An Object-Oriented Dynamic Logic with Updates

André Platzer

29th September 2004

Diploma Thesis

University of Karlsruhe
Department of Computer Science
Institute for Logic, Complexity and Deduction Systems

Responsible Supervisor: Prof. Dr. Peter H. Schmitt
Supervisor: Dr. Bernhard Beckert

Abstract

With the goal of this thesis being to create a dynamic logic for object-oriented languages, ODL is developed along with a sound and relatively complete calculus. The dynamic logic contains only the absolute logical essentials of object-orientation, yet still allows a “natural” representation of all other features of common object-oriented programming languages. ODL is an extension of a dynamic logic for imperative WHILE programs by function modification and dynamic type checks. A generalisation of substitutions, called updates, constitute the central technical device for dealing with object aliasing arising from function modification and for retaining a manageable calculus in practical application scenarios. Further, object enumerators realise object creation in a natural yet powerful way. Finally, completeness is proven relative to first-order arithmetic. Along with the soundness result, this proof constitutes the central part of this thesis and even copes with states containing uncomputable functions.

Acknowledgements

At this place I would like to thank all those who kindly supported me during the work on this thesis. First of all, I am very grateful to Dr. Bernhard Beckert, whose advice in the area of logic was crucial for the development of this approach, and who always knew how to come down to the essentials. I would also like to express my highest gratitude to Professor Peter Schmitt for providing the opportunity to investigate such an exciting and theoretically inspiring topic, and for his helpful comments on the work. Philipp Rümmer and Richard Bubel deserve many thanks for their continuous help on the implementation, and their extensive knowledge of the KeY System internals. Also I would like to thank Nicole Betz, Philipp Rümmer, Richard Bubel, Jan-Olaf Blech and Sonja Lauer for indispensable proofreading of the final thesis. And last but not least, I want to thank my parents for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

With the goal of this thesis being to create a dynamic logic for object-oriented languages, ODL is developed along with a sound and relatively complete calculus. For that purpose, the dynamic logic and its underlying programming language are bound to be defined so as to be able to cover all features of object-oriented programming languages. However, the dynamic logic is not intended to duplicate all peculiarities of current object-oriented programming languages. Instead of defining an overloaded object-oriented dynamic logic including native support for a large variety of language decorations with all resulting technical complications, ODL rather aims for a variation of common non-object-oriented imperative dynamic logics as simple as possible on the one hand, but still revealing the true logical essentials of object-orientation on the other hand. Furthermore, the dynamic logic to create should provide a structural representation of all features of object-oriented programming languages and an effective, "natural" construction should conduct the required translation of those features not having a direct counterpart in the dynamic logic.

A natural construction will treat the atomic elements of the programming language in a uniform, structural way and avoid exceptional cases or global dependencies. Further, it will amount to a relatively straightforward representation rather than depend on artificial encoding. For example, encoding whole object arrays into a single – suitably large – arithmetic number is computationally possible, but considered comparably unnatural. Moreover, almost all programming languages would collapse to one big equivalence class if arbitrary encodings were allowed for transformation. In other words, this representation follows the concept of schematology (Harel *et al.*, 2000). In this case, schematology demands the translation to be independent of the coding capabilities of a specific underlying domain of interpretation. Quoting from (Harel *et al.*, 2000):

> [schematology or the uninterpreted level of reasoning] is the most appropriate for comparing features of programming languages, since we wish such comparisons not to be influenced by the coding capabilities of a particular domain of interpretation.

This thesis consists of three parts and is organised as follows. First, the definition of a dynamic logic, ODL[1], is presented along with its underlying programming language. Second, an exemplary investigation why object-oriented programming languages can be translated into the dynamic logic by an effective though natural construction is shown. And third, a calculus for the dynamic logic is developed, about which the soundness and relative completeness proofs constitute the central part of this work. The relative completeness theorem proves completeness modulo first-order arithmetic. Vaguely speaking, in addition to the prevailing complexities of the domain of computation itself, such a program verification calculus will be complete. In combination with an adequate technique that handles the domain of computation, such a relatively complete calculus would constitute an algorithm with which all statements about programs could be verified automatically that Turing machines can hope to cover at all.

Even though one area of application will be the theoretical foundation of the KeY System (Ahrendt *et al.*, 2004) for the Java programming language (Gosling *et al.*, 1996), this thesis is not intended to treat only one particular programming language. Rather, the idea is to handle a broader range of object-oriented programming languages by uncovering the essential characteristics of object-oriented programming. Nevertheless, theoretical underpinnings of the KeY System and its update mechanism are a worthwhile goal. Especially §4.5.6 provides criteria for a calculus that allow to lift the relative completeness proof for ODL to a relative completeness statement about the full KeY System. The general hope is that a modular proof will be less complicated to conduct and to understand by splitting into a reduced ODL and an extended KeY part.

[1] ODL is short for Object Dynamic Logic. During informal statements, this work does not explicitly distinguish the programming language ODL from the logic ODL or the calculus presented for ODL, whenever confusions are impossible from the context.

1.2 ODL at a Glance

From a broad perspective the structure of the programming language underlying ODL is an extension of the classical non-object-oriented imperative WHILE programming language by parallel local updates of non-rigid function symbols. In contrast to rigid function symbols, non-rigid function symbols may have distinct interpretations in different states of a dynamic logic model. An update is the finite description of a (deterministic) modification of a non-rigid function during the transition from one state to another. Local updates only consider changes of a function at one single position. Parallel updates bundle multiple (but finite) simultaneous changes of possibly distinct function symbols into one combined update. A WHILE language on top of parallel updates essentially offers loops of conditional parallel updates to non-rigid function symbols.

Example 1.2.1 Consider the following update formula.

$$\langle f(x) \lhd a, f(y) \lhd b, g(x) \lhd c \rangle \phi$$

This formula holds in a state s if the formula ϕ holds in the state obtained from s by modifying the interpretation of f at (the position described by the value of) x to the value of a while simultaneously modifying the interpretation of f at y to b[2], and g at x to c. □

While in (Harel, 1984), the idea behind the regular propositional dynamic logic PDL_{reg}[3] was to find the essentials of imperative programming in order to build a more theoretically inclined foundation of the dynamic logic for WHILE, we now admit the generous control-structures of WHILE as a basis and aim to find the essentials of object-oriented programming on top of it. Despite their superficial similarities, common non-object-oriented imperative WHILE and variations of WHILE that are capable of naturally representing object-orientation do in fact differ. In contrast to the simple WHILE programming languages considered most often, object-oriented languages provide, for example, intrinsically unbounded data structures by a conglomerate of object

[2]The current stage does not yet permit to consider the case of ambiguous conflicting updates, where x and y evaluate to the same position, but a and b differ. Those so called clashes will have to receive a well-defined semantics, finally. Either by reducing clashes to a no-op operation, by stopping any further execution, or by letting one update predominate the effects of the other.

[3]The regular propositional dynamic logic PDL_{reg} allows assignment to atomic program variables, nondeterministic branching, nondeterministic repetition and conditional continuation as statements.

reference structures. Apart from Gödel encoding[4], those data structures of unbounded size cannot even be emulated in WHILE programming languages with (finite) static arrays without pointers.

Although there are a number of other approaches to object-oriented verification, ODL is a slimmer language with – in some cases considerably – less built-in language features, yet still proper object-orientation. KeY and Java^{light} (von Oheimb, 2001), for example, are based on rather huge languages. Additionally, ODL does not simulate a programming language interpreter or a Hoare calculus in an even more complicated logic like higher-order logic. Instead, ODL directly is the language of interest rather than providing yet another layer of abstraction. Except for JAVACARDDL, dynamic logic approaches like ODL are rare. Furthermore, contrary to the majority of approaches, ODL has a relative completeness proof and is the only one that has been proven complete relative even to first-order arithmetic, which is – in some sense – the strongest conceivable completeness statement by the *Unvollständigkeitssatz* of (Gödel, 1931).

In contrast to other approaches, ODL is its own assertion language. Thus, according to the common spirit of dynamic logics, there is no artificial distinction between programming language, assertion language and formalism for reasoning about program specifications. In the ODL logic all those aspects are neatly combined into one uniform concept. Especially, ODL does not impose unnecessary distinctions: there is no difference between program terms and terms of the logic, which is only possible because of the prohibition of side-effects and exceptional evaluation.

1.3 Architecture

In practical verification scenarios the predominant part of the project source code is written in some other object-oriented programming language than ODL. Nevertheless, the logic ODL yields a valuable contribution on the way to progress for practical system verification. It takes three conceivable roles in a verification scenario for a distinct source language, say, JAVA:

1. ODL on the specification level of Model Driven Architecture

2. ODL as intermediate verification language

3. ODL as a reliable basis for a full-custom verification calculus

[4]This encoding needs the infinite set of mathematical integers, because machine-sized integers do not suffice for encoding unbounded information.

Finally, independent of questions concerning the particular programming language under consideration, ODL serves as an adequate basis for theoretical investigations of object-oriented verification. This circumstance connected with the fact that ODL characterises the logical essentials of object-orientation that deserve most attention from a theoretical perspective.

As for 1, by the intuitive simplicity of the language, it is a viable suggestion to use ODL at the system specification level. The simple object-oriented programming language underlying ODL will then constitute a replacement for the UML action semantics in Model Driven Architecture (MDA) (Balcer & Mellor, 2002; MDA, 2003). In short, MDA is a top-down approach centring development around the model of the system. MDA-conform development starts with the construction of a system specification, which will be refined successively to produce the final product with as much source code as possible generated directly from the model. The model will always be the major artifact of development with the source code only of secondary importance in the hope that this priorisation of artifacts always leads to consistency of model and system. In order for this to be achieved, the model should be refined at some stage to incorporate a high-level operational specification as part of the action semantics of UML 2.0. Then the simplifying ODL is a good choice for an action semantics language in the case of an object-oriented target language.

To be said in behalf of 2, ODL is suitable as an intermediate language in practical verification scenarios. If, for instance, Java is the program source language, the translation of §3 allows to express the effects of the Java program in the simpler language ODL. This is assumed as the "default" scenario in the course of this thesis.

Concerning 3, the ODL calculus can be used as the basic ingredient for deriving a full-custom calculus for the particular source code language at hand. Although, just like in full-custom hardware design, this is a more intricate procedure than mere[5] translation on the program level. At least, a template-like construction on the basis ODL reduces the overhead in comparison to a design of a verification system from scratch. The nontrivial question of how to retain completeness in such an undertaking will be examined in §4.5.6.

The "default" ODL setting number 2 primarily considered in this work assumes a special verification architecture with ODL as its intermediate language. Suppose that there are programs in some object-oriented source language. More probably, there even is a collection of programs in various object-oriented (or at least purely imperative) programming languages. In

[5]Semantics-preserving source language translation already is a non-trivial undertaking. Although it is still much simpler than building a new verification system.

order to verify such a program conglomeration each programming language would need a dedicated verification system built on top of its own calculus plus appropriate means for communicating semantics between the verification systems for the respective languages. Unfortunately, building a sound and (relatively) complete verification calculus for an object-oriented programming language from scratch is a tremendous amount of work, not to mention the effort of crafting a corresponding (semi-)automatic proof system. Even retrofitting an existing program verification system to a new object-oriented programming language requires considerable endeavour. Especially exchanging the particular peculiarities of the old language, which are most probably spread throughout the calculus, with those of the new language can be utterly tedious and error-prone.

