^{\mathsf{type}}_{\ldots}.\forall t'^{\mathsf{type}}_{\ldots}.\ \neg\hat{c}(t_1,\ldots,t_n) \doteq \hat{d}(t'_1,\ldots,t'_m)) \mid c,d \in \Gamma, c \neq d\} \tag{4.9}$$

$$\cup \{(\forall t^{\mathsf{type}}_{\ldots}.\forall t'^{\mathsf{type}}_{\ldots}.\ \hat{c}(t_1,\ldots,t_n) \doteq \hat{c}(t'_1,\ldots,t'_n) \to \bigwedge_{i=1}^{n} t_{\ldots} \doteq t'_{\ldots}) \mid c \in \Gamma, \mathrm{ar}(c) = n\} \tag{4.10}$$

$$\cup \{(\forall x^{\mathsf{universe}}.\forall t_1^{\mathsf{type}}.\forall t_2^{\mathsf{type}}.\ \mathsf{instance}(x,t_1) \land \mathsf{instance}(x,t_2) \to t_1 \doteq t_2) \tag{4.11}$$

$$\cup \{(\forall x^{\mathsf{int}}.\ \mathsf{u2i}(\mathsf{i2u}(x)) \doteq x \land \mathsf{instance}(\mathsf{i2u}(x),\widehat{\mathsf{int}}))\} \tag{4.12}$$

$$\cup \{(\forall x^{\mathsf{universe}}.\ \mathsf{instance}(x,\widehat{\mathsf{int}}) \to (\exists y^{\mathsf{int}}.\ \mathsf{i2u}(y) \doteq x))\} \tag{4.13}$$

$$\cup \{(\forall x^{\mathsf{bool}}.\ \mathsf{u2b}(\mathsf{b2u}(x)) \doteq x \land \mathsf{instance}(\mathsf{b2u}(x),\widehat{\mathsf{bool}}))\} \tag{4.14}$$

$$\cup \{(\forall x^{\mathsf{universe}}.\ \mathsf{instance}(x,\widehat{\mathsf{bool}}) \to \mathsf{b2u}(\mathsf{true}) \doteq x \lor \mathsf{b2u}(\mathsf{false}) \doteq x)\} \tag{4.15}$$

$$\cup \{(\forall \hat{\alpha}^{\mathsf{type}}_{\ldots}.\forall x^{\mathsf{universe}}_{\ldots}.\ \mathsf{instance}(\hat{f}(\hat{\alpha}_1,\ldots,\hat{\alpha}_n,x_1,\ldots,x_m),\hat{T})) \tag{4.16}$$

$$\mid f : T_1 \times \ldots \times T_m \to T \in \mathrm{Fct}, \mathrm{typeVars}(f) = \{\alpha_1,...,\alpha_n\}\}$$

[5] To shorten the presentation, repeated quantifications are abbreviated by ellipses. Hence, $(\forall x^T_{\ldots}.\varphi)$ stands for $(\forall x^T_1.\ldots\forall x^T_n.\varphi)$.

$$\cup \{ (\forall \hat{\alpha}^{\mathsf{type}}_{\dots} . \forall x^{\mathsf{array(universe,universe)}}_{\dots} . \, \mathsf{instance}(\hat{b}(\hat{\alpha}_1, \dots, \hat{\alpha}_n, x_1, \dots, x_m), \hat{T})) \tag{4.17}$$
$$\quad \mid b : T_v \times T_1 \times \dots \times T_m \to T \in \mathrm{Bnd}, \mathrm{typeVars}(b) = \{\alpha_1, ..., \alpha_n\} \}$$
$$\cup \{ (\forall \bar{w} . \forall v . \, \mathsf{select}(\hat{A}^v_t(\bar{w}), v) \doteq \hat{t}) \mid t \text{ occurs as argument to a binder} \tag{4.18}$$
$$\quad \text{with bound variable } v, \mathrm{freeVars}(\hat{t}) = \bar{w} \}$$

The axioms in (4.9) and (4.10) characterise the structure of the domain $\mathcal{D}^{\mathsf{type}}$ of the type of types. It cannot be formalised that the constructors generate all possible types, yet it is ensured that two ground types are translated onto two distinct objects in $\mathcal{D}^{\mathsf{type}}$. Axiom (4.11) makes sure the relation $\mathsf{instance}$ is a partial function. As matter of fact, it is expected to be a total function, yet in practice this partial statement suffices. The axioms in (4.12) to (4.15) ensure the mappings from the built-in integers and boolean are bijections from and to the respective parts of the universe appointed to the types int and bool. The axioms (4.16) and (4.17) fix the result types for function and binder symbols as required. Finally, the axioms in (4.18) have been collected during the translation and fix the semantics of the auxiliary array symbols standing for the term parameters of binders.

Theorem 4.1 (Correctness of the translation to SMT) *Let* $\Phi \subseteq \mathrm{STrm}^{\mathsf{bool}}_{\Gamma,\Sigma}$ *be a set of closed schematic UDL formulas without program formulas and updates.* $\hat{\Phi} \in \mathrm{Trm}^{\mathsf{bool}}_{\hat{\Gamma},\hat{\Sigma}}$ *denotes its SMT-translation and* $\widehat{Ax} \subseteq \mathrm{Trm}^{\mathsf{bool}}_{\hat{\Gamma},\hat{\Sigma}}$ *the set of the additional axioms introduced during the translation. Then*

$$\Phi \textit{ satisfiable} \implies \widehat{Ax} \cup \hat{\Phi} \textit{ satisfiable,}$$

that is, the translation is correct.

The converse direction of the implication in the theorem does not hold as some aspects of the translation have not been faithfully fully formalised in the translation. The extend of the type of types is one such aspect.

PROOF (SKETCH) We will leave aside the bijection mappings between theory-induced types and $\mathsf{universe}$ and vice versa. Let $(\mathcal{D}, I)$ be a semantic structure over Γ, Σ with $I \models \Phi$. In the following, we construct a canonical structure $(\hat{\mathcal{D}}, \hat{I})$ over $\hat{\Gamma}, \hat{\Sigma}$ with $\hat{I} \models \hat{\Phi}$. The type domain is chosen as $\hat{\mathcal{D}}^{\mathsf{type}}_{\hat{\Gamma}} = \mathcal{T}^0_{\Gamma}$, the domain of the universe $\hat{\mathcal{D}}^{\mathsf{universe}}_{\hat{\Gamma}} = \mathcal{D}_{\Gamma} = \bigcup_{T \in \mathcal{T}^0_{\Gamma}} \mathcal{D}^T_{\Gamma}$ as the union of all domains in $\mathcal{D}$. Assume that the array types $\hat{\mathcal{D}}^{\mathsf{array}(T_1,T_2)} = \{ f : \mathcal{D}^{T_1} \to \mathcal{D}^{T_2} \}$ are interpreted as the total functions. The predicate $\mathsf{instance} : \mathsf{universe} \times \mathsf{type} \to \mathsf{bool}$ is assumed true if the first argument is element of the domain of the second type argument, that is $\hat{I}(\mathsf{instance})(d,t) = \mathit{tt} \iff d \in \mathcal{D}^t, t \in \mathcal{T}^0$

Let $\alpha_1, \dots, \alpha_n$ denote the type variables in the signature of a function symbol $f : T_1 \times \ldots \times T_m \to T \in \mathrm{Fct}$ or binder symbol $b : T_v \times T_1 \times \cdots \times T_m \to T \in \mathrm{Bnd}$. Their evaluations in $\hat{I}$ are defined as follows using $\tau = \{\alpha_1/U_1, \dots, \alpha_n/U_n\}$

$$
\begin{aligned}
\hat{I}(\hat{f})(U_1, \dots, U_n, d_1, \dots, d_m) &= I(f^{[\tau]})(d_1, \dots, d_m) \text{ for } d_i \in \mathcal{D}^{\tau(T_i)}, U_j \in \mathcal{T}^0 && (4.19)\\
\hat{I}(\hat{b})(U_1, \dots, U_n, a_1, \dots, a_m) &= I(b^{[\tau]})(a_1, \dots, a_m) \text{ for } a_i : \mathcal{D}^{\tau(T_v)} \to \mathcal{D}^{\tau(T_i)}, t_j \in \mathcal{T}^0 \,.
\end{aligned}
$$

These definitions do not fully specify $\hat{I}(\hat{f})$ or $\hat{I}(\hat{b})$, we leave them underspecified for the case that the arguments do not adhere to the typing constraints provided in the type arguments. For the evaluation of the translation of closed formulas, this does not play a role. As the evaluation of terms adheres to their types (see Theorem 2.1), a translated term will only produce compliant type-value combinations. To satisfy the typing constraints in $\widehat{Ax}$, we set $\hat{I}(\hat{f})$ and $\hat{I}(\hat{b})$ to a fixed value in $\mathcal{D}^{\tau(T)}$ if the arguments and types to not go together. The auxiliary function symbols $\hat{A}^v_t$ standing for term arguments to binder symbols are evaluated as

$$
\hat{I}(\hat{A}^v_t)(\bar{d}) = \{e \mapsto \mathrm{val}_{\hat{I},\hat{\beta}[\bar{w} \mapsto \bar{d}][v \mapsto e]}(\hat{t}) \mid e \in \mathcal{D}^{\mathsf{universe}}\} \text{ for } \bar{w} = \mathrm{freeVars}(t) \setminus \{v\}. \tag{4.20}
$$

$\hat{I}(\hat{A}^v_t)$ is a function in $\mathcal{D}^{\mathsf{universe}} \to \mathcal{D}^{\mathsf{universe}}$ and hence an element of the domain $\mathcal{D}^{\mathsf{array(universe,universe)}}$. $\hat{I}$ is constructed such that it satisfies the axioms in $\widehat{Ax}$. By structural induction, we show that $\mathrm{val}_{I,\tau,\beta}(t) = \mathrm{val}_{\hat{I},\hat{\beta}}(\hat{t})$ with $\hat{\beta}(\hat{x}) = \beta(x), \hat{\beta}(\hat{\alpha}) = \tau(x)$. Exemplarily, the two most interesting cases for the universal quantification and the general binder are shown. The other cases are similar.

Let $t = (\forall x^T . \varphi)$ be a quantified formula. By induction hypothesis, we have the equality $\mathrm{val}_{I,\tau,\beta[x \mapsto d]}(\varphi) = \mathrm{val}_{\hat{I},\hat{\beta}[\hat{x} \mapsto d]}(\hat{\varphi})$ for any $d \in \mathcal{D}^{\tau(T)}$. The translated variable $\hat{x}$ is of type $\mathsf{universe}$ and has thus the wider quantifier range $\mathcal{D}^{\mathsf{universe}}$. For the guard $G := \mathsf{instance}(\hat{x}, \hat{T})$ which precedes the formula $\hat{\varphi}$ in the translation of the quantifier (see Table 4.4), we have by definition of $\hat{I}(\mathsf{instance})$ that G does not hold if the variable $\hat{x}$ has a value outside of $\mathcal{D}^{\tau(T)}$. This implies that φ is true for all $d \in \mathcal{D}^{\tau(T)}$ for x if and only if $\hat{\varphi}$ is true for all $d \in \mathcal{D}^{\mathsf{universe}}$ for $\hat{x}$.

Let $t = (b^{[U]}\ x.\ u)$ be a binder term for the binder symbol $b : T_v \times T_1 \to T$ with $\mathrm{freeVars}(u) = \{x\}$, the generalisation to higher arities and free variables is obvious.

$$
\begin{aligned}
\mathrm{val}_{I,\tau,\beta}(b^{[U]}\ x.u) &\overset{\text{Def. 2.11}}{=} I(b^{[\tau(U)]})(\{d \mapsto \mathrm{val}_{\hat{I},\beta[x \mapsto d]}(u) \mid d \in \mathcal{D}^{\{\alpha/\tau(U)\}(T_v)}\})\\
&\overset{\text{i.h.}}{=} I(b^{[\tau(U)]})(\{d \mapsto \mathrm{val}_{\hat{I},\hat{\beta}[\hat{x} \mapsto d]}(\hat{u}) \mid d \in \mathcal{D}^{\{\alpha/\tau(U)\}(T_v)}\})\\
&\overset{(*)}{=} I(b^{[\tau(U)]})(\{d \mapsto \mathrm{val}_{\hat{I},\hat{\beta}[\hat{x} \mapsto d]}(\hat{u}) \mid d \in \mathcal{D}^{\mathsf{universe}}\})\\
&\overset{(4.20)}{=} I(b^{[\tau(U)]})(\hat{I}(\hat{A}^x_u))\\
&\overset{(4.19)}{=} \hat{I}(\hat{b})(\mathrm{val}_{\hat{I},\hat{\beta}}(\hat{U}), \hat{I}(\hat{A}^x_u)) = \mathrm{val}_{\hat{I},\hat{\beta}}(\hat{b}(\hat{U}, \hat{A}^x_u))\\
&= \mathrm{val}_{\hat{I},\hat{\beta}}(\widehat{(b^{[U]}\ x.\ u)})
\end{aligned}
$$

At $(*)$, a certain notational laxness has been used by applying $I(b^{[\tau(U)]})$ to a function with a larger domain than expected, for the evaluation only the restriction is needed.

This concludes the induction proof from which we have learnt that if $I, \tau, \beta \models \Phi$, then also $\hat{I}, \hat{\beta} \models \hat{\Phi}$. Since Φ has no free variables, structure $\hat{I}$ indeed satisfies $\widehat{Ax} \cup \hat{\Phi}$. Note that if Φ contained free variables, we could not deduce that $\hat{I}, \hat{\beta} \models \hat{\Phi}$ for all assignments β: A variable x^T is translated as variable $\hat{x}^{\mathsf{universe}}$ which may have a value also outside $\mathcal{D}^T$. No corresponding assignment β would exist to back up the claim. □

Evaluation

The presented translation has been applied to a number of small experiments in which the binder terms appeared isolated. Fig. 4.5 lists a number of properties highlighting various isolated aspects of binders. The translation for all of them are discharged by the solver Z3 in virtually no time. The translation engine was also used in the upcoming larger case studies of the thesis. Together with useful inference rules in the calculus and this reduction, binder symbols could be efficiently used in the specifications to come.

$\mathsf{in}(16, (\mathsf{setComp}\ x.\exists y. x \doteq y * y))$	A concrete instance of a collected property can be identified.
$(b\ v.p(v)) \doteq (b\ w.p(w))$	Equality under α-conversion is resolved even if for an uninterpreted binder symbol b.
$(\forall x. p(x) \doteq q(x)) \to$ $(b\ v.p(x)) \doteq (b\ v.q(x))$	Two binder expressions result in the same value if their arguments are equal in all positions. Here specified using a universally quantified equality, ...
$(b\ v.v + 1) \doteq (b\ v.1 + v)$	... or by two terms which can be proved equal by the solver.
$(\forall\!\!\forall \alpha.(\mathsf{setComp}\ x^\alpha.\ \mathsf{false}) \doteq \mathsf{empty}^{[\alpha]})$	Since type quantification can be reduced to ordinary quantification, polymorphic properties can as well be proved.

Figure 4.5: Properties proved automatically with the binder translation

While this translation works well with the quantifier instantiation mechanisms of the SMT solver, there are limits to it. If in a schematic axiom, a schematic term does not occur as a direct subterm of the binder symbol but as a subterm to one of the arguments, the matching mechanisms of the SMT solver cannot be triggered.

For instance, consider the schematic rewriting lemma

$$t \in \mathrm{Trm}^{\mathsf{nat}},\ \mathrm{freeVars}(t) = \{n^{\mathsf{nat}}\} \implies (\mathsf{min}\ n^{\mathsf{nat}}.t + 1) \rightsquigarrow (\mathsf{min}\ n^{\mathsf{nat}}.t) + 1$$

for the binder $\mathsf{min} : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ denoting the minimum value over the naturals described by its argument term. The rewrite rule uses a schematic integer entity t which stands for any term which may contain the free variable n^{nat}. The schematic term t does not appear as direct argument to the binder symbol but as a subterm to the argument. According to the translation described above, the equivalent formulation as an axiom for the SMT solver is

$$(\forall t^{\mathsf{array}(\mathsf{nat},\mathsf{nat})}.\widehat{\mathsf{min}}(\hat{A}^{n}_{t+1}(t)) \doteq \widehat{\mathsf{min}}(t) + 1)$$

together with the definition of

$$(\forall a^{\mathsf{array}(\mathsf{universe},\mathsf{universe})}.\forall n^{\mathsf{nat}}.\ \mathsf{select}(\hat{A}^{n}_{t+1}(a), n) \doteq \mathsf{select}(a, n) + 1) \tag{4.21}$$

in which the schematic term t has been translated as $a : \mathsf{array}(\mathsf{universe}, \mathsf{universe})$, a variable. Please note that a is a free variable in t and as such has been added to the Skolem function $\hat{A}^{v}_{t+1}$ as a parameter. The difference between formulas (4.5) defining $\mathsf{setComp}$ and (4.21) defining the lemma over min is that in (4.5) the argument to the binder is a quantified variable ($a^{\mathsf{array}(\mathsf{universe},\mathsf{universe})}$) whereas the argument to min is a function application ($\hat{A}_{t+1}(t)$). Hence, the former can be matched against any application of the binder while the latter is only defined on the symbol $\hat{A}^{v}_{t+1}$ which precludes that a sensible instantiation is found by matching.

This translation can be significantly be improved if "hybrid" symbols are supported which may have both value and functional arguments. The lifting of expressions using array constants is not needed then for the value arguments. The binder sum presented in Section 2.3.1 is a candidate for such a hybrid symbol in which the first two arguments (the lower and upper summation limits) are value-arguments while the term over which the summation operates is a functional argument.

Leino and Monahan (2009) present a translation for a fixed set of binders (there called *comprehensions*), namely the sum, product, minimum and maximum over a integer term. They introduce fresh function symbols such that every comprehension term can be formulated as an application of the new function symbols. The function symbols are then axiomatised using recursive axioms.

The translation of binder symbols to first order logic is similar to the task that has to be performed when higher order formulas are submitted to automatic first order theorem provers. The *sledgehammer* tool of the Isabelle/HOL theorem prover, described by Meng and Paulson (2008), maps functions symbol to constants, function applications to expressions using select and λ-expressions using combinators. This translation can be used very successfully with automatic theorem provers over uninterpreted symbols without built-in theories, but cannot be applied to solvers with theory support as the structure of the formulas is not maintained.

Blanchette et al. (2011) present an extension of the sledgehammer mechanism targeting SMT solvers. This translation lifts lambda expressions and creates a new constant per expression and is, thus, similar to the approach presented here. Examples 4 and 5 in Figure 4.5 can not be proved by Isabelle's SMT translation. Number 4 not even by the mighty *sledgehammer* tool.

As a final note: One notable exception amongst the available decision procedures is the solver *yices*, described by Dutertre and de Moura (2006), which supports function types, higher order functions and λ-expressions in its native input language. With this more expressive logic, the last Skolemisation step (the introduction of $\hat{A}^x_t$) of the translation can be omitted and λ-expressions be used instead. The *in vitro* examples above can be solved automatically by *yices* but the first (as *yices* does not support non-linear arithmetic).

4.3.3 Efficiency Issues

Due to the technology employed in SMT solvers, they can handle quantifiers significantly less efficiently than automatic first order theorem provers (ATP). Many of the supported theories in solvers are quantifier-free, the combination of some theories works only for ground instances. ATP can work more liberally with free variables while SMT solvers instantiate quantifiers with ground terms which results in a set of ground formulas on which the algorithms operate.

The instantiation of (universally) quantified formulas is performed using heuristic strategies built-in into the solver but can be controlled by means of so called *matching patterns* (sometimes also called *triggers*). The design of these patterns is delicate (see Moskal, 2009).

The SMT solver is fed the translation of the axioms defined in the system, the translated sequent under inspection and the translation of selected inference rules. Not every rule can be translated, meta conditions can possibly not be translated and schematic rules are impossible to translate in general. Rewrite rules, on the other hand, can efficiently be translated as equalities by quantifying other their schematic entities. Their left hand sides are herein used as matching pattern. This encoding reflects the behaviour of the rule in sequent calculus: the left hand side is matched and then replaced by the right hand side, that is, their equality is assumed.

Attention must be paid to recursive definitions. If the rewriting rule

$$\mathit{fac}(n) \rightsquigarrow (\text{if } n \leq 0 \text{ then } 1 \text{ else } \mathit{fac}(n-1)) \tag{4.22}$$

for the factorial $\mathit{fac} : \mathsf{int} \to \mathsf{int}$ was eagerly translated as an axiom for the decision procedure, an infinitely running expansion involving $\mathit{fac}(n), \mathit{fac}(n-1), \mathit{fac}(n-2), \ldots$ might be triggered. Seen as a rewriting system, the axioms provided to the decision procedure, should be terminating to avoid dead ends in the proof search.

4.4 Translating Java Bytecode to *UDL* Proof Obligations

Although the intermediate verification language is limited in its expressive means, it is ideal for the verification of real programming languages. The continuing success of the verification methodology of Boogie used in many real-language verification systems: VCC (for C, Dahlweid et al., 2009), Spec# (for C#, Barnett et al., 2011), AutoProof (for Eiffel, Tschannen et al., 2011) and more, shows that a general-purpose intermediate verification representation is a suitable stage in the process of verification. It also proves that the intermediate verification representation of many different programming languages and specification approaches can be expressed in a common formalism. The idiosyncrasies of the programming languages are removed during the reduction to intermediate code. As a consequence of the elementariness of the target language, the resulting code will usually be more extensive than the original code.

An unstructured intermediate language is particularly appealing if the orginal language is unstructured itself. This is the case for Java bytecode, the result language of the compilation process of the Java compiler. Verifying Java bytecode has two advantages over verifying Java source code:

1. The subject of verification is the result of the compilation, a process which the sources are subject to on their way to execution. As such, bytecode is closer to what is actually executed than the not yet processed source code. Bytecode is an interpreted intermediate language itself and different from the actually executed machine instructions, but is closer to them than the Java sources.

2. The bytecode language is machine-oriented and has a much more precise semantics description than the feature-rich source code language.

   Despite the fact that the Java language has a relatively strictly described semantics, capturing it formally is a much larger task than modelling the semantics of the low-level machine-oriented intermediate compilation code.

As a potential drawback of working on bytecode rather than on the source text, we identified that the human user who needs to interact in the proof does not want to work on the low technical level but wants to identify artifacts they know from their source text. We will address this topic in Section 4.4.10.

In this section, we briefly present a prototypical translation from Java bytecode to *UDL*. The translation and its technical details are described in extenso in Felden (2012). It addresses the translation of Java bytecode without taking type parameters (generics) into account. Generic types are removed from the bytecode at any rate and would have to be reconstructed from additional information stored in the binary files. The incorporation of generic types would generally fit into the framework but would have gone beyond the scope of a feasibility implementation. Ulbrich (2007) describes theoretically how generics can be handled in Java verification. The translation is also limited to sequential single-threaded programs.

One difference to other similar translations is that debugging and other source code information delivered with the bytecode is exploited to keep as much structure of the program as possible despite the intermediate representation. In particular, the names that have been used in the original Java specification and code are retained in the intermediate verification code. This significantly faciliates an interactive verification process and makes it more transparent.

This implementation is not the first to perform Java program verification on the level of bytecode. Quigley (2003) presents a first formalisation of a subset of Java bytecode in a higher-order Hoare logic and Bannwart and Müller (2005) present a Hoare-style logic for a sequential bytecode kernel language similar to Java bytecode directly without intermediate representation. Barthe et al. (2008); Pavlova (2007) introduce the Java Applet Correctness Kit tool (JACK), an automatic prover reducing annotated Java bytecode to SMT verification conditions. The appealing fact of this approach described by Burdy et al. (2007) is that it also compiles the source code specifications (in JML) into a bytecode representation (then called the *bytecode modelling language BML*) and embeds it into the bytecode. A proved sound translation from Java bytecode with embedded BML annotations to the intermediate language Boogie has been developed by Lehner and Müller (2007) in the context of the Mobius project (Mobius Consortium, 2006),

Java is a *structural* programming language: Its code is composed by nesting statement and statement blocks; repetition is encoded using *while* and similar loop constructors. The language lacks an unconfined goto statement. On the other hand, it is not a *strictly structural* language in the sense that control flow always would follow the nested block structure like it would be the case in a pure while-language. The Java language possesses the following constructs which make the flow non-strictly structural:

- `break` and `continue` statements preempt the execution of a block and continue the program at the point after/before the preempted block.
- `return` statements preempt the execution of the entire method (normally).
- Throwing exceptions also preempts the execution of the surrounding block and either continues at a point after the block or terminates the method (abnormally).

This makes a structural analysis and execution of Java code less straightforward and costs extra efforts when modelling the non-strictly structured behaviour in structured modelling languages. For instance, the invariant treatment for loops is considerably more complicated in dynamic logic for Java (presented by Beckert et al., 2007) than in a strictly structural language. In classical dynamic logic, it must be established that the loop invariant is inductive for the loop body. The postcondition and the code outside the loop needs not be considered at this point. In dynamic logic for Java, the postcondition and the code after the loop cannot be completely left aside in this examination as a statement in the body may preempt the loop body and then make these parts relevant.

However, the Java language does not support *arbitrary* transfer of control between statements within the scope of a method. Java bytecode as the target language of the compiler, on the other hand, is a language which resembles assembly language and it supports unconstrained control flow within the bounds of a method. The translation from bytecode to *UDL* as another goto language can be performed canonically and locally by considering one bytecode instruction after the other. The control flow of the *UDL* result replicates the control structures of the bytecode program.

Adding debugging information obtained from the information present[6] in the bytecode files allows an interactive stepwise verification on the level of source code even though the actual executed code has been compiled twice by two separate tools. The effect is similar to what one experiences when using a source code level debugger which allows the developper to operate (display, inspect or modify memory structures) on the source code level even though the debugger tool actually operates on bytecode level.

When designing the translation target for Java verification, a number of decisions have to be made on how entities are modelled. In the upcoming paragraphs, we will point out the important design features made in the reference implementation. In many of the features, the translation resembles that presented by Lehner and Müller (2007), other decisions have been inspired by Schmitt et al. (2010) and Weiß (2010). This prototypical translation is still presented in relative detail since the case studies in Chapter 6 make use of the formalisation that will be introduced here. For a fully detailed description presentation and formal soundness proofs, see the aforementioned references.

For the sake of better readability, symbolic targets will be used in the *ivil* program fragments instead of numerical indices over which goto statements of the *UDL* programming language are defined. This is only a notional syntactic sugar which is also available in the implementation. The beginning of a comments is marked with a "#".

4.4.1 Class Hierarchy

The Java language is an object-oriented programming language; its type system is therefore hierarchically organised and has a subtype relationship on the reference types. The logic *UDL*, however, supports parametrised types but has no subtype relationship. All (ground) types have disjoint domains. This is why integral primitive types (`int, byte, short, char, long`) are mapped to the type int, and all reference types (that is, all types inheriting from `java.lang.Object`) are subsumed in one type ref. The reference types themselves are brought onto the object level in another type named 'type'. The subtype relationship is modelled as a binary partial order predicate $<: : \mathsf{type} \times \mathsf{type} \to \mathsf{bool}$ on the types. A typing function $\mathsf{typeof} : \mathsf{ref} \to \mathsf{type}$ assigns to any reference object its unique dynamic type.

If no arrays or generic classes are allowed, the type type can be modelled to be the finite set of all class types. The subtype relationship on them can be made explicit. If array types or generics are taken into consideration, the type system cannot be precisely formalised any more (see Schmitt et al., 2009). What can be modelled is the

[6] if the corresponding compiler switches were set

direct parent relationship between class types and their supertypes. The fact that class D is a direct subtype of class C is encoded as

$$(\forall T^{\mathsf{type}}.\ D <: T \leftrightarrow (D \doteq T \vee C <: T))$$

Multiple inheritance (having more than one directly extended class type) is not allowed in Java. A class may, however, possibly implement one or more interfaces. The inheritance constraint can faithfully be encoded in *UDL* as

$$\begin{aligned}(\forall T^{\mathsf{type}}.\forall U_1^{\mathsf{type}}.\forall U_2^{\mathsf{type}}.\ & T <: U_1 \wedge T <: U_2 \rightarrow \\ & T \doteq \mathsf{typeof}(\mathtt{null}) \vee U_1 <: U_2 \vee U_2 <: U_1 \vee \mathit{interface}(U_1) \vee \mathit{interface}(U_2))\end{aligned}$$

in which *interface* : type → bool is a predicate which holds if and only if the argument stands for an interface declaration.

We silently assume that the given bytecode program is *wellformed*, in particular, it must be statically ensured that for every operation in the code, the types of its operands are compatible with the operation. With the static welltypedness assumed, the application of type checking must only be performed in the comparatively rare cases of dynamic type checks. Only `instanceof` operations and explicit downcast checks entail assertions involving the subtype relationship. For every class or interface C a constant symbol `C` : type is introduced. To construct an array type over another reference type, the constructor arrayType : type → type is used. The type of the distinct `null` object is subtype of every reference type as `null` can be cast to any other reference type.

4.4.2 Integers

Numerical datatypes in Java have a fixed bitwidth (32 bit for the type `int` for instance) while the type int in *UDL* represents the mathematical integers $\mathbb{Z}$. For the specification of a program, the mathematical interpretation is usually the intended. We assume that the author of a specification does not take overflow into account deliberately when writing a specification since it is against the natural understanding of operations (as observed by Chalin, 2003).

The reference implementation maps numerical operations in the Java program to the corresponding operations on mathematical integers. This deviates from the actual semantics of the operation, but greatly lightens the burden of the formulating and discharging specifications. Overflow-accounting specifications tend to be more lengthy than those disregarding the issue. Typically, constraints on the range of integer values need to be assumed as additional preconditions.

Instead of modelling the overflow, the implemented translation has got a switch which allows the verification to be sound again. With the overflow-check switched on, the translated program still operates on mathematical integers, but it ensures that these operations always coincide with what happens in the Java semantics. Additional range assertions are added after each integer manipulating instruction, thus ensuring that the value of each local variable, primitive heap and stack locations are always within the bounds defined by their primitive type, hereby ensuring that their being modelled as mathematical integers is sound. With the bytecode as input to the translation, it is particularly easy to identify the points at which the checks need to be performed. This is significantly more elaborate when done on the source code level.

To model the Java semantics faithfully with integer arithmetic, all integer operations would have to be embedded in *modulo* operations normalising the value range, like formulated by Beckert et al. (2007, chapter 12). Modern SMT solvers support the theory of finite bitvectors and arithmetic over them. This datatype seems to be the ideal choice for a translation of Java primitive integer types: a decidable theory supported by many decision procedures matching the definition exactly. However, there is one drawback. As specifications are to be interpreted in mathematical integers, formalising the operations in the program as bitvector operations requires that there be a formal connection between the two integer representations. The theory of bit vector and integer arithmetic operate on the same set of values and decision procedures for the two theories are usually organised differently, and are, hence, difficult to combine.

The primitive floating point datatypes `double` and `float` are not supported.

4.4.3 The Heap

The central data structure on which all data (with exception of local variables) within a Java program resides, is the heap. The efficiency and usefulness of its encoding into logic is crucial to the success of the entire task of verifying Java programs. The formalisation in *ivil* follows the ideas presented by Schmitt et al. (2010).

Heap access can be modelled in several different ways in first order logic, each with advantages and disadvantages. See Weiß (2010, Chapter 4) for a comparison of various first order representations of Java heaps. The *explicit heap model* proved to be the most flexible amongst them. It is called explicit as it introduces the concept of heaps and field names as expressions (in two new types) into the logic. This way, the information on the value of all fields of all objects in an execution state is condensed into one semantic value. This allows storing of intermediate heap states in variables, comparison of heap states, and quantification over heap states. The price to be paid for this flexibility is that formulating read or write accesses to the heap may require lengthy expressions.

The encoding of a heap is a two-dimensional map in the sense of McCarthy's (1962) theory of arrays. The new types heap and $\mathsf{field}(\alpha)$ are introduced together with the function symbols $\mathsf{select} : \mathsf{heap} \times \mathsf{ref} \times \mathsf{field}(\alpha) \to \alpha$ for retrieving from and $\mathsf{store} :$

$\mathsf{heap} \times \mathit{ref} \times \mathsf{field}(\alpha) \to \mathsf{heap}$ for updating a heap. The two functions are connected by the axiom

$$(\forall \alpha. \forall \beta. \forall h^{\mathsf{heap}}. \forall r_1^{\mathsf{ref}}. \forall f_1^{\mathsf{field}(\alpha)}. \forall v^{\alpha}. \forall r_2^{\mathsf{ref}}. \forall f_2^{\mathsf{field}(\beta)}.$$
$$\mathsf{select}(\mathsf{store}(h, r_1, f_1, v), r_2, f_2) \doteq (\text{if } r_1 \doteq r_2 \wedge f_1 \mathrel{\dot{\approx}} f_2 \text{ then } v \text{ else } \mathsf{select}(h, r_2, f_2)))$$

which is implemented by schematic rewriting rules in *ivil*. The weakly typed equality has to be used here to allow for the comparison $f_1^{\mathsf{field}(\alpha)} \mathrel{\dot{\approx}} f_2^{\mathsf{field}(\beta)}$ of two differently typed field variables to be correctly typed.

The intended semantics of the type of heaps is $\mathcal{D}^{\mathsf{heap}} = \{f : \mathcal{D}^{\mathsf{ref}} \times \mathcal{D}^{\mathsf{field}} \to \mathcal{D} \mid x \in \mathcal{D}^{\mathsf{ref}}, y \in \mathcal{D}^{\mathsf{field}(\alpha)} \implies f(x,y) \in \mathcal{D}^{\alpha}\}$, the set of functions which obey typing. The evaluation of select-terms is then the function application $I(\mathsf{select})(f,x,y) = f(x,y)$ and that of store the function update $I(\mathsf{store})(f,x,y,v) = f[(x,y) \mapsto v]$. To reflect this fact, we will in the following write $h[r,f]$ instead of the term $\mathsf{select}(h,r,f)$ and $h[(r,f) := v]$ instead of $\mathsf{store}(h,r,f,v)$ whenever it is unambiguous.

A distinct program variable $h : \mathsf{heap} \in \mathrm{PVar}$ is used to denote the current heap state during program execution, additional heap program variables are introduced to denote heaps in various states, like before the method execution (h_{pre}) or before a loop or method invocation (h_{before}).

For every field `x` of a reference type in class `C`, a constant symbol `C::x` : $\mathsf{field}(\mathsf{ref})$ is introduced. Arrays are also reference objects and the indices into an array can be considered as fields. Therefore, there is a function symbol $\mathsf{idxRef} : \mathsf{int} \to \mathsf{field}(\mathsf{ref})$ which turns an array index into a field value. Respective constant and function symbols exist also for fields or array access of boolean or integer type (idxBool, idxInt). The length of an array is invariant and needs not be encoded as a field of the array type. Instead it is denoted by a function symbol $\mathsf{arrlen} : \mathsf{ref} \to \mathsf{int}$.

A predicate $\mathsf{wellformed} : \mathsf{heap} \to \mathsf{bool}$ is used to formulate *wellformedness* of heaps. The set $\mathcal{D}^{\mathsf{heap}}$ of heaps is an overapproximation of the heap states reachable by a Java program. Assuming $\mathsf{wellformed}(h)$ sorts those semantic values out which represent valid Java heaps. The wellformedness condition implies the following properties:

- Only finitely many objects are created on the heap.
- A reference type field of an object points to null or to a created object.
- The heap is well-typed: A location of reference type points to an object of its declared classtype (or a subtype of it).

4.4.4 The Operand Stack

Java bytecode is a stack-oriented programming language, that is, most instructions take their arguments from the operand stack and replace their results back on the stack. Local variables are kept in registers[7].

[7] Byte code differs here from other low-level languages which keep local variable on the stack while the instructions operate on operands presented in registers.

The idea is the following: Every position $n \in \mathbb{N}$ of the stack is represented by a row $(stack_n^{\mathsf{int}}, stack_n^{\mathsf{bool}}, stack_n^{\mathsf{ref}})$ of program variables all standing for the same position depending on the type of the data stored in the positon of the stack. For this model to work we need to rely on properties of Java bytecode guaranteed by the specification of Java bytecode (Lindholm and Yellin, 1997, Section 4.10):

1. *The stack is type-safe.* It is not possible to read a numerical value from a stack position to which a reference value has been written (or vice versa).
2. *Stack layout is consistent.* The types stored on the stack in a particular bytecode statement must be the same irrespective of the execution path by which it is reached. That is that every loop iteration must have the same stack layout; joining branches after a conditional block likewise.

These properties allow us to perform a simple static analysis prior to the actual translation such that we can learn for each operation which stack positions and which types are involved. For every instruction within the bytecode block of a method definition, the stack variables upon which it operates can be computed statically, even if it lies within a loop. Other register-based (or assembly) languages may not have this property; it was introduced to Java to be able to efficiently ensure the integrity of a running JVM (the so called *bytecode verification*).

Let us look at a very simple example illustrating this analysis and corresponding translation: Let P be a Java program which contains the statement `x1 = x2 + x3`. We assume that integer variable `x`*i* is stored in register i. The above statement is then translated to bytecode as the following sequence of instructions

JAVA BYTECODE

```
iload 2
iload 3
iadd
istore 1
```

JAVA BYTECODE – 4.2

first loading the summands x_1 and x_2 onto the stack, then adding the two values and storing the result back to local variable x_1.

From the static analysis of P we know the stack layout for the first instruction of Listing 4.2, in particular the number $n \in \mathbb{N}$ of used stack elements[8]. The direct translation of this yields the following piece of intermediate code:

IVIL

```
$stack_{n+1}^{\mathsf{int}} := x_3$
$stack_n^{\mathsf{int}} := stack_n^{\mathsf{int}} + stack_{n+1}^{\mathsf{int}}$
$x_1 := stack_n^{\mathsf{int}}$
```

IVIL

[8] that is, positions $stack_0, \ldots, stack_{n-1}$ are taken

Herein, the stack positions n and $n+1$ are filled with the values of x_2 and x_3 respectively, subsequently added up in stack position n and finally written to variable x_1. This reflects the behaviour of the stack machine exactly. The translation can be performed locally, instruction by instruction.

During symbolic execution (as described in Section 3.2), this *ivil* code fragment will become translated into the parallel update

$$stack_n^{int} := x_2 \parallel stack_{n+1}^{int} := x_3 \parallel stack_n^{int} := x_2 + x_3 \parallel x_1 := x_2 + x_3 \ .$$

As mentioned above, the stack will not be used within specification elements such as pre- or postconditions, loop invariants, etc. since the stack is a concept of which the source code level is oblivious. This implies that the update can be equivalently replaced by the following, much simpler single assignment update

$$x_1 := x_2 + x_3 \ .$$

This applies accordingly to all bytecode instructions involving the stack. After the symbolic execution, the update removes the necessity of stack variables. The symbolic execution, hence, performs an on-the-fly transformation of the bytecode into a stackless representation. An external pre-processing step (as, for instance, proposed by Demange et al. (2010)) could have resolved the stack externally, but the update simplification handles this automatically.

An alternative stack representation which would not need the stack analysis beforehand is to model the stack explicitly using one variable of an abstract datatype *stack* supporting the operations $push : stack \times \alpha \to stack$, $top : stack \to \alpha$ and $pop : stack \to stack$. This modelling has a disadvantage, however, when it comes to loops. Since all the information stored on the stack is stored in one single variable, this variable would be subject to an anonymisation in the loop invariant rules. The loop invariant would need to explicitly express that the lower indices in the stack remain unchanged. This can be omitted with the model using explicit stack variables since the flow analysis automatically detects which variables are touched and which are not.

4.4.5 Exceptions

The Java language has a sophisticated language-based exception-handling mechanism. An exception object can be raised using a `throw` statement and can be handled in `catch` blocks. Some exceptions may be thrown *implicitly*, that is without that an explicit `throw` statement gave rise to them. Implicit exception types include `NullPointerException`, `IndexOutOfBoundsException`, `ArrayStoreException`.

In the logic, thrown exceptions are assigned to the program variable $exc : \mathsf{ref}$ holding the currently active exception or `null` if no exception has been raised. The control flow transfer at throw statements happens by translating the exception jump target table into a sequence of *UDL* statements.

It is often considered bad programming style if code can implicitly give rise to exceptions. Any exception thrown should have gone through an explicit statement.

That indicates that the error is intended and deliberately taken into consideration. For the translation to *UDL*, two modi can be considered: Either any implicit exception is considered a faulty program and rejected or the code for the handling of this is performed as if thrown explicitly. A heap access `o.field` for a reference variable *o* may hence be either translated as

IVIL

```
assert $\neg o \doteq \texttt{null}$
```

IVIL

rejecting a program with implicit exceptions, or as

IVIL

```
 goto null non_null
null:
 assume $o \doteq \texttt{null}$
 # create new NullPointerException object $nr$
 $exc := nr$
 # use the according exception handling table.

non_null:
 assume $\neg o \doteq \texttt{null}$
 # continue with the translation.
```

IVIL

fully handling the faulty case by creating a fresh exception and raising it. The reference implementation rejects implicit exceptions.

4.4.6 Design by Contract

This implemented Java verification system aims at a modular verification methodology, that is, every program method should be verified separately. This entails that every method must also be formally specified. Formal contracts – in the sense of *design by contract* (Meyer, 1988) – bind invoking and invoked method together. A formal method contract for a method $\texttt{T m(T}_1\texttt{ p}_1\texttt{,...,T}_n\texttt{ p}_n\texttt{)}$ in class `C` is comprised of

1. a precondition $pre_m \in \mathrm{Trm}^{\mathsf{bool}}$ describung the states in which *m* may be called,

2. a postcondition $post_m \in \mathrm{Trm}^{\mathsf{bool}}$ describing the states in which *m* may terminate,

3. a modification clause $mod_m \in \mathrm{Trm}^{\mathsf{set(ref)}}$ describing the set of objects whose fields may be changed by *m*,

4. an optional exception clause $signals_m \in \mathrm{Trm}^{\mathsf{type}}$ describing the class of Java exceptions which may be thrown by *m*,

5. a recursion variant $decreases_m \in \mathrm{Trm}$ used to provide evidence for the termination of programs.

All elements may refer to the heap and to the parameters of the method as well as the implicit `this` pointer. All but the postcondition are evaluated in the state at the beginning of the method execution. The postcondition is a two-state predicate and can evaluate both in the heap after the execution (using h) and in the heap before the execution (using h_{pre}). It can also use the special polymorphic program variable $\mathsf{res} : \alpha$ to refer to the result of the method. The proof obligation in dynamic logic for the correctness of a contract can be sketched as

$$pre_m \to \{h_{pre} := h\}[\![m]\!](post_m \land eqHeap(h,h_{pre},\{h := h_{pre}\}mod_m) \land \mathsf{typeof}(exc) <: signals_m) \,. \quad (4.23)$$

$eqHeap(h_1,h_2,S)$ is a predicate which is true if its two arguments $h_1,h_2 \in \mathrm{Trm}^{\mathsf{heap}}$ differ at most on locations which belong to objects in $S \in \mathrm{Trm}^{\mathsf{set}(\mathsf{ref})}$.

The sketch proof obligation leaves aside some vital preconditions which are guaranteed by the Java virtual machine without that they would have to be checked. Such conditions are called *free assumptions* and in this case include the wellformedness of the heap and of the arguments to the method. There are advantages if the proof obligation is not formulated as an implication like in (4.23) but as a *UDL* program with embedded assumptions and assertions. Thus, line number annotations can be made to the different conditions linking them more transparently to the original source code. The condition after the modality in (4.23) is split up into individual checks. If a single case fails, then it is easier to track down the problem.

IVIL

```
 assume $\neg this \doteq \mathtt{null}$                # dropped if m is a static method
 assume $\mathsf{typeof}(this) \doteq C$               # likewise
 assume $\mathsf{wellformed}(h)$
 assume $\mathsf{typeof}(arg_i) <: T_i$               # repeated for all reference type arguments
 assume $pre_m$
 $exc := \mathtt{null}$
 $h_{pre} := h$

 # the translated method body of $m$,
 # return statements and uncaught exceptions lead to the label endOfMethod

endOfMethod:
 assert $post_m$
 assert $exc <: \{h := h_{pre}\}signals_m$            # if a signals clause is given
 assert $exc \doteq \mathtt{null}$                    # if no signals clause has been given
 assert $(\forall \alpha. \forall o^{\mathsf{ref}}. \forall f^{\mathsf{field}(\alpha)}.\ h[o,f] \doteq h_{pre}[o,f] \lor \mathsf{in}(o,\{h := h_{pre}\}mod_m))$
                                                      # $= eqHeap(h,h_{pre},\{h := h_{pre}\}mod_m)$
```

IVIL – 4.3

With this formulation, a method must be "reverified" in any subclass not overriding it with new code. In line 2 exact coincidence of the type of the receiver and the class C are assumed. Alternatively, if line 2 is relaxed to assume $\mathsf{typeof}(\mathit{this}) <: C$, it is not needed to repeat the verification in subclasses. However, the proof obligation is significantly harder then since less is known about the receiver object.

In a modular verification context, it is vital to be able to specify which memory locations may be changed and which must remain untouched. Several methodologies have been developed addressing this the issue.

Separation Logic (O'Hearn and Pym, 1999; Distefano and Parkinson, 2008) introduces a new junctor in the logic which formulates separation by syntax. In ownership approaches (Clarke et al., 1998; Leino and Müller, 2004; Müller, 2002), operations on the heap are restricted to certain permissible cases. Ownership can be enforced by static checking, type checking or full verification.

The reference implementation addresses the framing problematic using *explicit dynamic frames* (Kassios, 2011) (also called *regions* (Banerjee et al., 2008a,b)). Dynamic frames are most liberal in their application but provide also the most specification notation overhead.

Framing clauses specify which objects are *modified*, not which are *assigned* to. A location for an object outside the modifies clause, may temporarily change its value as long it returns to the original value in the end. As an alternative to checking the framing condition after the method body, the checks can also be performed in situ right after a location has been written. That is, every assignment $h := h[(o, f) := v]$ is followed by an assertion assert $\mathsf{in}(o, \{h := h_{pre}\}mod_m)$. Yet, this checks for assignment instead of for modification which is a stricter condition than the former check. For the interaction, the immediate assignment check has the advantage that a possible violation of the modifies-contract can be located within the sources which is not considerably more difficult with the deferred check. The reference implementation has got a switch to choose between the two translations.

An important set that can be specified as modification clause is $\mathsf{freshObjects}(h) := (\mathsf{setComp}\ o.\ \neg h[o, created])$. It denotes the objects which have not yet been created in the heap provided as argument. A method whose modification clause contains $\mathsf{freshObjects}(h)$ is hence allowed to create fresh objects[9].

4.4.7 Method Invocations

The counterpart of proving a method contract correct in a modular setting is applying it to a method invocation. A method call $o.m()$ in Java source code is translated into an "invoke" bytecode instruction naming the method to be called. The receiver object and all arguments are put onto the operand stack before the invocation. A method call can be overapproximated using a contract defined and proved for the called method.

[9] This set corresponds to the semantics of a "pure" method in the Java Modeling Language

IVIL

```
1  assert $\neg stack_n^{\mathsf{ref}} \doteq \mathtt{null}$                  # receiver must not be null
2  assert $decreases'_m \prec \{h := h_{pre}\}decreases_n$ # if termination is examined
3  assert $pre'_m$
4  $h_{before} := h$
5  havoc $h$
6  havoc $res^T$                             # dropped if m is declared void
7  havoc $exc$
8  assume wellformed$(h)$                    # free assumption
9  assume $(\forall o^{\mathsf{ref}}.\ h_{before}[o, created] \to h[o, created])$ # no created object is ever deleted
10 assume typeof$(res^{[\mathsf{ref}]}) <: T$             # free assumption if the result
11                                           # type is a reference type
12 assume $post'_m$
13 assume $exc \doteq \mathtt{null}$                        # if no signals clause given
14 assume typeof$(exc) <: \{h := h_{before}\}signals'_m \wedge h[exc, created]$   # otherwise
15 assume $eqHeap(h_{before}, h, \{h := h_{before}\}mod'_m)$
16 goto exc no_exc
17 exc:
18   assume $\neg exc \doteq \mathtt{null}$
19   # use the according exception handling table.
20
21 no_exc:
22   assume $exc \doteq \mathtt{null}$
23   $stack_n^T := \mathsf{res}^T$                       # store the result onto the operand stack
```

IVIL – 4.4

The primed version of the contract elements in lines 3 (pre'_m), 12 ($post'_m$), 14 ($signals'_m$) and 15 (mod'_m) of Listing 4.4 stand for the version in which all *this* references and the method parameters have been adapted. The actual values reside in locations on the operand stack. The translation can statically compute which stack position the arguments are to be found in. For the invocation of a method `k(int p)` with one argument for instance, the general precondition $pre_k = h_{pre}[this, \mathtt{intField}] > p$ would be adapted by replacing the reference to *this* and p. The adapted precondition which would be assumed in line 3 reads $pre'_k = h_{before}[stack_{n-1}^{\mathsf{ref}}, \mathtt{intField}] > stack_n^{\mathsf{int}}$ wherein n is the index of the current top-most (head) element of the operand stack.

Pre- and postcondition have swapped their roles between the programs in Listings 4.3 and 4.4. While the precondition is assumed in the beginning of the proof obligation, it must be established when the method is called. The postcondition (together with the framing and exception clauses) may then be assumed in the after-state of the method call since their being true has been be proved in the proof obligation.

Replacing a method invocation by its contract is, in general, an *overapproximation* of the actual behaviour. For instance, for every trace $(I, 0), \ldots, (I', n)$ of the method body with $I \models pre_m$, it is guaranteed that $I' \models post_m$. But there can be pairs of states (J, J')

with $J \models pre_m$ and $J' \models post_m$ such that there is no corresponding run of the method body. This has a consequence: This translation can only be used if the program is examined for all traces using $[\cdot]$ or $[\![\cdot]\!]$ but cannot be used for $\langle\!\langle\cdot\rangle\!\rangle$, $\langle\cdot\rangle$. The box modalities formulate safety properties that hold for all runs and an overapproximation includes all cases. The diamond modalities formulate reachability conditions, in which one trace with a property (it fails) is searched for. An overapproximation bears the problem that the actual execution may have no trace with the property but the overapproximation can have. The proof would, in such cases, find a trace though there is none in the implementation. For such proof obligations, an underapproximating contract would be needed. For program verification, this is, however, not really a constraint since we usually prove that all traces do not fail and not the reachability of states.

There exists an alternative to applying a method contract: Unrolling the method body. In general, however, this way of handling method calls is *not* modular. At verification time, all extending classes of a type might not be known, and, hence, the implementation to be chosen for unrolling might not be available. A contract is valid also for all overriding implementations if we assume the *substitution principle* by Liskov and Wing (1994). Yet, there are cases in which method body unrolling is modular and thus acceptable: If the method body cannot possibly be refined in any inheriting class. This is the case if a method is declared `private`, `final` or if the entire class is declared `final`.

The *UDL* translation of the method body can be simply put in place in the translation of the invoking method:

IVIL

```
 assert $\neg stack_n^{\mathsf{ref}} \doteq \texttt{null}$
 $arg_1 := stack_{n+1}^{T_1}$          # assign the arguments to the method invocation from
 $arg_2 := stack_{n+2}^{T_2}$          # the stack to the variables $arg_1, \ldots$ of the parameters
 # ... repeated for all arguments

 # include the translation of the method body of the called method in which every
 # return statement or uncaught exception is redirected to this point:
afterMethodCall:
 goto exc noExc
exc:
  assume $\neg exc \doteq \texttt{null}$
  # jump to the according exception handler if defined or end program

noExc:
  assume $exc \doteq \texttt{null}$
  # continue execution ...
```

IVIL

Precautions may have to be taken to ensure that the local variables of the two methods are chosen disjoint. They may have to be renamed or chosen different by construction. Return instructions or uncaught exceptions statement of the called method must not terminate the calling method but return the control flow to the calling method. The method body must be followed by statements which handle a potentially uncaught exception thrown in the called method. See Section 4.4.5 for details. The stack indices must be shifted by the number of resident objects at method entry.

4.4.8 Object Creation

In Java, new objects can be allocated on the heap during the run of a program. This provides a challenge to the translation as in *UDL* all objects exist from the beginning on. The solution, also employed by Weiß (2010), is to use a purely verificational boolean class attribute `created` which stores on the heap whether or not an object is created on the heap. It must be ensured that this attribute is "monotone", that is, whenever an object has been created, its status cannot be changed[10] any more. Garbage collection is left aside in this model.

A Java constructor invocation `new C()` is decomposed into two separate steps on the bytecode level:

JAVA BYTECODE

```
new C
invokespecial C.<init>()
```

JAVA BYTECODE

First, a new object of the given class `C` is created, and then the synthetic method void `<init>()` is invoked on the freshly created object. The method call does not deviate from any other method calls and can be handled accordingly. The creation process is handled using a program variable nr : ref holding the reference to the fresh object.

IVIL

```
havoc $nr$
assume $\neg h[nr, created]$      # Choose an object reference which is not yet in use
$h := create(h, nr)$              # initialise all fields of $nr$ and mark it as in use
$stack_n^{\mathsf{ref}} := nr$    # push the freshly created reference onto the stack

# code for calling/inlining C.<init>()
```

IVIL

According to Lindholm and Yellin (1997, 2.5.1), prior to calling the initialising function, the fields of a freshly created object carry the default value of their respective

[10] See also the free assumption in line 9 of Listing 4.4.

type. In logic, this is captured by the heap constructor $create(h,o)$ which assigns to all fields of o the default value but true to the creation-flag field *created*:

$$(\forall \alpha. \forall o^{\mathsf{ref}}. \forall g^{\mathsf{field}(\alpha)}.\ create(h,o)[p,g] \doteq \\ \text{if } o \doteq p \text{ then } (\text{if } g \mathrel{\dot{\approx}} created \text{ then } \mathsf{true} \text{ else } \mathsf{defaultValue}^{[\alpha]}) \text{ else } h[p,g])$$

The polymorphic constant $\mathsf{defaultValue} : \alpha$ holds the default value for the argument type. The default value is fixed by the axiom

$$\mathsf{defaultValue}^{[\mathsf{int}]} \doteq 0 \quad \wedge \quad \mathsf{defaultValue}^{[\mathsf{bool}]} \doteq \mathsf{false} \quad \wedge \quad \mathsf{defaultValue}^{[\mathsf{ref}]} \doteq \mathtt{null}\,.$$

4.4.9 Specifications

Felden (2012) describes how specifications in the Java Modeling language (JML) can be translated into equivalent specifications in *UDL*. Although JML is the de-facto standard for specifying contracts for Java programs, this implementation operates on contracts in *UDL*. The rationale behind this decision is that in the face of formalism-crossing proof obligations in Chapter 5, a purely Java-oriented formal corset (like JML) would be disadvantageous.

The specification framework allows the annotation of the following specification elements:

1. *Method contracts* with the elements listed in Section 4.4.6.

2. *Model methods* which can be used to define heap dependent function symbols. JML class invariants and model fields can be implemented using model methods. Model methods are more flexible in that they can take additional arguments.

3. *Loop specifications* consisting of a loop invariant, a loop variant and a modification clause.

4. *Embedded ghost code* including embedded assertions which can be used to control the proof engine. Proof hints (as introduced in Section 4.2.5) may be used to fix the verification persistently from within the source code. Additional assignments to ghost fields and variables can be embedded as well.

Class invariants[11] are not a concept in their own right in the framework. There are fundamental questions open in the semantics of class invariants, as shown by Parkinson (2007). The approach of visible states as propagated by Chalin et al. (2005) proved not practical in a deductive setting. More recent approaches abolish the concept of class invariants altogether and replace it by specification functions (in the verification system *Dafny* by Leino, 2010a) or by model fields (see Weiß, 2010). In

[11] sometimes called *object invariants*

these approaches, the invariant needs to be mentioned explicitly if needed. Model methods can be used to realise a similar approach using *ivil*. Unlike in the above mentioned systems, however, wellfoundedness of the model method definitions is not checked.

The implementation adopts the notation of specification elements from JML. Specification elements are embedded in special comments embraced by the character sequences `/*@` and `*/`. The *UDL*-specifications used in the benchmarks of Section 4.4.11 do not use model methods and can all be expressed canonically in JML as well.

The most important autoactive annotation is the loop specification. A loop specification includes an invariant which must be inductive for the loop, a wellfounded variant and a modifies-clause denoting the set of objects whose locations may be modified by the loop body. Loop specifications are added as annotations to the translated program. In bytecode, the original structure of the Java sources has been replaced by unstructured code. The translation must hence find the point at which the invariant needs to hold by detecting the statement which performs the loop entry decision in the bytecode control flow graph. A simple code analysis can achieve this.

4.4.10 Bridging the Gap between Intermediate and Source Code

Since the objective of the translation is to obtain a result which can be used in an interactive verification process, special attention must be paid to retaining as much source code information as possible. But the starting point for the presented translation here is Java bytecode, already an intermediate representation. Thanks to the nature of Java bytecode and the format of bytecode class-files, it is possible to regain a lot of source code information from the compilation result.

Fortunately, the bytecode format has built-in capabilities to preserve source code information in the compilation result. This information is originally intended to enable dynamic debugging on source code level. But the same information used to connect a concrete execution of bytecode with the source code elements can be used to relate the source code entities with the statements in *UDL* programs. If the compiler is told to store debug information with the bytecode, lookup tables with source line numbers and names of local variables and method parameters are added to the saved files for the bytecode. Without the debug option, this information would be lost as the Java Virtual Machine itself does not have names for its registers or stack positions. The source line table can be used to connect the translated program statements and the original source code line number using the annotations available in *ivil*. The table for local variables can be used to name the created program variables according to their original name in the source code, increasing comprehensibility of the translation result.

In addition to this debug information, the modular design of the bytecode language preserves naming information as well. Since the virtual machine allows dynamic linking of classes, all references to methods, classes and fields are stored using the original symbolic names of these entities. In a less modularly compiled language, such references may not be given by name anymore but by numeric references.

We can therefore preserve the original class and field name constants in heap access instructions, method names in method invocations, class names in exception handling etc. in the translated *ivil* program.

Modern Java compilers do little or no optimisation at all on their resulting bytecode. They leave the optimisation to so-called *just-in-time* compilers which translate the bytecode to optimised native code right before executing it. This ensures a close semantic resemblance between sources and their bytecode which increases the comprehensibility of the verification conditions. In other systems with highly optimising compilers performing out of order execution, loop unwinding, and other modifications, this traceability would be considerably reduced.

4.4.11 Example and Evaluation

We conclude the presentation of the reference translation from Java bytecode to intermediate *ivil* verification code by an example and an evaluation.

Figure 4.6 shows a small Java method, its compilation result as a sequence of bytecode instructions and the ivil program which emerges from the translation of the bytecode to the intermediate language. It is evident that the translation made the program significantly longer, but the individual statements in the translation are of a finer granularity. The example method `incX(int p)` adds to an integer field `int x` in the class `C` the absolute value of the parameter `p` if that is not zero. The formal contract is given using *UDL* formulas within comments directly. The contract possesses one precondition (`requires`), one postcondition (`ensures`) and one modification clause (`modifies`). As it has no exception clauses, it is assumed that the method does not throw exceptions. The contract says that whenever the argument is not zero, the value of the heap location `this.x` strictly increases by invoking the method and at most the fields of the receiver object are modified.

The first lines up to Label1 are the preamble which sets the stage of for the method execution by assuming important known facts on the heap (in particular its well-formedness) and the invocation receiver `this`. The precondition $\neg p \doteq 0$ is assumed, too. The following lines implement the method body in the intermediate language. The last commands from LabelEnd onwards encode the checks which are to be performed after the method body: the exception clause (no exception thrown) in line 42, the postcondition in line 44 and the modification clause in line 46. The automatic verification of this tiny example happens instantaneously in *ivil*.

Evaluation We have applied the approach and the reference implementation to a number of standard benchmarks. These case studies were taken from the challenges presented in the verification contests VSTTE 2010 (Klebanov et al., 2011) and FM 2012. These benchmarks are of smaller complexity and size and were chosen to show the adequacy of this proof-of-concept implementation and to demonstrate that the dynamic logic over the intermediate language can be used for the verification of Java

JAVA & JAVA BYTECODE

```
  int x;
  /*@ contract
    @  requires $\neg p \doteq 0$
    @  ensures $h[this,\texttt{C::x}] > h_{pre}[this,\texttt{C::x}]$
    @  modifies $this$
    @*/
  void incX(int p) {
    if(p < 0) {
      p *= -1;
    }
    x += p;
  }
}
```

```
iload_1
ifge 7
iload_1
iconst_m1
imul
istore_1
aload_0
dup
getfield C::x
iload_1
iadd
putfield C::x
return
```

JAVA & JAVA BYTECODE

IVIL

```
program Java source ”C.jspec”
  sourceline 4
    assume $\neg p \doteq 0$
    assume wellformed$(h)$
    assume $\neg this \doteq \texttt{null}$
    assume $h[this, created]$
    assume typeof$(this) = C$
    $h_{pre} := h$
    $p_{pre} := p$
Label1:
 sourceline 9
    $stack_0^{\mathsf{int}} := p$
    $branchCond := stack_0^{\mathsf{int}} >= 0$
    goto Label2, Label3
Label2:
    assume $branchCond$
    goto Label4
Label3:
    assume $\neg branchCond$
Label5:
 sourceline 10
    $stack_0^{\mathsf{int}} := p$
    $stack_1^{\mathsf{int}} := -1$
    $stack_0^{\mathsf{int}} := stack_0^{\mathsf{int}} * stack_1^{\mathsf{int}}$
```

```
    $p := stack_0^{\mathsf{int}}$
Label4:
 sourceline 12
    $stack_0^{\mathsf{ref}} := this$
    $stack_1^{\mathsf{ref}} := stack_0^{\mathsf{ref}}$
    assert $\neg stack_1^{\mathsf{ref}} \doteq \texttt{null}$
    $stack_1^{\mathsf{int}} := h[stack_1^{\mathsf{ref}}, \texttt{C::x}]$
    $stack_2^{\mathsf{int}} := p$
    $stack_1^{\mathsf{int}} := stack_1^{\mathsf{int}} + stack_2^{\mathsf{int}}$
    assert $\neg stack_0^{\mathsf{ref}} \doteq \texttt{null}$
    $h := h[(stack_0^{\mathsf{ref}}, \texttt{C::x}) := stack_1^{\mathsf{int}}]$
Label6:
 sourceline 13
    $exc := \texttt{null}$
    goto LabelEnd
Label7:
LabelEnd:
    assert $exc \doteq \texttt{null}$
 sourceline 5
    assert $h[this, \texttt{C::x}] > h_{pre}[this, \texttt{C::x}]$
 sourceline 6
    assert $eqHeap(h, h_{pre},$
                    $\{h := h_{pre}\}($singleton$(this)))$
```

IVIL

Figure 4.6: An example Java program and its translations to bytecode and *ivil*

programs. The tool will also be used to discharge the proof obligations which arise from the refinement conditions in the case studies in Chapter 6.

Table 4.7 lists the benchmarks and their runtime on a Quadcore 2GHz machine with 4GB RAM. The needed runtimes are compared to the time that the KeY tools takes to verify a corresponding proof obligation. The KeY tool operates on the source level and must, hence, upon reading a proof obligation perform a lot of internal overhead work. Due to this, the trivial proof obligation $\vdash$ true takes more than 4 seconds to verify. Hence, the results for the KeY prover[12] are compensated for by this amount since the compilation to *ivil* code (which takes less than a second for each of these examples) has not been taken into account, too. The code together with the used specifications can be found in Appendix B.1.

Benchmark	Collection	*ivil*	KeY	
$\vdash$ true		0.5 sec	4.3 sec	–
Array Sum and Max	VSTTE 2010	5.2 sec	13.6 sec	9.3 sec
Longest Common Prefix	FM 2012	2.3 sec	10.6 sec	6.3 sec
ArrayList (4 POs)		30 sec	46 sec	29 sec
First in Linked List	VSTTE 2010	4.2 sec	33 sec	29 sec

Table 4.7: Benchmarks for *ivil* and comparison to KeY

4.5 Chapter Summary

The interactive verification system *ivil* has been presented in this chapter. The system implements the sequent calculus for *UDL* which has been proposed in the last chapter. *ivil* is both an interactive theorem prover for *UDL* proof obligations and an automatic verification system. The system has been implemented in Java and comprises a Java source code base of about 50.000 lines of code.

The graphical user interface is designed to incorporate ideas of other interactive theorem prover interfaces, in particular from the KeY tool (dynamic logic for Java, Beckert et al., 2007) and from Rodin (Event-B, Hallerstede, 2008). Its major novelty is the integration of the original source code together with the translated intermediate code. Concepts with which users are familiar from dynamic source code debuggers (breakpoints, stepwise execution) have been transferred into the interactive component of *ivil*.

The user interface provides a detailed insight on the goings-on under the hood of the verification engine and a tool to intervene on this level of technical detail. While

[12] The latest release 2.0 of KeY has been used for the benchmarks.

we deem this a necessity if intricate and challenging proofs are conducted, it is not always the level of detail which is to be chosen when an interactive system is desired. For many proof obligations no interaction should be necessary at all and for most obligations, the interaction should not require a look into the verification system but should happen on the level of the sources as annotations to them. The most widely accepted interaction which happens on the level of the source code annotations are loop invariants which are not part of the contract of the code, but help the verification tool to find a proof for the contract. Leino (2010b) has coined the term of *autoaction* for systems with this kind of interaction.

In purely automatic verification systems, a verification condition is generated and sent to an automatic prover. Since in *ivil*, the verification process can be interacted with during the course of the proof, there is the possibility to control the prover by annotations from within the program source code. We have presented a list of *proof hints* which can be annotated to statements in order to control the proof.

Though the calculus has been proved relatively complete in the last chapter, *ivil* does not rely on this completeness to discharge proof obligations which have been reduced to the underlying predicate logic. Instead, an industry-standard solver for satisfiability modulo theories (SMT) is driven to take care of proving the open proof goals. We have presented how, based on an idea by Leino and Rümmer (2010), *UDL* formulas are reduced to formulas in the simpler SMT input language. Particular attention has been spent on the efficient translation of binder terms. The translation has been proved correct.

Finally, a prototypical proof-of-concept implementation of a translation from Java bytecode to the intermediate language of *ivil* has been presented. There already exist similar translations and what sets this translation apart from them is that it manages to preserve source code information from the original Java source files, in the translation result. In particular, names of classes, methods, fields, method parameters and local variables can be retained, and intermediate code be brought in relation to the respective causing source code line. This makes the interactive verification of Java programs possible even if the program has been translated twice to an intermediate representation (first to bytecode, then to *ivil* code).

The implementation has been applied to standard benchmarks which could be verified automatically. On these benchmark, the presented verification system has proved competitive with the KeY tool, another semi-interactive Java verification system.

CHAPTER 5

Algorithm Refinement

In my humble opinion, the only
tool to master complexity is abstraction.
CLIFF JONES

In this chapter, a new methodology is devised which decomposes the task of verifying the implementation of an algorithm into two easier tasks:

1. *The algorithm is developed, specified and verified in abstract, mathematical terms.*
2. *A formal relationship between this abstract algorithm description and an actual implementation is established such that it is formally ensured that the code puts the abstract description into action.*

This process is called refinement. *We develop a notion of refinement using UDL and present how it can be used to verify method contracts. It is also shown that formal refinement with UDL coincides with the established notions of formal refinement.*

Chapter 6 will report on case studies with this new approach.

5.1 Separation of Concerns

When specifying and verifying a non-trivial algorithm implemented in a modern programming language, two difficulties come together and interfere with one another:

1. The complexity of understanding and fully formalising the *algorithm on a conceptual level*. This includes formalising the required precondition as well as coming up with a sufficiently strong postcondition; a task which can already be complex on its own. Even for a simple-looking specification, the required intermediate annotations may already exceed the length of the actual code considerably.

2. The complexity of the *implementation*. This includes the idiosyncrasies of the programming language, like side-effects in expressions, handling of underspecified or exceptional behaviour. Also the model of the data in implementations (often organised in a heap) has significantly more structure than in the mathematical model. Potential overlapping of heap data structures must be dealt with in the implementation verification.

The approach to be presented in this chapter separates these two concerns: The algorithm is formalised as a purely mathematical description in pseudocode removing any implementational detail from it. This algorithm is specified and verified on the higher level of abstraction.

An implementation is then written structurally coherent to the algorithmic description of the pseudocode. A refinement relationship between the implementation code and the abstract is established formally using coupling predicates formally connecting the abstract and the concrete model state spaces. The specification of the abstract pseudocode algorithm together with the coupling predicates lead then to a correct formal specification of the implementation.

Such a strict separation is not always feasible. It is a well-known fact that the verification of lightweight safety properties may be as difficult as a full functional verification and that the two aspects may not be separable. Consider an algorithm which performs a lengthy calculation of a number which is then used as denominator in a division. Showing that this number cannot be zero may require a full specification even if it seems to be a mere functional safety property. However, experience shows that there are many cases in which the complexity of the algorithm can be separated from the complexity of showing absence of error cases.

5.1.1 Pseudocode

We use *pseudocode* as the formal input language to describe algorithms on the abstract level. For the purposes of this work, pseudocode is a straightforward while-language over mathematical expressions as it can be found in algorithm engineering textbooks. Pseudocode has many advantages which make it the ideal choice of modelling language for algorithms:

- *Pseudocode is well suited for imperative descriptions.* In purely declarative description languages like the set-theoretic Z (Woodcock and Davies, 1996) or the relational Alloy (Jackson, 2006), algorithms can be specified by their effects (*what* is achieved) using before-after-predicates. Pseudocode allows a sequential decomposition of the work-flow elucidating *how* the effects are achieved.

- *Pseudocode is simple.* Effects which may occur in more complex languages do not show up in pseudocode: Aliasing of references, or overlapping of memory segments[1] are excluded by construction if the values are taken from abstract

[1] sometimes called the *abstract aliasing problem*

datatypes. If such effects are needed for the algorithm, they can be modelled explicitly but can equally be ignored if not relevant.

- *Pseudocode allows mathematical notation.* Mathematical expressions are an adequate tool to concisely and rigorously formulate complex facts. The richness of mathematical language may be hindering for the direct implementation of expressions in a programming language, but then, they can be refined to less involved expressions.

- *Pseudocode is self-explanatory.* The original intention of pseudocode language is to present algorithms in a comprehensible way (for instance, in a textbook). The abstraction capacity of the language as well as the simplicity of the available control-structures allow a convenient presentation of algorithms.

- *Pseudocode is widely accepted.* It is often used as the notation to present algorithms in various application fields and is not a formalism only known within the verification community.

- *Pseudocode is a good starting point for a refinement* to an imperative implementation. Many modern programming languages are imperative. If the target language of the algorithmic refinement is such a language, the modelling language should have the same property.

The pseudocode language which we will employ in this chapter is not, strictly speaking, a pseudo language. One major characteristic feature of pseudocode is that it lacks a formal syntactical and semantical corset but has its meaning been given by resorting to the reader's intuition. Since our goal is to perform formal refinements, this is unacceptable in our case. Hence, we sacrifice the freedom of being able to loosely formulate everything and formally fix the syntax and semantics of pseudocode. Appendix A.3 gives a syntax of the pseudocode used in this approach and its semantics by stating how to translate pseudocode to *UDL* programs. We still believe that, despite being formalised, the language matches the reader's intuition.

5.1.2 Code as Behavioural Specification

For functional verification, one challenge is to come up with good specifications. Normally, a specification is being thought of as a pair of pre- and postcondition or of a before-after-predicate which describe the results that an algorithm achieves. Both are purely declarative.

Sometimes such a purely result-oriented specification is not what is really meant to be specified. For an algorithm computing the shortest path between to places, it is a considerable difference to say *that* the result is the shortest distance between the places rather than to describe *how* the algorithm finds it. Of course, when seen from the outside, the former may be sufficient but when implementing the algorithm, the latter is a lot more valuable.

There may be situations in which it is impossible or at least practically infeasible to state a declarative specification. An algorithm traversing a data-structure may be canonically represented using a while-loop whereas its representation as a closed formula may be more involved, for instance, needing an operator for transitive closure or similar.

Of course, any specification which can be stated by a program can also be given by defining a recursive function. However, this may be unintuitive and may have little to do with the implementation. In such cases, an algorithm given in pseudocode may provide a formal, semantically unambiguous and comprehensible specification.

5.1.3 Verification versus Code Generation

An alternative to establishing a refinement relation between an algorithm and its implementation is to have the implementation code produced automatically from an abstract description by a code generator. If this generator can be verified correct (or adequately certified), the resulting code can be deemed correct by construction.

For code generation to be possible, the last description in the refinement chain must be of a form which can be automatically transferred into an implementation. It is in particular the B methodology (Abrial, 1996) whose models can span from high-level descriptions to synthesisable code descriptions, and industry-scale projects have already been realised using this approach (Behm et al., 1999, to name one).

However, such a proceeding is a good option only if the entire software system is modelled, refined and then its code generated. In such a scenario, the resulting code is fully under control of the code generator which can conduct the translation as it pleases – as long as it is correct.

In a more heterogeneous setting, the algorithm for which the refinement takes place may only account for a small part of a larger system. Other parts may not be subject to the idea of refinement and may, for instance, be provided by a third party which does not know about the internals of the code generator.

The realisation of data structures is not always necessarily canonical: An abstract sequence of values can be refined in many ways in the Java language: as an array, as an instance of a class implementing the interface `java.util.List` or by some other data structure implementation.

Also, it may be helpful to retain control over the implementational code such that it can realised more efficiently since generated code generally tends to be less efficient than manually written equivalents.

5.2 Refinement

Requirements for a complex (software or other) system are usually specified on more than one level of abstraction. In engineering, for instance, a first pencil sketch outlining an idea is refined via various intermediate stages to a fully precise construction drawing then used to eventually manufacture the part. Some properties of the object can be deduced already from a less detailed description of the object. To compute the dimensions of an article, for instance, no information concerning its interior are needed, the property can be computed from the outline of the object alone. The final construction drawing refines this outline sketch by adding detail information. The computation of the circumference made on the abstract sketch will still be valid.

We call the step from a more high-level description to a more detailed low-level description a *refinement*. Derrick and Boiten (2001, page 47) give a definition of a refinement called the *principle of substitutivity*: For an observer who is interested in a particular property, it must not be possible to tell if the description has been substituted by a refined description. For the engineer whose job it is to calculate the dimension of an article from the numbers in a drawing, whether or not details of the interior of the article are depicted does not make a difference. From the computations alone, one cannot tell which description has been used; substitutivity is given. The refinement may contain more details, but these do not invalidate calculations made on the more abstract level.

If the system is modelled using a formal description method with defined semantics, the refinement can be made equally formal by requiring that a refinement preserve the properties of a system. The benefit of a *formal refinement* is that a property shown for an abstract description then holds automatically also in its refined models. This is a desirable goal since it is very likely that the computation of a property can be performed more easily on an abstract system description than on the concrete.

System descriptions can be treated by more than one single refinement step. Usually, the development of a system starts with a rough overview description of the system followed by a series of refinements which all add detail and precision to various aspects of the system. This may serve documentation purposes as well as keeping the system comprehensible. We call the evolutionary series of sequentially refined descriptions a *refinement chain*.

5.2.1 Refinement in Formal Software Engineering

Formal refinement has a long standing tradition in formal methods and various related methodologies for refinement have been developed.

The Z specification language (Woodcock and Davies, 1996) is based on typed set theory and has a rich toolkit of operators which can be used to formulate declarative specifications. Operations are specified in form of before-after-predicates defining binary relations on the state space. The B method introduced by Abrial (1996) is inspired by Z and provides a methodology to refine system descriptions from a purely descriptive algorithmic description level down to actual executable code. Event-B (Hallerstede, 2008; Abrial, 2010) is an evolution of the B methodology in which the possibilities to formulate state transitions have been severely reduced and unified. Abstract State Machines (*ASM*, Börger and Stärk, 2003) provide a logical framework for definition of state transition system by means of a relatively small but general language of logical value updates. Morgan (1990) describes the *Refinement Calculus*, a method in which a system description is refined in a stepwise manner, beginning from a single pair of pre- and postcondition. Sequential decomposition of one operation by two consecutively executed operations is possible. The programs become more and more detailed with every refinement step, introducing conditional operations, loops, etc. The refinement calculus operates on the level of abstract algorithms (in the guarded command language devised by Dijkstra, 1975). Z and the refinement calculus are in a sense deliberately complementary: Z supports refinement of data structures, and has no means to sequentially decompose commands whereas the refinement calculus supports precisely this and does not incorporate data refinement.

All mentioned approaches propagate a multi-step refinement process in which the complexity is brought into the model step by step. Topmost descriptions are of a more declarative nature, explaining the objectives of a system in high-level notions. Refinements introduce more and more detailed aspects of the model, and towards the end of a refinement chain, the descriptions will become more and more technical, incorporating details of the implementation of the system, on the actual data structures or even on the way the code may be implemented.

Usually, the last refinement step in a refinement chain is the *implementation* which transfers the most detailed, yet still abstract, description into a working piece of executable code which can be run on a machine. This last step crosses a formalism border: The abstract descriptions are usually formulated in a language supporting abstract modelling concepts while the target formalism is usually a programming language. The last refinement step is, hence, not accompanied by a formal justification but done manually and inspected in a subsequent code review to establish that the implementation really realises the specification. Alternatively the code can be generated from the last element in the refinement chain by a code generator which may be proved to produce correct code refinements by construction. The last refinement step is often called the "jump to code".

In this chapter, we will attend to this last refinement step in particular and come up with a refinement method which allows also this last step to be done in a formal manner.

5.2.2 A Relational Notion of Refinement

The different established traditions of refinement share a common basic notion of what a refinement is though they differ in subtleties. Informally this definition says that given an abstract and a concrete system description, for every behaviour supported on the concrete level, there must be a corresponding behaviour on the abstract level.

We consider in the following two descriptions A (for abstract) and C (for concrete) of the same algorithm. Let us first look at the case in which they share a common state space S. The semantics of the algorithm descriptions is then given by two binary relations $R_A, R_C \subseteq S \times S$ containing pairs of corresponding before- and after-states. The algorithm may or may not terminate, fail or contain sources of indeterminism. The relations need hence not be partial (let alone total) functions.

A description may be given in various ways: It may be stated declaratively by a before-after-predicate φ describing the evolution of the data using a formula[2]. The state pairs $(s,s') \in R$ in the relation are in this case those in which φ holds.

But an algorithm may also be given as an imperative, sequential program π. In this case, the pair $(s,s') \in R$ is in the relation if and only if the program π started in state s may terminate in state s'. Programs can be indeterministically or non-terminating and may, hence, have none or several corresponding end states s' for a state s.