In order to solve those problems we propose an intermediate language approach. ODL has been designed as a sufficiently abstract common heart of object-oriented programming languages that is flexible enough to cover all features of object-oriented programming by translation. The actual source language programs, will then be translated into ODL. A translation on the level of the program is by far simpler than an attempt to "translate" a program verification calculus. The source language programs will be specified in some specification language, assumed to be ODL. Since ODL satisfies (almost) all needs for a proper specification language there is no advantage in constantly considering a further indirection during the course of this work, even though choosing UML/OCL (Rumbaugh *et al.*, 1998; Rumbaugh *et al.*, 1999) or JML (Leavens *et al.*, 1998) instead would be possible. It is a common observation that the language of programs is usually far more restricted by environmental and pragmatical constraints than the specification language.

What can turn out to be a further advantage of the ODL intermediate language approach in comparison to a dedicated JavaCardDL calculus is the greater simplicity. ODL is a language with a rather frugal and intuitive calculus. Illustrating the effects of a simpler logic in an application context is a more promising undertaking than explaining a very complex technical logic. Of course, at some stage, the inherent complexities of the source programming language have to be mastered by a verification system, anyway. Yet, a translation on the familiar programming language level by harmless program-transformation could be far more convenient than complicated techniques on the logical layer.

As Fig. 1.1 shows, from the source language, enriched with specifications in, say, ODL itself, a natural translation leads to one single language: ODL, in which correctness statements about programs can be expressed. ODL has a sound and relatively complete program verification calculus from which a proof system can be derived. This theorem prover system can come up with two different outcomes to the question of whether or not the source language program satisfies its specification: “Yes” or “No”. Of course, due to the undecidable nature of the verification problem a third potential behaviour is an indefinite run of the proof system, from which neither correctness of the source program nor the presence of a bug can be concluded.

With this approach, ODL allows to treat a broad range of object-oriented programming languages simultaneously by replacing the simple translation on the familiar level of the programming language rather than reinventing a verification calculus.

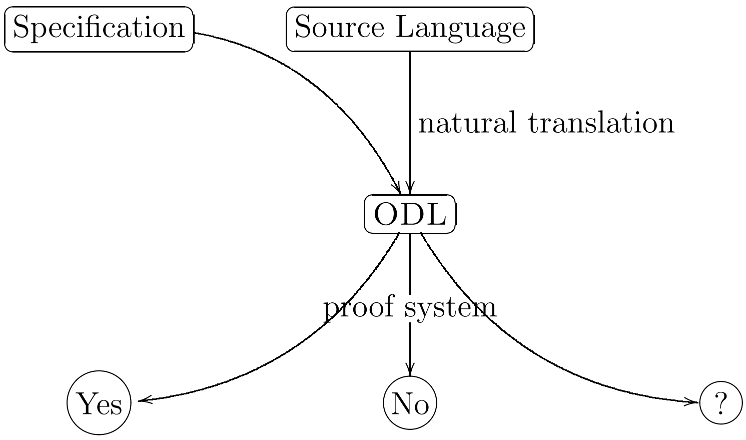

Figure 1.1: Verification Architecture. The source code program is translated into ODL along with its specification. On the basis of the ODL conjecture, the proof system outputs “Yes” along with a proof when the program meets its specification or “No” when it provably does not. The imaginary output “?” represents the case that the proof system cannot come to a final conclusion but continues to search for a proof indefinitely.

§2 presents the ODL language that has been depicted in the centre of Fig. 1.1. The questions of a natural translation from the source language will be investigated in §3. Finally, Chapt. 4 examines the proof system arcs of Fig. 1.1. That the output “Yes” is never produced when it would be unjustified is the topic of the soundness theorem in §4.4. That the proof system never outputs “No” but always “Yes” for a true specification would be the tenor of an absolute completeness statement. Since full completeness is impossible for ODL and the verification problem itself is not even semi-decidable, ODL only possesses the weaker form of relative completeness. The relative completeness theorem in §4.5 states that those cases in which the answer “Yes” fails to appear and "?" is produced[6] instead, this is – broadly speaking – not a deficiency of ODL but already due to the improvability of an equivalent formula of ordinary non-modal first-order arithmetic. A result of "No" is always justified.

1.4 Context

An implementation, called *i*ODL, of the program verification calculus ODL has been integrated into the KeY System. The KeY System (Ahrendt *et al.*, 2004) is a semi-automatic interactive proof system based on sequent calculus for dynamic logic for Java (called JavaCardDL). KeY adds formal specification and verification facilities to UML[7]-based software models by means of theorem proving. Due to the close integration into a familiar modelling environment and a real-world programming language, KeY has surpassing prospects of accomplishing its goal of bridging the gap between what academic case-studies demonstrate is possible with verification and what real industrial practice adopts. In order to facilitate a smooth integration into the modern development process and to permit a gradual involvement of formal methods, KeY combines with usual case-tools. Currently, KeY is available as a stand-alone prover and as a plug-in for the Together CASE-tool (TogetherSoft, 2003). The system supports UML+OCL specifications as well as JML specifications (Leavens *et al.*, 1998) for full JavaCard programs.

1.5 Related Work

(Stärk & Nanchen, 2001) present a calculus for a dynamic logic on abstract state machines (ASM) (Gurevich, 2000; Börger & Stärk, 2003). The calculus consists of about 30 schematic axioms and translates ASM rules into conditional ASM-updates, which again will be translated into modal formulas. Several update axioms depend on well-definedness conditions, for which appropriate inference rules are included as well. In this context, an ASM rule is well-defined if it terminates and does not produce clashes, i.e. contradictory update assignments. Thus, a typical proof will check termination according to axioms, continue to prove the absence of clashes and proceed with computing update predicates $\mathrm{upd}(\mathtt{R}, f, x, y)$[8]. Subsequently, those predicates

[6]Reasonably careful proof systems that examine proofs in a sufficiently systematic way will only emit "No" when justified.

[7]UML = Unified Modeling Language (Rumbaugh *et al.*, 1998; Rumbaugh *et al.*, 1999)

[8]$\mathrm{upd}(\mathtt{R}, f, x, y)$ says that the ASM rule R performs an update of the dynamic function symbol f at position x to the value y.

will be translated into elementary modal equations by a rule of the form $\text{upd}(\mathtt{R}, f, x, y) \rightarrow [R]f(x) \doteq y$. From those equations, the proof essentially consists of reasoning in equational multi-modal logic. By proving the ASM logic to be a definitional extension of first-order logic, the calculus is shown to be complete for ASMs without recursion and iteration[9]. Absolute completeness of the ASM logic is possible because – unlike the usual touch of abstract state machines – the modalities of the calculus only refer to single-step firing of ASM rules. The implicit repetition loop, which drives an ASM forward until termination, is not being considered.[10] Unfortunately, proving correctness of relatively simple ASMs requires an enormous effort with this calculus.

(von Oheimb, 2001) describes a Hoare calculus for Java^{light}, which is the first[11] logic for an object-oriented language proven sound and relatively complete for partial correctness. Java^{light} supports side-effects, recursion, dynamic dispatch, exception handling, static class initialisation, object creation, static fields and methods as well as static overloading. Like in (von Oheimb & Nipkow, 2001), the soundness and completeness proofs are formalised in Isabelle/HOL. However, the completeness is only relative and modulo (in-)completeness of higher-order logic. A disadvantage consists in the somewhat cumbersome assertion expressions that result from the explicit state parameters in each assertion.

(von Oheimb & Nipkow, 2001) describe a Hoare calculus for NanoJava, a Java-like language with conditions, loops, assignments to local variables and fields, object creation, casts, method calls and exclusively class types (especially there are no boolean tests but only references checks for equality to `null`). The calculus is formulated in Isabelle/HOL (Paulson, 1994; Nipkow *et al.*, 2002; Nipkow & Paulson, 2000)[12] and proven sound and complete relative to higher-order logic interactively. Despite the fact that NanoJava prohibits side-effects during expression evaluation, and even though the Hoare calculus is already simplified a lot in comparison to other object-oriented Hoare logic approaches, the calculus and proof examples still seem rather complicated and technical due to the general nature of Hoare calculi. The

[9]Essentially, the ASMs without recursion and iteration considered in (Stärk & Nanchen, 2001) boil down to sequential compositions of deterministic single-step firing of conditional parallel updates, which can be reduced equivalently to a set of conditional parallel updates.

[10]However, ASMs include an unbounded parallel forking command *forall*. In some sense, it evaluates a possibly countable infinite amount of instantiations of rules in parallel. Hence forall has a questionable computational complexity to consider as an atomic machine operation.

[11]Except for the smaller language SPOOL (de Boer, 1999).

[12]With HOL based on work of (Church, 1940).

notational and conceptional advantages of dynamic logics should achieve improvements on that issue.

(Nipkow, 2003) introduces a programming language, JINJA, with an operational semantics[13] exhibiting core features of JAVA. Big step and small step[14] operational semantics for JINJA are shown equivalent[15], and type safety is proven. JINJA provides boolean and integer as primitive types, references, null pointers, object creation, casting, field access and assignment, method calls (with static binding), local variable declarations, sequential composition, conditions, loops, exception throwing and catching. JINJA only provides overriding, not overloading, and has the peculiarity of LISP-like dynamic variable binding. Unfortunately, the operational semantics of JINJA consist of a fairly huge amount of rules. As an intermediate report, (Nipkow, 2003) does not yet present a calculus for JINJA.

(Igarashi *et al.*, 2001; Igarashi *et al.*, 1999) present a calculus, Featherweight Java, for a small pure non-imperative functional core of JAVA, with a touch very similar to the λ-calculus (Barendregt, 1984). As Featherweight Java concentrates on questions of type safety, it has a different focus than verification systems for functional behaviour. In addition to a treatment of constantly typed programs, the authors provide an extension that deals with parametric genericity except for generic type inference. The programming language underlying Featherweight Java provides (mutually recursive) class definitions, methods, fields, inheritance, method overriding, object construction, method invocation, field access, variables and casts. Featherweight Java does not provide assignments, loops, if-conditions, sequential composition, interfaces, overloading, super calls, null pointers, primitive types, abstract methods, field shadowing, access control (⌜**public**⌝ modifiers etc.), exceptions or side-effects. Without assignments and without side-effects, Featherweight Java represents a pure[16] functional object-oriented programming language. By nature of a pure functional language, objects do not change after the time of their construction. This implies that all expressions finally reduce[17] to newly constructed objects, which is the reason for the calculus

[13]Essentially, operational semantics define an interpreter for the programming language.

[14]In contrast to small step semantics, big step semantics evaluate expressions in a single "computation" straight to their final value. Hence, only small step semantics can evaluate an expression by iterated evaluations of different expressions in a sequence of distinct intermediate states.

[15]This means that the evaluation relation of the big step semantics coincides with the transitive closure of the small step semantics at the end of the evaluation, i.e. when only atomic values or raised exceptions remain.

[16]A functional language is *pure* if expression evaluation does not produce side-effects.

[17]Actually, the calculus essentially consists of syntactical reduction rules similar to the β-reduction of the λ-calculus.

only working on objects of the form ⌜**new** C(e_1,...,e_n)⌝. After all, the objects of Featherweight Java are essentially limited to mere immutable data containers: tuple types with a type hierarchy and method overriding.

(de Boer, 1999) describes a sound and complete Hoare calculus for a sequential object-oriented programming language, SPOOL, with main focus on pointers. SPOOL provides conditions, loops, sequential composition, pointers, null pointers, object creation, and only synchronised[18] methods. The logic for SPOOL has monotone[19] varying domain[20] semantics and also provides quantification over all finite sequences of objects of a particular type.

1.6 Basic Notions

1.6.1 Notation

This section briefly summarises the notation used in this paper. In the meta-language let $\Rightarrow$ denote the meta-level implication, and $\iff$ is used for meta-level equivalence.

That there is a possible transition in interpretation ℓ from state s to t when running the program α is denoted with $s\rho_\ell(\alpha)t$, as will be defined formally in Def. 2.3.1. This gives the modal accessibility relation for the program α as occurring in the definition of the semantics of $\langle\alpha\rangle$ and $[\alpha]$. If the state s is known from the context and no ambiguities arise, then "there is $s\rho_\ell(\alpha)t$" is a short notation for "there is t with $s\rho_\ell(\alpha)t$", and similar when t is known from the context.

In the course of this thesis, letters from the beginning of the alphabet, like c, d, will be preferred for constant symbols, while x, y, z are usually preferred for variable symbols. By convention, f, g are typical function symbols and p, q representative choices for predicate symbols. For a formula ϕ, $FV(\phi)$ denotes the set of free variables occurring in ϕ. Likewise, $FV(M) := \bigcup_{m\in M} FV(m)$ denotes the collective set of free variables in a set M. By an abuse of notation, write $FV(a, b, c)$ for $FV(\{a, b, c\})$.

1.6.2 Terminology

This section introduces some standard mathematical terminology.

[18]Contrary to the JAVA notion, here, synchronised means that at any time there can be at most one active method call within the stack trace. This additionally prevents recursion.

[19]Monotone domain semantics means that there only is object creation, no deallocation.

[20]Varying domain semantics implies that – due to object creation – the modal formula $\phi \land \langle\alpha\rangle\neg\phi$ can be true even if $Var(\phi) \cap Var(\alpha) = \emptyset$.

Definition 1.6.1 (Lattice) *A* lattice *is a partially ordered set L in which every (non-empty) finite subset has an infimum and supremum. Equivalently, a lattice is a set L with two binary operations $\cap$, $\cup$, which satisfy for each $a, b, c \in L$*

$$\begin{array}{lrcl}
\textit{idempotent} & a \cap a & = & a \\
 & a \cup a & = & a \\
\textit{commutative} & a \cap b & = & b \cap a \\
 & a \cup b & = & b \cup a \\
\textit{associative} & a \cap (b \cap c) & = & (a \cap b) \cap c \\
 & a \cup (b \cup c) & = & (a \cup b) \cup c \\
\textit{absorption} & a \cap (a \cup b) & = & a \\
 & a \cup (a \cap b) & = & a
\end{array}$$

A function $f : L \to N$ is a homomorphism *of lattices if for each $a, b \in L$*

$$\begin{array}{rcl}
f(a \cap b) & = & f(a) \cap f(b) \\
f(a \cup b) & = & f(a) \cup f(b)
\end{array}$$

Proof: The asserted equivalence of order-theoretic and algebraic lattices has to be proven. Given an order-theoretic lattice, it satisfies the axioms with operators $\cap$ for infimum and $\cup$ for supremum.

An algebraic lattice defines a partial order by defining $a \leq b :\iff a \cup b = b \iff a \cap b = a$, which yields an order-theoretic lattice. See (Davey & Priestley, 2002) for more detailed information. ■

Example 1.6.1 Consider the prototypical example of a lattice with two elements C and D, their supremum $C \cup D$ and their infimum $C \cap D$ in Fig. 1.2. The lattice further contains the top element $\top$ and the bottom element $\perp$. □

Remark 1.6.2 *Homomorphisms of lattices are* monotone, *i.e. for all $a, b \in L$ it is $a \leq b \Rightarrow f(a) \leq f(b)$.*

Definition 1.6.3 (Relational Composition) *For a subset $E \subseteq M$ and a relation ρ on M define the* relational composition *as*

$$\begin{array}{rcl}
E \circ \rho & := & \{t \in M \ : \ \text{there is } s \in E \ s\rho t\} \\
\rho \circ E & := & \{s \in M \ : \ \text{there is } t \in E \ s\rho t\}
\end{array}$$

Definition 1.6.4 (Multisets) *A* multiset *is a collection of elements of a domain D, in which the number of occurrences of each element matters but*

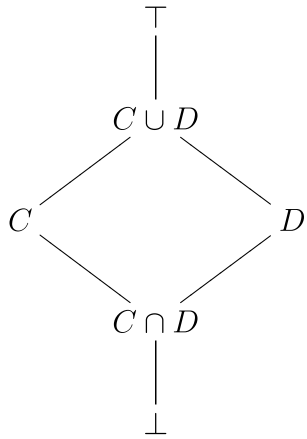

Figure 1.2: Dense Lattice Example

not the order of occurrence. Thus, a multiset M is a map $M : D \to \mathbf{N}$ with $M(a)$ being the multiplicity of occurrences of a in M. The set theoretical operations are extended to sets in an obvious way.

$$
\begin{array}{rcl}
x \in M & := & M(x) > 0 \\
M \cup N & := & M + N \\
M \cap N & := & \min(M, N) \ = \ (a \mapsto \min(M(a), N(a))) \\
M \setminus N & := & \max(M - N, 0) \ = \ (a \mapsto \max(M(a) - N(a), 0))
\end{array}
$$

As a notation for finite multisets, $\{\!\{a, b, b, b, c, d, d\}\!\}$ denotes the multiset M with $M(a) = M(c) = 1, M(b) = 3, M(d) = 2$ and $M(x) = 0$ otherwise.

Definition 1.6.5 (Transitive Closure) *The relation R^* is the (reflexive)* transitive closure *of the relation R and is defined as follows.*

$$xR^*y \iff \text{there are } x = x_0, x_1, x_2, \ldots, x_n = y \text{ such that } x_i R x_{i+1} \text{ for each } i$$

Definition 1.6.6 (Confluence) *A relation R on a set M is* confluent *if for each $x, y_1, y_2 \in M$ with xRy_1 and xRy_2 there is $z \in M$ such that $y_1 R^* z$ and $y_2 R^* z$. This leads to a situation as depicted in Fig. 1.3.*

Definition 1.6.7 (Noetherian) *A relation R on a set M is* Noetherian *(or terminating) if there is no infinite sequence $x_1, x_2, x_3, \ldots \in M$ with $x_i R x_{i+1}$ for each i. Noetherian relations prohibit situations like the one depicted in Fig. 1.4.*

$$\begin{tikzcd}
 & y_1 \arrow[dr, dashed, "\exists"] & \\
x \arrow[ur, "\forall"] \arrow[dr, "\forall"'] & & z \\
 & y_2 \arrow[ur, dashed, "\exists"'] &
\end{tikzcd}$$

Figure 1.3: Confluence Diagram

$$x_1 \longrightarrow x_2 \longrightarrow x_3 \longrightarrow x_4 \longrightarrow \cdots \longrightarrow x_i \longrightarrow x_{i+1} \longrightarrow \cdots$$

Figure 1.4: Noetherian Descending Chain Condition Diagram

Chapter 2

Update Logic

2.1 Overview

This section presents the actual structure of the update logic ODL, starting with a brief overview. ODL is a first-order dynamic logic with programs built from parallel updates by nesting of conditions and loops.

ODL can be obtained from first-order logic as follows: enrich first-order logic by $[]$ and $\langle\rangle$ modalities of a dynamic logic for the WHILE programming language, taking updates instead of mere program variable assignments as atomic programs.

Example 2.1.1 For the moment, assume $|$ to be a built-in divisibility[1] relation. As an introductory motivating example, consider the ODL program α below, for which the formal syntax and semantics will be introduced during this section.

$$\texttt{if}(2 \mid c)\,\{d{\lhd}c+2\}\texttt{else}\{d{\lhd}c+1\}$$

In this program, the statement $d{\lhd}c+1$ is called an update, which modifies the interpretation of d to be the successor of c. Depending on whether 2 divides c or not, α will perform the update $d{\lhd}c+2$ or the update $d{\lhd}c+1$, instead.

Consider the following (partial) specification of the behaviour of program α in case that c is initially odd.

$$\exists k{:}\texttt{nat}\ \ c \doteq 2 \cdot k + 1 \quad \rightarrow \quad [\alpha]d \doteq c + 1$$

This formula specifies that whenever c is odd, all executions of the program α lead to d having a value that is the successor of the value of c. During the

[1]The relation $x \mid y$ holds if and only if the integer x divides y.

course of proving that this is a correct specification for α, updates will also be promoted to subterms like the following.

$$([d \lhd c+1]d) \doteq ([d \lhd c+1](c+1))$$

From a semantical perspective, an update in front of a subterm leads to an evaluation of the subterm with a modified interpretation of d, regardless of the context evaluating the whole term. More syntactically speaking, moving updates to subterms promotes the effect of an update down to the subterms. This update promotion mechanism will further simplify the above equation to the following tautology.

$$c+1 \doteq c+1$$

After the necessary concepts have been introduced, program α will be examined in more detail again in Ex. 2.4.1 and proven correct in Ex. 4.3.1. $\square$

ODL_m is obtained from ODL by allowing method invocations. Method invocations are limited to the top-level expression after updates to local program variables, though: $x \lhd \mathtt{m}(o, e)$. This alleviates the problem of operator and operand evaluation order in the presence of side-effects, since ODL expressions like o and e cannot produce side-effects.

As will be seen in Lem. 4.2.7, multiple consecutive updates can be combined into a single update. This leads to ODL programs of a very simple structure: `while() {}` loops and `if() {}else{}` conditions nested around single parallel updates, and sequential compositions of loops or conditions. Since parallel updates can always be split into a sequential composition of singleton updates, the essential difference between dynamic logic for WHILE and ODL is that ODL allows modifications of structured data like $f(t, a)$ instead of just program variables like `x`. In other words, $\mathrm{DL}_{\mathrm{WHILE}}$ only allows assignments to atomic terms, whereas ODL allows assignments to arbitrarily nested compound terms.

For practical purposes, dynamic logics used in program verification often distinguish rigid from non-rigid symbols. Unlike (syntactically) non-rigid symbols, (syntactically) *rigid* symbols share the same value in every state of a particular interpretation per definition. In one particular interpretation, of course, syntactically non-rigid symbols may still happen to have the same interpretation in all states, but do not need do so to by *a priori* declaration. However, the syntactic structure of ODL does not formally involve rigidity declarations for pragmatic reasons. In spite of the practical relevance, this simplification clarifies the presentation. Furthermore, the addition of a formal concept of syntactic rigidity is straightforward. Still, several function symbols are informally classified as rigid or non-rigid in this presentation, in order to support the reader's intuition about what is intended to happen to those symbols.