The notion of refinement is defined on these state relations: A concrete description refines an abstract description if the concrete behaviour is subsumed by the abstract behaviour, that is

$$C \text{ refines } A \iff R_C \subseteq R_A\ . \tag{5.1}$$

From this follows the above mentioned principle of substitutivity: A user observing the operation can never tell if a A has been substituted by C[3]. This means that the refined description A may have more possible behaviours than the refining concrete description, but that every behaviour of the concrete system is "backed up" by the abstract system.

This refinement condition requires that both descriptions operate on the same state space. This is, for example, the case for *algorithmic refinement* in which R_A is a declarative description of the state relation (a before-after-predicate) and C is an imperative implementation operating on the same data model. It is quite common in a refinement chain to first manipulate the data and declarative notions of the operations (the notion of *what* happens) and then finally to look at the way, in which the operation is actually performed (the notion of *how* it happens). Usually for a refinement step in which the data remains unchanged, the operation descriptions evolves considerably.

In an object oriented context, a method which implements a contract which has been specified at its interface falls into this category of refinement as well. The code is the concrete description and the formal contract defines the abstract behaviour. There is a difference to (5.1), however (one of the mentioned subtleties): The implemented method must have a behaviour for a pre-state s if the contract has one. But it may have additional behaviour outside the cases covered by the contract. Formally, this can be expressed as $\operatorname{dom} R_C \supseteq \operatorname{dom} R_A \ \wedge\ R_C\big|_{\operatorname{dom} R_A} \subseteq R_A$. Behavioural subtyping and refinement are very related notions, but not the very same.

In the following we need the *forward composition* $A\,;B$ of two binary relations which is defined as $A\,;B := \{(x,y) \mid (\exists z.(x,z) \in A \wedge (z,y) \in B)\}$. It is the commutation of the composition $\circ$ in the sense that $A\,;B = B \circ A$.

The refinement requirement (5.1) is often too strong a restriction, a refinement may introduce new aspects into the state space which are to be considered unobservable in the abstract state space; or the data representation changes and abstract and concrete

[2] A before-after-predicate φ is a two-state formula and may syntactically incorporate two copies of the symbols in the signature (primed and unprimed).

[3] They may be able to tell the *opposite*. If a state transition $(s,s') \in R_A \setminus R_C$ occurs, they know that A and not C must have been used. A refinement is called *complete* if this cannot be learnt either.

state are then incomparable. To allow for the more relaxed situation, we consider the state spaces of A and C to be potentially different and denote them as S_A and S_C respectively. However, the states are not fully independent from another, there exists a relationship between abstract and refined state space which is captured in a binary relation $r \subseteq S_A \times S_C$. Relation r specifies which states correspond to one another. The nature of the relation depends on the type of the refinement, it will be an injection or bijection in many cases; but in general, r needs not have such properties.

The refinement requirement from (5.1) can be reformulated with the relation r acting as a mediator between the state spaces:

$$C \text{ refines } A \iff r\,;R_C \subseteq R_A\,;r \tag{5.2}$$

This condition can also be found in the formal definition of Z (Woodcock and Davies, 1996, §16.4) and B (Abrial, 1996, §11.2.4). Relation r defines the correspondence of abstract and concrete states and is an integral part of the refinement step. It is called the *coupling relation* or simply the *glue*.

The condition can also be visualised in form of a diagram:

$$\begin{tikzcd}
a_0 \arrow[r, "R_A"] \arrow[d, "r"'] & a_1 \arrow[d, "r"] \\
c_0 \arrow[r, "R_C"'] & c_1
\arrow[from=1-1, to=2-2, phantom, "\subseteq"]
\end{tikzcd}$$

In the beginning there is an arbitrary abstract state $a_0 \in S_A$. For every path from a_0 first following the mediating relation r (reaching a coupled concrete state $c_0 \in S_C$) and then the concrete operation R_C terminating in the concrete final state $s_1 \in S_C$, there must be a path from a_0 first performing the abstract operation R_A (resulting in $a_1 \in R_A$) followed by relation r which reaches the same state $c_1 \in S_C$.

There are a number of special cases for certain shapes of the coupling relation r. The algorithmic refinement has already been mentioned; condition (5.1) is a special case of the more general (5.2) for the case that $S_A = S_C$ and $r = \mathrm{id}_{S_A}$.

A refinement is called a *data refinement* if it changes the representation of the algorithms data. Changing the value space of a description necessarily requires that the algorithm description also be changed, but we may claim, that no behaviour is added or removed just the representation changed. A data refinement is characterised by a functional relationship $a : S_C \to S_A$ called the *abstraction function* which describes how a refined state is mapped to a state in the abstract state space. This function must be total[4] but needs not be injective or surjective.

Note that while one abstract state may have no, one, or many concrete representation mapping to it, no concrete state can map to more than one abstraction. This

[4] or at least defined on $\operatorname{dom} R_C$

is deliberate and sensible: A more concrete data structure has more room for redundancies than a mathematical counterpart. Consider the prototypical example of implementing sets of naturals as doublet-free value sequences. The refinement function would be the function $a_{set} : Seq(\mathbb{N}) \to 2^{\mathbb{N}}$. The empty set would be represented only by the empty sequence, $a(\langle\rangle) = \emptyset$. Finite sets with a cardinality of at least 2 have more than one representation in sequents, like $a(\langle 1,2 \rangle) = a(\langle 2,1 \rangle) = \{1,2\}$ while infinite sets do not have a single preimage. The coupling relation used for a refinement would be the inverse a^{-1} of the abstraction function seen as a relation.

Introduction refinement is the a data refinement which introduces new aspects into the system by extending the abstract state space S_A by a space S_{new} such that $S_C = S_A \times S_{new}$. The abstraction function $a : S_A \times S_{new} \to S_A$ is given by the projection $a((s_a, s_{new})) = s_a$, the common part of the state S_A must be identical while the new data aspects are discarded.

The refinement categories are not strict, and a refinement step may belong to more than one of the categories. A concretion of the used data model is clearly a data refinement. If the new data structure reveals more structure and does cover all cases which have been allowed on the abstract level (like implementing sets with finite sequences, see above), it is at the same time an algorithmic refinement. If the refinement step crosses the language border, it is likely that all three categories will be touched.

Remember that the rationale behind doing refinement in the first place is to automatically pass on a property P which holds on the abstract level to the concrete level.

Such a property may be a set $P \subseteq S_A$ approximating the set of reachable states of R_A. This set can, for instance, be induced by a postcondition which has been proved to be valid for A. For the relations, this means that on the abstraction $\operatorname{rng} R_A \subseteq P$ holds. If the coupling relation is the identity, then this directly induces that the same postcondition also holds for the refined description $\operatorname{rng} R_C \subseteq P$. If a non-identical coupling relation r is to be considered, the property inheritance is only modulo this relation and reads $\operatorname{rng}(r \mathbin{;} R_C) \subseteq P \mathbin{;} r$.

Thus, the refined property depends on P and r which shows that the choice of relation r is crucial for the refinement step. Let us consider the corner case of the empty relation $r = \emptyset$. Both compositions ($r \mathbin{;} R_C$ and $R_A \mathbin{;} r$) are also empty and the subsumption holds trivially. When we say that C refines A, we must always additionally state the coupling relation r modulo which the refinement has been established.

5.2.3 A Programmatic Notion of Refinement

Morgan (1990) has another definition of refinement which is based on weakest preconditions of programs rather than on subsumption of relations. Program C refines a program A in this definition if the weakest precondition of A implies the weakest precondition of C for any postcondition φ. This requirement can be conveniently be expressed in dynamic logic as

$$C \text{ refines } A \quad \iff \quad \models [A]\varphi \to [C]\varphi \text{ for all } \varphi \in \mathrm{Trm}^{\mathsf{bool}} \tag{5.3}$$

This condition can also be interpreted as the fact that whenever the formula φ is a valid postcondition for A, it is also a valid postcondition for C. Any argument relying only on valid postconditions[5] for the program A applies by construction also to C: The substitutivity principle holds.

Quite astonishingly, definition (5.3) and the relational refinement definition (5.2) are equivalent. We will capture this fact in Observation 5.3 and prove it, but better postpone this until after we have come up with the necessary definitions in the next section.

As has been mentioned before, Morgan's refinement calculus does not support data refinement and definition (5.3) does thus not allow for a mediating coupling relation.

5.3 Refinement using *UDL*

With the notion of a formal refinement established, we can now set out to find a way to express refinement conditions using *UDL*. Let us therefore look at system descriptions which are given as two *UDL* programs $A, C \in \Pi$ and let us for the moment also assume that the programs are assertion-free. The goal is to find a *UDL* formula which is valid if program C is a refinement of program A.

Let the function $\mathrm{PV} : \Pi \to 2^{\mathrm{PVar}}$ assign to every program $\pi \in \Pi$ the program variables which occur in π. We assume from now on that $\mathrm{PV}(A)$ and $\mathrm{PV}(B)$ are disjoint sets, that is, that the two programs do not share program variables. This is a sensible restriction since the two programs are considered individual separate descriptions of the algorithm which cannot in any way interact with one another. There are cases in which A and C share program variables, as we have seen, for instance, for an algorithmic refinement. This can be mended by replacing every variable $p \in \mathrm{PV}(A)$ in A by a fresh variable p' of the same type yielding an modified program A', which performs the same algorithm, yet operates on a different set of program variables.

The state space $S_{\mathcal{D}}$ depends on the domain $\mathcal{D}$ of the semantic structure in which the program is started. It is then the set $S_{\mathcal{D}} = \{I \mid (\mathcal{D}, I) \text{ is a semantic structure}\}$ of all interpretations over the fixed domain. The before-after relation induced by an assertion-free program $\pi \in \Pi$ is hence the binary relation R_π defined as

$$R_\pi = \{(I, I') \in S_{\mathcal{D}} \times S_{\mathcal{D}} \mid \text{there exists a finite trace } (I,0), \ldots, (I',k) \text{ with } \pi[k] = \mathsf{end}\} \tag{5.4}$$

[5] or rather their weakest preconditions

which is the set of pairs of before- and after-states of π. Since π modifies at most the variables in $\mathrm{PV}(\pi)$, most of the state remains unchanged under the execution of π. Therefore the states in a before-after state pair (I, I') may differ at most on the evaluation of the program variables in $\mathrm{PV}(\pi)$.

In the following we assume that the domain $\mathcal{D}$ is fixed. In the context of program verification, for many types the domain is assumed predefined and fixed at any rate (for instance, $\mathcal{D}^{\mathsf{int}} = \mathbb{Z}$). Uninterpreted types may have more than one conceivable domain, yet for the examination of traces we only compare states with the same domain. We drop hence the index $\mathcal{D}$ whenever the context is unambiguous.

Let us consider again the two programs A and C for the refinement. Under the assumption of disjoint program variable sets, the interpretation function I of a semantic structure giving semantics to all function and binder symbols can be partitioned into three disjoint functions:

$$I_C := I\big|_{\mathrm{PV}(C)} \quad I_A := I\big|_{\mathrm{PV}(A)} \quad I_0 := I\big|_{\complement(\mathrm{PV}(C)\cup\mathrm{PV}(A))} \quad \text{with } I = I_0 + I_C + I_A \tag{5.5}$$

in which we use $+$ to denote the disjoint union of functions. A program can neither depend on the values of the variables of the other program nor modify them. In any successful trace, the part of the interpretation for the complementary program is always constant. Since it cannot play a role in the execution, replacing it with another evaluation yields another successful trace. The disjointness of the program variables used in A and C allows us, hence, to make the following observation.

Observation 5.1 (Independence of program variable evaluation) *Let $I, I', X \in S$ be interpretations with the common domain $\mathcal{D}$.*

1. *If $(I, I') \in R_A$, then $(I_0 + X_C + I_A, I'_0 + X_C + I'_A) \in R_A$ and $I_0 = I'_0$ and $I_A = I'_A$.*
2. *If $(I, I') \in R_C$, then $(I_0 + I_C + X_A, I'_0 + I'_C + X_A) \in R_C$ and $I_0 = I'_0$ and $I_C = I'_C$.*

PROOF Direct consequence of the disjointness $\mathrm{PV}(A) \cap \mathrm{PV}(C) = \emptyset$. □

This lemma summarises the effects that a program execution can have. At most the program variables of the executed program may have their value changed and the values of the program variables belonging to the other program in the refinement process, may be replaced without effect.

In (5.2) we introduced a refinement coupling relation r which brings together abstract and concrete state space. In the context of a logical embedding, this relation is now induced by a formula $\psi \in \mathrm{Trm}^{\mathsf{bool}}$ which can make use both of the program variables in $\mathrm{PV}(C)$ and $\mathrm{PV}(A)$, thus binding concrete and abstract state together. We call this formula ψ a *coupling predicate*. It induces the coupling relation $r_\psi = \{(I, I) \mid I \models \psi\} \subseteq S^2$. It may seem strange that r_ψ is a subset of the identical relation. But recall that $I = I_0 + I_C + I_A$, and since both programs each disregard one of these components (see Observation 5.1), this can be thought of as the asymmetric relation $\{(I_0 + I_A, I_0 + I_C) \mid I \models \psi\}$.

Using *UDL* we can express the relational inclusion from (5.2). The execution of two programs must be combined in one expression to achieve this. The inclusion requires that for every post state of the concrete execution, there exists a coupled post-state for the execution of the abstract code such that the coupling predicate holds. Unlike triples for the Hoare Calculus, formulas in dynamic logic may contain more than one program formula. In particular, we can *nest* modal operators, and can therefore formulate

$$\psi \to [C]\langle A\rangle\psi \tag{5.6}$$

as a formula in the version of *UDL* with postfixed assertions (see Section 2.5.3). The dual modality formula $[C]\langle A\rangle\psi$ in (5.6) has the meaning that *for every* trace of C reaching an end statement, *there is* a successful trace of A ending at an end statement such that in the end ψ holds. The composed program formula expresses precisely the refinement relationship:

Theorem 5.2 (Refinement condition in *UDL*) *Let $A, C \in \Pi$ be two self-contained assertion-free programs such that $\mathrm{PV}(A) \cap \mathrm{PV}(C) = \emptyset$. Program C refines program A modulo the coupling relation induced by the formula $\psi : \mathrm{Trm}^{\mathsf{bool}}$ (that is, $r_\psi \,;R_C \subseteq R_A \,; r_\psi$) if the formula $\psi \to [C]\langle A\rangle\psi$ is valid.*

PROOF For this proof we will for once use quantifiers also on the meta level outside the object level of the logic as this clarifies the presentation a lot. We first expand the premiss

$$\models \psi \to [C]\langle A\rangle\psi$$

into the implication on the interpretations[6]:

$$(\forall I. I \models \psi \implies (\forall I'.(I, I') \in R_C \implies (\exists I''.(I', I'') \in R_A \wedge I'' \models \psi)))$$

Expanding the interpretations into partial interpretations as defined in (5.5) yields

$$\Big(\forall I_0.\forall I_A.\forall I_C.\ I_0 + I_C + I_A \models \psi \implies \\ (\forall I'_C.\ (I_0 + I_C + I_A, I_0 + I'_C + I_A) \in R_C \implies \\ (\exists I'_A.\ (I_0 + I'_C + I_A, I_0 + I'_C + I'_A) \in R_A \wedge I_0 + I'_C + I'_A \models \psi))\Big) .$$

This already uses Observation 5.1 as the quantifiers range only over that partial interpretation that can be changed by the execution of a program, the remainder is left untouched.

Now we can introduce abbreviations for the states: $a = c = I$, $c' = I_0 + I'_C + I_A$, $a' = I_0 + I'_C + I'_A$. We can also rewrite the quantifiers accordingly. Quantifying over the reachable post-states c' from c is as good as quantifying over the partial component I'_C. This gives us the more readable condition

$$(\forall a.\forall c.\ (a, c) \in r_\psi \implies (\forall c'.(c, c') \in R_C \implies (\exists a'.(a, a') \in R_A \wedge (a', c') \in r_\psi))) .$$

[6] The quantification over interpretations ranges over all interpretations for the fixed domain $\mathcal{D}$.

This can be transformed using predicate logic equivalence transformations to

$$(\forall a. \forall c'.\ (\exists c.(a,c) \in r_\psi \wedge (c,c') \in R_C) \implies (\exists a'.(a,a') \in R_A \wedge (a',c') \in r_\psi))$$

which is then equivalent to the relational $r_\psi \mathbin{;} R_C \subseteq R_A \mathbin{;} r_\psi$ after collapsing the relational operators. □

What we have shown is that the refinement proof obligation (5.6) directly implements the notion of refinement introduced for the classical discipline of refinement known from Z or the B method. This very notion is also captured by the definition using weakest preconditions of the refinement calculus as proposed in Section 5.2.3. We now have the necessary instruments to formulate and show the congruence of the notions of refinement.

Observation 5.3 (Congruence of refinement notions) *Let $A, C \in \Pi$ be assertion-free programs which operate on the* same *set of program variables* $\mathrm{PV}(A) = \mathrm{PV}(C)$ *with their state relations R_A and R_C defined as in* (5.4)*. Then*

$$R_C \subseteq R_A \quad \iff \quad \models [A]\varphi \to [C]\varphi \textit{ for all } \varphi \in \mathrm{Trm}^{\mathsf{bool}}$$

PROOF (OF OBSERVATION 5.3) First we need a short argument why we can quantify over all sets of states S instead of quantifying over all postconditions φ. If there are infinitely many states, not every set of states can be described by a postcondition in a given interpretation.

However, since all interpretations have to be taken into consideration, we can find a way to represent them. Assume that the signature contains an uninterpreted predicate symbol P which will stand for an arbitrary set of states. The symbol must be applicable to all program variables of A as its truth value may depend on them. Let the symbol provide a parameter for every $p \in \mathrm{PV}(a)$. We can, hence, write $P(\overline{\mathrm{PV}(A)})$ to denote the predicate application to all program variables of A. The right hand side of the claim includes the case for $\varphi = P(\bar{p})$. Since P is uninterpreted, it may assume any semantics and all sets of states are included as postconditions.

Expanding the right hand side using the definition of the relation R_A and R_C gives the equivalent condition (again using quantifiers on the meta level to ease presentation)

$$(\forall s \subseteq S.\ \forall I.\ (\forall I'.\ (I,I') \in R_A \implies I' \in s) \implies (\forall I''.\ (I,I'') \in R_C \implies I'' \in s))$$

which, after some relational equivalence transformations[7], is equal to

$$(\forall s \subseteq S.\ \forall I.\ \neg I \in (R_A \mathbin{;} \complement s) \implies \neg I \in (R_C \mathbin{;} \complement s))\ .$$

[7] The forward composition is here also applied between a binary relation R and a set s and is defined as $R \mathbin{;} s = \{r \mid \exists x.(r,x) \in R \wedge x \in s\}$.

This can be rewritten using propositional logic and the fact that quantifying over all sets s is as good as quantifying over their complements $\complement s$ as

$$(\forall s \subseteq S.\, \forall I.\, I \in (R_C \,;s) \implies I \in (R_A \,;s)) \,.$$

It is an easy exercise to show that this is equivalent to $R_C \subseteq R_A$. □

This raises a natural question at this point: Why cannot the implication $[A]\varphi \to [C]\varphi$ for all φ be used as the refinement proof obligation directly? The uninterpreted Skolem symbol P from the proof would allow us to formulate $[A]P(\overline{\mathrm{PV}(A)}) \to [C]P(\overline{\mathrm{PV}(A)})$ as the proof obligation. However, if a refinement coupling predicate ψ is needed for instance in a data refinement, the proof obligation would become as complicated as $\psi \land [A](\forall \bar{x}_C.\{\bar{p}_C := \bar{x}_C\}(\psi \to P(\bar{p}_C))) \to [C]P(\bar{p}_C)$. It is the additional quantifier which makes this proof obligation less useful. After finishing the symbolic execution of A, the quantifiers need to be instantiated with a matching tuple of values from the abstract state space. When doing the compositional approach ($[C]\langle A\rangle\psi$), these values are instantiated automatically as a result of the symbolic execution of C, a vital advantage when it comes to automatic (or semi-automatic) refinement proofs.

This nesting of modal operators is similar to a refinement condition formulated by Abrial (1996) in the B-book. Since Abrial does not use dynamic logic, however, and has only the operator corresponding to the $[\cdot]$ modality at hand, the formulation there would translate to dynamic logic as $\psi \to [C]\neg[A]\neg\psi$ which is logically equivalent to (5.6).

There are two different modalities involved in the definition of the refinement. The reason for this is that we want to ensure that for every possible execution of the concrete code there is (at least) one admissible run of the abstract code. In general, it cannot be shown that every run through the abstract program makes the coupling predicate true. This also mirrors the fact that an implementation may reduce the degree of indeterminism of an abstract description.

5.3.1 An Example for Refinement in *UDL*

It is time for a first example for refinement using *UDL*. We want to refine the operation of adding a value n_A : nat to a set s_A : set(nat) of natural numbers. We start on the most abstract level with the model which has for this operation two program variables $\mathrm{PV}(A) = \{n_A, s_A\}$. The abstract program A_{insert} is the obvious operation which adds the value n_A to the set using a single assignment to update s_A.

IVIL – A_{insert}

```
s_A  := union(s_A,singleton(n_A))
```

IVIL – 5.1

This program is now submitted to a data refinement changing the representation of a set towards an implementation by modelling it as a finite sequence of numbers. The concrete program variables are $\mathrm{PV}(C_{insert}) = \{n_C : \mathsf{nat},\ l_C : \mathsf{seq}(\mathsf{nat})\}$.

The refined program also optimises the process since it appends the value n_C to the sequence only if it has not yet occurred in it. When regarding the sequence as a set of values, adding a value already present for a second time is not necessary. The refinement is, hence, not a pure data refinement as it changes both the data and the structure of the program. The refinement step could be broken down into two steps which could then be attributed to one category only.

IVIL – C_{insert}

```
if $\neg(\exists i^{\mathsf{nat}}.\ i < \mathsf{seqLen}(l_C) \land \mathsf{seqGet}(l_C,i) \doteq n_C)$
then $l_C := \mathsf{seqAppend}(l_C,n_C)$
end
```

IVIL – 5.2

To formulate the coupling predicate binding the state of A_{insert} and C_{insert} together, we must find a way to express that the values in the list can be seen as a set. This can, for instance, be done by using a function $\mathsf{seqAsSet} : \mathsf{seq}(\alpha) \to \mathsf{set}(\alpha)$. This polymorphic function symbol is defined to result in the set consisting of the entries of the sequence by the polymorphic axiom $(\forall\alpha.\forall x^{\alpha}.\forall l^{\mathsf{seq}(\alpha)}.\ \mathsf{in}(x,\mathsf{seqAsSet}(l)) \doteq (\exists n^{\mathsf{nat}}.n < \mathsf{seqLen}(l) \land \mathsf{seqGet}(l,n) \doteq x))$. We employ seqAsSet to state the coupling predicate $\psi_{insert} := s_A \doteq \mathsf{seqAsSet}(l_C) \land n_A \doteq n_C$.

This refinement is obviously functional since for every abstract variable, we give an evaluation term to compute it from the concrete values. Had we had not the function seqAsSet available, we could have instead used the formula $(\forall x^{\mathsf{nat}}.x \in s_A \leftrightarrow (\exists n^{\mathsf{nat}}.n < \mathsf{seqLen}(l_C) \land \mathsf{seqGet}(l,n) \doteq x)$ equivalent to $s_A \doteq \mathsf{seqAsSet}(l_C)$. The coupling relation induced by the predicate without seqAsSet is still functional but this fact is far less easy to observe and it cannot be as easily exploited. We will later see that coupling predicates which contain equalities can be considered a substitution (see Observation 5.12 in Section 5.5.1).

The additional equality $n_A \doteq n_C$ is needed to couple the input variables of the abstract and the concrete program. Even though the semantics of the input value is not refined between A and C, the two variables must be different by the disjointness requirement. By assuming their equality they are semantically made equal again. It is an often occurring phenomenon that the refined algorithm shares some variables with the abstract program. We then assume that there are two copies of the variables and that equality between them is assumed, like we have done here.

The refinement proof obligation $\psi_{insert} \to [C_{insert}]\langle A_{insert}\rangle\psi_{insert}$ for this data refinement can be discharged using the *ivil* verification tool with one interaction instantiating a set rule.

The example can be taken one step further using an algorithmic refinement which transforms the existential quantifier in the condition of the if statement of C_{insert} into a loop to scan for the first relevant entry. The resulting program I_{insert} (for implementation) is

IVIL – I_{insert}

```
1   $i_I := 0$;
2   while $i_I < \mathsf{seqLen}(l_I)$
3     inv $0 \leq i_I \leq \mathsf{seqLen}(l_I) \land (\forall j.\ 0 \leq j < i_I \rightarrow \neg\mathsf{seqGet}(l_I, j) \doteq n_I)$
4     var $\mathsf{seqLen}(l_I) - i_I$
5   do
6     if $\mathsf{seqGet}(l_I, i_I) \doteq n_I$
7     then
8       return
9     end;
10    i_I := i_I + 1
11  end;
12
13  $l_I := \mathsf{seqAppend}(l_I, n_I)$
```

IVIL – 5.3

In addition to the program variables $n_I : \mathsf{nat}$ and $l_I : \mathsf{seq}(\mathsf{nat})$ inherited from C_{insert}, the program I_{insert} introduces a loop counter variable $i_I : \mathsf{nat}$. The coupling predicate for this refinement is obvious: Since no data is refined, equality is to be assumed: $n_C \doteq n_I \land l_C \doteq l_I$. The variable i_I has no counterpart in C_{insert}, and thus does not occur in the coupling predicate.

Program I_{insert} is less comprehensible in comparison to its predecessors A_{insert} and C_{insert}. The shorter program C_{insert} is an example of a functional specification which is not given as a postcondition but in form of a program itself. The small program is at least as good a specification for I_{insert} as a postcondition involving a conditional expression would be.

This program is also an example of a non-strictly structurally composed program. The return statement in line 8 abruptly terminates the program from within the loop. Using a purely structural while language or regular programs, this would have to be paraphrased differently. The translation of the return statement in *UDL* is an end statement which may appear at any point in the statement list.

The loop invariant and variant annotated in lines 3 and 4 allow the verifier *ivil* to discharge the refinement proof obligation automatically. In Section 5.5.2, we will come back to the example by refining the short algorithm to a Java method.

5.3.2 More than one Coupling Predicate

With declarative languages like Z, B or Event-B, refinement is often used to model the behaviour of *reactive systems* in which a sequence of operations (or events) sequentially modify the system state. The formal connection between an abstract and a concrete state description does not depend on the operation which has been performed and should be the same relation before and after any operation. The coupling relation is an *invariant* of the refinement.

For the verification of algorithms using refinement, this is different: The notion of which states are considered equivalent may differ considerably between the beginning and the end of an algorithm. Before the algorithm, the input values need to correspond between the abstraction and the implementation. After the executions, the results need to correspond. The description of which states are related may have changed.

Formally this is reflected by relaxing the refinement condition to $r_{pre}\,;R_C \subseteq R_A\,;r_{post}$ for two relations $r_{pre}, r_{post} \in S^2$ instead of the one relation r in (5.2). The *UDL* proof obligation then reads $\psi_{pre} \to [C]\langle A\rangle\psi_{post}$ with two different coupling predicates for the pre- and the post-state. We call ψ_{pre} and ψ_{post} the *coupling precondition* and *coupling postcondition*.

5.3.3 Verification using Program Products

The refinement condition which we have considered so far is a special case of a verification task in which not one but two programs are examined at a time. In dynamic logic, these verification conditions follow the schematic form $\psi \to [P_1]\langle P_2\rangle\psi$ for two programs $P_1, P_2 \in \Pi$ and a coupling predicate $\psi \in \mathrm{Trm}^{\mathsf{bool}}$. The program variables used in P_1 and P_2 are disjoint. The coupling formula ψ connects the states of P_1 and P_2 in a problem specific manner.

Refinement is not the only verification challenge that can be formulated using program products:

Information Flow Information flow analyses examine whether in a piece of code information may flow from variables with a high security level to locations which have a low security level. A program has an information leak if such a flow is possible.

To formalise this in dynamic logic, Darvas et al. (2005) propose to compose two copies of the same program P (so-called *self composition*) to show that P has no illegal information flow. The coupling formula used for this case is the equality over all program variables $L \subseteq \mathrm{PVar}$ which are considered to be of a low security level. The formula to express the partial information flow security in dynamic logic is

$$\bigwedge_{l\in L} l \doteq l' \to [P]\langle P'\rangle \bigwedge_{l\in L} l \doteq l' \;. \tag{5.7}$$

P' is a copy of P in which all occurrences of program variables have been replaced by their primed version. The partial verification condition has the theoretical leak that P might terminate for one high input and not terminate for another high input thus leaking information. In the literature, self composition is often described as the combination $[P][P']$ of modalities, or equivalently as $[P\,;P']$. This is the same proof obligation for a deterministic program P. For an indeterministic program, condition (5.7) still makes sense: It requires that the set of indeterministically achievable value constellations for the low variables l be the same regardless of the values of the high input variables. This does not incorporate the probabilistic distribution of the values but only their theoretical reachability.

Scheben and Schmitt (2011) apply the self-compositional approach to dynamic logic for the full Java language.

Program Equivalence If only small changes are made to a piece of code, ensuring that its behaviour has not changed may be important. In a safety-critical situation, formal evidence that the semantics of a program has not changed, may complement or replace regression testing. In dependence on this established technique, the formal proof of program equivalence is called a *regression verification*. The original implementation serves here as a specification in this situation as it precisely describes how the system should behave.

The change made to the code needs not be made manually. Verifying the result of a *code optimisation* (unrolling of loops, code alignment, etc.) as it may appear within an optimising compiler is another use case of program equivalence proof obligations.

Like information flow, program equivalence is a special case of algorithmic refinement: The data structures and the language are the same, only the program texts evolves. The verification condition hence reads

$$\bigwedge_{p \in \mathrm{PVar}} p \doteq p' \to [P]\langle Q' \rangle \bigwedge_{p \in \mathrm{PVar}} p \doteq p'$$

for the original program P and the optimised implementation Q' (in which all program variables are primed). Again, the issue of termination is left aside in this verification condition as the original problem may diverge where the optimised terminates.

This approach with nested modalities is *not* suited to prove properties of two concurrently executed interleaved programs even if it is another verification condition with two programs. The difference is that they do not adhere to the disjointness of the program variables and that they are decidedly assumed to interact. Beckert and Klebanov (2013); Klebanov (2009) describe how this kind of program composition can be dealt with in dynamic logic.

5.4 Synchronised Loops

Any non-trivial algorithmic description will contain a repetition structure of some kind, either in form of a loop or of a recursive invocation of the same algorithm. Repetition structures are always the difficult part in a verification process. They cannot be simply expanded using symbolic execution if the number of iterations is not bounded by a constant known a priori.

In Chapter 3, calculus rules for the treatment of loops have been devised for the case of verification conditions with a single program formula. Of course, this very mechanism could also be applied if two nested program formulas are used like in the refinement condition $[C]\langle A\rangle\psi$. However, this would require that every loop be fully annotated with a sufficiently strong loop invariant and a suitable variant. The calculus for *UDL* could then be used to reduce the refinement proof obligation to a verification condition in the underlying logic. This would always be possible under the assumption that one can formulate a sufficiently strong loop invariant for every program. But the enormous disadvantage of this proceeding would be that the coupling between the two abstraction levels is never exploited. Following the idea of separation of concerns from Section 5.1, it would be appreciated if the concrete program need not be annotated with a loop invariant. The invariant should be automatically induced from the abstract level by the coupling relation.

We assume now that the abstract program and its refinement have a structural resemblance; after all, they represent the same procedure on different abstraction levels. It is therefore safe to assume that their program flows follow similar patterns. For instance, if a branching condition is reached in both programs, always corresponding branches are taken if the program resemble each other. In particular it seems sensible that every abstract loop corresponds to one loop in the implementation. The loops correspond in the sense that every iteration in the concrete world corresponds to precisely one loop iteration on the abstract level modulo a coupling predicate. We call a pair of thus coupled loops *synchronised*.

The first definition for synchronised loops operates on structured while loops instead of their unstructured counterpart to bring out the intuition behind the definition.

Definition 5.1 (Synchronised loops) *Two loops*

$$\mathsf{while}\ cnd_A\ \mathsf{do}\ body_A\ \mathsf{end}\quad \textit{and}\quad \mathsf{while}\ cnd_C\ \mathsf{do}\ body_C\ \mathsf{end}$$

are called synchronised modulo the coupling predicate ψ *if*

$$\psi \wedge cnd_A \wedge cnd_C \;\rightarrow\; [body_C]\langle body_A\rangle(\psi \wedge (cnd_C \leftrightarrow cnd_A))\ .$$

is a valid formula.

If the programs are deterministic, this means that the coupling predicate is a loop invariant for the consecutive execution of the two loop bodies. Therefore we call ψ in these cases a *coupling invariant*. Moreover, the equivalence of the loop conditions $cnd_C \leftrightarrow cnd_A$ is also a loop invariant.

Consider for example the small program (a) in Figure 5.1 which takes one value $n_1 : \mathsf{nat}$ and computes the factorial $f_1 = n_1!$ iterating the numbers from 1 to n with the counter variable k_1. The program in (b) performs the same computation as (a): Given the coupling predicate $n_1 \doteq n_2$ in the pre-state, the coupling predicate in the post-state therefore is $f_1 \doteq f_2$. However, the counter in (b) does not start at 1 but at 0. To compensate, the value $k_2 + 1$ is used as the factor instead of k_2 in line 4 of (b), and the condition in line 3 is stricter. The loops in (a) and (b) are obviously synchronised modulo the coupling invariant $\psi_{\text{a-b}} = (f_1 \doteq f_2 \wedge k_1 \doteq k_2 + 1)$.

Are the loops in (a) and (c) also synchronised? The third program computes the factorial $f_3 \doteq n_3!$ as well, but its counter k_3 *de*creases from n_3 to 1. The intermediate results in f_1 and f_3 during the course of the algorithm differ considerably even though their values are equal in the end. Yet, there is a coupling invariant

$$\psi_{\text{a-c}} = (f_1 \cdot \frac{n_1!}{(k_1 - 1)!} \doteq f_3 \cdot k_3!) \wedge k_1 + k_3 \doteq n_1 + 1$$

modulo which (a) and (c) are synchronised. This invariant uses the fact that the "missing factors" of f_i to $n_i!$ can be expressed using k_i in both cases. While the synchronisation between (a) and (b) is obvious and intuitive, programs (a) and (c) are synchronised in a far less obvious way and the coupling has a more technical character. The loops are synchronised but in a far less natural way. Situations like the latter synchronisation do not usually occur in algorithmic refinements as this would mean that the structure of a loop would be fundamentally revised in a refinement step. Normally, refinements are devised in such a manner that the steps build on one another and do not contradict each other so severely.

Assuming for the fourth variant that n_4 is even, program (d) is equivalent to (a). Its loop performs two loop bodies of (a) at a time. For any even input value, program (a) will need twice as many loop iterations as (d). The loops cannot be synchronised.

```
k1 := 1;               k2 := 0;              k3 := n3;            k4 := 1;
f1 := 1;               f2 := 1;              f3 := 1;             f4 := 1;
while k1 <= n1 do      while k2 < n do       while k3 > 0 do      while k4 <= n do
  f1 := f1 * k1;         f2 := f2*(k2+1);      f3 := f3 * k3;       f4 := f4*k4*(k4+1);
  k1 := k1 + 1           k2 := k2 + 1          k3 := k3 - 1         k4 := k4 + 2
end                    end                   end                  end
```

(a) Counter from 1 **(b)** Counter from 0 **(c)** Decreasing counter **(d)** Double step

Figure 5.1: Example for synchronised loops: Computing factorials

We will in the next section concentrate on synchronised loops since they capture the idea of separation of concern best. They only require that coupling invariants be specified. While synchronisation between loops simplifies refinement a lot, it is not a requirement. If loops are not synchronised or if their coupling invariant is too complicated or impossible to state, the approach with individual loop invariants can also be performed. If more than one loop is involved, they are handled separatedly and can be treated differently. The approach is flexible enough.

Synchronised loops play a less prominent role in the other kinds of product program verification conditions presented in Section 5.3.3: For information-flow analyses, every path must be considered, loops in which the loop condition depends on variables with high security level are not necessarily synchronised with themselves[8].

[8] that is, of course, with a copy of themselves to keep the program variables separate.

It is a core business of compiler optimisation to partially unroll loops for various performance reasons. Proving program equivalence in the face of such optimisations requires the flexibility that only a number of loop iterations are synchronised and others are not. While it would be theoretically possible to generalise the presented approach to such cases, we leave this aside here.

5.4.1 Synchronised Refinement for Dynamic Logic

After the introduction to synchronised loops, we will now search for suitable sequent calculus rules to deal with them. Before turning our attention to the case of unstructured dynamic logic, some preparatory thoughts on structured dynamic logic set the stage.

The inference rules for loop invariants in structured dynamic logic base on the induction rule

$$\frac{\psi \vdash [\alpha]\psi}{\psi \vdash [\alpha^*]\psi}$$

which says that an invariant formula ψ maintained by a program α for all possible executions is also maintained by a repetition of α. A similar sound rule exists for loops in program products.

Observation 5.4 (Synchronised structured refinement) *Let α and β be regular programs for structured dynamic logic with disjoint program variables. If α refines β modulo the coupling predicate ψ, then this is also the case for the repeated executions of α and β. The rule*

$$\mathrm{PVar}(\alpha) \cap \mathrm{PVar}(\beta) = \emptyset \implies \frac{\psi \vdash [\alpha]\langle\beta\rangle\psi}{\psi \vdash [\alpha^*]\langle\beta^*\rangle\psi}$$

is sound.

This rule over the composition of two programs is only sound if the program variables involved in α and β are disjoint. This requirement is indeed necessary as the following shows: $x \doteq 2 \to [x := x - 1]\langle x := x + x\rangle(x \doteq 2)$ is a valid formula and the two programs share the variable $x : \mathsf{nat}$. If the first program ($x := x - 1$) is executed twice, $x \doteq 0$ will hold in the post-state. No repetition of the second program ($x := x + x$) can ever establish the invariant $x \doteq 2$ again. The conclusion does not hold.

The rule is sound if the used program variables are disjoint and we will briefly sketch why. By n-fold iteration of the premiss $\psi \vdash [\alpha]\langle\beta\rangle\psi$ we also know that $\psi \vdash ([\alpha]\langle\beta\rangle)^n\,\psi$ for any $n \in \mathbb{N}$. Since the variable sets of the programs do not overlap, the implication $\langle\beta\rangle[\alpha]\psi \to [\alpha]\langle\beta\rangle\psi$ is valid and can be used to reorder the modal operators in $\psi \vdash ([\alpha]\langle\beta\rangle)^n\psi$ to $\psi \vdash [\alpha]^n\langle\beta\rangle^n\psi$. This is now evidence for the claim since for any number of repetitions of α there is a number of repetitions of β (namely the same number) such that the invariant ψ is reached.