2.2 Syntax

This section presents the syntactic structure of the dynamic logic ODL. Given a vocabulary of symbols along with type information in a corresponding type-system the language ODL will be inductively defined by combining those symbols.

2.2.1 Type-System

The type-system of ODL needs to have some way to represent the type-system of the object-oriented source language sufficiently. This means that the calculus can simulate all statements of an effect that depends on (compile time or run-time) type information. In object-oriented programming languages, this especially includes the virtual dispatch of method calls. There are at least two ways for achieving this. First, a plug&play type-system with a formal description language for type-systems, which provides a way for plugging reasonable source type-systems into the calculus, perhaps similar to the UML Meta Model. Or second, an endogenous approach, where typing information is focused within a single predicate `instanceof`. Then statements with type dependent effects have to be unfolded into explicit low-level if-then-else type check cascades.

The endogenous approach is conceptually simpler and perhaps even more flexible. Yet, it results in a little more verbose and explicit proofs within ODL, it relies on a finite number of classes[2], and in naïve realisations it is rather inefficient for methods appearing with an identical name and arity in many classes because of its incompetence in distinguishing types in advance. However, this deficiency can be resolved quite easily by imposing unique naming conventions for conceptually distinct entities, as can be ensured by semantical name analysis preprocessing. In the spirit of simplicity the endogenous type system for dynamic types is chosen.

The construction of ODL begins by plugging in an arbitrary type-system stemming from the particular source language under consideration, say JavaCardDL. The type-system Typ is intended to contain all object types of some source language like JavaCardDL in the corresponding class hierarchy.

[2]More precisely, a finite amount of subclasses is assumed. Intra-language translations based on branching cascades usually require a finite number of *subclasses*.

Definition 2.2.1 *In the following, let* TYP *be an arbitrary decidable*[3] *type-lattice with* $\perp$ *as bottom type,* $\top$ *as top type,* $\cap$ *as infimum type or greatest common subtype, and* $\cup$ *as supremum type or smallest common supertype. Furthermore, let* TYP *have a designated type* `nat` $\in$ TYP *for natural numbers.* $\sigma \leq \tau$ *holds if* σ *is a subtype of* τ*.* $\sigma < \tau$ *holds if* $\sigma \neq \tau$ *is a proper subtype of* τ

Example 2.2.1 Note that in the case of JAVA the type-system will be bounded by a superficial bottom type $\perp$ which is a subtype of all classes and interfaces. Moreover, note that type-systems of conventional object-oriented programming languages will be rather coarse. Assuming the class hierarchy

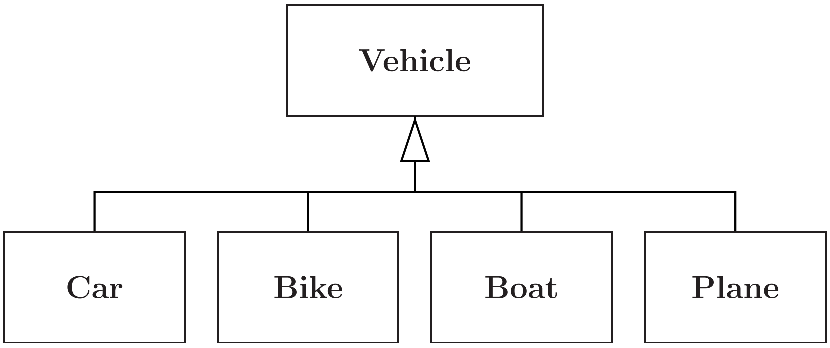

Figure 2.1: UML Class Diagram of Vehicle Type Hierarchy

in Fig. 2.1 of distinct classes `Car`, `Bike`, `Boat`, `Plane` which extend a common base class `Vehicle`. Then contrary to finer type-systems, rather different type suprema result in the very same type of the class hierarchy. Consider

$$\texttt{Car} \cup \texttt{Bike} = \texttt{Car} \cup \texttt{Boat} = \texttt{Plane} \cup \texttt{Boat} = \texttt{Vehicle}$$

Instead of supplementing the type-system with finer suprema, such a behaviour of the source code type-system is directly reflected in the type-lattice. Primitive types of the source language, like *int*, *short*, *boolean* will be integrated into the type-lattice. □

Example 2.2.2 As a simplified part of the standard JAVA class hierarchy, the type-system will contain, for instance, the lattice depicted in Fig. 2.2. □

ODL assumes external or exogenous symbol typing information instead of supporting explicit type declarations with the ODL language. Though,

[3]A type-lattice is decidable, if the membership and subtype relations are decidable. This is the usual case.

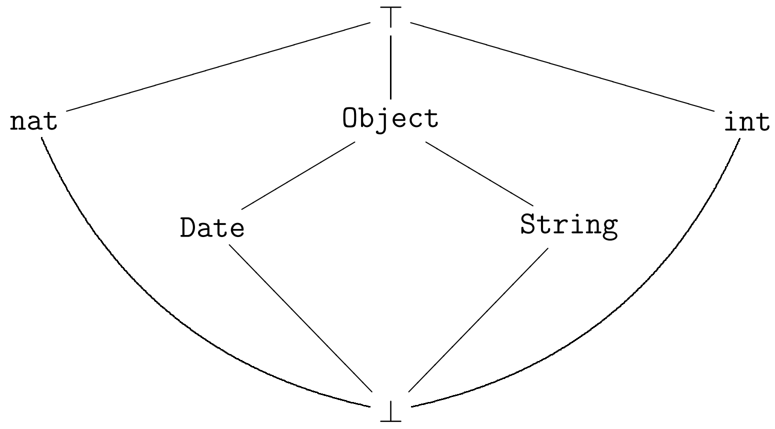

Figure 2.2: Lattice of an Excerpt of the JAVA Type-System

of course, insufficient from an application point of view, this is an adequate approach for symbol typing from a theoretical perspective. Just assuming that all symbols know their static type via some *a priori* mechanism outside of the logic simplifies the language and semantics a lot. Moreover, a classical semantical name and type analysis phase of a preprocessing compiler run can easily provide this type declaration functionality for free.

2.2.2 Signature

A *signature* Σ is the set of names (called *symbols*) of all entities nameable in a certain context. A signature is the vocabulary or alphabet of logical signs from which to build well-formed formulas. It is generally assumed to be given effectively. In addition to (constant) symbols for elements of the world, symbols also include variables from a set V. Σ is assumed to contain the usual signs $0, 1, +, \cdot, <, \leq, \geq, >$ for natural numbers. Moreover, Σ is always assumed to contain the rigid constant symbol `null` of type $\perp$, which is intended for the resolution of null pointers or otherwise undefined elements. Finally, Σ is always assumed to contain the rigid function symbols $\mathtt{obj}_\mathtt{C}$ of type $\mathtt{nat} \to C$, intended to represent newly created objects.[4] The $\mathtt{obj}_\mathtt{C}$ signs are called *object enumerator* symbols. For example, $\mathtt{obj}_\mathtt{C}(n)$ will be the newly created object of type C with object identifier n, and different from any newly created object with a different object identifier (*OID*) or different type. Symbols possess a type and are either individual, function or predicate symbols, with

[4]In the notation of the λP system of (Barendregt, 1992) an alternative for using the family of symbols $\mathtt{obj}_\mathtt{C}$ would be to use a single symbol obj of parametric generic type $\Pi\tau : *.(\mathtt{nat} \to \tau)$.

function and predicate symbols having a fixed arity. Variable symbols possess a type and are always *rigid*, i.e. their value is the same during all states of program execution. Constant symbols, however, are allowed to assume different values in distinct states during the program execution.

Be aware that, on the contrary, terminology on the level of programming languages like C++ says that constant program variables cannot change their value[5] by assignment. In the context of logic, however, variable symbols are primarily intended for quantification. This quantification should also suffice to establish a connection between the different states of program execution, because of which rigid variables constitute a significant ingredient to dynamic logic.[6]

All vocabularies are assumed to contain infinitely many variable symbols of each type, because of skolemisation, which – in the ODL calculus – works with variables.

2.2.3 Formulas

The terms $\mathrm{Trm}(\Sigma \cup V)$, formulas $\mathrm{Fml}(\Sigma \cup V)$ and programs $\mathrm{Prg}(\Sigma \cup V)$ of ODL will be defined by simultaneous induction in definitions 2.2.2, 2.2.3 and 2.2.4. They correspond to the classical notions of in a dynamic logic flavour. $\mathrm{Fml}_{FOL}(\Sigma \cup V)$ is the set of formulas of classical non-modal first-order logic.

Definition 2.2.2 (Terms) $\mathrm{Trm}(\Sigma \cup V)_\tau$ *is the set of* terms *of* ODL *of type* τ *with variables in* V *over the signature* Σ*. It is defined as the minimal set such that*

- $\mathrm{Trm}(\Sigma \cup V)_\sigma \subseteq \mathrm{Trm}(\Sigma \cup V)_\tau$ *for each* $\sigma < \tau$*.*
- *Every variable* $x \in V$ *of type* τ *is in* $\mathrm{Trm}(\Sigma \cup V)_\tau$*.*
- *If* $f \in \Sigma$ *is a function symbol of type* $\sigma_1 \times \cdots \times \sigma_n \to \tau$ *and* $t_1 \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_1}, \ldots, t_n \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_n}$ *then* $f(t_1, \ldots, t_n) \in \mathrm{Trm}(\Sigma \cup V)_\tau$*.*
- *If* $e \in \mathrm{Fml}(\Sigma \cup V)$ *and* $s, t \in \mathrm{Trm}(\Sigma \cup V)_\tau$ *then* $\mathit{if}\, e\, \mathit{then}\, s\, \mathit{else}\, t\, \mathit{fi} \in \mathrm{Trm}(\Sigma \cup V)_\tau$*.*

[5]Their instance fields are in general still subject to modification, though.

[6]In principle it is possible to make an orthogonal distinction between rigid and non-rigid function symbols, thereby allowing more than rigid variables and non-rigid constants. But non-rigid variables are of questionable utility, and the effect of constants syntactically declared as rigid can always be emulated with corresponding variables or ensured by meta lemmas ensuring rigid behaviour for particular constant symbols.

- *If* $\phi \in \mathrm{Trm}(\Sigma \cup V)$ *and* $\mathcal{U} \in \mathrm{Prg}(\Sigma \cup V)$ *is an update then* $\langle\mathcal{U}\rangle\phi \in \mathrm{Trm}(\Sigma \cup V)$.[7]

$\mathrm{Trm}(\Sigma \cup V) := \mathrm{Trm}(\Sigma \cup V)_\top = \bigcup_{\tau\in\textsc{Typ}} \mathrm{Trm}(\Sigma \cup V)_\tau$ *is the set of all terms of any type.*

Definition 2.2.3 (Formulas) $\mathrm{Fml}(\Sigma \cup V)$ *is the set of* formulas *of* ODL *with variables in* V *over the signature* Σ*. It is minimal with the following properties.*

- *If* $p \in \Sigma$ *is a predicate symbol of type* $(\sigma_1 \times \cdots \times \sigma_n)$ *and* $t_1 \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_1}, \dots, t_n \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_n}$ *then* $p(t_1, \dots, t_n) \in \mathrm{Fml}(\Sigma \cup V)$.