The same rule with the modalities exchanged is not sound. The validity of $\psi \to \langle\alpha\rangle[\beta]\psi$ does not imply the validity of $\psi \to \langle\alpha^*\rangle[\beta^*]\psi$. As a counter example consider $\alpha = (\mathsf{havoc}\ x_\alpha)$ and $\beta = (x_\beta := x_\beta + 1)$ with $\psi = x_\alpha \doteq x_\beta$. Obviously

$\psi \to \langle\alpha\rangle[\beta]\psi$ is valid (it reduces to $(\exists v^{\mathsf{nat}}.v \doteq x_\beta + 1)$) while $\psi \to \langle\alpha^*\rangle[\beta^*]\psi$ (equivalent to $(\exists v^{\mathsf{nat}}.\forall c^{\mathsf{nat}}.v \doteq x_\beta + c)$) is unsatisfiable.

We now turn again to the while programs mentioned in Definition 5.1. Their translations into regular programs are as follows:

$$\mathsf{while}\ cnd_C\ \mathsf{do}\ body_C\ \mathsf{end} = (cnd_C\mathbf{?}\,;body_C)^*\,;\neg cnd_C\mathbf{?}$$
$$\mathsf{while}\ cnd_A\ \mathsf{do}\ body_A\ \mathsf{end} = (cnd_A\mathbf{?}\,;body_A)^*\,;\neg cnd_A\mathbf{?}$$

The programs contain the repetition operator $*$, and we can hence apply Observation 5.4 to them. The refinement rule shows that synchronised loops always are in a refinement relationship.

Observation 5.5 (Refinement of synchronised loops) *For two while-loops which are synchronised modulo a coupling invariant $\psi \in \mathrm{Trm}^{\mathsf{bool}}$ as defined in Definition 5.1, the concrete loop program refines the abstract modulo the coupling predicate $\psi \wedge (cnd_A \leftrightarrow cnd_C)$.*

This observation implies that it suffices to prove for a refinement verification condition that all involved loops are pairwise synchronised. If all loops are synchronised, it is not necessary to come up with loop invariants for the loops in the concrete program if coupling invariants can be stated which may be significantly simpler.

PROOF Let $\psi' := \psi \wedge (cnd_A \leftrightarrow cnd_C)$, $\alpha = cnd_C\mathbf{?}\,;body_C$ and $\beta = cnd_A\mathbf{?}\,;body_A$. The formula

$$\psi' \to [cnd_C\mathbf{?}]\langle cnd'_A\mathbf{?}\rangle\psi' \tag{5.8}$$

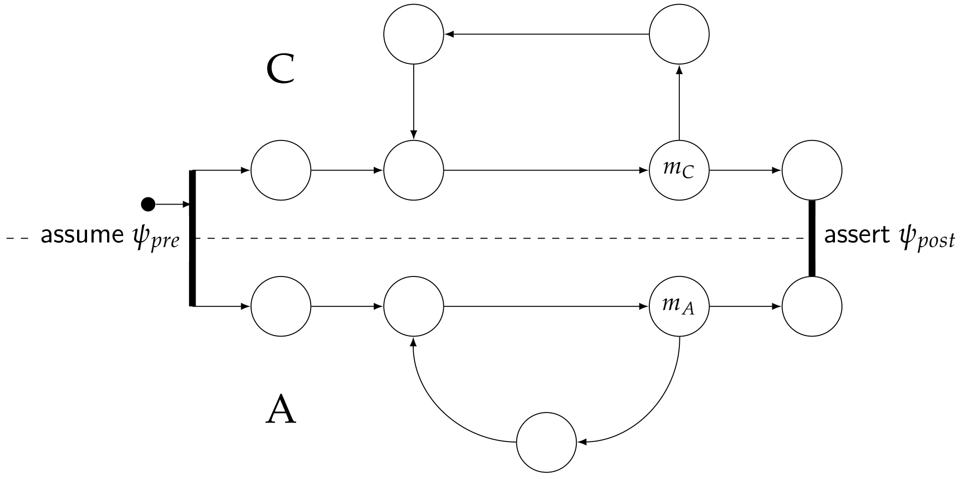
is valid since it is equivalent to the tautology $\psi' \to (cnd_C \to (cnd_A \wedge \psi'))$ in which the assumptions have been symbolically executed. Using this expansion also for α and β, it is easy to see that the condition

$$\psi \wedge cnd_A \wedge cnd_C \;\to\; [body_C]\langle body_A\rangle\psi'$$

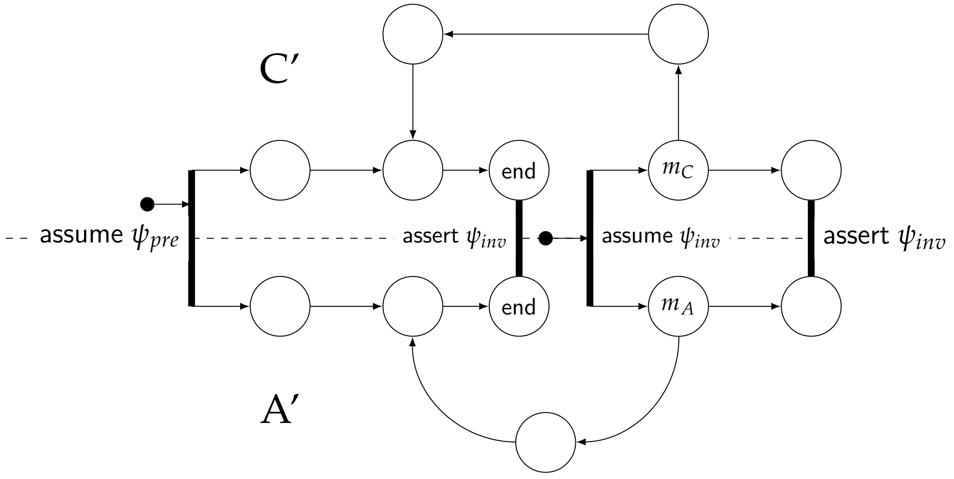
of Definition 5.1 is equivalent to the formula $\psi' \to [\alpha]\langle\beta\rangle\psi'$. By Observation 5.4, this implies also the validity of $\psi' \to [\alpha^*]\langle\beta^*\rangle\psi'$. Combining this result with (5.8) gives $\psi' \to [\alpha^*]\langle\beta^*\rangle[cnd_C\mathbf{?}]\langle cnd_A\mathbf{?}\rangle\psi'$. Since the programs do not share program variables by assumption, the valid implication $\langle\beta^*\rangle[cnd_C\mathbf{?}]\varphi \to [cnd_C\mathbf{?}]\langle\beta^*\rangle\varphi$ (for any condition φ) allows us to reorder the modal operators such that we have that the loops are in the refinement relationship: $\psi' \to [\alpha^*\,;cnd_C\mathbf{?}]\langle\beta^*\,;cnd_A\mathbf{?}\rangle\psi'$ □

5.4.2 A Synchronised Refinement Rule for *UDL*

The loop invariant inference rules presented in Section 3.3 allow the verification of programs with loops by introducing invariant split points at which a loop invariant predicate needs to hold whenever it is visited. The rules translate the loop invariant rule from structured to unstructured logic. The control flow of the unstructured



(a) Original programs A and C



(b) Modified programs A' and C'

Figure 5.2: Informal description of the modification of *UDL* programs for refinement with marks

program is thus broken up at one point such that the program control graph is made cycle-free. Figure 3.3 schematically depicts how a cycle in a program is suspended by the use of a loop invariant.

The idea of using an invariant split point to remove cycles can be transferred from the case with a single program to the situation that not one but the composition of *two* programs is examined. A synchronisation point consists of two corresponding indices in the refined program and its refinement. Corresponding in this context means that we can provide a coupling predicate between the abstract and concrete states which holds whenever the two programs reach their respective corresponding index.

Figure 5.2 transfers the intuition shown in Figure 3.3 onto the case with two composed programs. Both programs are sketched as finite automata. The composed program can then be thought of as the product automaton of C (above the dotted line) and A (below the line). Figure 5.2a shows the original refinement proof obligation: In the beginning, the coupling predicate ψ_{pre} is assumed and for every run of the automaton for C, a corresponding run must be found for the automaton A such that ψ_{post} holds in the end.

To resolve the cycles in both programs simultaneously using a common invariant, we identify two indices $m_A, m_C \in \mathbb{N}$ into the programs A and C which are both within the cycle of their respective loop and between which we can establish a formal refinement relationship in form of a coupling invariant ψ_{inv}. The two programs are then modified in such a manner that the control flow stops whenever m_A or m_C are reached. Instead of continuing, it is ensured that the corresponding index in the other automaton has been reached, and that ψ_{inv} holds, too, in the after-state. Like in the single-program loop invariant case, the programs are then also started in the marked indices with initial state (m_C, m_A) under the assumption of ψ_{inv}. If both program executions reach an end statement, the post-state coupling predicate ψ_{post} must hold. Figure 5.2b schematically sketches how the programs A' and C′ which result from A and C are organised after their modification. The cycles in both programs have been discontinued. Instead of one condition over cyclic conditions, two initial states must be considered in the modified environment.

For the implementation of this program modification with the means of *UDL*, we use the statement injection operator $\lhd$ introduced in Definition 3.3. It is necessary to remember whether the split indices m_C and m_A have been reached or whether the program terminated because it reached an end statement in the original code. The point of termination is encoded in two fresh program variables M_A, M_C : nat. They are used to store the nature of the programs' termination. They hold the value 1 if the synchronisation point has been reached and 0 in case of reaching a regular end statement.

Theorem 5.6 (Synchronised loop invariant rule) *Let $A, C \in \Pi$ be self-contained assertion-free UDL programs with $\mathrm{PV}(A) \cap \mathrm{PV}(C) = \emptyset$, $0 \leq m_A < |A|, 0 \leq m_C < |C|$ indices into the programs, M_A, M_C : nat fresh program variables which do not occur in the conclusion, $\psi_{post}, \psi_{inv} \in \mathrm{Trm}^{\mathsf{bool}}$ formulas without free variables.*

Then the sequent calculus inference rule

$$\frac{\Gamma \vdash \{\mathcal{U}\}\psi_{inv},\, \Delta \qquad \psi_{inv} \vdash \{M_A := 0 \parallel M_C := 0\}[m_C + 2; C']\langle m_A + 2; A'\rangle\Psi}{\Gamma \vdash \{\mathcal{U}\}[m_C; C]\langle m_A; A\rangle\psi_{post},\, \Delta}$$

with

$$\begin{aligned}\Psi &= (M_A \doteq M_C) \wedge (M_A \doteq 0 \to \psi_{post}) \wedge (M_A \doteq 1 \to \psi_{inv}),\\ C' &= C \lhd_{m_C} (M_C := 1;\mathsf{end}) \text{ and}\\ A' &= A \lhd_{m_A} (M_A := 1;\mathsf{end})\end{aligned}$$

is sound.

The idea behind the proof of this theorem is that the premisses ensure that every loop iteration from m_C to m_C in C induces an according loop from m_A to m_A in A. The coupling loop invariant ψ_{inv} binds the abstract and concrete state together after each loop iteration. The first premiss of the rule ensures that the coupling loop invariant holds in the beginning.

PROOF We may without loss of generality assume that $\Delta = \emptyset$. Let I be an interpretation with $I \models \bigwedge\Gamma$ and $I_1 := I^{\mathcal{U}}$ the interpretation with the update $\mathcal{U}$ applied. Furthermore, assume there is a successful trace $(I_1, m_C), \ldots, (I_3, k_C)$ of C with $C[k_C] = \mathsf{end}$. The trace may visit statement $C[m_C]$ several times since it may lie within a loop. We use induction over the number N of states in the trace which visit $C[m_C]$ (that is, the number of loop iterations) to show that there is a trace $(I_3, m_A), \ldots, (I_4, k_A)$ with $A[k_A] = \mathsf{end}$ and $I_4 \models \psi_{post}$.

$N = 1$: The trace $(I_1, m_C), \ldots, (I_3, k_C)$ has no loop which goes through m_C and the trace ends with $C[k_C] = \mathsf{end}$. By Obs. 3.4, there is a corresponding trace $(I_C, m_C + 2), \ldots, (I_3, k'_C)$ of C' with $C'[k'_C] = \mathsf{end}$. We have that $I_1 \models \psi_{inv}$ by the first premiss and thus that $I_3 \models \langle m_A + 2; A'\rangle\Psi$ by the second premiss. That means there must be a trace $(I_3, m_A + 2), \ldots, (I_4, k'_A)$ with $I_4 \models \Psi$, which implies $I_4(M_A) = I_4(M_C)$. But the trace for C' could not pass through a statement changing M_C (this would imply a loop involving m_C). Therefore, $I_4(M_C) = I_4(M_A) = 0$, and the abstract program terminates in a statement with $A'[k'_A] = \mathsf{end}$ and $I_4 \models \psi_{post}$. By Obs. 3.5, there is a corresponding trace $(I_3, m_A + 2), \ldots, (I_4, k_A)$ with $A[k_4] = \mathsf{end}$ for A.

The induction step is schematically sketched in Figure 5.3. Programs can only modify the part of state which belongs to their respective program variables. We hence restrict the presentation of the states to the part of the interpreting function responsible for the respective program. The rigidly interpreted I_0 is constant. The state for the complementary program can be varied, the sequence of states remains a trace, see also Obs. 5.1.

$N > 1$: There is at least one loop iteration from m_C to m_C. Let $(I_{2|C}, m_C)$ be the first occurrence of m_C in the trace after I_1. The partial trace $(I_{1|C}, m_C), \ldots, (I_{2|C}, m_C)$ then has the required properties of Obs. 3.4 and there exists a partial trace $(I_{1|C}, m_C +$

2),...,$(I_{2|C}, m_C)$ of C' which will lead by one more execution step to the successful trace $(I_{1|C}, m_c + 2), \ldots, (I^{[1]}_{2|C}, m_C + 1)$ for $I^{[1]}_{2|C} := I_{2|C}[M_C \mapsto 1]$. By the second premiss, it must be that $I_0 + I_{1|A} + I^{[1]}_{2|C} \models \langle m_A + 2; A' \rangle \Psi$, i.e., there is a trace $T'_1 := (I_{1|A}, m_A + 2), \ldots, (I_{2|A}, m_A + 1)$ for A' with $I_0 + I_{2|A} + I_{2|C} \models \Psi$. This trace must end in $m_A + 1$ as the only manipulation of M_A ensuring $I_{2|A}(M_A) = I_{2|C}(M_C) = 1$ is at $A'[m_A]$. But then $I_2 \models \psi_{inv}$.

The remaining trace $(I_{2|C}, mc), \ldots, (I_{3|C}, k_C)$ has now only $N - 1$ loop iterations and $I_2 \models \psi_{inv}$. By induction hypothesis, there exists a trace $T_2 := (I_{2|A}, m_A), \ldots, (I_{3|A}, k_A)$ such that $I_0 + I_{3|A} + I_{3|C} \models \psi_{post}$.

The trace T'_1 for A' gives rise to a partial trace $T_1 := (I_{1|A}, m_A), \ldots, (I_{2|A}, m_A)$ of A according to Obs. 3.5. The parts T_1 and T_2 can be concatenated to obtain a trace $(I_{1|A}, m_A), \ldots, (I_{3|A}, k_A)$ for A. By defining $I_4 := I_0 + I_{3|A} + I_{3|C}$ we have found the sought trace for A with $I_4 \models \psi_{post}$. □

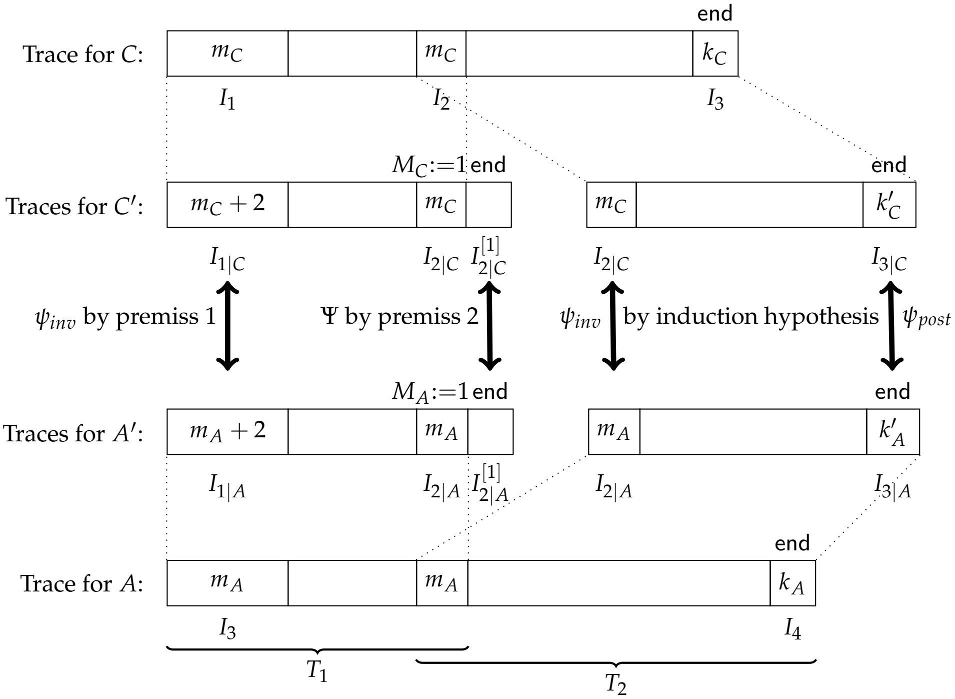


Figure 5.3: Visualisation of the induction step for the proof of Thm. 5.6

5.4.3 Improved Synchronised Refinement Rules

While Theorem 5.6 already allows the verification of refinement proof obligations using a coupling invariant, there is still room for improvement to obtain rules which are better suited for the application in practice. Section 3.3.5 covers a similar challenge in the context of single-program invariant rules. One rule for the treatment of loop invariants has been presented as Theorem 3.8 which allowed retaining information from the context Γ, Δ and $\mathcal{U}$ not changed within the loop for the step case.

The context may contain additional information which is needed to conduct the proof. This information could be added to the invariant and thereby made available, but a relatively small adaption of the rule relieves us from the obligation to identify this information by retaining as much as possible of the original context. This can be achieved by *anonymising* the program variables modified within the loop body. The same technique used in Theorem 3.8 can be applied here using a reachability analysis for both programs. The goal is to relax the second premiss in the last theorem to a sequent in which the context is preserved and is structured like $\Gamma, \mathcal{U}\psi_{inv} \vdash \mathcal{U}\{...\}[...]\langle...\rangle\Psi, \Delta$.

However, this relaxed sequent would give an unsound rule variation as some program variables used in the context may have been changed in the course of the program and some propositions in the context may not hold any longer in some intermediate state. For the improved rule to be sound, the variables which are under the influence of assignments and havoc statements within the loop under inspection must have their value deleted using an additional update $\mathcal{V}$ which assigns to every modified program variable p the value of a fresh uninterpreted function symbol p'. Variables which are not touched in the loop may keep their respective values from before the loop and the propositions of the context applies to them.

In the theorem for the improved refinement rule, we come back to Definition 3.4 from Section 3.3.4 where the set $mod(n,\pi) \subseteq \text{PVar}$ has been defined as the set of all program variables which may be assigned to during the course of a loop iteration from $\pi[n]$ to $\pi[n]$. The rule requires that the set of program variables changed in the course of the inspected loops in A and C be computed. The modification sets for the programs can be computed by two independent simple static code analyses.

Observation 5.7 (Improved synchronised loop invariant rule) *Let the requirements of Theorem 5.6 hold. Additionally, let the finite set $\{p_1,\dots,p_r\}$ with $mod(m_c,C) \cup mod(m_A,A) \subseteq \{p_1,\dots,p_r\} \subseteq \text{PVar}$ approximate the variables which are modified in the loops of A and C. $p'_1,\dots,p'_r$ denote fresh constant function symbols of the same types as $p_1,\dots,p_r$ which do not occur in the conclusion, $\mathcal{V} := \{p_1 := p'_1 \parallel \dots \parallel p_r := p'_r\}$.*

Then the sequent calculus inference rule

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\psi_{inv},\ \Delta \\ \Gamma, \{\mathcal{U} \parallel \mathcal{V}\}\psi_{inv} \vdash \{\mathcal{U} \parallel \mathcal{V} \parallel M_A := 0 \parallel M_C := 0\}[m_C+2; C']\langle m_A+2; A'\rangle\Psi, \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[m_C; C]\langle m_A; A\rangle\psi_{post},\ \Delta}$$

with Ψ, C' and A' like in Theorem 5.6 is sound.

PROOF In the proof of Theorem 5.6, the second premiss is only applied to states which have evolved from the initially considered interpretation I with $I \models \Gamma$. It suffices to show that the validity of the modified second premiss implies that the original second premiss holds in these reachable states.

Let I' be a state which is reachable via symbolic execution of A and C from $I^{\mathcal{U}}$ and let $I'' := I[p'_1 \mapsto I'(p_1)] \dots [p'_r \mapsto I'(p_r)]$. The second premiss is assumed valid, therefore it holds in I'' which coincides with I on all symbols but the newly introduced anonymisation symbols p'_i. Due to the assumptions that $\Delta = \emptyset$ and $I \models \bigwedge\Gamma$ in the previous proof, it must be that $I'' \models \bigwedge\Gamma$, too, and $I'' \models \{\mathcal{U} \parallel \mathcal{V}\}\psi_{inv} \vdash \{\mathcal{U} \parallel \mathcal{V} \parallel M_A := 0 \parallel M_C := 0\}[m_C + 2; C']\langle m_A + 2; A'\rangle\Psi$. This is equivalent to $I''^{\mathcal{U}\parallel\mathcal{V}} \models \psi_{inv} \vdash \{M_A := 0 \parallel M_C := 0\}[m_C + 2; C']\langle m_A + 2; A'\rangle\Psi$, the second premiss in Theorem 5.6.

It remains to be shown that the interpretation $I''^{\mathcal{U}\parallel\mathcal{V}}$ is equal to I': By construction, the possibly modified symbols p_i are equal as update $\mathcal{V}$ ensures that $I''^{\mathcal{U}\parallel\mathcal{V}}(p_i) = I''(p'_i) = I'(p_i)$. Any other symbol s cannot have been changed by the programs and has the value $I^{\mathcal{U}}(s)$ it had the beginning of the trace: $I'(s) = I^{\mathcal{U}}(s)$. The symbols s is not amongst the program variables anonymised in $\mathcal{V}$ such that $I''^{\mathcal{U}\parallel\mathcal{V}}(s) = I^{\mathcal{U}}(s)$ holds. Thus we have shown that I' fulfils the second premiss of the original rule: We can safely apply the proof of Theorem 5.6 to show correctness of this rule. □

The verification of a refinement step can also be used to simultaneously show termination of the algorithm. The proof obligation is in this case formulated using the terminating modality $[\![C]\!]$ instead of the partial $[C]$. A variant expression whose value is decreased in every loop iteration has to be specified as a witness for the termination.

Like in Theorem 3.7 in the case of single program conditions, the value of the variant is stored in a fresh program variable nc before the loop iteration, and $var \prec nc$ must hold after each loop iteration.

Observation 5.8 (Synchronised loops with termination) *Let the requirements of Theorem 5.6 hold. Additionally, let nc be a fresh program variable which do not occur in the conclusion and var* : $\mathrm{Trm}^{\mathrm{ty}(nc)}$ *a variable-free variant term.*

Then the sequent calculus inference rule

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\psi_{inv}, \Delta \\ \psi_{inv} \vdash \{M_A := 0 \parallel M_C := 0 \parallel nc := var\}[\![m_C + 2; C']\!]\langle m_A + 2; A'\rangle\Psi^{\prec}\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\![m_C; C]\!]\langle m_A; A\rangle\psi_{post}, \Delta}$$

with C' and A' as above and

$$\Psi^{\prec} = \Psi \wedge (M_A \doteq 1 \to var \prec nc),$$

is sound.

PROOF The program formula in the conclusion is equivalent to the conjunction $[\![m_c;C]\!]\mathsf{true} \wedge [m_C;C]\langle m_A;A\rangle\psi_{post}$. The second conjunct is a conclusion of the premisses as we have already seen in Theorem 5.6 (The second premiss here implies the second premiss of Thm. 5.6). Only the termination $[\![m_C;C]\!]$ remains to be shown. The argument is an alteration of the one for Theorem 3.7.

Assume that there is an infinite trace $(I_1,m_C),\ldots,(I_2,m_C),\ldots,(I_3,m_C),\ldots$ of C which can be partitioned into partial traces from m_C to m_C. Each loop synchronisation point is evaluated in an interpretation with $I \models \psi_{inv}$. By assumption 2 (which uses $[\![\cdot]\!]$) none of these partial traces can be infinite themselves, thus, there must be infinitely many repetitions of m_C to constitute the infinite trace. The sequence $(\mathrm{val}_{I_1}(var),\mathrm{val}_{I_2}(var),\mathrm{val}_{I_3}(var),\ldots)$ must be infinite. Due to the added variant reduction condition in $\Psi^{\prec}$, the chain must be strictly decreasing. But this cannot be as there is no infinite strictly decreasing chain for $\prec$. □

This rule is helpful if the termination of the algorithm is to be shown during the refinement process. But termination may already have been proved for the abstract program A prior to its formal refinement. It may be part of the functional specification and verification of algorithm A. In such situations, there seems to be no need to provide another termination argument during refinement; the refined program should inherently be terminating as well. This is indeed the case, and we can adapt the rule from Theorem 5.6 for total correctness if we require one additional premiss.

Theorem 5.9 (Synchronised loops with inherited termination) *Let the requirements of Theorem 5.6 hold. Then the rule*

$$\frac{\begin{array}{rcl}\Gamma & \vdash & \{\mathcal{U}\}[\![m_A;A]\!]\mathsf{true},\ \Delta\\ \Gamma & \vdash & \{\mathcal{U}\}\psi_{inv},\ \Delta\\ \psi_{inv} & \vdash & \{M_A := 0 \parallel M_C := 0\}[\![m_C+2;C']\!]\langle m_A+2;A'\rangle\Psi\end{array}}{\begin{array}{rcl}\Gamma & \vdash & \{\mathcal{U}\}[\![m_C;C]\!]\langle m_A;A\rangle\psi_{post},\ \Delta\end{array}}$$

with Ψ, C' and A' as in Theorem 5.6 is sound.

In contrast to the rule for $[\cdot]$, this inference rule requires that termination of A is known. This is represented by the premiss $\Gamma \models \{\mathcal{U}\}[\![m_A;A]\!]\mathsf{true},\Delta$ in the antecedent of the conclusion. This property may be known as a consequence from another proof of a functional property.

PROOF Let $\Delta = \emptyset$ and I with $I \models \bigwedge\Gamma$. Assume we had an infinite trace $(I^{\mathcal{U}},m_C),\ldots,(I_1,m_C),\ldots,(I_2,m_C),\ldots$ for C. Due to the premisses, this trace must visit statement $C'[m_C]$ infinitely often. Otherwise the last subtrace from the synchronisation point m_C onwards would be infinite which contradicts the third premiss which uses the terminating modality.

But also by the third premiss, each of the partial traces from m_C to m_C in C induces a corresponding partial trace from m_A to m_A of program A. Putting together these partial traces for A results in an infinite trace for A starting in $I^{\mathcal{U}}$ which is a contradiction with the added first premiss. □

This rule needs the termination of A begun at index m_A. Usually termination will have been shown for A from the beginning of the program not from an intermediate proof state. We will use the result of Theorem 5.9 for the terminating variant of the next, more powerful rule which requires termination of A started at its beginning.

5.4.4 A Refinement Rule with Multiple Synchronisation Points

The most important improvement of the synchronised loop treatment showed up in practice when doing refinement proofs. Applying all program modifications as the first step in the proof turned out to be superior to interjecting execution and splitting of the program. A single rule application is easier to apply and introduces less management overhead in the proof obligations. Also the symbolic execution does not need to synchronise the two programs and inhibit symbolic execution for one program to wait for the other party to arrive at the synchronisation point.

If several synchronisation points are specified, the control flow is broken up accordingly and simultaneously for all points. The rule has as many premisses as there are synchronisation points (plus one for the initial case, and another one if termination is considered).

Theorem 5.10 (Synchronised multi-loop invariant rule) *Let* $A, C \in \Pi$ *be self-contained assertion-free programs with* $\mathrm{PVar}(A) \cap \mathrm{PVar}(C) = \emptyset$, $0 \leq m_A^1 < m_A^2 < \ldots < m_A^N < |A|, 0 \leq m_C^1 < m_C^2 < \ldots < m_A^N < |C|$ *indices into the programs,* $M_A, M_C : \mathsf{nat}$ *fresh program variables which do not occur in the conclusion,* $\psi_{post}, \psi_{inv}^1, \ldots \psi_{inv}^N \in \mathrm{Trm}^{\mathsf{bool}}$ *formulas without free variables.*

Then the sequent calculus inference rules

$$\frac{\begin{array}{rcl} \Gamma & \vdash & \{\mathcal{U}\}\{M_A := 0 \parallel M_C := 0\}[C']\langle A'\rangle\Psi,\ \Delta \\ \psi_{inv}^1 & \vdash & \{M_A := 0 \parallel M_C := 0\}[m_C^1{+}2;C']\langle m_A^1 + 2; A'\rangle\Psi \\ & \vdots & \\ \psi_{inv}^N & \vdash & \{M_A := 0 \parallel M_C := 0\}[m_C^N + 2N;C']\langle m_A^N + 2N; A'\rangle\Psi \end{array}}{\Gamma \vdash \{\mathcal{U}\}[C]\langle A\rangle\psi_{post},\ \Delta}$$

and

$$\frac{\begin{array}{rcl} \Gamma & \vdash & \{\mathcal{U}\}[\![A]\!]\mathsf{true},\ \Delta \\ \Gamma & \vdash & \{\mathcal{U}\}\{M_A := 0 \parallel M_C := 0\}[\![C']\!]\langle A'\rangle\Psi,\ \Delta \\ \psi^1_{inv} & \vdash & \{M_A := 0 \parallel M_C := 0\}[\![m^1_C{+}2;C']\!]\langle m^1_A + 2;A'\rangle\Psi \\ & \vdots & \\ \psi^N_{inv} & \vdash & \{M_A := 0 \parallel M_C := 0\}[\![m^N_C + 2N;C']\!]\langle m^N_A + 2N;A'\rangle\Psi \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\![C]\!]\langle A\rangle\psi_{post},\ \Delta}$$

with

$$\Psi = (M_A \doteq M_C) \wedge (M_A \doteq 0 \rightarrow \psi_{post}) \wedge \bigwedge_{i=1}^{N} (M_A \doteq i \rightarrow \psi^i_{inv}),$$

$$C' = C \lhd_{m^N_C} (M_C := N;\mathsf{end}) \lhd_{m^{N-1}_C} (M_C := N-1;\mathsf{end}) \lhd_{m^{N-2}_C} \ldots \lhd_{m^1_C} (M_C := 1;\mathsf{end})$$

$$A' = A \lhd_{m^N_A} (M_A := N;\mathsf{end}) \lhd_{\mathrm{m}^{N-1}_{\mathrm{A}}} (M_A := N-1;\mathsf{end}) \lhd_{m^{N-2}_A} \ldots \lhd_{\mathrm{m}^1_{\mathrm{A}}} (M_A := 1;\mathsf{end})$$

are sound.

In contrast to the rules introduced so far, this rule is not applied upon reaching the synchronisation points by symbolic execution but prior to doing any execution steps. This has the advantage that no synchronisation is needed between the symbolic executions of A and C, the automation can be applied without modification. Accordingly, the first premiss also contains a program formula unlike in earlier invariant rules to reach the first synchronisation point.

PROOF A detailed proof is omitted here as it follows the same lines as that for Theorem 5.6 but is more technically elaborate.

If a trace for C does not visit any of the insertion points $m^1_C, \ldots, m^N_C$, the first premiss ensures the existence of an according trace for A directly.

If the trace visits one or more synchronisation points, the inductive argument applied earlier can be adapted. It must be canonically generalised a little since there is not one but N synchronisation points of which one is used in the induction step.

That the terminating rule is sound can be shown using the argument from the proof of Observation 5.8. If the there was an infinite trace for C, then the premisses would allow the construction of a corresponding infinite trace for A which cannot be due to the first premiss requiring that there is no infinite trace for A in the given context. □

The inference rule with multiple synchronisation points explains why M_A and M_C were modelled as natural numbers. In Theorem 5.6, boolean flags with only two possible values would have sufficed, but with N disjoint exit options to distinguish, natural numbers model this more concisely. The assignments to M_C and M_A ensure that the traces always go side-by-side. For every synchronisation point $C[m^i_C]$ visited by the trace of C, the trace of A must visit $A[m^i_A]$ for the same index i or the postfixed condition Ψ cannot hold.

In the previous rules, the synchronisation point at which the program had to be modified was implicitly given by the current indices in the conclusion. This is no longer the case for the multi-split rules. The indices for which the splits should be applied are parameters to the rule since they do not occur in the conclusion. The rules have an additional degree of freedom in this respect.

To make the rule usable in a more automatic fashion, we add autoactive annotations to both the abstract program A and its refinement C. The annotations are synchronisation markers which can be annotated to the statements in programs without changing their semantics. Each marker carries a natural number indicating the value for M_A (or M_C) to be assigned at this point. For the annotations to be wellformed, it is necessary that the set of annotated markers is the same for both programs and that every number occurs at most once in a marker in a program. The invariants ψ_i for the used synchronisation markers are stated in a table.

The example which follows next demonstrates on a small example how the synchronised loop splitting can be used to prove a refinement proof obligation.

5.4.5 Example: Summation

The small example code to demonstrate the invariant rules for synchronised loops computes the sum of a finite set of integers. The abstract program has a program variable $s_A : \mathsf{set}(\mathsf{int})$ for the set. The simple summation algorithm takes an element out of that set, removes it from the set and adds it to the interim sum $sum_A : \mathsf{int}$. This is repeated until the set is empty.

Similar to the process in Section 5.3.1, this program is then subjected to a data refinement using a more implementation-friendly data-structure, namely sequences instead of sets. The set variable is refined by the program variable $l_C : \mathsf{seq}(\mathsf{int})$ holding a sequence of integer values. The refined algorithm uses an iteration variable $i_C : \mathsf{nat}$ to iterate over the indices into the sequence. Each value $\mathsf{seqGet}(l_C, i_C)$ within the sequence is then added to the sum $sum_C : \mathsf{int}$.

Both programs are formulated in pseudocode. We present the abstract program A_{sum} and concrete C_{sum} side by side in the following listing:

```
  PSEUDOCODE
$sum_A$ := 0;                                 $sum_C$ := 0;
while $\neg s_A \doteq \{\}$ do                  $i_C$ := 0;
  choose $x_A$ such that $x_A \in s_A$;         while $i_C$ < seqLen($l_C$) do
  $s_A$ := $s_A \setminus \{x_A\}$;
  mark 1; . . . . . . . . . . . . . . . .mark 1;
  $sum_A$ := $sum_A$ + $x_A$;                      $sum_C$ := $sum_C$ + seqGet($l_C$,$i_C$);
                                                $i_C$ := $i_C$ + 1
end                                           end
                                                            PSEUDOCODE
```

It is important to observe that the algorithm in both programs follows the same general idea, and that their loops are synchronised. While in the abstract description an arbitrary value may be taken from the set, the order of the summed values in the refined program is fixed by the order of the values in the sequence. But the order in C_{sum} is backed up by the indeterministic choice in A_{sum} such that the presented proof method for synchronised programs can be applied.

Line 5 plays an important role in both programs as it marks the synchronisation points which are needed to couple the two programs. They lie within the respective loops of the programs such that breaking up the control flow at these points will make the modified programs loop-free. No loop invariant needs to be annotated for C_{sum}, only a coupling invariant on the data structures must be found.

In the case of this example, a functional coupling predicate can be given. We can even provide an explicit functional relationship by using the polymorphic functions $\mathsf{seqAsSet} : \mathsf{seq}(\alpha) \to \mathsf{set}(\alpha)$ and $\mathsf{seqSub} : \mathsf{set}(\alpha) \times \mathsf{int} \times \mathsf{int} \to \mathsf{set}(\alpha)$ The former denotes the set of all components of the sequence while the latter describes the subsequence of the first argument which starts at the second and ends at the third. For a formal definition of the symbols see Appendix A.2.

The first refinement condition which comes to mind says that if set and sequence are coupled in the beginning, the sums are equal in the end:

$$s_A \doteq \mathsf{seqAsSet}(l_C) \to [C_{sum}]\langle A_{sum}\rangle sum_C \doteq sum_A$$

But this proof obligation cannot and must not be discharged. The condition does not describe a valid refinement. A counter example which could, for example, emerge from an analysis of a failed interactive proof attempt reveals that a value may still be present in the sequence even if it has been removed from the set: If a value is contained in the sequence more than once, this is not reflected in the abstraction as a set. The loops are no longer synchronised: Removing a value from the set does not necessarily mean that it is also removed from the refined data structure.

Therefore, the coupling for the precondition must be strengthened and additionally require that the sequence contains no duplicates. The refinement condition can now be discharged. The formal definition of the refinement comes together with the definitions of the two programs and reads as follows:

PSEUDOCODE REFINEMENT

```
1  refine $A_{sum}$ as $C_{sum}$
2    requires $s_A \doteq \mathsf{seqAsSet}(l_C) \land$
3        $(\forall k. \forall j.\ 0 \le k < j < \mathsf{seqLen}(l_C) \to \neg \mathsf{seqGet}(l_C, k) \doteq \mathsf{seqGet}(l_C, j))$
4    ensures $sum_A \doteq sum_C$
5    mark 1 inv $sum_A \doteq sum_C \land s_A \doteq \mathsf{seqAsSet}(\mathsf{seqSub}(l_C, i_C, \mathsf{seqLen}(l_C))) \land$
6        $0 \le i_C < \mathsf{seqLen}(l_C)$
```

PSEUDOCODE REFINEMENT

There are three coupling predicates involved in the refinement: The coupling predicate used as precondition line 2, the postcondition coupling in line 4 and the coupling invariant for the synchronisation point in line 5.

5.5 Refinement from Pseudocode to Java

Let us come back to the goal outlined in the first section of the chapter: Separation of Concerns, that is, the verification of an algorithm implemented in a real programming language with all its technical challenges by means of refinement. The refinement tool developed in the last sections can now be used to achieve precisely this. We can formally link a Java implementation to its abstract algorithm description.

The refinement step from the algorithmic description to the implementation crosses a language border as one description is given as a Java method implementation while the other is stated as an algorithm in pseudocode. Yet, the two languages have to be formally coupled. We have described in detail in Section 4.4 that the execution state of a Java program manifests itself in the program variables which represent the local variables and method parameters and a special program variable capturing the entire heap as a two-dimensional array. The pseudocode algorithm operates on its own set of program variables which is disjoint from the ones available in Java. The predicate logic provides the common ground to formulate coupling predicates between the pseudocode level and the Java level.

For the refinement approach with dynamic logic, both algorithm formalisations need to be combined within one verification condition. At this point, the intermediate verification language proves particularly valuable. By reducing Java and pseudocode algorithms to the same intermediate format, we are able to consider the two formulations as two programs in the same language and do not need special constructs which connect two different notions of symbolic execution.

5.5.1 Extracting Contracts from Refinement

An algorithm will usually not stand on its own but be embedded in a system context in which it interacts with other parts of the software. For this reason, it is important that the implemented algorithms have a formal contract which can then be used in the verification even if the refinement model does not cover the entire system.

Refinement can be used to transfer proofs for properties of abstract descriptions to contracts on the level of the implementation. The proof of the properties need not be repeated on the implementational level. The refinement condition has been defined such that every property of the abstract level is automatically one of the concrete level, possibly modulo a coupling relation.