- *If* $t \in \mathrm{Trm}(\Sigma \cup V)$ *and* $C \in \textsc{Typ}$ *is a type then* $t\,\mathtt{instanceof}\,\mathtt{C} \in \mathrm{Fml}(\Sigma \cup V)$.

- *If* $s \in \mathrm{Trm}(\Sigma \cup V)_\sigma, t \in \mathrm{Trm}(\Sigma \cup V)_\tau$ *then* $s \doteq t \in \mathrm{Fml}(\Sigma \cup V)$.

- *If* $\phi, \psi \in \mathrm{Fml}(\Sigma \cup V)$ *then* $\neg\phi, (\phi \vee \psi), (\phi \wedge \psi), (\phi \rightarrow \psi) \in \mathrm{Fml}(\Sigma \cup V)$.

- *If* $\phi \in \mathrm{Fml}(\Sigma \cup V)$ *and* $x \in V$ *is a variable then* $\forall x\, \phi, \exists x\, \phi \in \mathrm{Fml}(\Sigma \cup V)$.

- *If* $\phi \in \mathrm{Fml}(\Sigma \cup V)$ *and* $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ *then* $[\alpha]\phi, \langle\alpha\rangle\phi \in \mathrm{Fml}(\Sigma \cup V)$.

Usually, V is implicitly assumed to be a set of infinitely many variables $x_1, x_2, x_3, \dots$ for each type. Bisubjunction $\phi \leftrightarrow \psi$ is treated as an abbreviation for mutual implication $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ in the calculus. Likewise, $s \neq t$ is an abbreviation of $\neg(s \doteq t)$.

The usual rules for bracket compaction apply, with operator precedence as follows. Quantifiers and modalities bind strong instead of extending far to the right. The precedence order is thus $\leftrightarrow, \rightarrow, \vee, \wedge, \neg, \exists, \forall, [], \langle\rangle, \doteq, \neq, <, \leq, >, \geq$. Especially note that those binding preferences imply

$$\begin{array}{rcccl} (\forall i\, \phi) \rightarrow \psi & = & \forall i\, \phi \rightarrow \psi & \neq & \forall i\, (\phi \rightarrow \psi) \\ (\langle\alpha\rangle\phi) \rightarrow \psi & = & \langle\alpha\rangle\phi \rightarrow \psi & \neq & \langle\alpha\rangle(\phi \rightarrow \psi) \end{array}$$

While the precise semantics of the syntactic constructs in formulas will be defined in §2.3, their intuitive meaning is already explained here. The logical connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ have their standard meanings. $\doteq$ is the equality on

[7] Updates only occur on term level during the incremental rewrite process. Term level updates are of no importance for the initial programs or program specifications.

terms, while the quantification $\forall x\, \phi$ holds if the formula ϕ holds for all assignments of the variable x. The conditional term $\mathit{if}\, e\, \mathit{then}\, s\, \mathit{else}\, t\, \mathit{fi}$ evaluates to the value of s whenever e is true in the current state, but evaluates to t otherwise. The type check formula $t\,\mathtt{instanceof\,C}$ expresses that the dynamic type of the object referred to by t is a subtype of class C. The modal formula $[\alpha]\phi$ expresses that the formula ϕ always holds after program execution, i.e. ϕ holds in all states that can be reached from the current state by executing the program α. Likewise, $\langle\alpha\rangle\phi$ holds if ϕ is *sometimes* true after program execution, i.e. in *some* state that can be reached by executing program α.

2.2.4 Programs

Definition 2.2.4 (Programs) $\mathrm{Prg}(\Sigma \cup V)$ *is the set of* programs *of* ODL *with variables in V over the signature Σ. It is defined to be the minimal set with*

- *If* $f_1(t_1) \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_1}, \ldots, f_n(t_n) \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_n}$ *and* $s_1 \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_1}, \ldots, s_n \in \mathrm{Trm}(\Sigma \cup V)_{\sigma_n}$*, then* $f_1(t_1) \lhd s_1, \ldots, f_n(t_n) \lhd s_n \in \mathrm{Prg}(\Sigma \cup V)$ *is an atomic program.*[8]

- *(For* ODL_m*) If* $\mathtt{c} \in \Sigma_\tau$ *is a constant symbol,* $\mathtt{m}$ *a method of type* $\sigma \to \tau$ *in class* ζ *and* $o \in \mathrm{Trm}(\Sigma \cup V)_\zeta, t \in \mathrm{Trm}(\Sigma \cup V)_\sigma$*, then* $\langle \mathtt{c} \lhd \mathtt{m}(o, t) \rangle \in \mathrm{Prg}(\Sigma \cup V)$ *is an atomic program.*

- *If* $\alpha, \gamma \in \mathrm{Prg}(\Sigma \cup V)$ *then* $\alpha; \gamma \in \mathrm{Prg}(\Sigma \cup V)$.[9]

- *If* $\phi \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *and* $\alpha, \gamma \in \mathrm{Prg}(\Sigma \cup V)$ *then* $\mathtt{if}(\phi)\,\{\alpha\}\mathtt{else}\{\gamma\} \in \mathrm{Prg}(\Sigma \cup V)$.

- *If* $\phi \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *and* $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ *then* $\mathtt{while}(\phi)\,\{\alpha\} \in \mathrm{Prg}(\Sigma \cup V)$.

For convenience, formulas occurring within program examples often use the Java logical operators like ⌜&⌝ instead of $\wedge$, and ⌜|⌝ instead of $\vee$.

Note that $f(s)$ is neither a proper identifier nor a location but only a descriptor, since it does not unambiguously and absolutely identify a location, but only describes a value relative to the current context of evaluation of s.

[8] In case of a syntactical separation of rigid and non-rigid terms, the f_i are assumed non-rigid here. Update programs like $f_i(r_1, \ldots, r_n) \lhd s$ are defined similarly for higher arities of the f_i.

[9] Note that program composition via ; is associative, which means that no grouping brackets are required for program composition.

The intuitive effect of the program $\alpha;\gamma$ is the effect of the sequential composition of the programs α and γ, i.e. the program that begins with the execution of α and – after its termination – continues to execute γ. The branching statement $\mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\gamma\}$ has an effect that depends on the value of the condition e in the current state. If e evaluates to true the branching statement $\mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\gamma\}$ produces the same effects as α, otherwise it operates like γ. The effect of the loop $\mathtt{while}(e)\,\{\alpha\}$ is to repeat the execution of α as long as e evaluates to true. The effect of an update $f_1(t_1)\lhd s_1,\ldots,f_n(t_n)\lhd s_n$ is to simultaneously modify the interpretations of the (non-rigid) function symbols f_i at the respective positions t_i to the value of s_i. The formal semantics of the above program constructs will be defined in §2.3.

What will be added to the program language for convenience are skip statements and single-side branches. They can be defined as abbreviations for other programs, and – by the nature of mere syntactic sugar – do not need a special treatment in the semantics or calculus.

- $\mathtt{skip} := x\lhd x$ is a no-operation statement, where x is a new constant symbol not occurring anywhere else.
- $\mathtt{if}(e)\,\{\alpha\} := \mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\mathtt{skip}\}$ is a single-side branch statement.

2.3 Semantics

By interpretation, a meaning will be associated with the formulas of the language defined by syntax. By virtue of their interpretation, some formulas express true facts about the world while others do not. With sound logical reasoning this truth entails from the premises of arguments to their conclusion. As the value of a constant symbol can change from state to state, the meaning of an ODL formula depends on the state of the program in which it is interpreted.

2.3.1 Domain of Computation

The domain of computation comprises integer arithmetic as well as adequate referents for object types. The constant symbol `null` stands for a specific object of type $\perp$. Especially, within the logic, any function may get assigned arbitrary values at the position `null`. This results in a very simple formulation of ODL, which does not have to care about questions of partial functions and multi-valued logics. For the purpose of achieving an equivalent translation from any source language to ODL, of course, the translation will have to include explicit `null` pointer checks and exceptional treatment. Even though this may lead to translations that attain less readability than the original JavaCardDL program, a major advantage of the ODL approach is the much greater simplicity of concepts. Another advantage of the uniform handling of `null` pointers in comparison to partiality during expression evaluation consists in $c.x \doteq c.x + 2$ always consistently being false, even in the case of $c \doteq \mathtt{null}$. This reduces the amount of exceptions for the validity of formulas.

2.3.2 Interpretation

This section defines the central concepts of the denotational semantics of ODL: primitive interpretations that relate syntactic structures and objects of a (model) world.

Definition 2.3.1 (Interpretation) *In general dynamic logics, an* interpretation *ℓ of the signature Σ is a non-empty set W of worlds, and an* accessibility *relation $\rho_\ell(\alpha) \subseteq W \times W$ for each atomic program $\alpha \in \mathrm{Prg}(\Sigma \cup V)$. For* ODL, *$W$ can be any set of states that is stable under transition with $\rho_\ell(\alpha)$ under atomic programs, while the accessibility relation $\rho_\ell(\alpha)$ for atomic programs will be fixed according to Def. 2.3.8.*[10] *Therefor, a* state *is a first-order interpretation, i.e. it consists of a universe – which is the same for all states – and an association of the symbols of the signature with elements in the world. More precisely, for each state $w \in W$ this amounts to an association of:*

- *bijections $val_\ell(w, \mathtt{obj}_\mathtt{C}) : \ell(\mathtt{nat}) \to O_\mathtt{C}$ into disjoint sets $O_\mathtt{C}$*[11] *with the object enumerator symbols $\mathtt{obj}_\mathtt{C}$.*
- *the set $\ell(\tau) := \bigcup_{\sigma \leq \mathtt{C}} O_\sigma$ with each type $\tau \in \textsc{Typ}$.*
- *an object $val_\ell(w, c) \in \ell(\tau)$ with each individual symbol $c \in \Sigma \cup V$ of type τ.*
- *a function $val_\ell(w, f) : \ell(\sigma_1) \times \cdots \times \ell(\sigma_n) \to \ell(\tau)$ with each function symbol f of type $\sigma_1 \times \cdots \times \sigma_n \to \tau$.*

[10] For W, this amounts to including all finite modifications, i.e. all states that only differ at finitely many points of finitely many functions.

[11] Intuitively, $O_\mathtt{C}$ is intended to contain those objects of most specific type `C`, i.e. which have no proper subtype of `C` as type but `C` itself. Hence, the universe splits into a family of disjoint sets $O_\mathtt{C}$.

- *a relation* $val_\ell(w,p) \subseteq \ell(\sigma_1) \times \cdots \times \ell(\sigma_n)$ *with each predicate symbol* p *of type* $(\sigma_1 \times \cdots \times \sigma_n)$.