However, not every property of an implementation can be inherited from its abstraction. There are aspects which are better shown on the implementation level directly rather then synthesising them downwards from the abstract level. Exception handling and framing properties concerning the memory footprint of code of the implementation highly depend on the data structures and their implementations. They are not part of the abstract model. These properties can and should be proved on the concrete level directly without an additional refinement layer. Also, the absence of unexpected runtime errors (like null pointer dereferencing, out of bounds accesses, etc.) can and should be treated on that level directly. Showing functional safety in this sense may be as hard as full functional verification in some cases, but in practice, technical issues can often be separated from the algorithmic complexity.

Let us assume that an abstract algorithmic description $A \in \Pi$ is given as well as its implementation $J \in \Pi$ as a Java method, each translated to a *UDL* program. To obtain a functional contract for the implementation through the formal refinement from A to J, the following proof obligations must be discharged:

1. *The technical functional safety* is the part of the verification best performed on the implementation directly:

   $$pre_{tech} \to [J]post_{tech} \tag{5.9}$$

   The pair of technical pre- and postcondition pre_{tech} and $post_{tech}$ give a contract for the program J directly. This also ensures that under the technical precondition, no assertion of J ever fails.

   If desired, termination of J can also be proved on the implementation directly; in that case the modality $[\![J]\!]$ is to be used instead of $[J]$ in the proof obligation. Termination may be inherited from the abstract algorithm as has been shown in Observation 5.8 such that the termination modality needs not be applied here but may be used in (5.10) instead.

2. *The algorithmic property*

   $$pre_{algo} \to [\![A]\!]post_{algo} \tag{5.10}$$

   must be shown to hold for the abstract algorithm. The verification can be made on the abstract level solely ignorant of a later refinement step. Termination suggests itself to be proved during this full functional algorithm verification of the algorithm.

   However, it may be that the abstract algorithm description does not necessarily imply termination, but that the implementation always *does* terminate. Then termination must be shown on J directly, and the weaker modality $[A]$ can be used in (5.10) instead of $[\![A]\!]$.

3. *The refinement condition* modulo the coupling predicates ψ_{pre} and ψ_{post}

   $$\psi_{pre} \to [\![J^{af}]\!]\langle A^{af}\rangle\psi_{post} \tag{5.11}$$

   requires assertion-free variants A^{af} and J^{af} of A and J. We can safely remove[9] the assertions from the programs as only cases in which we have shown that assertions hold will be considered for the refined contract. In items 1 and 2 we have shown that, under the respective precondition, J and A do not violate their embedded assertions.

   If termination is proved on the implementation, it suffices to use the partial modality $[J^{af}]$.

[9] that is, replace every statement assert φ by skip.

From the combination of these three verification conditions, a contract for the implementation J can be extracted. The contract comprises a pre- and postcondition, but not the other elements we had identified as elements for contracts in Section 4.4.6. They are more of a technical nature and should be proved on the implementation directly (for instance as part of the technical postcondition $post_{tech}$). This also includes the framing condition mod_J describing the set of objects possibly modified by J.

The extracted contract makes use of an abstraction from the values of the variables in $\mathrm{PV}(A)$. This is achieved by a combination of existential quantifiers and updates which assign to the program variables the values of the quantified variables. For the set $\mathrm{PV}(A) = \{p_A^1, \ldots, p_A^n\}$ of abstract program variables, the abstraction of a formula φ reads $(\exists x_1 \ldots \exists x_n.\ \{p_A^1 := x_1 \parallel \ldots \parallel p_A^n := x_n\}\varphi)$. For better readability, we abbreviate this as $(\exists \bar{x}.\{\bar{p}_A := \bar{x}\}\varphi)$.

Theorem 5.11 (Contract extraction) *Let $A, J \in \Pi$ be self-contained programs with $\mathrm{PV}(A) \cap \mathrm{PV}(C) = \emptyset$. Let $pre_{tech}, post_{tech} \in \mathrm{Trm}^{\mathsf{bool}}$ be formulas in which the abstract program variables $\mathrm{PV}(A)$ do not occur and $pre_{algo}, post_{algo} \in \mathrm{Trm}^{\mathsf{bool}}$ formulas in which the implementation program variables $\mathrm{PV}(C)$ do not occur. The coupling predicates $\psi_{pre}, \psi_{post} \in \mathrm{Trm}^{\mathsf{bool}}$ may make use of all program variables.*

If the formulas (5.9), (5.10) *and* (5.11) *are valid, then the implication $pre \to [\![J]\!]post$ with*

$$\begin{aligned} pre &= (\exists \bar{x}.\{\bar{p}_A := \bar{x}\}(\psi_{pre} \wedge pre_{algo})) \wedge pre_{tech} \quad \textit{and} \\ post &= (\exists \bar{x}.\{\bar{p}_A := \bar{x}\}(\psi_{post} \wedge post_{algo})) \wedge post_{tech} \end{aligned}$$

is also valid.

Both the pre- and the postcondition of the extracted contract $pre, post$ for J contain existential quantifiers which decouple the abstract from the concrete state space. The extracted contract does not depend on the abstract program variables though they may syntactically appear in it. The conditions talk about the concrete Java state and only about a coupled (modulo ψ_{pre} or ψ_{post} respectively) abstract state implicitly. The resulting contract is self-contained on the implementational level and can also be used in a context oblivious of the abstract algorithm description.

PROOF Let I be an interpretation with $I \models \psi_{pre} \wedge pre_{algo} \wedge pre_{tech}$. Then clearly $I \models [J]post_{tech}$, $I \models [\![A]\!]post_{algo}$ and $I \models [\![J^{af}]\!]\langle A^{af}\rangle\psi_{post}$ by the three assumptions (5.9), (5.10) and (5.11).

Let $I^{[\![J]\!]}$ be a state which is reached by executing J starting in $(I,0)$ (ending at an end-statement):

(A)	$I^{[\![J]\!]} \models pre_A$	since $I \models pre_A$ and program J did not modify the program variables that pre_A depends on.
(B)	$I^{[\![J]\!]} \models [\![A]\!] post_A$	due to (A) and assumption (5.10)
(C)	$I^{[\![J]\!]} \models \langle A^{af} \rangle \psi_{post}$	by assumption (5.11). Modality $[\![J^{af}]\!]$ could be replaced by J as by assumption (5.9), J is known not to fail from I on.
(D)	$I^{[\![J]\!]} \models \langle A \rangle \psi_{post}$	due to (B) no trace of A from $I^{[\![J]\!]}$ can fail
	$I^{[\![J]\!]} \models \langle A \rangle post_A$	by (D) there *is* a final state of A and (B) ensures that $post_A$ holds.

Put together, this means that $I \models [\![J]\!] \langle A \rangle (\psi_{post} \wedge post_{algo})$. Note that the assumption of assertion-freeness has been dropped here.

The second modality can be abstracted from by an anonymising update. Instead of claiming that there exists a post-state of A, we can formulate the weaker statement that there is a modification of the variables in $\mathrm{PV}(A)$ with the same effects:

$$I \models [\![J]\!] (\exists \bar{x}. (\{\bar{p}_A := \bar{x}\} \psi_{post} \wedge post_{algo}))$$

The interpretation I has been chosen such that $pre_{tech} \wedge pre_{algo} \wedge \psi_{pre}$ holds. The last condition is, hence, equivalent to claiming that

$$\models \psi_{pre} \wedge pre_{algo} \wedge pre_{tech} \ \rightarrow\ [\![J]\!] (\exists \bar{x}. \{\bar{p}_A := \bar{x}\} \psi_{post} \wedge post_{algo})$$

is valid. This is equivalent to the condition in which the values of the abstract program variables are anonymised prior to the evaluation, that is

$$\models \big(\forall \bar{y}. \{\bar{p}_C := \bar{y}\} \big(\psi_{pre} \wedge pre_{algo} \wedge pre_{tech} \ \rightarrow\ [\![J]\!] (\exists \bar{x}. \{\bar{p}_A := \bar{x}\} \psi_{post} \wedge post_{algo})\big)\big) \ .$$

The conclusion of the implication does not depend on the value of the symbols in $\mathrm{PV}(A)$ as the program J does not use them and the postcondition of J has the values of all variables $\mathrm{PV}(A)$ overwritten by the anonymising assignment in the update. Also the technical precondition is independent of $\mathrm{PV}(A)$. We can therefore conclude that

$$\models (\exists \bar{y}. \{\bar{p}_C := \bar{y}\} (\psi_{pre} \wedge pre_{algo})) \wedge pre_{tech} \ \rightarrow\ [\![J]\!] (\exists \bar{x}. \{\bar{p}_A := \bar{x}\} \psi_{post} \wedge post_{algo}) \ .$$

which is the claim if we consider that the technical postcondition $post_{tech}$ has already been proved a postcondition of J in (5.9). □

The quantifiers in the extracted contract make them cumbersome. They are disadvantageous since whenever an extracted method contract is to be applied, the precondition needs to be established. This means for these cases that a witness needs to be provided giving evidence that the algorithmic precondition is satisfiable. This is an undesired effect since the method contract is meant to be an artifact on the level of the implementation solely and, as such, should not rely on algorithmic entities like the abstract precondition any more. To overcome the necessity of this step, an additional proof obligation can be introduced, the so-called *feasibility condition*

$$pre_{tech} \to (\exists \bar{x}.\{\bar{p}_A := \bar{x}\}\psi_{pre} \land pre_{algo})$$

to establish that the technical precondition is strong enough to imply the existence of a coupled abstract state in which the algorithmic precondition holds. By discharging this obligation, no witness needs to be provided on the individual applications of the extracted contract.

A formula is called *syntactically functional* if it is of the form $\bigwedge_{p \in \mathrm{PV}(A)} p \doteq e_p$ such that the expression e_p is a term over the concrete program variables in which the abstract program variables do not occur. It has been mentioned earlier that an explicitly stated functional coupling predicate has advantages over a merely semantically functional formula. The reason is the following: The quantifiers in the extracted contract can be eliminated if the coupling predicates are syntactically functional.

Observation 5.12 (Contract extraction, syntactically functional) *If the coupling predicates*

$$\psi_{pre} = \bigwedge_{p \in \mathrm{PV}(A)} p \doteq e_p^{pre} \quad \textit{and} \quad \psi_{post} = \bigwedge_{p \in \mathrm{PV}(A)} p \doteq e_p^{post}$$

in Theorem 5.11 are syntactically functional, the extracted contract pre $\to [\![J]\!]$*post can be rewritten without existential quantifiers as*

$$pre = \{\mathop{\|}_{p \in \mathrm{PV}(A)} p := e_p^{pre}\}pre_{algo} \land pre_{tech} \quad \textit{and}$$

$$post = \{\mathop{\|}_{p \in \mathrm{PV}(A)} p := e_p^{post}\}post_{algo} \land post_{tech} \ .$$

PROOF This is a direct consequence of the equivalence of $(\exists x.\{p := x\}(p \doteq e \land \varphi))$ and $\{p := e\}\varphi$ in predicate logic (if x does not occur in φ). □

In most cases, the algorithmic pre- and postconditions will not themselves contain program formulas. The updates in Observation 5.12 can hence be applied as syntactical substitutions to the formulas using the equivalences in the update simplification rules of Theorem 3.2 in Section 3.2. This yields a logically equivalent contract for the Java implementation in which the abstract program variables $\mathrm{PV}(A)$ do not occur any more.

5.5.2 Refinement Example Revisited

We come back to the example begun in Section 5.3.1 and provide now a straightforward refinement of the most concrete description onto an implementation given as a Java method. Recall that the example models the operation of adding a value to a set of values which is implemented as a sequence. The natural choice for the implementation of a sequence of integers in Java is an integer-array. The auxiliary method `add(int[] a, int v)` creates a new array of length `a.length+1`, copies the values of `a` into the result and appends the value `v`.

JAVA

```java
int[] insertInSet(int[] l, int n) {
  for(int i = 0; i < l.length; i++) {
    if(l[i] == n) {
      return l;
    }
  }

  return add(l, n);
}
```

JAVA – 5.4

Let us denote the *UDL* translation of the code of the method `insertInSet` as J_{insert}. This method implements the algorithm given as program I_{insert} (see page 144) on the mathematical level in a straightforward manner. It preserves the structure of the algorithm while the implementation language changes.

Like in a previous step in the refinement chain on the mathematical level, we introduce a function to map from the concrete data onto the abstract state space. In this case a sequence of integers must be extracted from a Java reference pointing to an array of `int`s. The function $\mathsf{intArrayAsSeq} : \mathsf{heap} \times \mathsf{ref} \to \mathsf{seq}(\mathsf{int})$ assigns to a Java reference (of type `int[]`) the sequence of the values stored in the array. These values in the array depend on the heap in which it is evaluated. The function is hence defined as

$$(\forall h. \forall r.\ \mathsf{intArrayAsSeq}(h,r) \doteq (\mathsf{seqDef}\ i.\ \ 0, \mathsf{arrlen}(r), h[r, \mathsf{idxInt}(i)]))\ .$$

According to last section, in order to extract a contract for the Java method from the abstract description, three proof obligations need to be examined:

1. The implementation has no runtime violations.

2. The abstract algorithm adds the argument n to the set.

3. The Java code J_{insert} refines I_{insert}.

For the first verification task, the program must be annotated with trivial loop specifications (shown on the left hand side of Figure 5.4) which are simple enough that they could also be inferred using one of the many invariant inference techniques. As technical precondition, $\mathtt{l} \neq \mathtt{null}$ is used. This task is hence concluded by discharging the obligation $\mathtt{l} \neq \mathtt{null} \to [\![J]\!]\mathsf{true}$.

We have not yet specified the property for which the contract should be synthesised by refinement. The obvious property already established by the most

JAVA + IVIL FRAGMENT

```
/*@ maintains 0 ≤i ∧ i≤ arrlen(l)
  @ decreases arrlen(l)−i
  @ modifies empty                           i_I := 0;
  @*/                                        while i_I < seqLen(l_I)
for(int i = 0; i < l.length; i++) {          do
  //@ mark 1;. . . . . . . . . . . . . . . . . mark 1;
  if(l[i] == n) {                              if seqGet(l_I,i_I) ≐ n_I
    return l;                                  then return end
  }                                            i_I := i_I + 1;
}                                            end;
```

JAVA + IVIL FRAGMENT – 5.5

Figure 5.4: Loop refinement from pseudocode to Java

abstract description A is that the algorithm adds its argument n to its representation of the set. This can be expressed on the abstract level as the implication $l_I^0 \doteq l_I \to [\![I_{insert}]\!]\mathsf{seqAsSet}(l_I) \doteq \mathsf{seqAsSet}(l_I^0) \cup \{n_I\}$ in which the fresh function symbol $l_I^0 : \mathsf{seq}(\mathsf{int})$ has been used to remember the old value of l_I prior to the method.

For the refinement task, we use different coupling predicates for pre- and postcondition.

$$\begin{aligned}
\psi_{pre} &:= l_I \doteq \mathsf{intArrayAsSeq}(h,\mathtt{l}) \land l_I^0 \doteq \mathsf{intArrayAsSeq}(h_{pre},\mathtt{l}) \land n_I = \mathtt{n}\\
\psi_{post} &:= l_I \doteq \mathsf{intArrayAsSeq}(h,res^{[\mathsf{ref}]}) \land l_I^0 \doteq \mathsf{intArrayAsSeq}(h_{pre},\mathtt{l}) \land n_I = \mathtt{n}\\
\psi_{inv} &:= l_I \doteq \mathsf{intArrayAsSeq}(h,\mathtt{l}) \land l_I^0 \doteq \mathsf{intArrayAsSeq}(h_{pre},\mathtt{l}) \land\\
&\qquad n_I \doteq \mathtt{n} \land i_I \doteq \mathtt{i} \land 0 \leq \mathtt{i} \land \mathtt{i} < \mathsf{arrlen}(\mathtt{l})
\end{aligned}$$

In the pre-state (using ψ_{pre}), l_I is coupled to the method parameter `l` while in the post-state (ψ_{post}), it is coupled with the result program variable $res^{[\mathsf{ref}]} : \mathsf{ref}$ of the method. The loops are synchronised and we can use ψ_{inv} as coupling invariant to prove the condition $\psi_{pre} \to [\![J^{af}_{insert}]\!]\langle I^{af}_{insert}\rangle\psi_{post}$.

The refinement is syntactically functional. The specification extraction described in Section 5.5.1 lets us hence arrive at the following contract for `insertInSet` in which the updates have already been simplified:

Java

```
/*@ contract
  @  requires ¬l ≐ null
  @  ensures    seqAsSet(intArrayAsSeq(h,res^[ref]))
  @             ≐ seqAsSet(intArrayAsSeq(h_pre,l)) ∪ {n}
  @*/
int[] insertInSet(int[] l, int v) { ... }
```

Java – 5.6

The contract does not show signs of the refinement it has undergone and operates only on items from the Java world.

5.5.3 Assumptions and Assertions in Refinement

We have accepted throughout the chapter that both the abstract code A and its refinement C have been assumed to be free of assertion statements. The reasons for this restriction differ from C to A.

For the concrete implementation, it has been done for performance reasons: As part of the separation of concerns, the assertions in C should be discharged in a proof outside the refinement condition. This allows the refinement to concentrate on the relation between C and A.

The situation is different for the abstract program A. If it contains an assertion which is not met, then every non-failing program refines the abstraction. This is a dangerous implication of the modelling and should better be avoided. Consider the program $A = (\mathsf{assert\ false})$ which has a failing assertion. The refinement condition $\psi_{pre} \to [C]\langle A\rangle\psi_{post}$ is valid regardless of the employed (non-failing) C and ψ_{pre}/ψ_{post}. This is due to the equivalence $\langle\mathsf{assert\ false}\rangle\varphi \equiv \neg([\mathsf{assert\ false}]\neg\varphi) \equiv \neg(\mathsf{false} \wedge \varphi) \equiv \mathsf{true}$.

The implication of this is that if a program description has got a failing trace, then any program refines it. While this is strictly an effect coming from the logical framework, it is unexpected and unwanted behaviour. To prevent us from running into such difficulties, assertions in abstract programs were generally forbidden.

We can relax that restriction a little now. In those cases in which we *know* that an assertion can never fail, it may be included. If $pre \to [A]\mathsf{true}$ has been verified and ψ_{pre} implies pre, then we can take A instead of the assertion-free variant A^{af} to build the refinement condition; the relevant traces for A and A^{af} are the same.

Moreover, assertions can even be used to provide help during the refinement proof. If a property φ is needed during the refinement proof at a particular point of the abstract program, it can be added as additional $\mathsf{assert}\ \varphi$ to the program. When verifying A, it is, amongst other things, shown that φ holds whenever the assertion is reached. Upon reaching the same statement under the $\langle\cdot\rangle$ modality, the asserted φ (due to the duality; compare Obs. 2.5) becomes an additionally assumed formula. This can be thought of as a small in-situ contract deep-embedded into the algorithm.

The contract is proved correct during the algorithm verification and used during the refinement proof.

A similar local contract effect can be made on the concrete program. Since they appear always under a box modality, the duality between the modalities cannot be exploited. However, when injecting an additional assertion (assert φ) to a program, it is safe[10] to inject an assumption of the same formula right after the assertion (assert φ; assume φ). During the technical verification of C, the assertion is proved correct. For the refinement condition the assertion-free C^{af} is used in which only the assumption remains, which can then be used in the refinement proof.

5.6 Related Work

Literature on refinement in the various refinement traditions is extensive. However, most research effort has been put in refinement on abstract levels. As far as refinement from abstract descriptions onto program code is concerned, fewer notable works exist.

Grandy et al. (2007) present a method within a case study in which they refine Abstract State Machines to Java programs. They apply the approach to verify security protocols implemented in the mobile edition of the Java language. The approach uses the interactive theorem prover KIV (Reif et al., 1995). The systems under consideration are reactive systems and not algorithmic descriptions, their code does not contain loops, loop synchronisation is not an issue. Leino and Yessenov (2012) present a methodology to perform statementwise algorithmic refinement within the Chalice language, an experimental language for specification and verification of concurrent programs (Leino and Müller, 2009). It is closely related to Morgan's refinement calculus and extends it to an object-oriented setting. They provide a mechanism for supplying witnesses when refining non-deterministic programs. Tafat et al. (2011) compare model fields in the Java Modeling Language together with representation clauses to refinement in the B method. Their focus, however, lies on the question of abstract aliasing rather than on code refinement.

Apart from the results in the application of refinement techniques, another field of related research are approaches that prove equivalence of programs. Godlin and Strichman (2009) give an overview to the topic. They describe an approach which uses bounded verification techniques to automatically verify program equivalence between two similar implementations, especially abstracting away from recursive function calls. Barthe et al. (2011) propose a calculus and a framework to prove program congruence. They combine the two programs for which equivalence is to be proved into one single program. Synchronised loops can hereby be resolved as one single loop with additional checks the loop conditions always coincide. They explicitly support the case that loops are not perfectly synchronised by unrolling a fixed number of loop iterations. This is helpful especially for the regression proofs

[10] The reason is the *local lemma* property mentioned in Section 3.2.

for compiler-optimised programs. The experiments are conducted using the Why tool (Filliâtre and Paskevich, 2013). Almeida et al. (2010) devise a similar method for regression verification using program composition to prove equivalence of cryptographic implementations using Frama-C (Cuoq et al., 2009).

The approach described by Feng and Hu (2005) comes from a more technical domain. They apply the technique of *cut-points* used in hardware verification for the equivalence verification of low-level software given in machine code. Cut-points are very similar to what we called synchronisation points in Section 5.4. Their intention for using cut-points, however, is to reduce the state space for the model checker they apply on the problem. They describe also how to find cut-points for the analysis automatically.

The mentioned equivalence approaches serve a similar goal as refinement, yet have two major differences: Data refinement is not taken into consideration and the approaches assume that programs are always deterministic. This is a sensible assumption if both programs are implemented in code; for refinement, this is not sufficient.

5.7 Chapter Summary

In this chapter, a new approach to prove Java implementation of algorithms has been presented. The refinement technique allows separation of concerns: An algorithmic property is proved on a high-level abstraction of the algorithm formalised in pseudocode and then refined to a contract of the implementation. At the same time, technical issues which have their origin in the used programming language are proved on the level of the implementation directly.

The refinement notion has been motivated and formally introduced with reference to existing definitions in established abstract formal systems like B, Z or the refinement calculus. A refinement condition for two programs in the programming language of *UDL* has been proposed. The concrete program is a refinement of the abstract program if every possible concrete behaviour is backed by an abstract behaviour. Since the state spaces may differ between abstract and concrete program, the refinement conditions have been defined modulo a *coupling predicate* binding the state spaces together. It has been shown that the notion of refinement induced by the *UDL* refinement condition is equivalent to the established notion of refinement.

Particular emphasis has been put on the treatment of loops. The loop in a program and its refinement are likely to be similar: Each abstract loop iteration corresponds to precisely one loop iteration in the concretion. Loops with this property are called *synchronised loops*. Inference rules for the sequent calculus have been presented which can handle synchronised loops using a *coupling invariant* binding abstract and concrete state space together. Like in Section 3.3 where rules for treatment of loops in *UDL* have been proposed, improved rules have been introduced which allow a more efficient application during a proof attempt in practice.

Finally, the refinement from abstract descriptions to Java code has been elucidated. Since the intermediate programming language provides a common language ground for the abstract and for the Java programming language, this refinement step is, from the theoretical point of view, not different to other algorithmic refinement steps. The extraction of contracts for the Java method under investigation has been explained. From the discharged proof obligations for (1) the technical issues on the Java code directly, (2) the abstract contract on the abstract program, and (3) the refinement condition, a contract for the Java method can be synthesised. If the used coupling is *syntactically functional*, the contract contains no syntactical traces of the abstract refinement layer.

Throughout the chapter, small examples have been presented to motivate and illustrate the introduced concepts.

CHAPTER 6

Case Studies

The case studies in this chapter bring together the threads begun in the previous chapters. Algorithms are presented as abstract mathematical descriptions and then refined to Java implementations applying the refinement approach presented in Chapter 5. The proof obligations which emerge from the verification are discharged using the verification system ivil introduced in Chapter 4.

The first smaller case study shows how a sorting algorithm can be formally refined into an implementation in the Java programming language. The algorithmic descriptions can be annotated with intermediate specifications such that the proofs can be performed automatically.

The second more extensive and more complex case study is an algorithm for computing shortest paths in graphs. This case study shows that an algorithm which is sufficiently complex already on the abstract level can be refined all the way down to a non-trivial implementation in the Java programming language.

6.1 Selection Sort

The first non-trivial algorithm onto which we apply the refinement chain from a most abstract description down to an implementation is the *selection sort* algorithm. Selection sort is a procedure bringing n elements within a sequence into an ascending order with $O(n^2)$ many comparisons. The runtime of selection sort is hence longer than that of other sorting algorithms (like heap sort, merge sort, etc.), but this is of no importance for our purposes.

This easier kind of algorithm has been chosen deliberately for this first case study to demonstrate how the approach can be used to verify the implementation of an algorithm through refinement without user interaction in the theorem prover. Interaction for this verification example can be restricted to annotations on the level of the sources.

The selection sort procedure is defined by the semi-formal pseudocode description in Figure 6.1a which takes a sequence $a : \mathsf{seq}(\mathsf{int})$ as input and gives another sequence $b : \mathsf{seq}(\mathsf{int})$ as output. The algorithm sorts a into b by successively finding the minimum value in the tail sequence of b from i and replacing it with its target index i.

This description relies on natural language and needs to be brought into a formalised setting first before any formal refinement can take place. This first formalisation step can evidently not be formal, we can only try to capture the meaning of the English text and bring it into the formal variant of pseudocode as closely as possible. Natural language always bears the danger of impreciseness. Here, line 8 says that t should become the value of the minimum from indices i to n. It is not mentioned whether this range from i to n is inclusive or exclusive the bounds. In this case, it is that i is meant inclusive while n is to be handled exclusive. A formal transcript of the algorithm clarifies such imprecisenesses.

This translated program `AbstractSelectionSort`, which will be used as the starting point for the refinement, is depicted in Figure 6.1. The algorithmic property which is to be proved for this algorithm is obvious: The algorithm must sort the values of a into b. This obligation has two aspects: The result sequence must contain the same values as the input (b must be a permutation of a), and the values in b must be in good order. Formally, this can be expressed as the proof obligation

$$\mathsf{true} \to [\![\texttt{abstractSelectionSort}]\!](\mathsf{seqPerm}(a,b) \land \mathsf{seqSorted}(b)) \ . \tag{6.1}$$

The two symbols $\mathsf{seqSorted} : \mathsf{seq}(\mathsf{int}) \to \mathsf{bool}$ and $\mathsf{seqPerm} : \mathsf{seq}(\alpha) \times \mathsf{seq}(\alpha) \to \mathsf{bool}$ capture the two mentioned aspects of the contract. For their formal definitions see the description of the sequence data type in Appendix A.2.2.

Proof obligation (6.1) cannot verify automatically without additional annotations. The theorem prover needs help in form of intermediate specifications telling it how to proceed at certain points. These are annotations for the following two statements:

1. The loop invariant and variant

$$\begin{aligned}\varphi_1 = {} & 0 \le i \land i < n \land \mathsf{seqPerm}(a,b) \land \mathsf{seqLen}(b) \doteq n \land \\ & (\forall k.\forall l.\ 0 \le k \land k \le l \land l \le i \to \mathsf{seqGet}(b,k) \le \mathsf{seqGet}(b,l)) \land \\ & (\forall k.\forall l.\ 0 \le k < i \land i \le l \land l < n \to \mathsf{seqGet}(b,k) \le \mathsf{seqGet}(b,l)) \land \\ \nu_1 = {} & n - i\end{aligned}$$

   need to be annotated to the while loop in line 14. The invariant ensures that sequence b is always a permutation of a, that b is sorted up to index i inclusively and that the values beyond that index are all larger than the values in the already sorted part. The variant ν_1 is the typical termination expression for a counting loop.

2. The choice operator[1] in line 16 incorporates an assertion that there exists an element with the propagated attributes.

   To symbolically execute such a statement under the $[\![\cdot]\!]$ modality, a *witness* must be provided which shows that there is a value with the chosen property. It is not so easy to find this witness for the minimum value in the sequence here. Again

[1] see Appendix A.3 for the semantics of the `choose` statement.

INFORMAL PSEUDOCODE

```
algo InformalSelectionSort
do
  b := a;
  i := 0;
  while i < (length of a) - 1
  do
    choose t such that it is the index of the minimum of the values
      in the sequence from index i to n;
    swap i and t in b;
    i := i+1
  end
end
```

INFORMAL PSEUDOCODE — (a)

PSEUDOCODE

```
algo AbstractSelectionSort
  input  a: seq(int)
  output b: seq(int)

  ensures seqSorted(b)
  ensures seqPerm(a,b)

  var i,t : int
do
  b := a;
  i := 0;
  n := seqLen(a);
  while i < n-1
    inv $\varphi_1$ var $\nu_1$
  do
    choose t such that $i \leq t < n \land (\forall k.\ i \leq k < n \rightarrow \mathsf{seqGet}(b,t) \leq \mathsf{seqGet}(b,k))$
    b := seqSwap(b, i, t);
    i := i+1
  end
end
```

PSEUDOCODE — (b)

Figure 6.1: Pseudocode for the abstract description of selection sort

we rely on a symbol from the library for sequence, in this case the polymorphic binder $\mathsf{argmin} : \alpha \times \mathsf{bool} \times \mathsf{int} \to \alpha$. The application $(\mathsf{argmin}\ x.\varphi, e)$ denotes one of the elements of type α for which the formula $\varphi \in \mathrm{Trm}^{\mathsf{bool}}$ holds and $e \in \mathrm{Trm}^{\mathsf{int}}$ has the minimum value. It is formalised by the schematic axiom

$$(\exists l. \forall x. \varphi \to e \geq l) \land (\exists x.\ \varphi) \to \varphi[x/(\mathsf{argmin}\ x.\varphi, e)] \land \\ (\forall x. \varphi \to e[x/(\mathsf{argmin}\ x.\varphi, e)] \leq e)\ .$$

The term can only denote a minimum if there is a lower bound to the values that satisfy φ. This can, for instance, be established by showing that $\mathsf{finite}((\mathsf{setComp}\ x.\varphi))$ holds. The binder argmin is a generalisation of the special description operator $\boldsymbol{\tau}_{min}$ which has been used to show completeness of the calculus in Section 3.4.2 due to $(\boldsymbol{\tau}_{min}\ x.\varphi) \equiv (\mathsf{argmin}\ x.\varphi, x)$.

Hence, we annotate as an autoactive hint to the program code in line 16 the witness

$$(\mathsf{argmin}\ k.\ i \leq k \land k < n, \mathsf{seqGet}(b,k)))$$

for the existence of the chosen value t. Using the binder opens side branches in the proof tree. For the minimum to exist, the finiteness and non-emptiness of the set of values which satisfy the guard must be shown.

The verification system *ivil* can discharge the proof obligation for the thus annotated algorithmic code automatically and instantaneously.

6.1.1 Abstract Refinement

The description `abstractSelectionSort` is too abstract for a direct refinement to an implementation in Java since the "choose" operator is a concept which has no direct realisation in the Java language. This is why a refinement on the abstract level precedes the implementation. This refinement step is an algorithmic refinement as it does not touch the representation of the data at all, but only the way in which they are processed. The performed change expands the before-mentioned choose operator by a loop searching for the minimum value amongst the relevant values in the remainder sequence.

The refined program `selectionSort` is shown in Figure 6.2. The two commented lines will become relevant in the refinement step to the implementation in the next section. The program uses the same identifiers for program variables as the abstract program even though we still adhere to the assumption of disjointness of the program variables. Using the same names is suggestive for a purely algorithmic refinement since the respective program variables are coupled by their equality. If necessary, we differentiate the variables in the refinement condition by adding primes to the abstract program variables.

Instead of the indeterministic choice in the abstract program, a loop using the newly introduced counter variable j finds the index t of the minimum entry in the sequence b from i to its end.

PSEUDOCODE

```
algo selectionSort
  input  a : seq(int)
  output b : seq(int)
  var i,j,t,n : int
do
  b := a;
  i := 0;
  n := seqLen(a);

  if n $\doteq$ 0 then return end;

  while i < n-1 do
    mark 1;  /* abstract */
    t := i;
    j := i+1;

    while j < n
      inv $\varphi_2$ var $\nu_2$
    do
      if seqGet(b,j) < seqGet(b,t) then t := j end;
      /* implementation refinement: mark 1; */
      j := j+1
    end;
    b := seqSwap(b, i, t);
    i := i+1
    /* implementation refinement: mark 2; */
  end
end
```

PSEUDOCODE – 6.1

Figure 6.2: Pseudocode for the refined abstract selection sort algorithm

The intention is to prove now that the two descriptions are equivalent. The refinement condition for the two programs according to Theorem 5.2 is

$$\psi_{eq} \to [\![\texttt{selectionSort}^{af}]\!]\langle\texttt{abstractSelectionSort}^{af}\rangle\psi_{eq}$$

with $\psi_{eq} = (a' \doteq a \wedge b' \doteq b \wedge i' \doteq i \wedge t' \doteq t)$. The box modality $[\![\cdot]\!]$ is chosen to show termination on the fly during the refinement verification. Using Theorem 5.9, termination can be inherited from the termination of `abstractSelectionSort` obtained by the proof of (6.1).

The refined program has two nested loops whereas the original description only had one. It is obvious that the outer loop is synchronised with the loop of the original description modulo ψ_{eq}. The inner loop however has no counterpart and cannot be verified using this technique. Instead a loop invariant and variant can be specified describing its behaviour adequately:

$$\begin{aligned} &\varphi_2 = (\forall k. i \leq k \land k < j \rightarrow \mathsf{seqGet}(b,t) \leq \mathsf{seqGet}(b,k)) \land i+1 \leq j \land j \leq n \land i \leq t \leq n \\ &\nu_2 = (n-j)+1 \end{aligned} \tag{6.2}$$

The outer loop requires the placement of one synchronisation point in line 13 of Figure 6.2. Synchronisation points must be matched between abstract and concrete program; and we assume the counterpart placed between lines 15 and 16 in Figure 6.1. Then the refinement in the pseudocode language can be stated by giving the involved coupling predicates:

PSEUDOCODE REFINEMENT

```
1 refine abstractSelectionSort as selectionSort
2   requires a' ≐ a
3   ensures b' ≐ b
4   mark 1
5     inv i' ≐ i ∧ b' ≐ b' ∧ n' ≐ n ∧ seqLen(b) ≐ n ∧ i < n
```

PSEUDOCODE REFINEMENT – 6.2

Note that in the pre-state the equality only is relevant for the input sequence and does not matter for the other program variables. Likewise, in the terminal state only the value b is visible after the execution of the sorting algorithm and must equal the abstract result. In the intermediate coupling predicate (line 5) these equalities cannot be spared out. Additionally, one condition on the counter of the outer loop must be preserved. The synchronised invariant rules need to abstract from the actual programs, and it may be necessary to preserve some information in addition to the coupling relation in the synchronisation invariants.

Since no data refinement has taken place in this refinement step and the coupling predicates are equalities, the property proved on the abstract description is passed on automatically to `abstractSelectionSort` without any further proof.

In the verification tool *ivil*, this refinement step can be performed automatically if one autoactive annotation is added to the abstract program. The abstract choose operation of line 16 in $\texttt{abstractSelectionSort}^{af}$ (Figure 6.1) corresponds in its assertion-free variant to the two *UDL* statements ($\mathsf{havoc}\ t'$, $\mathsf{assume}\ \varphi$). Let us assume these are at indices k and $k+1$ in the program. When symbolic execution of the refinement condition proceeds over this point $\langle k; \texttt{abstractSelectionSort}^{af} \rangle \psi_{eq}$, it yields the formula $(\exists x. \{t' := x\}(\varphi \land \langle k+2; \texttt{abstractSelectionSort}^{af} \rangle \psi_{eq}))$. An existential quantifier has been introduced by the havoc statement under the $\langle \cdot \rangle$ modality which must satisfy assumption φ.

This quantifier in the proof obligation needs to be instantiated with a sensible value to proceed the verification. It is not possible to infer the instantiation from the sequent on which the quantifier appeared as the further course of the program may be important to deduce this right instantiation.

This is why a witness for this existential quantifier needs to be stated. Yet, this instantiation reflects an important decision which has to be made in the refinement. It couples the one course of the concrete algorithm to one of the many possible courses of the abstract algorithm. By annotating another autoactive proof hint (**witness** *t*) to the choice statement (the second to this line!) we allow the symbolic execution to continue over this position. The canonical choice for the indeterministic choice is the obvious value which is the result computed by the inner loop of the refined program.

With this additional annotation, *ivil* proves the obligation instantaneously. The program has no embedded assertions, the technical proof obligation

$$\mathsf{true} \rightarrow [\texttt{selectionSort}]\mathsf{true}$$

is hence trivially valid.

6.1.2 Implementation Refinement

The algorithm description last examined in the last section is finally implementation-friendly enough to refine it to a Java method. The algorithm has been modelled such that we can preserve the algorithmic structure of the program (its control flow and loops). The step will refine the data representation severely.

Figure 6.3 shows the Java implementation onto which the pseudocode algorithm `selectionSort` will be refined. It consists of a single static method in a single class which sorts the array which it receives as argument. Unlike the abstract algorithm which uses separate variables *a* for input and *b* for output, this implementation operates in place on the input array and does not return a new copy of the array.

The first proof of this refinement step is to show the integrity of the implementation `sort`. The condition $\neg array \doteq \texttt{null} \rightarrow [\texttt{sort}]\mathsf{true}$ shows that no unexpected exceptions are raised in the program. Loop invariants which ensure that the values of *i*, *j* and *t* remain within the array boundaries must be annotated. The technical precondition pre_{tech} under which the program behaves as expected is the requirement that the parameter is different from `null`. More details on this verification step are omitted here since they do not provide new insights.

For the actual refinement condition, synchronisation points must be identified in the programs `selectionSort` and `sort`. It proved convenient to place them at the end of the two loops. The synchronisation points are marked in the Java source code as comments `/*@ mark ...; @/` with special semantics in lines 14 and 19. The corresponding synchronisation points of the refined pseudocode program are shown in Figure 6.2 embedded in comments.

The coupling relation is functional and we can reuse the function $\mathsf{intArrayAsSeq}$ which we already have encountered in the example in Section 5.5.2. It couples the elements of `array` in the current heap with the abstract sequence of integers; in the pre-state with *a* and in the post-state with *b*. Thus, the in-place sorting of `sort` can be matched to the formulation with abstract datatypes. The refinement description in Figure 6.4 uses again primed variables to distinguish the abstract

ANNOTATED JAVA

```
class SelSort {

    /*@ contract
      @  requires ¬array ≐ null
      @*/
    static void sort(int[] array) {

        for(int i = 0; i < array.length - 1; i++) {
            int t = i;
            for(int j = i+1; j < array.length; j++) {
                if(array[j] < array[t]) {
                    t = j;
                }
                /*@ mark 1; @*/
            }
            int tmp = array[i];
            array[i] = array[t];
            array[t] = tmp;
            /*@ mark 2; @*/
        }
    }
}
```

ANNOTATED JAVA – 6.3

Figure 6.3: Java implementation of selection sort

program variables from the program variables from the concrete Java signature. The pseudocode language allows the definition of abbreviations for formulas. The abbreviation `@idx_range` stating that the loop variables are within the array range is added to the coupling invariants in the synchronisation points.