Furthermore, $\texttt{nat} \in \textsc{Typ}$*, along with* $0, 1, +, \cdot, <, \leq, \geq, > \in \Sigma$*, are restricted to have the natural numbers*[12] $\mathbf{N}$ *as a fixed interpretation. As notation, further define* $\mathrm{dom}(\ell) := W$ *for occasional use.*

Note that in this definition, constant symbols and variable symbols are treated homogeneously. Since, in ODL, the interpretation of function symbols differs from state to state, the state w defines the value $val_\ell(w,\cdot)$ on Σ, while the proper interpretation ℓ defines it on V, independent from any states. Moreover, for simplicity, this definition follows the tradition of constant domain semantics in contrast to varying domain semantics. *constant domain semantics* enforces all states to share the same universe, while *varying domain semantics* permits distinct universes for different states.

As a matter of convenience, the case of $\perp$ can be integrated into the common $\texttt{obj}_{\texttt{C}}$ semantics with $\texttt{obj}_\perp$ being the constant mapping into the singleton set $O_\perp = \{val_\ell(w, \texttt{null})\}$.

Remark 2.3.2 *For* $\tau \in \textsc{Typ}$*, the* $\ell(\tau)$ *form a lattice with set inclusion* $\subseteq$*, set intersection* $\cap$ *and union* $\cup$*. Moreover,* ℓ *respects subtypes, which means that* ℓ *is a monotone map on types, i.e.* $\ell(\sigma) \subseteq \ell(\tau)$ *for each* $\sigma \leq \tau$.[13]

Proof: $\ell(\sigma) = \bigcup_{\rho \leq \sigma} O_\rho \overset{\sigma \leq \tau}{\subseteq} \bigcup_{\rho \leq \tau} O_\rho = \ell(\tau)$. ∎

Definition 2.3.3 (Semantic Modification) *The operation of a* semantic modification $[f(e) \mapsto d]$ *of the symbol* $f : \sigma \to \tau$ *at the position* $e \in \ell(\sigma)$[14] *to the value* $d \in \ell(\tau)$ *transforms the state* w *of the interpretation* ℓ *by*

$$val_\ell(w[f(e) \mapsto d], a) := \begin{cases} f' & \Leftarrow a = f \\ val_\ell(w, a) & \Leftarrow a \neq f \end{cases}$$

$$\text{with } f' := \left(b \mapsto \left\{ \begin{array}{ll} d & \Leftarrow b = e \\ val_\ell(w,a)\big(val_\ell(w,b)\big) & \Leftarrow b \neq e \end{array} \right\}\right)$$

---

[12] An isomorphic copy of $\mathbf{N}$ as domain for the type nat would be sufficient.

[13] However note that ℓ usually is no homomorphism of lattices on types. For example $\ell(\texttt{Car} \cup \texttt{Boat}) = \ell(\texttt{Vehicle}) \neq \ell(\texttt{Car}) \cup \ell(\texttt{Boat})$, because the common super class of Car and Boat would be Vehicle. Yet there are vehicles that are neither cars nor boats but planes.

[14] Note that $f(e)$ is not a term here, since e is just an element of the domain of interpretation, not a term. Further, note that a similar definition applies in the case of higher or lower arities of f.

The semantic modification of ℓ at $x \in V$ to d is defined similarly by

$$val_{\ell[x \mapsto d]}(w, a) \;:=\; \begin{cases} d & \Leftarrow a = x \\ val_\ell(w, a) & \Leftarrow a \neq x \end{cases}$$

2.3.3 Valuation of Formulas

A homomorphic continuation extends the interpretation ℓ of symbols to valuations of compound formulas as follows. The valuation $val_\ell(w, c)$ is already defined for symbols $c \in \Sigma \cup V$ according to Def. 2.3.1, and will be extended by homomorphic continuation to all formulas. Pursuing the inductive structure of expressions in the definitions 2.2.2-2.2.4, the semantics of a term, formula and program will be defined by simultaneous induction in the following definitions 2.3.4-2.3.8.

Definition 2.3.4 (Valuation of Terms) *Let w be a state of an interpretation ℓ. The* valuation of terms *with respect to ℓ and w is defined as follows.*

1. $val_\ell(w, f(t_1, \ldots, t_n)) \;:=\; \big(val_\ell(w, f)\big)\big(val_\ell(w, t_1), \ldots, val_\ell(w, t_n)\big)$, *when f is a function symbol of arity n and $t_1, \ldots, t_n \in \mathrm{Trm}(\Sigma \cup V)$ are terms.*

2. $$val_\ell(w, \mathit{if}\, \phi\, \mathit{then}\, r\; \mathit{else}\, t\; \mathit{fi}) \;:=\; \begin{cases} val_\ell(w, r) & \Leftarrow val_\ell(w, \phi) = \mathit{true} \\ val_\ell(w, t) & \Leftarrow val_\ell(w, \phi) = \mathit{false} \end{cases}$$ *when $r, t \in \mathrm{Trm}(\Sigma \cup V)$ are terms.*

3. $val_\ell(w, \langle \mathcal{U} \rangle t) \;:=\; val_\ell(w', t)$ *with $w \rho_\ell(\mathcal{U}) w'$ like for formulas.*[15]

Definition 2.3.5 (Valuation of Formulas) *Let ℓ be an interpretation and $w \in \mathrm{dom}(\ell) = W$ a state. The* valuation of formulas *with respect to ℓ and w is defined as follows. Let $t, t', t_1, \ldots, t_n \in \mathrm{Trm}(\Sigma \cup V)$, $\phi \in \mathrm{Fml}(\Sigma \cup V)$, $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ and $x \in V$ a variable of type τ.*

1. $val_\ell(w, p(t_1, \ldots, t_n)) \;:=\; \big(val_\ell(w, p)\big)\big(val_\ell(w, t_1), \ldots, val_\ell(w, t_n)\big)$, *when p is a predicate symbol of arity n.*

2. $val_\ell(w, t\; \mathtt{instanceof\ C}) = \mathit{true} \;:\Longleftrightarrow\; val_\ell(w, t) \in \ell(C)$[16]*, when $C \in$* Typ *is a type.*

[15] This is a simpler case, as updates are deterministic and terminating.

[16] $\ell(C)$ is the set of all individual objects in the universe of interpretation ℓ that belong to the type C or a subtype.

3. $val_\ell(w, t \doteq t') = true \;:\Longleftrightarrow\; val_\ell(w,t) = val_\ell(w,t')$.

4. $val_\ell(w, \phi \wedge \psi)$ *is defined as usual. The same holds for* $\vee, \neg, \rightarrow, \leftrightarrow$.

5. $val_\ell(w, \forall x\, \phi) = true \;:\Longleftrightarrow\;$ *for each* $d \in \ell(\tau)$ $true = val_{\ell[x \mapsto d]}(w, \phi)$.

6. $val_\ell(w, \exists x\, \phi) = true \;:\Longleftrightarrow\;$ *there is* $d \in \ell(\tau)$ $true = val_{\ell[x \mapsto d]}(w, \phi)$.

7. $val_\ell(w, [\alpha]\phi) = true \;:\Longleftrightarrow\;$ *for each* $w' \in W$ *with* $w \rho_\ell(\alpha) w'$ $true = val_\ell(w', \phi)$.

8. $val_\ell(w, \langle\alpha\rangle\phi) = true \;:\Longleftrightarrow\;$ *there is* $w' \in W$ *with* $w \rho_\ell(\alpha) w'$ $true = val_\ell(w', \phi)$.

Remark 2.3.6 *Quantification is type-dependent. For variables x and y of distinct types σ and τ, the formulas $\forall x\, \phi$ and $\forall y\, \phi$ express different facts since $\ell(\tau) \neq \ell(\sigma)$, in general. The meaning of a quantification is always well-defined on the basis of the predefined type of the bound variable. To support the readability of formulas, the notation $\forall x : \tau \;\; \phi$ is short for $\forall x\, \phi$ with an external declaration of the type of x being τ.*

2.3.4 Execution of Programs

Definition 2.3.7 (Clash) *The update $f_1(s_1) \lhd t_1, \ldots, f_n(s_n) \lhd t_n$ produces a* clash *in state w of interpretation ℓ, whenever some s_i and s_j with $f_i = f_j$ happen to evaluate to the same location $val_\ell(w, s_i) = val_\ell(w, s_j)$ in the current context of valuation, but t_i and t_j do not. Otherwise, the update is called* clash-free.

As notation, the relation $s \rho_\ell(\alpha) t$ holds if in interpretation ℓ, state t can be reached from s by execution of the program α.

Definition 2.3.8 (Valuation of Programs) *Let s, t be states of an interpretation ℓ. The* valuation of programs *with respect to ℓ is defined as follows.*

1. $\rho_\ell(\alpha; \gamma) \;:=\; \{(s,t) \;:\; s\rho_\ell(\alpha)u\,,\, u\rho_\ell(\gamma)t$ *for any* $u\}$.

2. $s\rho_\ell(\mathtt{if}(\phi)\,\{\alpha\}\mathtt{else}\{\gamma\})t \;:\Longleftrightarrow$

   - $val_\ell(s, \phi) = true$ *and* $s\rho_\ell(\alpha)t$, *or*
   - $val_\ell(s, \phi) = false$ *and* $s\rho_\ell(\gamma)t$.

3. $s\rho_\ell(\mathtt{while}(\phi)\,\{\alpha\})t \;:\Longleftrightarrow$ there is $n \in \mathbf{N}$, $s = s_0, s_1, s_2, \ldots, s_n = t \in W$ *with*

- *for each* $0 \leq i < n$ $s_i \rho_\ell(\alpha) s_{i+1}$.
- *for each* $0 \leq i < n$ $val_\ell(s_i, \phi) = true$.
- $val_\ell(s_n, \phi) = false$.

4. $\rho_\ell(f_1(t_1) \lhd s_1, \ldots, f_n(t_n) \lhd s_n) := \{(s,t) : s \in W,$ $t = s[f_1(val_\ell(s,t_1)) \mapsto val_\ell(s,s_1)] \ldots [f_n(val_\ell(s,t_n)) \mapsto val_\ell(s,s_n)]\}$ *for clash-free updates. Similarly for higher arities.*

5. $\rho_\ell(\mathtt{c} \lhd \mathtt{m}(o,t))$ *is defined to be the smallest relation* R *satisfying the equation* $R = \rho_\ell(\alpha')$*, when* α *is the method body of* m *with parameter* y *and return-value parameter*[17] r*.* α' *is obtained from* α *by replacing* y *by* t *and* r *by* $\mathtt{c}$*.*[18] *Note that in the case of arbitrary recursion, the forming of the fixed-point is necessary to yield a well-defined semantics.*

Like JavaCardDL and While, ODL does not consider statements about the behaviour of non-terminating programs runs other than that they do not terminate.

ODL chooses to relate object allocation and quantification by (relativised) constant domain. This means that on the level of the logic ODL, all objects initially exist to persist throughout the interpretation of the execution. Then object allocation amounts to setting a flag for the particular new object specifying that it really has been created by a program statement execution (refer to §3.4.2 for a more circumstantial discussion). The other objects just wait passively for their activation by program activity.

In order to avoid technical subtleties ODL disallows side-effects resulting from expression evaluation[19]. Note that there is no need to specify the evaluation order of ODL expressions because of the lack of side-effects. Any order will lead to the same results in the same states. Just $\mathtt{m}(o,t)$ statements have side-effects, but are syntactically limited to occur as the only expression of a statement. Thus, in a certain way, there are no proper *side*-effects, but only one complex "major" effect per statement. See §3.4.1 for more details on this issue.

ODL did not introduce a syntactic distinction between rigid and non-rigid constants. All constant symbols are allowed to take a different value in distinct states. However, some symbols never will do so, as for example the term $2+7$ always ought to evaluate to the same number. So even though this

[17] This work does not assume a fixed formal notation for methods and formal parameters but relies on the reader's intuition, instead.

[18] For simplicity assume that the argument t is rigid for α. Otherwise consider the method body $b \lhd t; \alpha_y^b$, instead, with b being a new constant symbol.

[19] Except for return-value evaluation during direct method calls of the form $\mathtt{c} \lhd \mathtt{m}(o,t)$.

term has the function symbol $+$ as top-level symbol and is thus non-rigid, $2+7$ will never make use of this degree of freedom. Similarly, an update like $\mathtt{obj}_\mathtt{C}(5) \lhd \mathtt{obj}_\mathtt{C}(2)$ to object enumerators would destroy the bijection property of the interpretation of $\mathtt{obj}_\mathtt{C}$ postulated in Def. 2.3.1, which is why they will be forbidden.

Remark 2.3.9 ODL *is restricted in order to disallow updates to terms with top-level symbol* $0, +, \cdot, \mathtt{obj}_\mathtt{C}, \mathtt{next}_\mathtt{C}$.[20] *For this reason, those constant symbols behave rigidly, i.e. for each program* α *it is* $\vDash \forall x\, (0 \doteq x \to [\alpha]0 \doteq x)$.

2.3.5 Consequences

The next definitions introduce the standard concepts of satisfaction and consequence relation. The concept of semi-local consequence is from (Fitting & Mendelsohn, 1998).

Definition 2.3.10 (Satisfaction and Validity) *For a state* w *of an interpretation* ℓ *and a formula* $\phi \in \mathrm{Fml}(\Sigma \cup V)$ *define the* satisfaction *relation* $\vDash$ *as*

$$\begin{aligned} \ell, w \vDash \phi \quad &:\Longleftrightarrow \quad val_\ell(w, \mathrm{Cl}_\forall\, \phi) = true \\ \ell \vDash \phi \quad &:\Longleftrightarrow \quad \text{for each } w \in \mathrm{dom}(\ell)\ \ell, w \vDash \phi \end{aligned}$$

Where $\mathrm{Cl}_\forall\, \phi := \forall x_1 \ldots \forall x_n\, \phi$ *is the* universal closure *of the formula* ϕ *with the free variables* $\{x_1, \ldots, x_n\}$.

Definition 2.3.11 (Semi-Local Consequence) *The* consequence *relation* $\vDash\rhd$ *between a formula* $\chi \in \mathrm{Fml}(\Sigma \cup V)$ *and a set of local premises* $\Psi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *with a set of global premises* $\Phi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *is defined as*

$$\begin{aligned} \Phi \vDash \Psi \rhd \chi \quad :\Longleftrightarrow \quad & \text{for each interpretation } \ell \text{ with } \ell \vDash \Phi : \\ & \quad \text{for each state } w\ (\text{if } \ell, w \vDash \Psi \text{ then } \ell, w \vDash \chi) \end{aligned}$$

Where $\ell \vDash \Phi$ *means that* $\ell \vDash \phi$ *for each* $\phi \in \Phi$, *and* $\ell, w \vDash \Psi$ *means that* $\ell, w \vDash \psi$ *for each* $\psi \in \Psi$. *The* local consequence *relation* $\vDash_l$ *and the* global consequence *relation* $\vDash_g$ *are defined as*

$$\begin{aligned} \Psi \vDash_l \chi \quad &:\Longleftrightarrow \quad \vDash \Psi \rhd \chi \\ \Phi \vDash_g \chi \quad &:\Longleftrightarrow \quad \Phi \vDash\rhd \chi \end{aligned}$$

In the case of $\Phi = \emptyset$ *the notation* $\vDash_l \psi$ *shall be sufficient instead of* $\emptyset \vDash_l \psi$.

[20] $\mathtt{next}_\mathtt{C}$ will be required and therefore introduced in §3.4.2.

Remark 2.3.12 (Consequence Correspondence) *The local consequence relation is, in general, stronger than global consequence:* $\vDash_l \subseteq \vDash_g$*, i.e.*

$$\Phi \vDash_l \chi \quad \Rightarrow \quad \Phi \vDash_g \chi$$

Moreover, in the case of $\Phi = \emptyset$*, local and global consequence coincide, because of which it is possible to write* $\vDash$ *instead of* $\vDash_l$ *in such situations.*

$$\vDash_l \chi \quad \Longleftrightarrow \quad \vDash_g \chi$$

Lemma 2.3.13 (Local Deduction Theorem) *Let* $\Phi, \Psi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *and* $F, \chi \in \mathrm{Fml}(\Sigma \cup V)$*.*

$$\Phi \vDash \Psi \cup \{F\} \rhd \chi \quad \Longleftrightarrow \quad \Phi \vDash \Psi \rhd (\mathrm{Cl}_\forall F \to \chi)$$

This deduction theorem is global with respect to variables.

Proof: simple by Def. 2.3.11 ■

Lemma 2.3.14 (Global Deduction Theorem) *Let* $\Phi, \Psi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *and* $F, \chi \in \mathrm{Fml}(\Sigma \cup V)$*.*

$$\Phi \cup \{F\} \vDash \Psi \rhd \chi \quad \Longleftrightarrow \quad \Phi \vDash \Psi \cup \bigcup_{n \in \mathbf{N}} \{\Box^n F\} \rhd \chi$$
$$\textit{with } \{\Box^n F\} \quad := \quad \{[\alpha_1] \dots [\alpha_n] F \ : \ \alpha_i \in \mathrm{Prg}(\Sigma \cup V)\}$$

A proof of Lem. 2.3.14 is contained in §A.3 along with the required machinery.

The definitions 2.3.10 and 2.3.11 cause that local and global consequence relations are global with respect to variables, which leads to an implicit universal treatment of free variables. Global consequence for variables is a non-critical convenience.

Remark 2.3.15 *The stronger condition below is sufficient for* $\Phi \vDash_l \psi$ *to hold. Unlike the definition of* $\vDash_l$*, this stronger condition treats both, states* and *free variables locally.*

$$\textit{for each interpretation } \ell \textit{ for each state } w$$
$$(\textit{for each } \phi \in \Phi \ \mathit{val}_\ell(w, \phi) = \mathit{true}) \ \textit{ implies } \ \mathit{val}_\ell(w, \psi) = \mathit{true}$$

Definition 2.3.16 (Local Equivalence) *The* local equivalence *relation* $\equiv$ *between formulas* ϕ *and* ψ *is defined as*

$$\phi \equiv \psi \quad :\Longleftrightarrow \quad \textit{for each interpretation } \ell \textit{ for each state } w$$
$$(\ell, w \vDash \phi \iff \ell, w \vDash \psi)$$

2.3.6 Clash Semantics

Def. 2.3.8 only defined the semantics of clash-free parallel updates, while the meaning of an update in the presence of clashes has been left undefined until now. Now it is time for a discussion of the possible choices for clash semantics, which exhibit considerable impact on the calculus. The justification for the actual ODL choice of clash semantics alternatives will therefore have to be delayed until Def. 4.2.5, when all concepts affected by the choice of the clash semantics have been introduced. A premature judgement – as would be possible at the current stage – could not cope with the demands of an appropriate discussion.

Clashes occur in an update $\langle f(s_1) \lhd t_1, f(s_2) \lhd t_2 \rangle$ whenever s_1 and s_2 happen to evaluate to the same location in the current context (i.e. state) of valuation, but t_1 and t_2 do not. In this situation, $\langle f(s_1) \lhd t_1 \rangle$ resp. $\langle f(s_2) \lhd t_2 \rangle$ represent contradictory state updates and one has to select which modification to the interpretation of f to execute, since performing both at once is impossible.

There are at least five essentially different choices for clash semantics. *Last-win* semantics defines that the last update to a location takes precedence over earlier updates to the same location. In the example, $f(s_2) \lhd t_2$ will be performed, while $f(s_1) \lhd t_1$ takes no effect in case of a clash. Although seeming an arbitrary and odd choice from a logical perspective, last-win semantics is closest to the semantics of imperative programming languages.

Lock semantics is guided by the intuition that conflicting updates somehow seem ill-defined and should thus produce no sensible state transition. Whenever an update contains a clash, execution stops (by error) as there is no next state. More formally, in case of a clash, $\rho_\ell(f(s_1) \lhd t_1, f(s_2) \lhd t_2) = \emptyset$. This is the approach taken for ASM updates.

Skip semantics is guided by the intuition that conflicting updates somehow seem ill-defined and should thus be ignored. Whenever an update contains a clash, the whole update is discarded and no change occurs at all. More formally, $\rho_\ell(f(s_1) \lhd t_1, f(s_2) \lhd t_2) = \{(s, s) : s \in \mathrm{dom}(\ell)\}$ holds for those states that produce a clash.

Nondeterministic clash semantics resolves the update execution choice nondeterministically, i.e. whenever updates conflict because multiple values should be assigned to the same location, one of the updates is chosen in an unpredictable way. Especially this intrinsically unpredictable selection process may come to different choices in very similar formal contexts. Formally, $\rho_\ell(f(s_1) \lhd t_1, f(s_2) \lhd t_2) = \rho_\ell(f(s_1) \lhd t_1) \cup \rho_\ell(f(s_2) \lhd t_2)$.

Arbitrary clash semantics resembles nondeterministic clash semantics insofar as some choice is taken from the conflicting updates, but the choice is fixed in some arbitrary way. Thus, contrary to the nondeterministic clash semantics, arbitrary clash semantics still is deterministic. Last-win semantics is a special case of arbitrary clash semantics.

Although no choice of clash semantics is clearly superior to all others, we choose last-win semantics for reasons that will be discussed in §4.2.5 after the calculus has been introduced and all impact for the overall verification is conceivable.

2.4 Exemplary Application

The purpose of ODL is threefold. First, ODL defines an underlying programming language for object-oriented programming. Second, the logic ODL is a specification language that can be used to specify the behaviour of programs written in ODL. And third, ODL comes along with a verification calculus, which is able to prove conjectures about programs. The ultimate ODL scenario will thus begin by writing an object-oriented program in ODL, specify its behaviour in the ODL logic, and prove that the program meets its specification in the ODL calculus. As already mentioned in case 2 of §1.3 other application scenarios involve standard object-oriented programming languages like JAVA, C++ or C# and work by transformation into ODL.