The extracted Java contract for `sort` according to Observation 5.12 reads then

JAVA CONTRACT

```
/*@ contract
  @  requires ¬ array ≐ null
  @  ensures seqSorted(intArrayAsSeq(h, array))
  @  ensures seqPerm(intArrayAsSeq(h_pre, array), intArrayAsSeq(h, array))
  @*/
```

JAVA CONTRACT – 6.5

PSEUDOCODE REFINEMENT

```
abbreviation
  @idx_inrange := 0 ≤ i < arrlen(array) ∧
                  0 ≤ j ≤ arrlen(array) ∧
                  0 ≤ t < arrlen(array)

refine selectionSort as Java
  requires ¬array≐null ∧ a ≐ intArrayAsSeq(h, array)

  ensures b' ≐ intArrayAsSeq(h, array)

  mark 1
    inv b' ≐ intArrayAsSeq(h, array) ∧ n' ≐ seqLen(b) ∧
        i' ≐ i ∧ j' ≐ j ∧ t' ≐ t ∧ j < n ∧ @idx_inrange

  mark 2
    inv b' ≐ intArrayAsSeq(h, array) ∧ n' ≐ seqLen(b) ∧
        i' ≐ i ∧ j' ≐ j ∧ @idx_inrange
```

PSEUDOCODE REFINEMENT – 6.4

Figure 6.4: The refinement description for the implementation of selection sort

This convenient and comprehensible specification is self-contained, contains no more signs of the abstraction which has taken place for the refinement and is a sensible and concise full[2] functional specification of the method.

6.1.3 Remarks

It may be questioned whether for such a seemingly simple algorithm a refinement with three steps is too fine-grained a division. Apart from the intention to demonstrate the procedure in this section, the finer granularity of the steps has its benefits even for the small example:

The very first formal description had only a single loop, the second loop was introduced by the first refinement step. In the first algorithmic verification the effort could be concentrated on finding a suitable loop invariant for the outer loop while the inner loop was not an issue but has been abstracted from adequately.

With the outer loop specified and verified, the verification of the second program could fully concentrate on the inner loop, the outer loop was treated automatically by the invariant mechanism of synchronised loops. In fact, the loop invariant that served as specification for the inner loop in (6.2) can be chosen a lot more concisely if the loop can be considered alone. If both loops are verified at once, the inner loop invariant must imply parts of the outer loop invariant to allow the latter to be inductive.

[2] The examination in this section does not cover a `modifies`-clause for the method `sort`, but separation of concerns applies and the modifies clause $\{array\}$ can be verified in the technical verification step.

This examples shows also another thing: It is tempting to use the idea of refinement to formulate an algorithm on the set of mathematical integers and then refine it to the numerical datatype which is present in the implementation language, Java's 32-bit signed integers for instance. The problems that arise here are the same technical challenges mentioned in Section 4.4.2. It is difficult to handle situations in which bounded and unbounded integer values exist side by side. The overflow check assertions mentioned in Section 4.4.2 can, of course, also be applied within a refinement proof to show that the bounded values behave like the mathematical integers.

6.2 Breadth First Search

The second refinement example is more complex. It shows that pseudocode is an adequate formulation language for algorithms and that the refinement concept can be used to synthesise sophisticated contracts for an implementation from an abstract description. This example will be refined by two steps from a set-theoretical pseudocode description down to an efficient implementation in Java using boolean arrays.

The example has been taken from the offline verification competition which took place prior to the conference *Verified Software: Theories, Tools, Experiments* (VSTTE 2012). The organisers give an overview of the competition in (Filliâtre et al., 2012) including a brief description of the assignments. The assignment has been worked on by many participants in the competitions. Most of the solutions for them are on a higher abstraction level. There are solutions for the tools VCC (Dahlweid et al., 2009) and VeriFast (Jacobs et al., 2011) which also operate on the implementational level, but we did not have access to their solutions to compare.

The assignment was to verify an algorithm which computes the shortest distance between two vertices in a directed graph using breadth first search. The edges in the graph are not weighted, that is, the shortest path is the path with the fewest visited nodes. The algorithm as it has been presented on the occasion is depicted in Figure 6.5a. In the algorithm, the function application $succ(n)$ is used to denote the successors of a node n in the graph.

Figure 6.5b shows the verbatim transliteration of the problem description into our formalised pseudocode language. The two descriptions resemble each other very much, the differences are only owing to concrete syntax. This resemblance shows that the pseudocode language is indeed a suitable means to formulate algorithms on an abstract level.

The verification process of the algorithm takes the following steps:

1. Verification of the algorithm on the purely algorithmic level using the mathematical notions of the original problem description.

2. Refinement onto a pseudo-code algorithm which uses more implementation-friendly, yet still abstract data structures: sequences instead of sets and integer indices instead of vertices.

3. Implementation of the algorithm in a Java class in accordance with this last refined description. A lightweight verification is conducted that the implementation is free of exceptions.

4. Refinement from the intermediate refined algorithm to the Java implementation.

```
V <- {source};
C <- {source};
N <- {};
d <- 0;
while C is not empty do

  remove one vertex v from C;
  if v = dest then return d; endif
  for each w in succ(v) do
    if w is not in V then
      add w to V;
      add w to N;
    endif
  endfor
  if C is empty then
    C <- N;
    N <- {};
    d <- d+1;
  endif
endwhile
fail "no path"
```

(a) Original problem description

```
V := {src};
C := {src};
N := {};
d := 0;
while !(C = {}) do
  choose v such that v in C;
  C := C \ {v};
  if v = dest then return end;
  iterate succ(v) with w in tovisit do
    if not w in V then
      V := V \/ {w};
      N := N \/ {w}
    end
  end;
  if C = {} then
    C := N;
    N := {};
    d := d+1
  end
end;
d := -1
```

(b) Pseudocode transliteration

Figure 6.5: Description of the breadth-first-algorithm

The case study is significantly more complex than the selection sort algorithm presented in the last section. The data refinements make larger changes, the set-theoretic expressions involved in the algorithm are non-trivial and their translation in the refinement give more complex proof obligations. The verification conditions are not necessarily larger but more intricate formulas. As a consequence, the proofs for this section cannot be conducted automatically or autoactively by the *ivil* system. At various occasions, manual user interaction has been necessary to support the theorem prover.

Astonishingly, these interactions where sometimes purely of propositional nature. If the proof obligation contained a conjunction $A \wedge B$ of two parts to be shown, a split of the current sequent into one for A and one for B allowed the SMT solver to discharge both parts relatively fast (in a matter of seconds) while the conjunction could not be proved even within minutes sometimes. This interaction was definitely not a theoretically required input for the verification engine, but was a helpful guidance for the prover which showed it where to split the proof condition.

6.2.1 Algorithm Verification

For the logical encoding we model the set of vertices as an unparametrised new type *vertex*. The successor map is then a function $succ : vertex \to \mathsf{set}(vertex)$ on this type.

The path searching algorithm operates in rounds. Every round explores the nodes which have a certain distance $d : \mathsf{int}$ from the source *src*. In round 0, only the source itself is visited; in round 1, all nodes which can be reached with one step are considered, and so on. Three sets of vertices are used in the algorithm: The set $V : \mathsf{set}(vertex)$ (for *visited*) contains all vertices which have been considered by the algorithm so far. The set $C \subseteq V$ (for *current*) contains the nodes of the currently visited distance level which are still to be checked while $N \subseteq V$ (for *next*) contains the nodes of the next distance level which have been discovered lately. When all nodes in C have been checked, the algorithm steps over to the next distance level and N becomes the next input set C.

Figure 6.6 shows an example directed graph during the visitation. The graph is drawn in such a way that the horizontal distance of a vertex to *src* equals its minimum distance to it. For instance, nodes a and b are one step away from *src* while d is two steps away. There are paths to vertices which are longer than the minimum distance but they do not matter. The sets V, C and N are outlined in the sketch. At the time of the visitation, the current distance level is $d = 2$. At the beginning of round 2, C equalled $\{d,e,f\}$. Node d has already been explored and gave rise to adding h,g into N. Vertex e is to be examined next. This examination of e will ensure that i is added to N as well (h is already in N). Examination of the successors of f does not yield any new findings. Then the next round for $d = 3$ commences in which $\{g,h,i\}$ will be used as set C of current nodes.

Whenever the algorithm reaches the destination node, it finishes and reports the current level as the minimum distance between source and destination. The algorithm is thus a simplified version of Dijkstra's shortest path algorithm for the special case that all edges have the same weight 1.

The distance between vertices is modelled using two primitive recursive predicates. The first predicate $connect : vertex \times vertex \times \mathsf{int} \to \mathsf{bool}$ formalises reachability between vertices and is defined by

$$(\forall a.\forall b.\forall n.\ connect(a,b,n) \doteq \\ \text{if } n \leq 0 \text{ then } a \doteq b \text{ else } (\exists x.connect(a,x,n-1) \land b \in succ(x))))$$

and $minconnect : vertex \times vertex \times \mathsf{int} \to \mathsf{bool}$ axiomatised as

$$(\forall a.\forall b.\forall n.\ minconnect(a,b,n) \doteq \\ (connect(a,b,n) \land (\forall m.\ 0 \leq m \land m < n \to \neg connect(a,b,m)))) \ .$$

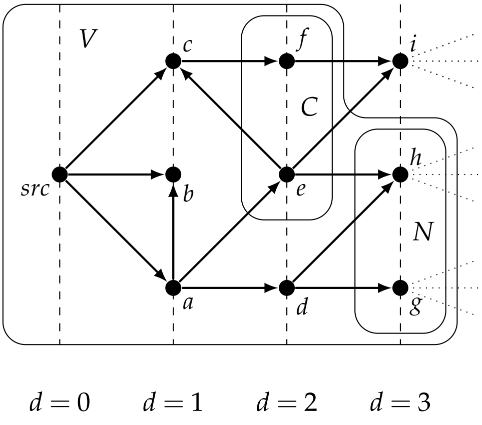


Figure 6.6: Example for a directed graph during breadth-first-search

The property that we want to show is that the algorithm (we denote the translation to a *UDL* program as *bfs*) with the input variables $src, dest : vertex$ and the result variable d : int returns the shortest distance between src and dest in d, that is that $minconnect(src, dest, d)$ holds. If the two vertices are not connected, the value $d = -1$ should be returned. The verified property can be expressed as the *UDL* contract

$$\begin{aligned}\mathsf{finite}(\mathsf{fullset}^{[\mathsf{set}(vertex)]}) \to [\![bfs]\!](d \geq -1 \\ \wedge\ (d \doteq -1 \to (\forall m. m \geq 0 \to \neg minconnect(src, dest, m))) \\ \wedge\ (d > 0 \to minconnect(src, dest, d)))\ .\end{aligned}$$

The atomic formula $connect(a, b, n)$ mans that vertex b can be reached from a with (precisely) n steps in the graph. $minconnect(a, b, n)$ additionally implies that there is no shorter connection between a and b.

The finiteness in the precondition is a necessary condition as the algorithm might not finish if there is no connection from *src* to *dest* and the graph is infinite. The only small difference of the verified algorithm to the problem description is that, if no path exists between src and dest, a negative result is given instead of raising a failure.

The algorithm has two nested loops (one while loop and one set iteration). If these loops are furnished with sufficiently strong loop invariants and variants, the proof can be conducted almost automatically. The sufficient loop invariant employed for the outer while loop is the formula

$$
\begin{aligned}
&d \geq 0 \;\wedge\; \neg dest \in (V \setminus N) \setminus C \qquad && (6.3)\\
\wedge\ &(\forall x.\, x \in C \rightarrow minconnect(src, x, d)) && (6.4)\\
\wedge\ &(\forall y.\, y \in N \leftrightarrow minconnect(src, y, d+1) \wedge y \in V) && (6.5)\\
\wedge\ &(\forall z.\, z \in V \setminus N \leftrightarrow (\exists n.\, 0 \leq n \leq d \wedge connect(src, z, n))) && (6.6)\\
\wedge\ &(\forall w.\, minconnect(src, w, d+1) \rightarrow (\exists c.\, w \in succ(c) \wedge c \in C) \vee a \in N) && (6.7)\\
\wedge\ &(C \doteq \emptyset \rightarrow N \doteq \emptyset)\,. && (6.8)
\end{aligned}
$$

This invariant is a straightforward and concise formalisation of the state during a run of the algorithm. The fact that this invariant spans, already on this high level of abstraction, over six lines is a strong indicator that the verification better be started on this higher level leaving aside technical issues of an implementation. (6.3) captures information on the current round and that the destination has not yet been reached yet. (6.4), (6.5) and (6.6) define the contents of the sets C, N and V. The last two (6.7) and (6.8) are needed to deduce the postcondition in case that no path between source and destination exists.

The inner iterate-loop is less complex, it merely iterates over all successors $succ(v)$ of v and adds them to N and V if they had not yet been visited. The overall effect of the loop is hence that $V = V_0 \cup succ(v)$ and $N = N_0 \cup (succ(v) \setminus V_0)$ where V_0 and N_0 are the values of V and N prior to the inner loop. For the loop invariant, the vertices in $tovisit \subseteq succ(v)$ have not been visited yet and the invariant is the weaker $V = V_0 \cup (succ(v) \setminus tovisit) \;\wedge\; N = N_0 \cup ((succ(v) \setminus tovisit) \setminus V_0)$.

Termination can be guaranteed using the decreasing variant expressions $\mathtt{C}(V \setminus (C \cup N))$ for the outer loop and $tovisit$ for the inner loop. Both invariants are of set type and the well-founded precedence relation for finite sets is fixed as $A \prec B :\leftrightarrow A \subseteq B \wedge (\exists x.\, \neg x \in A \wedge x \in B)$.

The proof needs some interaction: Expansion of set theoretical properties, quantifier instantiation, case distinctions guide the SMT solver to the solution.

6.2.2 Refinement on Algorithmic Level

We have now verified that the algorithm given in the pseudocode indeed computes the shortest distance in an unweighted graph. This description is now in a first refinement step subjected to a data refinement transforming its data structures into concepts closer to an implementation in a programming language.

The abstract description uses typical elements of an algorithm explanation in pseudocode: A problem-specific data-type ($vertex$) is used to denote the elements on which the algorithm operates. Abstract data structures over them (here $\mathsf{set}(vertex)$) are used to define structures over the elements.

The data refinement step refines precisely these two aspects without changing the control flow of the algorithm. The logical type $vertex$ introduced for this challenge is removed and the nodes in the graph are represented by integer numbers in a well-specified range. The finite sets of vertices used in the abstract description are replaced by sequences of boolean values. A sequence of boolean values can be seen as a function from its index range to true or false, and thus as the characteristic function of a subset of its index range. By ensuring that the sequences have the integer values which represent vertices in their index range, they can be used to refine the sets in the algorithm.

Vertices and these index integers must be identified in the coupling predicates. The index range of a sequence is always an initial sequence of the natural numbers, vertices are therefore refined as numbers between 0 and the number of vertices. We introduce function symbols $vi : vertex \to \mathsf{int}$ and $iv : \mathsf{int} \to vertex$ for this partial bijection which are defined as

$$(\forall v^{vertex}.\ iv(vi(v)) \doteq v \land 0 \leq vi(v) < \mathsf{card}(\mathsf{fullset}^{[vertex]})) \land \\ (\forall i^{\mathsf{int}}.\ 0 \leq i < \mathsf{card}(\mathsf{fullset}^{[vertex]}) \to vi(iv(i)) \doteq i)\ .$$

This definition only describes a conservative extension if the set of vertices is known to be finite, a fact we had already assumed before.

Such a bijection relationship is typical for a data refinement where the behaviour does not change, but the data model is modified. Having the explicit function symbols *vi* and *iv* for the bijection helps for the formulation of the coupling predicates and during the proofs.

The structure of the algorithm is not changed in the refinement, yet the refined procedure possesses less indeterminism than the original: In the second loop of *bfs*, an arbitrary element w of the not-yet-visited successors of v may be chosen (line 18 in Fig. 6.8b). The refinement narrows this operation and always chooses the vertex with the minimum index $vi(w)$. The purpose of this concretion is to save the expenses of keeping an explicit list of nodes still to be visited. Selecting the minimum in all cases allows us to keep a single integer value as index to the part yet to be traversed.

Reducing indeterminism is a typical operation during a refinement step. On the abstract level, things are kept deliberately open to not restrict the possibilities of an implementation. In the case of the example, the particular order in which the set is traversed does not play a role in the correctness of the algorithm or its result. However, when the set becomes concretised as a sequence, the elements are ordered, the data has more structure. This additional information on the concrete level can be exploited to optimise the algorithm.

We can now proceed and present the algorithm bfs_{seq} which refines *bfs*. To differentiate between variables on the original description level and the refinement, we suffix every variable in the refined algorithm with *seq* (as it uses *seq*uences). There is one program variable $p_{seq} \in \mathrm{PVar}(bfs_{seq})$ for every abstract variable $p \in \mathrm{PVar}(bfs)$. The refinement step applied here is a functional refinement, the coupling predicates give terms to compute the abstract values from the concrete. The program variables and their coupling predicates are shown in Table 6.7.

Variable in *bfs*	**refined in** bfs_{seq}
Coupling predicate	
$succ : vertex \rightarrow \mathsf{set}(vertex)$	$succ_{seq} : \mathsf{int} \rightarrow \mathsf{seq}(\mathsf{bool})$
$(\forall v.\ succ(v) \doteq (\mathsf{setComp}\ x.\ \mathsf{seqGet}(succ_{seq}(vi(v)), vi(x)))$	
$src, dest : vertex$	$src_{seq}, dest_{seq} : \mathsf{int}$
$src \doteq iv(src_{seq}),\ dest \doteq iv(dest_{seq})$	
$d : \mathsf{int}$	$d_{seq} : \mathsf{int}$
$d \doteq d_{seq}$	
$V : \mathsf{set}(vertex)$	$V_{seq} : \mathsf{seq}(\mathsf{bool})$
$V \doteq (\mathsf{setComp}\ x.\mathsf{seqGet}(V_{seq}, vi(x))) \land \mathsf{seqLen}(V_{seq}) \doteq size$	
$C : \mathsf{set}(vertex)$	$C_{seq} : \mathsf{seq}(\mathsf{bool})$
$C \doteq (\mathsf{setComp}\ x.\mathsf{seqGet}(C_{seq}, vi(x))) \land \mathsf{seqLen}(C_{seq}) \doteq size$	
$N : \mathsf{set}(vertex)$	$N_{seq} : \mathsf{seq}(\mathsf{bool})$
$N \doteq (\mathsf{setComp}\ x.\mathsf{seqGet}(N_{seq}, vi(x))) \land \mathsf{seqLen}(N_{seq}) \doteq size$	
$v, w : vertex$	$v_{seq}, w_{seq} : \mathsf{int}$
$v \doteq iv(v_{seq}),\ w \doteq iv(w_{seq})$	
$tovisit : \mathsf{set}(vertex)$	—
$tovisit \doteq (\mathsf{setComp}\ x.\ vi(x) > w_{seq}) \cap succ(v)$	

Table 6.7: Abstract and refined variables and coupling predicates

The coupling for the distance output value is trivial (the result must be the same), the currently examined vertices are kept in program variables v, w and their counterparts must hold the same nodes modulo the bijection iv. The coupling for the sets V, C and N is more involved and it says that the sets must contain precisely those nodes x^{vertex} for which the corresponding entry in the according characteristic sequence V_{seq}, C_{seq} or N_{seq} holds the value true. The set *tovisit* of vertices which are still to be processed has no counterpart in the refinement but can be computed from the concrete variables w_{seq} and v_{seq}.

The original program *bfs* and its refinement bfs_{seq} are listed in Figure 6.8. It is evident that, compared to the concise formalisation of the set-theoretic description, the refined program is more extensive and less intuitive to comprehend. Both pseudocode texts have been annotated with synchronisation points (`mark 1`, …, `mark 6`) marking the program indices to be used with the synchronised loop invariant rule with multiple synchronisation points from Section 5.4.4. Each pair of identical mark statements marks program states at which a coupling predicate must hold.

Before we check that the refinement condition is valid, it is advisable to prove $\mathsf{true} \to [bfs_{seq}]\mathsf{true}$, that is, that the embedded assertions of bfs_{seq} hold. In this case, there is only one assertion, associated to the second choose statement (line 18 in Fig. 6.8b) selecting the minimum element with the property. Using the binder symbol argmin that we have already used for the first refinement in the selection sort algorithm, we can specify the smallest witness for the existence of the chosen predicate as $(\mathsf{argmin}\ x^{\mathsf{int}}.\ t_{seq} \leq x \land x < size_{seq} \land \mathsf{seqGet}(succ_{seq}(v_{seq}), x),\ x)$.

The program contains two loops; it would, hence, suffice to use only two synchronisation points (for instance mark 1 and mark 3), one in each loop to break up the cyclic structure of the programs. This is along the same lines as applying the invariant rule twice during the correctness proof. However, experience showed that having more check points in the programs makes the proof significantly shorter and easier to conduct. The reason is the following: Whenever the program flow may split the execution into several paths, it is likely that a similar effect also appears on the run of the other program. The overhead (measured in the number of proof goals) grows exponentially in the number of branches. Providing more synchronisation points reduces this load and splits the proof on a rather high level into subproofs. The costs of this reduction in proof size is the specification of extra coupling invariants. In the case of this case study, and we believe this to be prototypical, however, the invariants are relatively similar to one another.

The proof of the refinement condition $\psi_{pre} \to [\![bfs^{af}_{seq}]\!]\langle bfs^{af}\rangle\psi_{post}$ requires eight coupling predicates, one coupling precondition, one coupling postcondition and one coupling invariant per synchronisation point. The predicates used in the proof are

```
V := {src};
C := {src};
N := {};
d := 0;

while ¬(C $\doteq$ {}) do
  mark 1;

  choose v such that v $\in$ C;
  C := C \ {v};
  if v $\doteq$ dest then
    return
  end;
  mark 2;

  iterate succ(v)
  with w in tovisit do



    mark 3;
    if ¬w$\in$V then
      V := V $\cup$ {w};
      N := N $\cup$ {w}
    end
    mark 4;

  end;
  mark 5;
  if C $\doteq$ {} then
    C := N;
    N := {};
    d := d+1
  end
  mark 6;
end;
d := -1
```

(a) *bfs*: Using sets of vertices

```
$V_{seq}$ := (seqDef i. 0, $size_{seq}$, i $\doteq$ $src_{seq}$);
$C_{seq}$ := (seqDef i. 0, $size_{seq}$, i $\doteq$ $src_{seq}$);
$N_{seq}$ := (seqDef i. 0, $size_{seq}$, false);
$d_{seq}$ := 0;

while ($\exists$k. 0$\leq$k<$size_{seq}$ $\land$ seqGet($C_{seq}$, k)) do
  mark 1;

  choose $v_{seq}$ such that 0$\leq$$v_{seq}$<$size_{seq}$ $\land$ seqGet($C_{seq}$, $v_{seq}$);
  $C_{seq}$ := seqUpdate($C_{seq}$, $v_{seq}$, false);
  if $v_{seq}$ $\doteq$ $dest_{seq}$ then
    return
  end;
  mark 2;
  $w_{seq}$ := 0;
  while ($\exists$i. $w_{seq}$$\leq$i<$size_{seq}$ $\land$ seqGet($succ_{seq}$($v_{seq}$), i)) do
    $t_{seq}$ := $w_{seq}$;
    choose $w_{seq}$ such that $t_{seq}$$\leq$$w_{seq}$<$size_{seq}$
       $\land$ seqGet($succ_{seq}$($v_{seq}$), $w_{seq}$)
       $\land$ ($\forall$j. $t_{seq}$$\leq$j<$w_{seq}$ $\rightarrow$ ¬seqGet($succ_{seq}$($v_{seq}$), j));
    mark 3;
    if ¬seqGet($V_{seq}$, $w_{seq}$) then
      $V_{seq}$ := seqUpdate($V_{seq}$, $w_{seq}$, true);
      $N_{seq}$ := seqUpdate($N_{seq}$, $w_{seq}$, true)
    end;
    mark 4;
    $w_{seq}$ := $w_{seq}$ + 1
  end;
  mark 5;
  if ¬($\exists$i. 0$\leq$i<$size_{seq}$ $\land$ seqGet($C_{seq}$, i)) then
    $C_{seq}$ := $N_{seq}$;
    $N_{seq}$ := (\seqDef i. 0, $size_{seq}$, false);
    $d_{seq}$ := $d_{seq}$+1
  end;
  mark 6
end;
$d_{seq}$ := -1
```

(b) bfs_{seq}: Using sequences of boolean values

Figure 6.8: Data refinement from sets to sequences

(various) conjuncts of the basic predicates presented in Table 6.7, in particular we have the pre- and postconditions

$$\begin{aligned}\psi_{pre} := {} & \mathsf{fullset}^{[vertex]} \doteq (\mathsf{setComp}\ v.\ (\exists i.\ 0 \leq i \land i < size \land v \doteq iv(i))) \land \\ & size \doteq size_{seq} \land src \doteq iv(src_{seq}) \land dest \doteq iv(dest_{seq}) \land \\ & (\forall v.\ succ(v) \doteq (\mathsf{setComp}\ x.\ \mathsf{seqGet}(succ_{seq}(vi(v)), vi(x)))) \\ \psi_{post} := {} & d \doteq d_{seq}\end{aligned} \tag{6.9}$$

which are syntactically functional.

Termination of the refined algorithm is directly inherited from the abstract algorithm.

6.2.3 Refinement to Implementation

The second refinement step is now the implementation of the algorithm as described in bfs_{seq} in a Java program. The sets of the original algorithm description, the sequences of the first refinement will there be represented as boolean arrays.

The intermediate data refinement step was devised in such a manner that the second refinement step is relatively canonical now. The data structures need not be tremendously altered any more. We adopt in the Java implementation the concept of boolean sequences standing for the characteristic functions of the sets of vertices. The sequences are, quite naturally, implemented by arrays of `booleans` in the Java program. The successor function $succ_{seq} : \mathsf{int} \to \mathsf{seq}(\mathsf{bool})$ is refined as an array `boolean adjacency[][]` representing the adjacency matrix as a two-dimensional array.

Unlike in the case for selection sort, this algorithm is not implemented in one solitary method. Theoretically, it could be, but both for the sake of better code quality and to ease the burden of the verification, several aspects of the algorithm can be sourced out into small helper methods. These methods handle, in particular, the management of the arrays which implement the sets of vertices. Figure 6.9 lists the declarative skeleton of the implementing class `BFS`. It contains the declarations of fields and methods and the contracts of the helper methods. The main method which performs the actual computation is `minDistance`. It and its contract will be listed later. Appendix B.2 fully lists the class with its methods, their contracts and additional specification comments.

Two of the methods (`isEmpty` and `first`) are queries; `isEmpty` computes whether an index in the array exists which holds the value `true`; `first` retrieves the first such index. The method `copy` copies the contents of one array to another array of the same length, and `clear` sets all values in the array to `false`. All methods have relatively straightforward tasks to implement, but each needs a for-loop to finish their job. The complexity of their tasks is moderate and when annotated with canonical loop invariants and variants, the fully automatic verification of the contracts of the helper function takes *ivil* approximately 4 seconds.

JAVA

```
class BFS {
    int size;
    boolean[][] adjacency;

    /*@ contract
      @   requires ¬array ≐ null
      @   ensures -1 ≐ res^int ∨ from ≤ res^int ∧ res^int < arrlen(array)
      @   ensures res^int ≥ from  → h[array, idxBool(res^int)] ∧
      @               (∀i. from≤i ∧ i<res^int → ¬h[array, idxBool(i)])
      @   ensures res^int ≐ -1 → (∀i. 0≤i ∧ i<arrlen(array) →
      @                            ∧ h[array,idxBool(i)])
      @   modifies empty
      @*/
    int first(boolean[] array, int from) { ...  }

    /*@ contract
      @   requires ¬array ≐ null
      @   ensures res^bool ≐ ¬(∃i. 0 ≤ i ∧ i < arrlen(array) ∧
      @                            h[array, idxBool(i)])
      @   modifies empty
      @*/
    boolean isEmpty(boolean[] array) { ... }

    /*@ contract
      @   requires ¬array ≐ null
      @   ensures (∀i. 0 ≤ i ∧ i < arrlen(array) →
      @                            ¬h[array, idxBool(i)])
      @   modifies {array}
      @*/
    void clear(boolean[] array) { ... }

    /*@ contract
      @   requires ¬target ≐ null
      @   requires ¬source ≐ null
      @   requires arrlen(source) ≐ arrlen(target)
      @   ensures (∀i. 0 ≤ i ∧ i < arrlen(target) →
      @     h[target, idxBool(i)] ≐ h[source, idxBool(i)])
      @   modifies {target}
      @*/
    void copy(boolean[] target, boolean[] source) { ... }

    /**
     * This method computes the distance between src and dest using
     * the breadth-first-search algorithm.
     */
    int minDistance(int src, int dest) { ... }
}
```

JAVA – 6.6

Figure 6.9: Skeleton of the Java class BFS implementing the breadth first search

Let us now turn to the implementation `minDistance` of the actual algorithm. The code of the method and its technical contract are listed in Figure 6.10. The method body possesses six synchronisation points which directly correspond to the six marks in Figure 6.8b.

The technical contract which is needed to show that the implementation behaves well with respect to throwing unexpected exceptions is more extensive for this method than for the smaller examples we have looked at so far. It contains technical details which were not an issue on the abstract descriptions. In particular, the contract requires that all references in the adjacency matrix do not reference to `null` and that all rows in the matrix have the same length `size`. This contract together with the minimal technical loop annotations allow the verification of the technical proof obligation $\mathit{pre}_{\mathit{tech}} \to [\mathtt{minDistance}](\mathit{res}^{[\mathsf{int}]} \geq -1)$. The verification system *ivil* can do this proof without any further user interaction, but it takes the system approximately 20 seconds. The method is significantly more complex than other methods we have examined: It possesses two nested loops, several conditional statements and six method calls which must be handled by their respective method contracts.

This shows that the separation of concerns as propagated in Section 5.1 has been achieved for the example. The technical issues have been factored out and are considered separated from the algorithm. The non-trivial technical precondition proves that the technical aspects are *not* negligible and must be considered. A verification of all concerns at the same time would have been drastically more challenging.

With these technical issues proved correct, we can safely remove the assertions from the program `minDistance` and use the assertion-free variants for the refinement proof condition. Since six synchronisation marks have been specified, eight coupling predicates need to be specified in the refinement declaration: one for the pre-, one for the postcondition, and one for each synchronisation point. The coupling invariants comprise several aspects and are very similar to another. Therefore, three *abbreviations* have been introduced such that these aspects need not always be spelled out. The coupling predicates become thus more comprehensible.

Figure 6.11 shows the refinement definition which accompanies the algorithm descriptions. The abstract variable names (of program $\mathit{bfs}_{\mathit{seq}}$) are primed again. The introduced abbreviations (lines 1–21) precede the actual refinement definition.

Abbreviation *@vars_coupled* captures the functional part of the refinement relation. The values of the abstract program variables $V', C', N', d', \mathit{src}', \mathit{dest}'$ and size' can be computed from their Java counterparts. The boolean sequences use the abstraction function boolArrayAsSeq which is analogous to the earlier used intArrAsSeq but for `boolean` values rather than for `int`. The technical coupling invariant *@arrays* is needed to guarantee some properties of the arrays which must be maintained. They must not alias with another nor with the adjacency matrix. Additionally, their length must always be constant and equal to size'. The abbreviation *@succ* finally captures the refinement of the successor function succ' into the `adjacency` matrix. The refinement is again functional and the abstraction for each relevant $\mathit{succ}'(v)$ is given using the function boolArrayAsSeq.

JAVA

```
class BFS {
  int size;
  boolean[][] adjacency;
  // ...

  /*@ contract
    @   requires ¬h[this, adjacency] ≐ null
    @   requires h[this, size] > 0
    @   requires arrlen(h[this, adjacency]) ≐ h[this, size]
    @   requires (∀i. 0 ≤ i ∧ i < h[this, size] →
    @               !h[h[this, adjacency], idxRef(i)] ≐ null
    @             ∧ arrlen(h[h[this, adjacency], idxRef(i)]) ≐ h[this, size])
    @   requires 0 ≤ src ∧ src < h[this, size]
    @   requires 0 ≤ dest ∧ dest < h[this, size]
    @   ensures -1 ≤ res^int
    @   modifies freshObjects(h)
    @   decreases 1
    @*/
  int minDistance(int src, int dest) {
    boolean[] V = new boolean[size];
    boolean[] C = new boolean[size];
    boolean[] N = new boolean[size];

    V[src] = true;
    C[src] = true;
    int d = 0;

    /*@ maintains 0 ≤ d
      @ decreases 2
      @ modifies {V, C, N}
      @*/
    while(!isEmpty(C)) {
      //@ mark 1;
      int v = first(C, 0);

      C[v] = false;
      if(v == dest) {
        return d;
      }

      /*@ maintains 0 ≤ w
        @ decreases 2
        @ modifies {V, N}
        @*/
      //@ mark 2;
      int w = 0;
      while(true) {
        w = first(adjacency[v], w);
        if(w == -1) {
          break;
        }

        //@ mark 3;
        if(w < size && !V[w]) {
          V[w] = true;
          N[w] = true;
        }

        //@ mark 4;
        w++;
      }

      //@ mark 5;
      if(isEmpty(C)) {
        copy(C, N);
        clear(N);
        d++;
      }

      //@ mark 6;
    }
    return -1;
  }
}
```

JAVA – 6.7

Figure 6.10: Java method implementing breadth first search

With these auxiliary abbreviations defined, the actual refinement (from line 23 on) can be specified. The refinement precondition (ψ_{pre}, line 24) is followed by the postcondition (ψ_{post}, line 25) and the coupling invariants for the six synchronisation points. It is obvious that the coupling invariants are very similar and each have only small additions to the abbreviated conditions.

The verification of the refinement condition needs heavy user interaction in the verifier: Most notably, the choice operator in the abstraction must be instantiated with the corresponding choice on the more concrete level. This corresponds to an observation made for the selection sort verification. The choose operator of line 18 in the program bfs_{seq} (Figure 6.8b) is actually deterministic. Still, we must provide the instantiation of the extensional quantifier to proceed with the symbolic execution. Luckily, we can take the value obtained by the execution of the Java code and use it as witness.

Other user interactions are owing to the complexity of the proof obligation. The underlying decision procedure is often overburdened by the verification conditions spanning over two levels of abstraction. As has been mentioned and reported earlier, simple propositional splits and small canonical lemmas (introduced using the "cut" rule of the calculus interactively) guided the theorem prover. We use the decision procedure as a black box and cannot look inside it when it cannot decide a proof obligation; we could therefore only speculate about the reasons for this kind of behaviour.

6.2.4 Extracting a Method Contract

Two formal refinement steps have been presented and all arisen verification conditions have been successfully discharged. It remains, in a final step, to extract a functional contract for the main method of the Java implementation. This contract should not make any references to the more abstract formulations using sequences or sets of vertices but be completely in terms of the Java implementation.

Recall that the formulated contract of the original most abstract algorithm *bfs* is the following:

$$\begin{aligned}\mathsf{finite}(\mathsf{fullset}^{[\mathsf{set}(vertex)]}) \to [\![\mathit{bfs}]\!]\big(&d \geq -1 \\ \wedge\ &(d \doteq -1 \to (\forall m.m \geq 0 \to \neg \mathit{minconnect}(src,dest,m))) \\ \wedge\ &(d > 0 \to \mathit{minconnect}(src,dest,d))\big)\end{aligned} \tag{6.10}$$

The result d is either -1 if $src : vertex$ and $dest : vertex$ are not connected, or it holds the minimum distance between the two nodes.

The coupling predicates in the first refinement step (6.9) are syntactically functional as they provide definitions for abstract entities in terms of the concretion. In particular, the extend of the type *vertex* is fixed by means of the equality $\mathsf{fullset}^{[vertex]} \doteq (\mathsf{setComp}\ v.\ (\exists i.\ 0 \leq i \wedge i < size \wedge v \doteq iv(i)))$. The elements of the domain $\mathcal{D}^{vertex}$ are the images of the interval $\{0,\ldots,size-1\}$ under the partial bijection iv. This immediately implies the domain is finite, and the precondition $\mathsf{finite}(\mathsf{fullset}^{[vertex]})$ from (6.10) is under these circumstances equivalent to true and can be dropped. Using the functional coupling predicates of (6.9), the following contract for the intermediate program bfs_{seq} can be extracted:

$$\begin{aligned}\mathsf{true} \to [\![bfs_{seq}]\!](\quad & d_{seq} \geq -1 \\ & \wedge (d_{seq} \doteq -1 \to (\forall m.m \geq 0 \to \neg minconnect(iv(src_{seq}), iv(dest_{seq}), m))) \\ & \wedge (d_{seq} > 0 \to minconnect(iv(src_{seq}), iv(dest_{seq}), d_{seq})))\end{aligned}$$

This contract has still got references to the abstract world in form of the function applications of *minconnect*. If according new function symbols $connect_{seq}, minconnect_{seq} : \mathsf{int} \times \mathsf{int} \times \mathsf{int} \to \mathsf{bool}$ are defined in the same way but not with respect the abstract succession function *succ* but using its refinement $succ_{seq}$, then (6.10) can be reformulated without reminiscences to the abstract world as

$$\begin{aligned}\mathsf{true} \to [\![bfs_{seq}]\!](\quad & d_{seq} \geq -1 \\ & \wedge (d_{seq} \doteq -1 \to (\forall m.m \geq 0 \to \neg minconnect_{seq}(src_{seq}, dest_{seq}, m))) \\ & \wedge (d_{seq} > 0 \to minconnect_{seq}(src_{seq}, dest_{seq}, d_{seq}))) \; .\end{aligned} \tag{6.11}$$

For the refinement of this contract to a contract for the Java program, we need to look at the refinement definition given in Figure 6.11, in particular at the coupling pre- and postcondition in line 24 and 25. They are both functional and the program variables in (6.11) can be replaced by their equally named Java counterparts.

To remove dependency on the abstract algorithms, we need to reformulate the postcondition (like before) replacing the function symbols $connect_{seq}$ and $minconnect_{seq}$ by symbols that are not defined in terms of abstract entities but solely over the elements present in the Java world. As the connection of nodes on the Java level depends on entries of the adjacency matrix, the corresponding function symbols $connect_{Java}, minconnect_{Java} : \mathsf{heap} \times \mathsf{ref} \times \mathsf{int} \times \mathsf{int} \to \mathsf{bool}$ for Java must hence depend on the heap and the `BFS` object. They are defined by the axioms

$$\begin{aligned}&(\forall h.\forall r.\forall a.\forall b.\forall n.\ connect_{Java}(h,r,a,b,n) \doteq \\ &\text{if } n \leq 0 \text{ then } a \doteq b \text{ else } (\exists x. 0 \leq x \wedge x < h[r, \mathtt{size}] \wedge connect_{Java}(h,r,a,x,n-1) \wedge \\ &\qquad\qquad h[h[h[r, \mathtt{adjacency}], \mathsf{idxRef}(x)], \mathsf{idxBool}(b)]))\end{aligned}$$

and

$$\begin{aligned}&(\forall h.\forall r.\forall a.\forall b.\forall n.\ minconnect_{Java}(h,r,a,b,n) \doteq \\ &\qquad (connect_{Java}(a,b,n) \wedge (\forall m.\ 0 \leq m \wedge m < n \to \neg connect_{Java}(a,b,m)))) \; .\end{aligned}$$

The clumsy-looking expression $h[h[h[r, \mathtt{adjacency}], \mathsf{idxRef}(x)], \mathsf{idxBool}(b)]))$ in the defining axiom for $connect_{Java}$ is the logical equivalent of the corresponding Java expression `r.adjacency[x][b]`.

These definitions can now replace the functions $\mathit{connect}_{seq}$ and $\mathit{minconnect}_{seq}$ in (6.11). When we bring this contract together with the technical contract of the Java method, we obtain the following Java method contract:

JAVA CONTRACT

```
/*@ contract
  @   requires ¬h[this, adjacency] ≐ null
  @   requires h[this, size] > 0
  @   requires arrlen(h[this, adjacency]) ≐ h[this, size]
  @   requires (∀i. 0 ≤ i ∧ i < h[this, size] →
  @                ¬h[h[this, adjacency], idxRef(i)] ≐ null
  @               ∧   arrlen(h[h[this, adjacency], idxRef(i)])
  @                ≐ h[this, size])
  @   requires 0 ≤ src ∧ src < h[this, size]
  @   requires 0 ≤ dest ∧ dest < h[this, size]
  @   ensures res^int ≥ -1
  @   ensures res^int ≐ -1 → (∀m. m≥0 → ¬minconnect_Java(src,dest,m))
  @   ensures res^int > 0 → minconnect_Java(src,dest, res^int)
  @*/
```

JAVA CONTRACT – 6.9

This contract is completely dissociated from the more abstract definitions of the algorithm which have preceded the Java implementation. It can stand on its own. It is apparent that an attempt to prove this contract correct directly on the implementation would be a far more complex undertaking as the relatively simple refinement steps. The separation of concerns has worked well for this example.

It should finally be admitted that this contract was not obtained by a direct syntactical replacement of a more abstract definition. The equivalence of the various *minconnect* variants needs to be established semantically since the different successor relation encodings are not functional.

6.3 Chapter Summary

In two case studies which cover examples of different degree of complexity and difficulty, we have shown that the refinement approach which has been proposed in Chapter 5 is a practicable way to formalise, specify and verify abstract algorithms and their implementations. The software verification system *ivil* presented in Chapter 4 for the program logic *UDL* has been used to prove all verification conditions which arose during the refinement process. The translation of Java bytecode to *UDL* programs outlined in Section 4.4 has been used to bring in the Java implementation.

The first and smaller case study has formalised and refined the *selection sort* algorithm. In the most abstract pseudocode description, a part of the algorithm is abstracted from by an indeterministic choice operation. This makes the program more comprehensible and easier to verify. The indeterministic operation is then formally refined to an implementable algorithmic description which is, in another step, formally refined to its implementation as a Java method. The program code could be annotated with autoactive proof hints introduced in Section 4.2 such that the proof obligations could be discharged automatically without further human interaction in the interactive verification tool.

The second case study has been more sophisticated and more extensive; it covers an algorithmic problem which had been given as a challenge during a software verification competition. A breadth first search algorithm for the shortest-path computation in a graph was to be verified. The original problem description fitted seamlessly into our proposed pseudocode language. This description uses presentation-friendly data structure like sets to transport a good intuition of the procedure. A formal refinement has step been proposed transferring this data representation to more implementation-friendly data structures. These have then, in a second formal refinement step, been brought into connection with the Java implementation of the algorithm. In the verification of the Java implementation, regular contract-based verification meets with the refinement-based approach.

For both case studies, formal contracts for the implementing Java methods without reference to abstractions could be achieved by means of the presented refinement technique.

PSEUDOCODE REFINEMENT

```
abbreviation
   @vars_coupled := V’ ≐ boolArrayAsSeq(h, V) ∧ C’ ≐ boolArrayAsSeq(h, C) ∧
                    N’ ≐ boolArrayAsSeq(h, N) ∧ d’ = d ∧ src’ = src ∧
                    dest’ = dest ∧ size’ = h[this, size]

abbreviation
   @arrays :=
       ¬(V≐N) ∧ ¬(N≐C) ∧ ¬(V≐C)  ∧ ¬(V≐h[this, adjacency]) ∧
       ¬(N≐h[this, adjacency]) ∧ ¬(C≐h[this, adjacency]) ∧
       ¬(V≐this) ∧ ¬(N≐this) ∧ ¬(C≐this) ∧
       (∀i; 0 ≤ i < size’ →
          ¬h[h[this, adjacency], idxRef(i)] ≐ V ∧
          ¬h[h[this, adjacency], idxRef(i)] ≐ N ∧
          ¬h[h[this, adjacency], idxRef(i)] ≐ C) ∧
       arrlen(V) ≐ size’ ∧ arrlen(N) ≐ size’ ∧ arrlen(C) ≐ size’

abbreviation
   @succ :=
     (∀v.0≤v<size’ →
            succ’(v) ≐ boolArrayAsSeq(h, h[h[this,adjacency],idxRef(v)])) ∧
     (∀i.0≤i<size’ → arrlen(h[h[this, adjacency], idxRef(i)]) ≐ size’)

refine bfs_seq as minDistance
  requires @succ ∧ src’ ≐ src ∧ dest’ ≐ dest ∧ size’ ≐ h[this, size]
  ensures  d’ ≐ res^int

  mark 1
    inv @varscoupled ∧ @arrays ∧ @succ ∧ (∃i. 0≤i<arrlen(C) ∧ h[C, idxBool(i)])

  mark 2
    inv @varscoupled ∧ @arrays ∧ @succ ∧ v’ ≐ v ∧ 0 ≤ v < size’

  mark 3
    inv @varscoupled ∧ @arrays ∧ @succ ∧ v’ ≐ v ∧ 0≤v<size’ ∧ w’ ≐ w ∧ 0≤w<size’

  mark 4
    inv @varscoupled ∧ @arrays ∧ @succ ∧ v’ ≐ v ∧ 0≤v<size’ ∧ w’ ≐ w ∧ 0≤w<size’

  mark 5
    inv @varscoupled ∧ @arrays ∧ @succ ∧ v’ ≐ v ∧ 0 ≤ v < size’

  mark 6
    inv @varscoupled ∧ @arrays ∧ @succ ∧ v’ ≐ v ∧ 0 ≤ v < size’
```

PSEUDOCODE REFINEMENT – 6.8

Figure 6.11: The refinement declaration for the breadth first search

CHAPTER 7

Conclusion

7.1 Summary

The goal of this thesis has been to make deductive verification of Java implementations more practicable. Two means have been proposed to reduce the workload which is inherent to this heavyweight formal method:

A novel logical framework based on an intermediate verification language has been presented which allows the formulation of verification conditions in a very flexible way. Its flexibility has been achieved by combining two successful paradigms of implementation verification: *dynamic logic* and *intermediate verification languages*. The new logic inherits the flexibility to compose programs with disseminated embedded assertions from the intermediate language it incorporates. At the same time, it retains the flexibility to combine several programs within a single verification condition from dynamic logic.

Deductive verification can be a difficult business, and it cannot always succeed fully automatically, but may require input from the user. Therefore, the interactive theorem prover *ivil* for the new logic has been devised. It permits the user to intervene in the process of symbolic execution and proof. The interaction provides insight into the current proof situation and allows the user to give stimuli in form of interactive proof steps which help the automatic proof component to eventually find the proof.

The interactive framework is designed such that the communication between user and verification system can happen in terms of elements also present in the source code program even if the verification is actually performed on a translation to the intermediate representation. The user interface has got interaction features like breakpoints and stepwise execution which are similar to familiar features known from dynamic source-level debugging tools.

The second instrument that has been presented to deal with the difficult task of full functional verification is *refinement*. It is a well-established technique in formal methods which operate on abstract models. Beginning from the most abstract coarse system model, refinement is used to make the models more and more detailed in a stepwise fashion. Properties satisfied by an abstract model are passed down to its refining descriptions.

We have extended the reach of formal refinement to include the implementation in code as the final step in the formal refinement process. Thus, a *separation of concerns* between the conceptual questions and the implementational details of a verification task has been achieved. The technical issues that an implementation in a modern programming language necessarily brings with it can be dealt with on the implementational level directly. The implementation is verified with lightweight specifications to establish technical goals like the absence of runtime errors or compliance with a framing contract. The conceptual, still challenging proof of properties which are inherent to the algorithm not the implementation, is conducted on a suitable abstraction of the implementation. This abstract model of the implementation and the implementation are brought into a formal relationship using coupling predicates. The results proved on the abstraction are finally transferred to the implementational level resulting in a correct formal contract for the implementation.

In recent software verification competitions[1], most of the proposed challenges have been of an abstract, algorithmic nature. They show that full functional verification is difficult already on the conceptual level without the added difficulty of an implementation in a real-world programming language.

The novel refinement approach makes full use of the flexibility introduced by the new logic: The intermediate language is important to prepare a common ground to accommodate quite different source languages. Both programs in an abstract pseudocode language and Java implementations (given as bytecode) must be translated to the intermediate language. The fact that the logic is a dynamic logic is important since the refinement proof obligation requires that proof obligations contain two non-trivially composed programs.

7.2 Future Work

The approach for refinement to implementation code by using dynamic logic and symbolic execution presented here is a new method; the provided examples and case studies can only be considered the first steps into this direction. But the results are promising since a non-trivial algorithm could already be provided with a sound formal contract using the new method.

Besides the evident need of more case studies to explore the possibilities of the refinement approach, more theoretical research questions remain to be answered:

The approach as it has been presented here is limited to programs which use loops; algorithmic descriptions which rely on recursion rather than on loops cannot be considered. A canonical extension of the presented approach could hence also allow recursive programs to be formally refined. However, the programming languages in structured or unstructured dynamic logic do not have a concept of recursion; they base on *regular programs*. Harel et al. (2000, §9.1) have loosened this restriction

[1] in particular the competitions associated with the conferences VSTTE 2010, FoVeOOS 2011, VSTTE 2012 and FM 2012

theoretically for structured dynamic logic by relaxing the programs in modalities from regular to *context-free programs*. These results would have to be transferred to the unstructured variant of the logic affecting its basic definitions and the calculus. Loops and recursion are related patterns, and it is to be expected that a verification pattern similar to the presented case for loops would emerge for recursion as well. Instead of synchronised loops, the approach would introduce a concept of synchronised recursive program invocations. The regression verification method by Godlin and Strichman (2009) (a topic closely related to refinement) relies heavily on coupled recursive function calls.

The implementation abstractions that have been explored in the case studies were conceptually similar; all involved arrays in the implementation and sequences in the abstraction. The arrays in the Java code were abstracted from using function symbols which have a heap argument. Such a proceeding is by no means limited to abstracting arrays to sequences, a similar abstraction function can be devised, for instance, to retrieve an element of the abstract datatype tree from a heap structure. A collection and classification of relevant datatypes together with their abstraction functions from heap structures should be worked out.

The refinement approach would not be the only scenario in deductive verification where such a collection would prove valuable. Modular specifications of interfaces which do not want to reveal details of their implementations also require means to abstract from the implementational data structures, and could employ the same abstraction techniques as the refinement.

The potential of this refinement approach should, in particular, be examined for implementations involving rich heap structures in which many objects and references between them are involved. On such structures, it appears particularly interesting to separate the algorithmic questions from the technical issues. The most relevant technical aspects of heap-intensive structures are framing issues. It is desirable to factor this question out as much as possible.

As far as the interactive verification tool *ivil* is concerned, two further development ideas come to mind: First, the user experience could be brought closer to that of a dynamic source-level debugging tool. In particular, the presentation of data to the user could be conceptually reconsidered. Instead of displaying the proof state as a logical sequent, a more data-oriented presentation approach which separates path condition, variable assignment and heap state would be appreciated. Ideas from tools like the visual debugger presented by Hähnle et al. (2010) that present a symbolic system state graphically could be incorporated in the context of the prover.

Secondly, storing interactively applied proof steps persistently remains an interesting open question. Interaction in *ivil* can be provided by means of user input in the interface or as autoactive proof hints annotated to the source code controlling the verification system. It would simplify the process of interactive verification if these two kinds of interaction could be unified. A suitable generalising concept which allows the formulation of *proof scripts* should be defined. These scripts should be comprehensible enough to be written manually while they should serve at the same time as the persistent store format for interactive rule applications. If only small and irrelevant changes are made to the input description, such scripts would be robust enough to reproduce proofs even if an earlier proof cannot be replayed verbatim.

Appendix A

Formal Definitions

A.1 Syntax and Semantics of Terms

The definition of the syntax and semantics of terms in *UDL* has been split into several smaller incremental definitions for the sake of a clearer presentation. This section merges for these definitions the parts scattered over several places in Chapter 2 into one spot.

A.1.1 Syntax

Definition A.5 (Terms) *Let the type signature* Γ *and the signature* Σ *be given. For every type* $T \in \mathcal{T}_\Gamma$ *be a type, the set* Trm_Σ^T *of* terms of type T *is defined as the smallest set such that the following inductive conditions hold:*

1.
$$\begin{array}{rl} & x^T \in \mathrm{Var}_\Gamma \\ \Longrightarrow & x^T \in \mathrm{Trm}_\Sigma^T \end{array}$$

2.
$$\begin{array}{rl} & f \in \mathrm{Fct}_\Sigma,\ \tau : \mathrm{typeVars}(\mathrm{ty}_\Sigma(f)) \to \mathcal{T}_\Gamma \\ & \langle T_1, \ldots, T_n, T\rangle = \tau(\mathrm{ty}_\Sigma(f)), \\ & t_i \in \mathrm{Trm}_\Sigma^{T_i}\ (1 \le i \le n) \\ \Longrightarrow & f^{[\tau]}(t_1, \ldots, t_n) \in \mathrm{Trm}_\Sigma^T \end{array}$$

3.
$$\begin{array}{rl} & b \in \mathrm{Bnd}_\Sigma,\ \tau : \mathrm{typeVars}(\mathrm{ty}_\Sigma(b)) \to \mathcal{T}_\Gamma \\ & \langle T_v, T_1, \ldots, T_n, T\rangle = \tau(\mathrm{ty}_\Sigma(b)) \\ & x^{T_v} \in \mathrm{Var}_\Gamma,\ t_1 \in \mathrm{Trm}^{T_1}, \ldots, t_n \in \mathrm{Trm}^{T_n} \\ \Longrightarrow & (b^{[\tau]}\ x^{T_v}.\ t_1, \ldots, t_n) \in \mathrm{Trm}^T \end{array}$$

4.
$$\begin{array}{rl} & \alpha \in \mathrm{TVar},\ \varphi \in \mathrm{Trm}^{\mathsf{bool}} \\ \Longrightarrow & (\forall \alpha.\ \varphi),\ (\exists \alpha.\ \varphi) \in \mathrm{Trm}^{\mathsf{bool}} \end{array}$$

5.
$$\begin{array}{rl} & p_1, \ldots, p_n \in \mathrm{PVar}_\Sigma \text{ and } t_1 \in \mathrm{Trm}_\Sigma^{\mathrm{ty}(p_1)}, \ldots, t_n \in \mathrm{Trm}_\Sigma^{\mathrm{ty}(p_n)},\ t \in \mathrm{Trm}_\Sigma^T \\ \Longrightarrow & \{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t \in \mathrm{Trm}_\Sigma^T \qquad \textit{(Update term)} \end{array}$$

6. $\pi \in \Pi_\Sigma, n \in \mathbb{N}$
$\Longrightarrow$ $[n;\pi]$, $[\![n;\pi]\!] \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}$ *(Program formula)*

This definition consolidates Definitions 2.5, 2.10, 2.12 and 2.15.

The set of freely occurring variables freeVars(t) in a term $t \in \mathrm{Trm}^T$ is inductively defined for as follows:

$$\begin{aligned}
\mathrm{freeVars}(x^T) &= \{x^T\}, \quad x^T \in \mathrm{Var} \\
\mathrm{freeVars}(f^{[\sigma]}(t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \mathrm{freeVars}(t_i), \quad f \in \mathrm{Fct} \\
\mathrm{freeVars}((b^{[\sigma]}\ v.\ t_1,\ldots,t_n)) &= \bigcup_{i=1}^{n} \mathrm{freeVars}(t_i) \setminus \{v\}, \quad b \in \mathrm{Bnd} \\
\mathrm{freeVars}(\forall\alpha.\varphi) &= \mathrm{freeVars}(\varphi) \\
\mathrm{freeVars}(\{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t_0) &= \bigcup_{i=0}^{n} \mathrm{freeVars}(t_i) \\
\mathrm{freeVars}([n;\pi]) = \mathrm{freeVars}([\![n;\pi]\!]) &= \bigcup \left\{ \mathrm{freeVars}(t) \,\middle|\, \begin{array}{l} \textsf{assert } t \textsf{ or assume } t \textsf{ or} \\ p := t \textsf{ is a statement in } \pi \end{array} \right\}
\end{aligned}$$

If-then-else terms have not been officially introduced and they have been used confiding in their intuitive meaning. Formally, there is a polymorphic function $\mathsf{cond} : \mathsf{bool} \times \alpha \times \alpha \to \alpha$ which is axiomatised as $(\forall\alpha.\forall x^\alpha.\forall y^\alpha.\ \mathsf{cond}(\mathsf{true},x,y) \doteq x \wedge \mathsf{cond}(\mathsf{false},x,y) \doteq y)$. Instead of the term $\mathsf{cond}(t,u,w)$ we use the established notation (if t then u else w).

A.1.2 Evaluation

Definition A.6 (Evaluation of Terms) *Let $D = (\mathcal{D}, I)$ be a semantic structure, τ a type variable assignment and β a variable assignment compatible with τ. The term evaluation function* $\mathrm{val}_{I,\tau,\beta} : \mathrm{Trm} \to \mathcal{D}$ *is inductively defined as follows:*

1. $x^T \in \mathrm{Var}_\Gamma$
$\Longrightarrow$ $\mathrm{val}_{I,\tau,\beta}(x^T) = \beta(x^T)$

2. $f \in \mathrm{Fct}$, $\sigma : \mathrm{typeVars}(\mathrm{ty}(f)) \to \mathcal{T}$, $t_i \in \mathrm{Trm}_\Sigma^{T_i}$ $(1 \le i \le n)$
$\Longrightarrow$ $\mathrm{val}_{I,\tau,\beta}(f^{[\sigma]}(t_1,\ldots,t_n)) = I(f^{[\tau\circ\sigma]})(\mathrm{val}_{I,\tau,\beta}(t_1),\ldots,\mathrm{val}_{I,\tau,\beta}(t_n))$

3. $b \in \mathrm{Bnd}$, $\sigma : \mathrm{typeVars}(\mathrm{ty}(b)) \to \mathcal{T}$, $t_i \in \mathrm{Trm}_\Sigma^{T_i}$ $(1 \le i \le n), v \in \mathrm{Var}$,
$\mathrm{ty}(b) = \langle T_v, T_1, \ldots, T_n \rangle$
$\mathit{eval}_i : \mathcal{D}^{\tau(\sigma(T_v))} \to \mathcal{D}^{\tau(\sigma(T_i))}$,
$\mathit{eval}_i(d) = \mathrm{val}_{I,\tau,\beta[v\mapsto d]}(t_i)$, *for all* $d \in \mathcal{D}^{\tau(\sigma(T_v))}$
$\Longrightarrow$ $\mathrm{val}_{I,\tau,\beta}((b^{[\sigma]}\ v.\ t_1,\ldots,t_n)) = I(b^{[\tau\circ\sigma]})(\mathit{eval}_1,\ldots,\mathit{eval}_n)$

4. $\varphi \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}$, $\alpha \in \mathrm{TVar}$
$\Longrightarrow \mathrm{val}_{I,\tau,\beta}(\forall\alpha.\varphi) = \mathit{tt} \iff I,\tau[\alpha \mapsto T],\mathrm{coerce}(\beta,\tau[\alpha \mapsto T]) \models \varphi$ *for all* $T \in \mathcal{T}^0$

5. $p_1,\ldots,p_n \in \mathrm{PVar}_\Sigma$, $t \in \mathrm{Trm}_\Sigma$, $t_i \in \mathrm{Trm}_\Sigma^{\mathrm{ty}(p_i)}$ $(1 \le i \le n)$
$I' = I[p_1 \mapsto \mathrm{val}_{I,\tau,\beta}(t_1)][p_2 \mapsto \mathrm{val}_{I,\tau,\beta}(t_2)]\ldots[p_n \mapsto \mathrm{val}_{I,\tau,\beta}(t_n)]$
$\Longrightarrow \mathrm{val}_{I,\tau,\beta}(\{p_1 := t_1 \parallel \ldots \parallel p_n := t_n\}t) = \mathrm{val}_{I',\tau,\beta}(t)$

6. $\pi \in \Pi, n \in \mathbb{N}, \varphi \in \mathrm{Trm}_\Sigma^{\mathsf{bool}}$
$\Longrightarrow \mathrm{val}_{I,\tau,\beta}([n;\pi]) = \mathit{tt} \iff$ *Every trace starting in* (I,n) *is successful.*
$\mathrm{val}_{I,\tau,\beta}([\![n;\pi]\!]) = \mathit{tt} \iff \mathrm{val}_{I,\tau,\beta}([n;\pi]) = \mathit{tt}$ *and* π *has no infinite trace starting in* (I,n)

This definition consolidates Definitions 2.8, 2.11, 2.13, 2.16 and 2.18.

A.2 Definitions of Abstract Data Types

At various points in the thesis, abstract data types were used to model aspects of systems. In particular sets (as the polymorphic type set) and sequences (as the polymorphic type seq) are used. To clarify their meaning, we list the defined symbols with their signatures and the defining axioms for the two data types.

The induction schema for has been mentioned in Section 4.2.5. It is the Peano axiom for induction over natural numbers generalised to a set of integers with a lower bound $l : \mathrm{Trm}^{\mathsf{int}}$.

$$\frac{\Gamma \vdash \varphi[n/l],\ \Delta \qquad \Gamma \vdash (\forall n.\ n \ge l \land \varphi \to \varphi[n/n+1]),\Delta}{\Gamma \vdash (\forall n.\ n \ge l \to \varphi),\Delta}\,\mathrm{intInduction}$$

A.2.1 Sets

The formalisation of sets in *ivil* bases (like axiomatic set theory) on the $\in$-relation. All operators on set are defined in terms of $\in$. An extensionality axiom is added which says that sets are considered equal if the contain the same elements.

The theory of sets contains the function symbols

$\mathsf{in} : \alpha \times \mathsf{set}(\alpha) \to \mathsf{bool} \quad (\in)$
$\mathsf{empty} : \mathsf{set}(\alpha) \quad (\emptyset)$
$\mathsf{fullset} : \mathsf{set}(\alpha)$
$\mathsf{singleton} : \alpha \to \mathsf{set}(\alpha) \quad (\{\cdot\})$
$\mathsf{union} : \mathsf{set}(\alpha) \times \mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \quad (\cup)$
$\mathsf{intersect} : \mathsf{set}(\alpha) \times \mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \quad (\cap)$

$\mathsf{diff} : \mathsf{set}(\alpha) \times \mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \quad (\setminus)$
$\mathsf{complement} : \mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \quad (\complement)$
$\mathsf{seqAsSet} : \mathsf{seq}(\alpha) \to \mathsf{set}(\alpha)$
$\mathsf{finite} : \mathsf{set}(\alpha) \to \mathsf{bool}$
$\mathsf{card} : \mathsf{set}(\alpha) \to \mathsf{int}$

$$
\begin{aligned}
(\forall\alpha.\forall s^{\mathsf{set}(\alpha)}.\forall t^{\mathsf{set}(\alpha)}.\ (\forall x^{\alpha}.\ (\mathsf{in}(x,s) \leftrightarrow \mathsf{in}(x,t))) \rightarrow s \doteq t)\\
(\forall\alpha.\forall x^{\alpha}.\ \neg\mathsf{in}(x,\mathsf{empty}))\\
(\forall\alpha.\forall x^{\alpha}.\ \mathsf{in}(x,\mathsf{fullset}))\\
(\forall\alpha.\forall x^{\alpha}.\forall y^{\alpha}.\ \mathsf{in}(x,\mathsf{singleton}(y)) \leftrightarrow x \doteq y)\\
(\forall\alpha.\forall x^{\alpha}.\forall s^{\mathsf{set}(\alpha)}.\ \mathsf{in}(x,\mathsf{complement}(s)) \leftrightarrow \neg\mathsf{in}(x,s))\\
(\forall\alpha.\forall x^{\alpha}.\forall s^{\mathsf{set}(\alpha)}.\forall t^{\mathsf{set}(\alpha)}.\ \mathsf{in}(x,\mathsf{union}(s,t)) \leftrightarrow (\mathsf{in}(x,s) \vee \mathsf{in}(x,t)))\\
(\forall\alpha.\forall x^{\alpha}.\forall s^{\mathsf{set}(\alpha)}.\forall t^{\mathsf{set}(\alpha)}.\ \mathsf{in}(x,\mathsf{intersect}(s,t)) \leftrightarrow (\mathsf{in}(x,s) \wedge \mathsf{in}(x,t)))\\
(\forall\alpha.\forall x^{\alpha}.\forall s^{\mathsf{set}(\alpha)}.\forall t^{\mathsf{set}(\alpha)}.\ \mathsf{in}(x,\mathsf{diff}(s,t)) \leftrightarrow (\mathsf{in}(x,s) \wedge \neg\mathsf{in}(x,t)))\\
(\forall\alpha.\forall s^{\mathsf{set}(\alpha)}.\forall t^{\mathsf{set}(\alpha)}.\ \mathsf{subset}(s,t) \leftrightarrow (\forall x^{\alpha}.\ (\mathsf{in}(x,s) \rightarrow \mathsf{in}(x,t))))\\
(\forall\alpha.\forall s^{\mathsf{set}(\alpha)}.\ \mathsf{finite}(s) \leftrightarrow (\exists t^{\mathsf{seq}}.\ s \doteq \mathsf{seqAsSet}(t)))\\
(\forall\alpha.\forall x^{\alpha}.\ \mathsf{in}(x,(\mathsf{setComp}\ x^{\alpha}.\ \varphi)) \doteq \varphi)\\
\text{for all } \varphi \in \mathrm{Trm}^{\mathsf{bool}}, \mathrm{freeVars}(\varphi) \subseteq \{x\}
\end{aligned}
$$

Figure A.1: Axioms for the abstract data type set

and the set comprehension binder symbol

$$\mathsf{setComp} : \alpha \times \mathsf{bool} \rightarrow \mathsf{set}(\alpha) \ .$$

We use the typical mathematical notation for the set functions for more concise presentation if the context is clear. The according mathematical symbols are annotated to the function symbol declarations above.

Sets are axiomatised by the set of axioms in Figure A.1 in which the last axiom is schematic (containing the schematic formula φ) .

A.2.2 Finite Sequences

A similar theory of finite sequences has also been covered more extensively by Schmitt (2011, Section 2.6).

The domain for finite sequences of type $T \in \mathcal{T}^0$ is fixed as the set $\mathcal{D}^{\mathsf{seq}(T)} = (\mathcal{D}^T)^* = \bigcup_{n=0}^{\infty}(\mathcal{D}^T)^n$ of finite words over $\mathcal{D}^T$ to match the intuition. There are two relevant observer symbols in the theory: $\mathsf{seqLen} : \mathsf{seq}(\alpha) \rightarrow \mathsf{int}$ and $\mathsf{seqGet} : \mathsf{seq}(\alpha) \times \mathsf{int} \rightarrow \alpha$. The former gives the length of the sequence and the latter retrieves a value from the sequence at a specified index:

$$I(\mathsf{seqLen}^{[\mathsf{T}]})(\langle v_1,\ldots,v_n\rangle) = n \qquad I(\mathsf{seqGet}^{[\mathsf{T}]})(\langle v_1,\ldots,v_n\rangle,i) = \begin{cases} v_{i+1} & \text{if } 0 \leq i < n \\ \perp^{[T]} & \text{otherwise} \end{cases}$$

If a value is to be retrieved from a sequent outside its valid index range, an error value $\bot^T \in \mathcal{D}^T$ is given. The error value is represented in the logic as the underspecified polymorphic constant $\mathsf{seqError} : \alpha$.

Unlike the sets which are axiomatised using an observer symbol, sequence operators are defined by means of a constructor symbol. The binder $\mathsf{seqDef} : \mathsf{int} \times \mathsf{int} \times \alpha \to \mathsf{seq}(\alpha)$ can be used to define sequences. Its semantics is fixed by the schematic axioms with $l,h \in \mathrm{Trm}^{\mathsf{int}}, v \in \mathrm{Trm}^T$, $\mathrm{freeVars}(l) = \mathrm{freeVars}(h) = \emptyset$, $\mathrm{freeVars}(v) \subseteq \{i^{\mathsf{int}}\}$:

$$(\forall j^{\mathsf{int}}.\ \mathsf{seqGet}^{[T]}((\mathsf{seqDef}^{[T]}\ i^{\mathsf{int}}.\ l,h,v),j) \doteq \text{if } 0 \le j < h-l \text{ then } v[i/j+l] \text{ else } \mathsf{seqError}^{[T]})$$
$$\mathsf{seqLen}^{[T]}((\mathsf{seqDef}^{[T]}\ i^{\mathsf{int}}.\ l,h,v),j) \doteq \text{if } l < h \text{ then } h-l \text{ else } 0$$

The remaining symbols of the theory

$$\begin{aligned}
\mathsf{seqEmpty} &: \mathsf{seq}(\alpha) & \mathsf{seqSingleton} &: \alpha \to \mathsf{seq}(\alpha)\\
\mathsf{seqConcat} &: \mathsf{seq}(\alpha) \times \mathsf{seq}(\alpha) \to \mathsf{seq}(\alpha) & \mathsf{seqSub} &: \mathsf{seq}(\alpha) \times \mathsf{int} \times \mathsf{int} \to \mathsf{seq}(\alpha)\\
\mathsf{seqUpdate} &: \mathsf{seq}(\alpha) \times \mathsf{int} \times \alpha \to \mathsf{seq}(\alpha) & \mathsf{seqAsSet} &: \mathsf{seq}(\alpha) \to \mathsf{set}(\alpha)\\
\mathsf{seqPerm} &: \mathsf{seq}(\alpha) \times \mathsf{seq}(\alpha) \to \mathsf{bool} & \mathsf{seqSorted} &: \mathsf{seq}(\mathsf{int}) \to \mathsf{bool}\\
\mathsf{seqSwap} &: \mathsf{seq}(\alpha) \times \mathsf{int} \times \mathsf{int} \to \mathsf{seq}(\alpha) & &
\end{aligned}$$

have their semantics fixed by the axioms in Figure A.2. All formulas are implicitly universally quantified. The quantifiers were left out to shorten the presentation. An auxiliary predicate $\mathit{isPermN} : \mathsf{seq}(\mathsf{int}) \to \mathsf{bool}$ is used. The formula $\mathit{isPermN}(p)$ is true if the parameter is a permutation of the first $\mathsf{seqLen}(p)$ natural numbers: $(\forall i.\ 0 \le i < \mathsf{seqLen}(p) \to (\exists j.\ 0 \le j < \mathsf{seqLen}(p) \land \mathsf{seqGet}(p,j) \doteq i))$.

The following theorem is relevant for the proof of the selection sort algorithm (Section 6.1): Swapping two elements within the sequence preserves the permutation property

$$\begin{aligned}
0 \le a^{\mathsf{int}} < \mathsf{seqLen}(s_1^{\mathsf{seq}(\alpha)}) \land 0 \le b^{\mathsf{int}} < \mathsf{seqLen}(s_2^{\mathsf{seq}(\alpha)}) \land\\
\mathsf{seqPerm}(s_1,s_2) \to \mathsf{seqPerm}(\mathsf{seqSwap}(s_1,a,b),s_2))\ .
\end{aligned}$$

This can be proved within *ivil* to be a consequence of the definitions of the function symbols.

A.3 Formal Definition of Pseudocode

Pseudocode is used as the input language for the algorithmic descriptions on higher abstration level in Chapter 5. The rationales of using pseudocode as modelling language have been discussed in Section 5.1.1.

The pseudocode that we consider uses *UDL* as inner language, that is, every expression in a pseudocode program can use the logical symbols which are defined the signature in use. This includes (type) quantifiers and other binders, updates

$$
\begin{aligned}
\mathsf{seqEmpty}^{[\alpha]} &\doteq (\mathsf{seqDef}\ i.\ 0{,}0{,}\mathsf{seqErr}^{[\alpha]})\\
\mathsf{seqSingleton}^{[\alpha]}(v^{\alpha}) &\doteq (\mathsf{seqDef}\ i.\ 0{,}1{,}v)\\
\mathsf{seqConcat}^{[\alpha]}(s_1^{\mathsf{seq}(\alpha)},s_2^{\mathsf{seq}(\alpha)}) &\doteq (\mathsf{seqDef}\ i.\ 0{,}\mathsf{seqLen}(s_1)+\mathsf{seqLen}(s_2),\\
&\qquad \text{if } i<\mathsf{seqLen}(s_1)\ \ \text{then } \mathsf{seqGet}(s_1,i)\\
&\qquad\qquad\qquad\qquad\quad \text{else } \mathsf{seqGet}(s_2,i-\mathsf{seqLen}(s_1)))\\
\mathsf{seqSub}^{[\alpha]}(s^{\mathsf{seq}(\alpha)},a^{\mathsf{int}},b^{\mathsf{int}}) &\doteq (\mathsf{seqDef}\ i.\ a{,}b{,}\mathsf{seqGet}(s,i))\\
\mathsf{seqUpdate}^{[\alpha]}(s^{\mathsf{seq}(\alpha)},a^{\mathsf{int}},v^{\alpha}) &\doteq (\mathsf{seqDef}\ i.0{,}\mathsf{seqLen}(s){,}\text{if } i\doteq a \text{ then } v \text{ else } \mathsf{seqGet}(s,a))\\
\mathsf{seqAsSet}^{[\alpha]}(s^{\mathsf{seq}(\alpha)}) &\doteq (\mathsf{setComp}\ x.(\exists i.0\le i\wedge i<\mathsf{seqLen}(s)\wedge \mathsf{seqGet}(s,i)\doteq x))\\
\mathsf{seqSorted}^{[\alpha]}(s^{\mathsf{seq}(\alpha)}) &\doteq (\forall i.0\le i<\mathsf{seqLen}(s)\rightarrow\\
&\qquad (\forall j.\ 0\le j<\mathsf{seqLen}(s)\wedge i<j\rightarrow \mathsf{seqGet}(s,i)\le\mathsf{seqGet}(a,j))))\\
\mathsf{seqSwap}^{[\alpha]}(s^{\mathsf{seq}(\alpha)},a^{\mathsf{int}},b^{\mathsf{int}}) &\doteq (\mathsf{seqDef}\ t.0{,}\mathsf{seqLen}(s),\\
&\qquad \text{if } t\doteq a \text{ then } \mathsf{seqGet}(s,b) \text{ else}\\
&\qquad\quad (\text{if } t\doteq b \text{ then } \mathsf{seqGet}(s,a) \text{ else } \mathsf{seqGet}(s,t)))\\
\mathsf{seqPerm}^{[\alpha]}(s_1^{\mathsf{seq}(\alpha)},s_2^{\mathsf{seq}(\alpha)}) &\doteq \mathsf{seqLen}(s_1)\doteq\mathsf{seqLen}(s_2)\wedge\\
&\quad (\exists p.isPermN(p)\wedge\mathsf{seqLen}(p)\doteq\mathsf{seqLen}(s_1)\wedge\\
&\qquad (\forall i.0\le i<\mathsf{seqLen}(p)\rightarrow\\
&\qquad\quad \mathsf{seqGet}(s_1,i)\doteq\mathsf{seqGet}(s_2,\mathsf{seqGet}(p,i))))
\end{aligned}
$$

Figure A.2: Axioms for finite sequences

and even program formulas. The pseudocode language heavily depends on sensible datatypes which can be used to model algorithms. The datatype signatures and definitions which are used for the presented algorithms are listed in Appendix A.2.

Pseudocode programs can easily be translated to *UDL* programs, in fact, pseudocode is a syntactical frontend to *UDL* and *ivil*. The Boogie language which shares its primitives with *ivil* has an input language which features a rich set of syntactic sugar to enable expressive freedom. This pseudocode frontend provides now a similar feature-rich language to the user, but with a different goal: While for the Boogie system, the input language is an intermediate representation format generated automatically and not written by the end-user (and rarely read), pseudocode is intended for the designer of an algorithm as input language.

The following grammar in extended Backus-Naur-Form describes the general structure of a pseudocode declaration.

EBNF

```
Declaration =
   { Algorithm }
   [ Refinement ]

Algorithm =
   "algo" Identifier
   [ "input" VariableDeclarations ]
   [ "output" VariableDeclarations ]
   [ "var" VariableDeclarations ]
   { "requires" Expression }
   { "ensures" Expression }
   "do"
     Command
   "end"

VariableDeclarations =
   { Identifier ":" Type }
```

```
Refinement =
   "refine" Identifier "as" Identifier
   "requires" Expression
   "ensures" Expression
   { "mark" NaturalNumber
       "inv" Expression }

Command =
   | Command ";" Command
   | Assume | Assert
   | Identifier := Expression
   | Mark
   | While | Repeat | Choose | Note
   | If | Iterate | Return
```

EBNF

A pseudocode algorithm description starts with the declaration of the program variables which are used in the course of the program. They may be marked as parameters to the algorithm (input), as result of the algorithm (output) or internal for the algorithm (var). Then, pre- and postcondition formulas of the code are specified. The actual code follows after these declarations. A full example of a pseudo algorithm description is the following:

PSEUDOCODE

```
algo addNumbers
  input S : set(int)
  output sum : int
  var T : set(int)
      x : int
  requires finite(S)
  ensures sum = (sum i. i ∈ S)
do
  sum := 0;
  iterate S with x as T
     inv sum = (sum i. i ∈ S \ T)
  do sum := sum + x end
end
```

PSEUDOCODE

The commands of the language are given their semantics by describing how they are reduced to the intermediate language of *ivil*. Assumptions, assertions and assignments are directly adopted in the translation. Mark statements are translated to annotations in the program. Figure A.3 shows the structure of the remaining pseudocode command constructors and their translation in *ivil*.

Besides the while loop whose semantics is as expected, there are two more loop statements: A repeat-loop which allows the repetition of a block indeterministically often (the correspondent to the Kleene-star) and an iterate-loop which allows the iteration over a set of values. Iterations take the iterated set S as argument and the program variable x with which the set is iterated. Program variable T is used to denote the elements in S which are still to be iterated. See the above small pseudocode program for an example with these program variables.

The looping constructors (while, repeat and iterate) have an annotated invariant φ and variant ν. These are brought into the unstructured program as annotations to the suggested loop invariant places which are labelled with (*) in Figure A.3. For the iterate statement, the suggested loop invariant point is in (**). The variant is the set of remaining elements $T \subset S$; the loop terminates if T is finite. An implicit additional invariant for iterate loops is $T \subseteq S$.

A pseudocode description may contain several programs and a concluding refinement description. The description

PSEUDOCODE REFINEMENT

```
refine A as C
  requires $\psi_{pre}$
  ensures $\psi_{post}$
  mark 1 inv $\psi_1$
  ...
  mark n inv $\psi_n$
```

PSEUDOCODE REFINEMENT

is translated into the proof obligation $\psi_{pre} \to [C]\langle A\rangle\psi_{post}$. The intermediate coupling invariants ψ_i are annotated together with the mark statements in the programs and will be used by the strategies during the verification of the refinement condition.

PSEUDOCODE/IVIL

```
while $\kappa$          loop:
  inv $\varphi$           goto after, body
  var $\nu$               # (*)
do $P$ end              body:
                          assume $\kappa$
                          # translate P
                          goto loop
                        after:
                          assume $\neg\kappa$
```

```
choose $x$              assert
  such that $\varphi$      $(\exists v^T.\{x := v^T\}\varphi)$
                          havoc $x$
                          assume $\varphi$
```

```
if $\varphi$            goto then, else
then $P$                then:
else $Q$                  assume $\varphi$
end                       # translate P
                          goto after
                        else:
                          assume $\neg\varphi$
                          # translate Q
                        after:
```

```
repeat                  loop:
  inv $\varphi$           # (*)
  var $\nu$               goto after, body
do $P$ end              body:
                          # translate P
                          goto loop
                        after:
```

```
note $\varphi$            assert $\varphi$
                          assume $\varphi$
```

```
iterate $S$               $T := S$
  with $x$ as $T$       loop:
  inv $\varphi$           # (**)
do $P$ end                goto after, body
                          assume $\neg T \doteq$ empty
                          havoc $x$
                          assume in$(x, T)$
                          $T := T \setminus \{x\}$
                          # translate P
                          goto loop
                        after:
                          assume $T \doteq$ empty
```

```
return                    end
```

PSEUDOCODE/IVIL

$x \in$ PVar, $\kappa, \varphi \in \text{Trm}^{\text{bool}}$, $S, T \in \text{Trm}^{\text{set}(\text{ty}(x))}$, $\nu \in$ Trm, P and Q pseudocode programs

Figure A.3: Translation from pseudocode to *ivil*

Appendix B

Java Code

This appendix lists Java code excerpts which have been considered in the various examples throughout the thesis.

B.1 Examples for Evaluation

Here we list the examples which have been used in Section 4.4.11 as benchmarks to evaluate the capacities of *ivil* as a Java verification system.

Sum and Max In this example, a method computes the maximum and the sum of the values given to it as a parameter. Additionally, the lemma that the sum is at most as large as the length of the array multiplied with its maximum value.

This benchmark has been proposed as an assignment during the verification competition VSTTE10 (covered by Klebanov et al., 2011).

Java

```
class SumAndMax {
    int sum, max;

    /*@ contract normal_behaviour
      @   requires ¬a ≐ null
      @   requires (∀i. 0 ≤ i ∧ i < arrlen(a) → 0 ≤ h[a,idxInt(i)])
      @   modifies {this}
      @   ensures  (∀i. 0 ≤ i ∧ i < arrlen(a) → h[a,idxInt(i)] ≤ h[this,max])
      @   ensures  (arrlen(a) > 0 → (∃i. 0 ≤ i ∧ i < arrlen(a) ∧ h[a,idxInt(i)]≐h[this,max]))
      @   ensures  h[this,sum] ≐ (sum i. 0,arrlen(a),h[a,idxInt(i)])
      @   ensures  h[this,sum] ≤ arrlen(a) ∗ h[this,max]
      @*/
    void sumAndMax(int[] a) {
        int sum = 0;
        int max = 0;
        int k = 0;

        /*@ maintains
          @       $0 \leq k \land k \leq \mathsf{arrlen}(a) \land (\forall i; 0 \leq i \land i < k \to h[a, \mathsf{idxInt}(i)] \leq max)$
          @     $\land (k = 0 \to max \doteq 0) \land (k > 0 \to (\exists i.\ 0 \leq i \land i < k \land max \doteq h[a, \mathsf{idxInt}(i)]))$
          @     $\land sum \doteq (\mathsf{sum}\ \ i.\ 0, k, h[a, \mathsf{idxInt}(i)]) \land sum \leq k * max$
          @
          @  modifies $\mathsf{empty}$
          @  decreases $\mathsf{arrlen}(a) - k$
          @*/
        while(k < a.length) {
            if(max < a[k]) {
                max = a[k];
            }
            sum += a[k];
            k++;
        }

        this.sum = sum;
        this.max = max;
        //@ note $k \doteq \mathsf{arrlen}(a)$;
    }
}
```

JAVA

Least Common Prefix This simple method is part of a verification benchmark proposed within the verification competition at the Formal Methods conference 2012 in Paris.

The method takes an array a of integers and two indices x, y into the array and returns the number of elements which equal from both indices counted onwards.

JAVA

```
class LeastCommonPrefix {
    /*@
      @ contract normal_behaviour
      @  requires $0 \leq x \land x < \mathsf{arrlen}(a)$
      @  requires $0 \leq y \land y < \mathsf{arrlen}(a)$
      @  requires $\neg a \doteq null$
      @  ensures $0 \leq res^{[\mathsf{int}]}$
      @  ensures $res^{[\mathsf{int}]} \leq \mathsf{arrlen}(a) - x$
      @  ensures $res^{[\mathsf{int}]} \leq \mathsf{arrlen}(a) - y$
      @  ensures $(\forall j.\ 0 \leq j \land j < res^{[\mathsf{int}]} \to h[a, \mathsf{idxInt}(x + j)] \doteq h[a, \mathsf{idxInt}(y + j)])$
      @  ensures $res^{[\mathsf{int}]} \doteq \mathsf{arrlen}(a) - x \lor res^{[\mathsf{int}]} = \mathsf{arrlen}(a) - y \lor$
      @            $\neg h[a, \mathsf{idxInt}(x + res^{[\mathsf{int}]})] \doteq h[a, \mathsf{idxInt}(y + res^{[\mathsf{int}]})]$
      @  modifies $\mathsf{empty}$
      @*/
    @spec.Include("options.p")
    int lcp(int[] a, int x, int y) {
        int i = 0;

        /*@ maintains
          @    0 ≤ i ∧ i ≤ arrlen(a) − x ∧ i ≤ arrlen(a) − y ∧
          @    (∀j. 0 ≤ j ∧ j < i → h[a,idxInt(x + j)] ≐ h[a,idxInt(y + j)])
          @ decreases arrlen(a) − i
          @ modifies empty
          @*/
        while(x+i < a.length && y+i < a.length &&
              a[x+i] == a[y+i]) {
            i++;
        }

        return i;
    }
}
```

Java

Array List This example has not been part of a competition. It implements a list backed up by an array. To the user, the list is abstracted by a sequence of the references stored in a ghost field `S`. The methods add and get are specified with respect to this ghost field.

It has been mentioned that object invariants are not supported by the translation. Instead, a model method `boolean listInv()` is used to describe when an object is valid. It is translated to a fresh predicate symbol *listInv* : heap × ref → bool in the logic. A model method may depend on the heap and on the object upon which it is called (the *receiver*). Both are added as arguments.

The methods are annotated with a decreases clause which is a constant. This is necessary since any method call may only call a method with a strictly lower value for their clause to avoid infinite method call cycles. A more sophisticated implementation would compute these constants for non-recursive call graphs like this automatically.

Java

```
final class ArrayList {

    private Object[] data;
    private int len;
    //@ ghost seq(ref) S;

    /*@ model boolean listInv() {
      @   return 0 ≤ h[r,len] ∧ (h[r,data] ≐ null → h[r,len] ≐ 0)
      @       ∧ (¬h[r,data] ≐ null → h[r,len] ≤ arrlen(h[r,data])
      @             ∧ typeof(h[r,data]) ≐ arrayType(java.lang.Object))
      @       ∧ seqLen(h[r,S]) ≐ h[r,len]
      @       ∧ (∀i. 0 ≤ i ∧ i < h[r,len] → seqGet(h[r,S],i) ≐ h[h[r,data],idxRef(i)]);
```

```
     @ }
     @*/

    /*@ contract normal
     @   requires listInv(h,this)
     @   ensures h[this,len] < arrlen(h[this,data])
     @   ensures h[this,S] ≐ h_pre[this,S]
     @   ensures h[this,len] ≐ h_pre[this,len]
     @   ensures listInv(h,this)
     @   ensures ¬h[this,data] ≐ null
     @   ensures h[this,data] ≐ h_pre[this,data] ∨ ¬h_pre[h[this,data],created]
     @   modifies {this} ∪ freshObjects(h)
     @   decreases 0
     @ */
    private void ensureSpace () {
        if (data == null) {
            data = new Object[10];
        } else if (data.length == len) {
            Object[] newData = new Object[len + 10];
            /*@ maintains 0 ≤ i ∧ i ≤ h[this,len] ∧
             @    (∀j. 0 ≤ j ∧ j < i → h[newData,idxRef(j)] ≐ h[h[this,data],idxRef(j)])
             @ decreases h[this,len] − i
             @ modifies {newData}
             @ */
            for (int i = 0; i < len; i++) {
                newData[i] = data[i];
            }
            data = newData;
        }
    }

    /*@ contract normal
     @   requires listInv(h,this)
     @   ensures h[this,S] ≐ seqConcat(h_pre[this,S],seqSingleton(d))
     @   ensures listInv(h,this)
     @   modifies {this,h[this,data]} ∪ freshObjects(h)
     @   decreases 1
     @ */
    public void add (Object d) {
        ensureSpace ();
        data[len] = d;
        len++;
        /*@ inline
         @    h[this,S] := seqConcat(h_pre[this,S],seqSingleton(d));
         @ */
    }

    /*@ contract normal
      @   requires listInv(h,this)
      @   requires 0 ≤ i ∧ i < h[this,len]
      @   ensures res^[ref] ≐ seqGet(h[this,S],i)
      @   modifies empty*/
    /*@ contract exceptional
      @   requires i < 0 ∨ i >= h[this,len]
      @   signals java.lang.IndexOutOfBoundsException
      @   decreases 1
      @   modifies freshObjects(h)
      @*/
    public Object get (int i) {
        if (i < 0 || i >= len) {
            throw new IndexOutOfBoundsException ();
        }
        return data[i];
    }
}
```

Java

First in Linked List This benchmark is like the first taken from the VSTTE 2010 competition. The assignment was to write a program which returns the index of the first 0 in a linked list of integer values.

The linked list structure is more complex as it involves several objects on the heap over which the information of the list is distributed. The list is, therefore, augmented by a ghost field of type seq(int) in which the sequence of values in the list from the current element on is stored. As object invariants are not supported by the translation, a model method `boolean nodeInv()` is declared which captures that the object is in a good state. Its correspondent in the logic is the predicate symbol $nodeInv : \mathsf{heap} \times \mathsf{ref} \to \mathsf{bool}$ which gets the heap and the receiver object as additional parameters. This parallels the modelling in the last benchmark.

There is one difference to the last predicate *listInv*. The definition of *nodeInv* is recursive in the sense that on right hand side of the rewrite rule, the symbol is used again (for the next node in the list). An unbridled replacement using the rewriting rule leads to an infinite expansion of the definition. Instead we use autoactive annotations to remark that these definitions should be expanded twice on the branch.

Java

```
class Node {
    Node next;
    int value;
    /*@ ghost seq(int) data; */

    /*@ model boolean nodeInv() {
      @    return $(h[r,\mathtt{next}] \doteq \mathtt{null} \rightarrow h[r,\mathtt{data}] \doteq \mathsf{seqSingleton}(h[r,\mathtt{value}]))$
      @        $\land (\neg h[r,\mathtt{next}] \doteq \mathtt{null} \rightarrow$
      @              $h[r,\mathtt{data}] \doteq \mathsf{seqConcat}(\mathsf{seqSingleton}(h[r,\mathtt{value}]),h[r,\mathtt{data}]))$
      @        $\land (\neg h[r,\mathtt{next}] \doteq \mathtt{null} \rightarrow \mathit{nodeInv}(h,h[r,\mathtt{next}]))$;
      @ }
      @*/

    /*@ contract normal
      @  requires $\mathit{nodeInv}(h,\mathit{this})$
      @  ensures $0 \leq \mathit{res}^{[\mathsf{int}]} \land \mathit{res}^{[\mathsf{int}]} \leq \mathsf{seqLen}(h[\mathit{this},\mathtt{data}])$
      @  ensures $\mathit{res}^{[\mathsf{int}]} < \mathsf{seqLen}(h[\mathit{this},\mathtt{data}]) \rightarrow \mathsf{seqGet}(h[\mathit{this},\mathtt{data}],\mathit{res}^{[\mathsf{int}]}) \doteq 0$
      @  ensures $\mathit{res}^{[\mathsf{int}]} = \mathsf{seqLen}(h[\mathit{this},\mathtt{data}]) \rightarrow$
      @               $(\forall x.\ 0 \leq x \land x < \mathit{res}^{[\mathsf{int}]} \rightarrow \neg\mathsf{seqGet}(h[\mathit{this},\mathtt{data}],x) \doteq 0)$
      @  modifies empty
      @*/
    public int search() {
        int i = 0;
        Node n = this;
        /*@ maintains $0 \leq \mathtt{i} \land \mathtt{i} \leq \mathsf{seqLen}(h[\mathit{this},\mathtt{data}]) \land$
          @   $(\neg\mathtt{n} \doteq \mathtt{null} \rightarrow h[\mathtt{n},\mathtt{data}] = \mathsf{seqSub}(h[\mathit{this},\mathtt{data}],\mathtt{i},\mathsf{seqLen}(h[\mathit{this},\mathtt{data}]))) \land$
          @   $(\mathtt{n} \doteq \mathtt{null} \rightarrow \mathtt{i} \doteq \mathsf{seqLen}(h[\mathit{this},\mathtt{data}])) \land$
          @   $(\forall x.\ 0 \leq x \land x < \mathtt{i} \rightarrow \neg\mathsf{seqGet}(h[\mathit{this},\mathtt{data}],x) \doteq 0) \land$
          @   $(\neg\mathtt{n} \doteq \mathtt{null} \rightarrow \mathit{nodeInv}(h,\mathtt{n}))$
          @ decreases $\mathsf{seqLen}(h[\mathit{this},\mathtt{data}]) - \mathtt{i}$
          @ modifies empty
          @*/
        while(n != null && n.value != 0) {
            n = n.next;
            i++;
            //@ hint "(expand invDef 2)";
        }

        //@ hint "(expand invDef 2)";
        return i;
    }
}
```

JAVA – B.1

B.2 Case Study – Breadth First Search

In Section 6.2, a breadth first search algorithm has been presented. It is implemented in the class BFS of which excerpts have been depicted in the section. We deliver here the missing implementation code of the auxiliary methods of the class. The main method has already been listed in Figure 6.10.

The typeset for the formal annotation changes a little in comparison to the last section. There the different types of symbols should be pointed out, an aspect we want to emphasise less in this example. Originally, all files are annotated with specifications in ASCII.

JAVA

```java
class BFS {

    int size;
    boolean[][] adjacency;

    /*@ contract
      @   requires ¬array ≐ null
      @   ensures -1 ≐ res^int ∨ from ≤ res^int ∧ res^int < arrlen(array)
      @   ensures res^int ≥ from  → h[array, idxBool(res^int)] ∧
      @               (∀i. from≤i ∧ i<res^int → ¬h[array, idxBool(i)])
      @   ensures res^int ≐ -1 → (∀i. 0≤i ∧ i<arrlen(array) →
      @                          ∧h[array,idxBool(i)])
      @   modifies empty
      @*/

    int first(boolean[] array, int from) {
        /*@ maintains
          @ from ≤ i ∧ i < arrlen(arry) ∧
          (∀j. 0 ≤ j ∧ j < i → ¬h[array, idxBool(j)])
          @ decreases arrlen(array) - i
          @ modifies empty
          @*/
        for(int i = from; i < array.length; i++) {
            if(array[i]) {
                return i;
            }
        }
        return -1;
    }

    /*@ contract
      @   requires ¬array ≐ null
      @   ensures res^[bool] ≐¬(∃i. 0 ≤ i ∧ i < arrlen(array) ∧
      @                          h[array, idxBool(i)])
      @   modifies empty
      @*/
    boolean isEmpty(boolean[] array) {
        /*@ maintains
          @   0 ≤ i ∧ _i ≤ arrlen(array) ∧
          @   (∀ j. 0 ≤ j  ∧ j < i -> ¬h[array, idxBool(j)])
```

```
     @ decreases arrlen(array) - i
     @ modifies empty
     @*/
    for(int i = 0; i < array.length; i++) {
      if(array[i]) {
        return false;
      }
    }
    return true;
  }

  /*@ contract
   @  requires ¬array ≐ null
   @  ensures (∀i. 0 ≤ i ∧ i < arrlen(array) →
   @               ¬h[array, idxBool(i)])
   @  modifies {array}
   @*/
  void clear(boolean[] array) {
    /*@ maintains
     @  0 ≤ i ∧ i ≤ arrlen(array) ∧
     @  (∀j. 0 ≤ j ∧ j < i → ¬h[array, idxBool(j)])
     @ decreases arrlen(array) - i
     @ modifies {array}
     @*/
    for(int i = 0; i < array.length; i++) {
      array[i] = false;
    }
  }

  /*@ contract
   @  requires ¬target ≐ null
   @  requires ¬source ≐ null
   @  requires arrlen(source) ≐ arrlen(target)
   @  ensures (∀i. 0 ≤ i ∧ i < arrlen(target) →
   @   h[target, idxBool(i)] ≐ h[source, idxBool(i)])
   @  modifies {target}
   @*/
  void copy(boolean[] target, boolean[] source) {
    /*@ maintains
     @  0 ≤ i ∧ i ≤ arrlen(source) ∧
     @  (∀j. 0 ≤ j ∧ j < i →
     @     h[target, idxBool(j)] ≐ h[source, idxBool(j)])"
     @ decreases arrlen(source) - i
     @ modifies  {target}
     @*/
    for(int i = 0; i < source.length; i++) {
```

```java
                target[i] = source[i];
            }
        }

        /*
         * The main method minDistance(int src, int dest) has already
         * been listed in Figure 6.10.
         */

}
```

JAVA

Bibliography

Jean-Raymond Abrial (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. (Cited on pages 4, 132, 133, 136 and 142.)

Jean-Raymond Abrial (2010). *Modeling in Event-B - System and Software Engineering*. Cambridge University Press. (Cited on pages 4 and 133.)

Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe (2009). Abstract object creation in dynamic logic – to be or not to be created. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings, Formal Methods, Second World Congress (FM 2009)*, volume 5850 of *LNCS*, pages 612–627. Springer. (Cited on page 39.)

José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira (2010). Deductive verification of cryptographic software. *Innovations in Systems and Software Engineering*, 6(3):203–218. (Cited on page 171.)

Anindya Banerjee, Michael Barnett, and David A. Naumann (2008a). Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *Proceedings, Verified Software: Theories, Tools, Experiments, Second International Conference (VSTTE 2008)*, volume 5295 of *LNCS*, pages 177–191. Springer. (Cited on page 119.)

Anindya Banerjee, David A. Naumann, and Stan Rosenberg (2008b). Regional logic for local reasoning about global invariants. In Jan Vitek, editor, *Proceedings, 22nd European conference on Object-Oriented Programming (ECOOP 2008)*, volume 5142 of *LNCS*, pages 387–411. Springer. (Cited on page 119.)

Fabian Bannwart and Peter Müller (2005). A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):255–273. (Cited on page 110.)

Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter (2011). Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91. (Cited on pages 4 and 109.)

Mike Barnett and K. Rustan M. Leino (2010). To goto where no statement has gone before. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments 2010 (VSTTE 2010)*, volume 6217 of *LNCS*, pages 157–168. Springer. (Cited on page 37.)

Clark Barrett, Aaron Stump, and Cesare Tinelli (2010). The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings, 8th International Workshop on Satisfiability Modulo Theories (SMT 2010)*. (Cited on pages 10 and 88.)

Gilles Barthe, Juan Manuel Crespo, and César Kunz (2011). Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *Proceedings, Formal Methods - 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 200–214. Springer. (Cited on page 170.)

Gilles Barthe, Benjamin Grégoire, and Mariela Pavlova (2008). Preservation of proof obligations from Java to the Java virtual machine. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Proceedings, Automated Reasoning, 4th International Joint Conference (IJCAR 2008)*, volume 5195 of *LNCS*, pages 83–99. Springer. (Cited on page 110.)

Bernhard Beckert and Daniel Bruns (2012). Dynamic trace logic: Definition and proofs. Technical Report 2012-10, Department of Informatics, Karlsruhe Institute of Technology. (Cited on page 3.)

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors (2007). *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer. (Cited on pages 3, 34, 57, 87, 110, 113 and 127.)

Bernhard Beckert and Vladimir Klebanov (2013). A Dynamic Logic for deductive verification of multi-threaded programs. *Formal Aspects of Computing*, 25(3):405–437. (Cited on page 146.)

Bernhard Beckert, Steffen Schlager, and Peter H. Schmitt (2005). An improved rule for while loops in deductive program verification. In Kung-Kiu Lau and Richard Banach, editors, *Proceedings, 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 315–329. Springer. (Cited on pages 69 and 85.)

Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier (1999). Météor: A successful application of B in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings, Volume I, Formal Methods, World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*, volume 1708 of *LNCS*, pages 369–387. Springer. (Cited on page 132.)

Yves Bertot (2008). A short presentation of Coq. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Proceedings, Theorem Proving in Higher Order Logics, 21st International Conference (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 12–16. Springer. (Cited on pages 11 and 95.)

Yves Bertot, Gilles Kahn, and Laurent Théry (1994). Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Proceedings, Theoretical Aspects of Computer Software, International Conference (TACS 1994)*, volume 789 of *LNCS*, pages 141–160. Springer. (Cited on page 90.)

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson (2011). Extending sledgehammer with smt solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proceedings, 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 116–130. Springer. (Cited on page 107.)

Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff (2008). HOL-Boogie—An interactive prover for the Boogie program-verifier. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Proceedings, Theorem Proving in Higher Order Logics, 21st International Conference (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 150–166. Springer. (Cited on page 95.)

Egon Börger and Robert F. Stärk (2003). *A Method for High-Level System Design and Analysis*. Springer. (Cited on pages 4 and 133.)

Janusz A. Brzozowski (1964). Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494. (Cited on page 47.)

Richard Bubel, Andreas Roth, and Philipp Rümmer (2008). Ensuring the correctness of lightweight tactics for JavaCard dynamic logic. *Electronic Notes in Theoretical Computer Science*, 199:107–128. (Cited on pages 53 and 88.)

Lilian Burdy, Marieke Huisman, and Mariela Pavlova (2007). Preliminary design of BML: A behavioral interface specification language for Java bytecode. In Matthew B. Dwyer and Antónia Lopes, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *LNCS*, pages 215–229. Springer. (Cited on page 110.)

Patrice Chalin (2003). Improving JML: For a safer and more effective language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings, International Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *LNCS*, pages 440–461. Springer. (Cited on page 112.)

Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll (2005). Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Revised Lectures, Formal Methods for Components and Objects, 4th International Symposium (FMCO 2005)*, volume 4111 of *LNCS*, pages 342–363. Springer. (Cited on page 123.)

Chia-Hsiang Chang and Robert Paige (1992). From regular expressions to DFA's using compressed NFA's. In *Proceedings, Third Annual Symposium on Combinatorial Pattern Matching (CPM 1992)*, volume 644 of *LNCS*, pages 90–110. Springer. (Cited on page 47.)

Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric (2007). A reachability predicate for analyzing low-level software. In Orna Grumberg and Michael Huth, editors, *Proceedings, 13th International Conference, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *LNCS*, pages 19–33. Springer. (Cited on page 4.)

David G. Clarke, John M. Potter, and James Noble (1998). Ownership types for flexible alias protection. In *Proceedings, 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64. ACM Press. (Cited on page 119.)

Stephen A. Cook (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90. (Cited on page 77.)

Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti (2009). Experience report: Ocaml for an industrial-strength static analysis framework. In Graham Hutton and Andrew P. Tolmach, editors, *Proceedings, 14th ACM SIGPLAN international conference on Functional programming (ICFP 2009)*, pages 281–286. ACM Press. (Cited on page 171.)

Markus Dahlweid, Michal Moskał, Thomas Santen, Stephan Tobies, and Wolfram Schulte (2009). VCC: Contract-based modular verification of concurrent C. In *Companion Volume, 31st International Conference on Software Engineering (ICSE 2009)*, pages 429–430. IEEE Press. (Cited on pages 4, 95, 109 and 182.)

Luis Damas and Robin Milner (1982). Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *Proceedings, 9th Annual ACM Symposium on Principles of Programming Languages, (POPL 1982)*, pages 207–212. ACM Press. (Cited on page 15.)

Ádám Darvas, Reiner Hähnle, and David Sands (2005). A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proceedings, Security in Pervasive Computing, Second International Conference, (SPC 2005)*, volume 3450 of *LNCS*, pages 193–209. Springer. (Cited on page 145.)

Leonardo Mendonça de Moura and Nikolaj Bjørner (2008). Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer. (Cited on page 88.)

Anatoli Degtyarev and Andrei Voronkov (2001). Equality reasoning in sequent-based calculi. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 10, pages 611–706. Elsevier and MIT Press. (Cited on pages 55 and 74.)

Delphine Demange, Thomas P. Jensen, and David Pichardie (2010). A provably correct stackless intermediate representation for Java bytecode. In Kazunori Ueda, editor, *Proceedings, Programming Languages and Systems - 8th Asian Symposium (APLAS 2010)*, volume 6461 of *LNCS*, pages 97–113. Springer. (Cited on page 116.)

John Derrick and Eerke A. Boiten (2001). *Refinement in Z and object-Z : foundations and advanced applications*. Formal approaches to computing and information technology (FACIT). Springer. (Cited on page 133.)

David Detlefs, Greg Nelson, and James B. Saxe (2005). Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473. (Cited on page 10.)

Edsger W. Dijkstra (1968). Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148. (Cited on pages 34 and 62.)

Edsger W. Dijkstra (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457. (Cited on pages 3, 32 and 134.)

Dino Distefano and Matthew J. Parkinson (2008). jStar: towards practical verification for Java. In Gail E. Harris, editor, *Proceedings, 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA 2008)*, pages 213–226. ACM Press. (Cited on page 119.)

Gilles Dowek, Thérèse Hardin, and Claude Kirchner (2002). Binding logic: Proofs and models. In Matthias Baaz and Andrei Voronkov, editors, *Proceedings, Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference (LPAR 2002)*, volume 2514 of *LNCS*, pages 130–144. Springer. (Cited on pages 24 and 31.)

Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy (2011). Correct code containing containers. In Martin Gogolla and Burkhart Wolff, editors, *Proceedings, Tests and Proofs - 5th International Conference (TAP 2011)*, volume 6706 of *LNCS*, pages 102–118. Springer. (Cited on page 1.)

Bruno Dutertre and Leonardo Mendonça de Moura (2006). A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Proceedings, Computer Aided Verification, 18th International Conference (CAV 2006)*, volume 4144 of *LNCS*, pages 81–94. Springer. (Cited on page 108.)

Herbert B. Enderton (1972). *A Mathematical Introduction to Logic*. Harcourt Academic Press. (Cited on page 10.)

Christian Engel (2009). *Deductive Verification of Safety-Critical Java Programs*. PhD thesis, Universität Karlsruhe. (Cited on page 3.)

Manuel Fähndrich, Michael Barnett, and Francesco Logozzo (2010). Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings, 2010 ACM Symposium on Applied Computing (SAC 2010)*, pages 2103–2110. ACM Press. (Cited on page 37.)

Timm Felden (2011). Introducing the Boogie methodology to USDL. Studienarbeit, Karlsruhe Institute of Technology. (Cited on page 88.)

Timm Felden (2012). Design and implementation of a verification framework for Java bytecode using unstructured dynamic logic. Diploma thesis, Karlsruhe Institute of Technology. (Cited on pages 109 and 123.)

Xiushan Feng and Alan J. Hu (2005). Cutpoints for formal equivalence verification of embedded software. In Wayne Wolf, editor, *Proceedings, 5th ACM International Conference On Embedded Software (EMSOFT 2005)*, pages 307–316. ACM. (Cited on page 171.)

Jean-Christophe Filliâtre and Claude Marché (2007). The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Proceedings, Computer Aided Verification, 19th International Conference (CAV 2007)*, volume 4590 of *LNCS*, pages 173–177. Springer. (Cited on page 4.)

Jean-Christophe Filliâtre and Andrei Paskevich (2013). Why3 – where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings, Programming Languages and Systems, 22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer. (Cited on pages 95 and 171.)

Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump (2012). The 2nd verified software competition: Experience report. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff Sutcliffe, editors, *Proceedings, 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE)*, volume 873 of *CEUR Workshop Proceedings*, pages 36–49. CEUR-WS.org. (Cited on page 182.)

Michael J. Fischer and Richard E. Ladner (1977). Propositional modal logic of programs. In *Proceedings, 9th annual ACM Symposium on Theory of Computing (STOC 77)*, pages 286–294. ACM Press. (Cited on page 45.)

Michael J. Fischer and Richard E. Ladner (1979). Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211. (Cited on page 45.)

Robert W. Floyd (1967). Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32. (Cited on pages 3 and 32.)

Gerhard Gentzen (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431. (Cited on pages 51 and 84.)

Benny Godlin and Ofer Strichman (2009). Regression verification. In Andrew B. Kahng, editor, *Proceedings, 46th Design Automation Conference (DAC 2009)*, pages 466–471. ACM Press. (Cited on pages 170 and 203.)

Holger Grandy, Kurt Stenzel, and Wolfgang Reif (2007). A refinement method for Java programs. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Proceedings, Formal Methods for Open Object-Based Distributed Systems, 9th International*

Conference (FMOODS 2007), volume 4468 of *LNCS*, pages 221–235. Springer. (Cited on page 170.)

Sarah Grebing (2012). Evaluating and improving the usability of interactive verification systems. Diploma thesis, Universität Koblenz-Landau. (Cited on pages 56 and 91.)

Reiner Hähnle (2005). Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the Interest Group in Pure and Applied Logics*, 13(4):415–433. (Cited on page 24.)

Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe (2010). A visual interactive debugger based on symbolic execution. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *Proceedings, 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pages 143–146. ACM. (Cited on page 203.)

Stefan Hallerstede (2008). Incremental system modelling in Event-B. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain, editors, *Revised Lectures, Formal Methods for Components and Objects, 7th International Symposium (FMCO 2008)*, volume 5751 of *LNCS*, pages 139–158. Springer-Verlag. (Cited on pages 87, 127 and 133.)

David Harel (1979). *First-Order Dynamic Logic*, volume 68 of *LNCS*. Springer. (Cited on page 32.)

David Harel, Dexter Kozen, and Jerzy Tiuryn (2000). *Dynamic Logic*. MIT Press. (Cited on pages 3, 32, 33, 48, 74 and 202.)

David Harel and Rivi Sherman (1985). Propositional dynamic logic of flowcharts. *Information and Control*, 64(1-3):119–135. (Cited on pages 46 and 47.)

Stefan Hetzl, Alexander Leitsch, Daniel Weller, and Bruno Woltzenlogel Paleo (2008). Herbrand sequent extraction. In Serge Autexier, John A. Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *Proceedings, Intelligent Computer Mathematics, 9th International Conference (AISC 2008), 15th Symposium Calculemus, 7th International Conference (MKM 2008)*, volume 5144 of *LNCS*, pages 462–477. Springer. (Cited on page 76.)

J. Roger Hindley (1969). The principle type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60. (Cited on pages 10 and 15.)

C. Antony R. Hoare (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580. (Cited on pages 3 and 32.)

Daniel Jackson (2006). *Software Abstractions - Logic, Language, and Analysis*. MIT Press. (Cited on page 130.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens (2011). VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings, NASA Formal Methods - Third International Symposium (NFM 2011)*, volume 6617 of *LNCS*, pages 41–55. Springer. (Cited on page 182.)

John Arnold Kalman (2001). *Automated Reasoning with OTTER*. Rinton Press. (Cited on page 10.)

Ioannis T. Kassios (2011). The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288. (Cited on page 119.)

James C. King (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394. (Cited on page 57.)

Vladimir Klebanov (2009). *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Universität Koblenz-Landau. (Cited on pages 3 and 146.)

Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß (2011). The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer. Materials available at `www.vscomp.org`. (Cited on pages 125 and 215.)

Dexter Kozen (1997). Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443. (Cited on page 33.)

Hermann Lehner and Peter Müller (2007). Formal translation of bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science*, 190(1):35–50. (Cited on pages 4, 110 and 111.)

K. Rustan M. Leino (2008). This is Boogie 2. KRML Manuscripts. (Cited on page 233.)

K. Rustan M. Leino (2010a). Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Revised Selected Papers, Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference (LPAR 16)*, volume 6355 of *LNCS*, pages 348–370. Springer. (Cited on pages 4 and 123.)

K. Rustan M. Leino (2010b). Usable auto-active verification. Workshop on Usable Verification 2010, Redmond, Washington. (Cited on pages 94 and 128.)

K. Rustan M. Leino and Rosemary Monahan (2009). Reasoning about comprehensions with first-order smt solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings, 2009 ACM Symposium on Applied Computing (SAC 2009)*, pages 615–622. ACM Press. (Cited on page 107.)

K. Rustan M. Leino and Peter Müller (2004). Object invariants in dynamic contexts. In Martin Odersky, editor, *Proceedings, Object-Oriented Programming, 18th European Conference (ECOOP 2004)*, volume 3086 of *LNCS*, pages 491–516. Springer. (Cited on page 119.)

K. Rustan M. Leino and Peter Müller (2009). A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Proceedings, Programming Languages and Systems, 18th European Symposium on Programming (ESOP 2009)*, volume 5502 of *LNCS*, pages 378–393. Springer. (Cited on pages 4 and 170.)

K. Rustan M. Leino and Philipp Rümmer (2010). A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference (TACAS 2010)*, volume 6015 of *LNCS*, pages 312–327. Springer. Further details in Leino (2008). (Cited on pages 10, 28, 34, 48, 97, 100, 103 and 128.)

K. Rustan M. Leino and Kuat Yessenov (2012). Stepwise refinement of heap-manipulating code in chalice. *Formal Asp. Comput.*, 24(4-6):519–535. (Cited on page 170.)

Tim Lindholm and Frank Yellin (1997). *The Java Virtual Machine Specification*. Addison-Wesley. (Cited on pages 115 and 122.)

Peter Linz (1997). *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers. (Cited on page 47.)

Barbara Liskov and Jeannette M. Wing (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841. (Cited on page 121.)

Claude Marché, Christine Paulin-Mohring, and Xavier Urbain (2004). The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106. (Cited on page 4.)

John McCarthy (1962). Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Proceedings, Information Processing Congress (IFIP 1962)*, pages 21–28. (Cited on pages 101 and 113.)

Thomas F. Melham (1993). The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3:7–24. (Cited on page 28.)

Jia Meng and Lawrence C. Paulson (2008). Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60. (Cited on page 107.)

Bertrand Meyer (1988). *Object-Oriented Software Construction, 1st editon*. Prentice-Hall. (Cited on page 117.)

Mobius Consortium (2006). Deliverable 3.1: Bytecode specification language and program logic. (Cited on page 110.)

J. Donald Monk (1976). *Mathematical Logic*. Springer. (Cited on pages 77 and 80.)

Carroll Morgan (1990). *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall. (Cited on pages 4, 133 and 137.)

Michał Moskal (2009). Programming with triggers. In Bruno Dutertre and Ofer Strichman, editors, *Proceedings, 7th International Workshop on Satisfiability Modulo Theories (SMT 2009)*, pages 20–29. ACM Press. (Cited on page 108.)

Yannick Moy (2009). *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud. (Cited on page 4.)

P. Müller (2002). *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag. (Cited on page 119.)

Peter W. O'Hearn and David J. Pym (1999). The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244. (Cited on page 119.)

Martin Otto (2000). Epsilon-logic is more expressive than first-order logic over finite structures. *Journal of Symbolic Logic*, 65(4):1749–1757. (Cited on page 31.)

Matthew Parkinson (2007). Class invariants: The end of the road? *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO 2007)*. Position Paper. (Cited on page 123.)

Lawrence C. Paulson (1986). Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3(3):237–258. (Cited on page 95.)

Mariela Pavlova (2007). *Vérification de bytecode et ses applications*. PhD thesis, Ecole superieure en Science Informatiques de Sophia Antipolis. (Cited on page 110.)

André Platzer (2010). *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer. (Cited on page 3.)

Vaughan R. Pratt (1976). Semantical considerations on Floyd-Hoare logic. Technical Report MIT/LCS/TR-168, Massachusetts Institute of Technology, Cambridge, MA, USA. (Cited on page 32.)

Claire L. Quigley (2003). A programming logic for Java bytecode programs. In David A. Basin and Burkhart Wolff, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*, pages 41–54. Springer. (Cited on page 110.)

Silvio Ranise and Cesare Tinelli (2003). The SMT-LIB format: An initial proposal. *Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2003)*. (Cited on page 10.)

Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel (1995). Interactive correctness proofs for software modules using KIV. In Bonnie Danner, editor, *10th annual conference on computer assurance, IEEE (COMPASS 1995)*, pages 151–162. IEEE Press. (Cited on pages 3, 34 and 170.)

Christoph Scheben and Peter H. Schmitt (2011). Verification of information flow properties of java programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *FoVeOOS*, volume 7421 of *LNCS*, pages 232–249. Springer. (Cited on pages 3 and 146.)

Peter H. Schmitt (2011). A computer-assisted proof of the Bellman-Ford lemma. Karlsruhe Reports in Informatics 2011-15, Karlsruhe Institute of Technology. (Cited on pages 24 and 208.)

Peter H. Schmitt, Mattias Ulbrich, and Michael Walter (2009). A first-order logic with first-class types. In *Tableaux 2009 Position Papers and Workshop Proceedings, Research Report 387, University of Oslo*, pages 43–60. (Cited on pages 32 and 111.)

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß (2010). Dynamic frames in java dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 138–152. Springer. (Cited on pages 111 and 113.)

Natarajan Shankar and Sam Owre (1999). Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *LNCS*, pages 37–52, Toulouse, France. Springer-Verlag. (Cited on pages 11 and 57.)

Kurt Stenzel, Holger Grandy, and Wolfgang Reif (2008). Verification of Java programs with generics. In *Proceedings, 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *LNCS*, pages 315–329. Springer. (Cited on page 3.)

Asma Tafat, Sylvain Boulmé, and Claude Marché (2011). A refinement methodology for object-oriented programs. In Bernhard Beckert and Claude Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 153–167. Springer. (Cited on page 170.)

Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif (2004). Interactive verification of statecharts. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *SoftSpez Final Report*, volume 3147 of *LNCS*, pages 355–373. Springer. (Cited on page 3.)

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer (2011). Verifying Eiffel programs with Boogie. *Computing Research Repository*, abs/1106.4700. (Cited on pages 4 and 109.)

Mattias Ulbrich (2007). Software verification for Java 5. Diploma thesis, Universität Karlsruhe. (Cited on pages 10 and 109.)

Mattias Ulbrich (2011). A dynamic logic for unstructured programs with embedded assertions. In Bernhard Beckert and Claude Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 168–182. Springer. (Cited on pages 48 and 84.)

Benjamin Weiß (2010). *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institue of Technology. (Cited on pages 3, 111, 113, 122 and 123.)

Markus Wenzel (1999). Isar – a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Proceedings, Theorem Proving in Higher Order Logics, 12th International Conference, (TPHOLs 1999)*, volume 1690 of *LNCS*, pages 167–184. Springer. (Cited on page 96.)

Jim Woodcock and Jim Davies (1996). *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall. (Cited on pages 4, 130, 133 and 136.)

Valentin Wüstholz (2009). Encoding Scala programs for the Boogie verifier. Master's thesis, ETH Zurich. (Cited on page 4.)

Index

Index