Example 2.4.1 As an introduction, consider the following JAVA program fragment.

```
int round(int c) {
  if (c % 2 == 0)
      c = c + 2;
  else
      c = c + 1;
  return c;
}
```

For a concise ODL translation of the above program to ODL assume | as a built-in divisibility relation.[21] This relation is not necessary but simplifies the notation. Then the translation leads to the following ODL program α.

$$\texttt{if}(2 \mid c)\,\{c \lhd c+2\}\texttt{else}\{c \lhd c+1\}$$

[21]Divisibility is definable by adding the following axiom.

$$a \mid b \;\leftrightarrow\; \exists k\!:\!\texttt{int}\; k \cdot a \doteq b$$

For simplicity, consider the simpler variant α', first, which has already been presented in Ex. 2.1.1.

$$\mathtt{if}(2 \mid c)\,\{d \lhd c+2\}\mathtt{else}\{d \lhd c+1\}$$

For specifying the behaviour of α' in ODL, there are a number of possibilities. It is known what happens in the case that the branching condition is true. When 2 actually divides the current value of c, then after all executions of the program α' the program variable d will have the value of c plus 2. In formulas:

$$2 \mid c \rightarrow [\alpha']d \doteq c+2$$

Similarly, the following formula specifies the behaviour of program α in case that c initially is odd, though without reference to the $\mid$ condition.

$$\exists k{:}\mathtt{nat}\ \ c \doteq 2 \cdot k + 1 \quad \rightarrow \quad [\alpha]d \doteq c+1$$

This knowledge can be combined with the description of the behaviour in the case of a true branching condition to form a specification of α'.

$$(2 \mid c \rightarrow [\alpha']d \doteq c+2)\ \wedge\ (\neg(2 \mid c) \rightarrow [\alpha']d \doteq c+1)$$

The specification below directly repeats the case distinction performed in the program but in retrospect after the modality. This is only possible because α' does not modify c. For a general solution see (Platzer, 2004).

$$[\alpha']\big((2 \mid c \rightarrow d \doteq c+2)\ \wedge\ (\neg(2 \mid c) \rightarrow d \doteq c+1)\big)$$

From this a more "integrated" specification of α' can be obtained.

$$[\alpha']d \doteq (\mathit{if}\, 2 \mid c\ \mathit{then}\, c+2\ \mathit{else}\, c+1\ \mathit{fi})$$

In some scenarios, a more satisfactory solution could, however, be a more intentional specification of α': One describing the quality of the outcome rather than describing the process of computation carried out to reach it. What α' computes is a function that rounds its input up to the next greater even number. This result can be described with the following specification.

$$[\alpha'](d > c \wedge 2 \mid d \quad \wedge \quad \forall y'\,(y' > c \wedge 2 \mid y' \ \rightarrow\ y' \geq d))$$

With some standard mathematical notation, however, this specification can be refined to a far simpler version. Let $c \div d$ denote the integer division of c

by d.[22] Then the following ODL formula is a specification of α'.

$$[\alpha']d \doteq 2 \cdot (c \div 2) + 2$$

Moving back to the original ODL program α the question remains what an analogue specification could look like. As far as α is concerned, there is a problem with referents: Since α modifies its input variable c one needs some way to distinguish the value of c prior to the run of α and the value of c after the execution of the program α. For this purpose the specification remembers the prestate value of c in a (rigid) variable c_0 for later referral. This gives the following specification of α.

$$\forall c_0 \, (c_0 \doteq c \ \rightarrow \ [\alpha]c \doteq 2 \cdot (c_0 \div 2) + 2)$$

By the implicit universal quantification of free variables this is equivalent to the following specification.

$$c_0 \doteq c \ \rightarrow \ [\alpha]c \doteq 2 \cdot (c_0 \div 2) + 2$$

This formula says that – presuming a start in a state in which the prestate value of c has been remembered in the logical variable c_0 – α has the effect that after each possible (successful) execution, c will be the sum of the previous value of c rounded down to the closest even number, and of 2. What this specification does not talk about is whether the program α has any (successful) terminating runs at all. The following specification says that there is a successfully terminating run in which the postcondition holds. Yet, it does not specify the effect of any other termination cases.

$$c_0 \doteq c \ \rightarrow \ \langle\alpha\rangle c \doteq 2 \cdot (c_0 \div 2) + 2$$

In the case of deterministic programs like α there can be at most one terminating run. Therefore the formula $\phi \rightarrow [\alpha]\psi$ expresses that *if* the program terminates, the poststate reached will surely satisfy ψ provided that the initial state satisfies ϕ. The ODL formula $\phi \rightarrow \langle\alpha\rangle\psi$, on the other hand, expresses that the program really *does* terminate and the poststate reached will satisfy ψ provided that the initial state has satisfied ϕ. □

[22] This integer division takes two integers and gives the integer part of the division of c by d with fractions round down. It is of course definable in first-order logic.

$$z \doteq c \div d \ \leftrightarrow \ \exists r{:}\texttt{nat} \ (r < d \wedge c \doteq z \cdot d + r)$$

In general it is $c \div d \neq \frac{c}{d}$.

2.5 Variation

Definition 2.5.1 (Substitution) *A (uniform variable)* substitution *is a total endomorphism* $\sigma : \mathrm{Trm}(\Sigma \cup V) \to \mathrm{Trm}(\Sigma \cup V)$[23] *of finite support, i.e.*

$$\begin{array}{lrcl}
(m) & \sigma\big(f(t_1,\dots,t_n)\big) & = & f\big(\sigma t_1,\dots,\sigma t_n\big) \quad \textit{for each } f(t_1,\dots,t_n) \\
(var) & \sigma|_\Sigma & = & \mathrm{id} \\
(fin) & \sigma|_V & = & \mathrm{id} \ \textit{p.t.}^{24} \\
& \textit{i.e. } \mathrm{dom}(\sigma) & := & \{x \in \Sigma \cup V \ : \ \sigma x \neq x\} \textit{ is finite}
\end{array}$$

Substitutions are assumed to be type-safe*, i.e. they respect the type graduation: for each* $\tau \in \mathrm{T{\scriptsize YP}}$ *map* σ *has the type* $\mathrm{Trm}(\Sigma \cup V)_\tau \to \mathrm{Trm}(\Sigma \cup V)_\tau$*. The precise inductive effect of substitutions is summarised in Fig. 2.3, with a straightforward component-wise generalisation to updates of higher-arities or simultaneous updates.* $[s \mapsto t]$ *is the substitution replacing* s *by* t*, and* ϕ^t_s *denotes the result of applying* $[s \mapsto t]$ *to* ϕ*.*

$$\begin{array}{rcl}
\sigma(\mathit{if}\, e\; \mathit{then}\, s\; \mathit{else}\, t\; \mathit{fi}) & = & \mathit{if}\, \sigma e\; \mathit{then}\, \sigma s\; \mathit{else}\, \sigma t\; \mathit{fi} \\
\sigma(t\, \mathtt{instanceof}\, \mathtt{C}) & = & (\sigma t)\, \mathtt{instanceof}\, \mathtt{C} \\
\sigma(s \doteq t) & = & (\sigma s) \doteq (\sigma t) \\
\sigma(\phi \vee \psi) & = & (\sigma\phi) \vee (\sigma\psi) \\
\sigma\forall x\, \phi & = & \forall x\, \sigma\phi \\
\sigma\langle\alpha\rangle\phi & = & \langle\sigma\alpha\rangle\sigma\phi \\
\sigma[\alpha]\phi & = & [\sigma\alpha]\sigma\phi \\
\sigma(\mathtt{if}(\chi)\,\{\gamma\}\mathtt{else}\{\delta\}) & = & \mathtt{if}(\sigma\chi)\,\{\sigma\gamma\}\mathtt{else}\{\sigma\delta\} \\
\sigma(f(s) \lhd t) & = & (f(\sigma s) \lhd \sigma t) \\
\sigma(\gamma; \delta) & = & (\sigma\gamma)\ ;\ (\sigma\delta) \\
\sigma(\mathtt{while}(\chi)\,\{\gamma\}) & = & \mathtt{while}(\sigma\chi)\,\{\sigma\gamma\}
\end{array}$$

Figure 2.3: Inductive Substitution: This figure describes the effect of substitutions on terms and formulas. For simplicity assume that $x \notin \mathrm{dom}(\sigma)$ has been ensured by α-renaming of bound variables.

As an important technical device, ODL needs the concept of admissible and wary substitutions. They constitute a syntactical approximation of equivalence classes of modality levels and are based on standard first-order notions of "compatible" substitutions.

[23]Respectively $\sigma : \mathrm{Fml}(\Sigma \cup V) \to \mathrm{Fml}(\Sigma \cup V)$ for substitutions on formulas. Formulas are considered as a special case of terms, here. Then, of course, the condition for the function symbol f generalises to predicates and operators.

[24]i.e. $\sigma|_V = \mathrm{id}$ holds except for a finite number of variables.

Definition 2.5.2 (Admissible)

1. *A substitution σ is* first-order admissible *or* free of collisions *for a formula ϕ, if no free variable x occurs within the scope of a quantifier binding a variable of σx.*

2. *A substitution $[s \mapsto t]$ is* admissible *(or denotation-preserving) for ϕ, if it is first-order admissible and, during the process of substituting s by t in ϕ for the formation of ϕ_s^t, neither s nor t trespass modalities for which they are not* rigid, *i.e. no s occurs in the scope of a modality updating a constant symbol of s or t.*

Example 2.5.1 For the following formula ϕ,

$$x \doteq c \;\rightarrow\; \langle c \lhd c+1 \rangle (c \geq x+1)$$

the substitution $[x \mapsto c]$, which replaces all occurrences of x by c, is not admissible. This is due to the fact that for the forming of ϕ_x^c as

$$c \doteq c \;\rightarrow\; \langle c \lhd c+1 \rangle (c \geq c+1)$$

the inductive substitution process trespasses the modality $\langle c \lhd c+1 \rangle$ for which c is not rigid. Hence, within the scope of the modality, constant symbol c suddenly denotes a different value than outside the modality, thereby destroying the property of the occurrences of x, or – after the substitution – c, to share the same value throughout the formula. Instead, a substitution of x by $d+1$ in ϕ to form ϕ_x^{d+1} is admissible for different constant symbols d. □

Remark 2.5.3 *In this thesis all substitutions are assumed to be admissible. Name clashes can always be resolved by α-renaming bound variables, which is assumed to happen implicitly.*

Definition 2.5.4 (Wariness) *The* wary substitution $\widehat{[s \mapsto t]}$ *corresponding to a first-order admissible substitution $[s \mapsto t]$ works like $[s \mapsto t]$ but discontinues the substitution process in front of modalities for which s or t are not rigid*[25]*. Furthermore, by definition, substitutions never replace the top-level symbol of the left hand side in an update.*

Similarly, $\widehat{\sigma}$ is the wary[26] substitution corresponding to a (non-singleton) substitution σ.

[25]Of course, here, this only amounts to a syntactic criterion for rigidity like no update to a non-rigid function symbol of t occurs within the modality.

[26]The notation $\widehat{\sigma}$ comes from the German word for wary, "*behutsam*", of which an etymological root is "*Hut*", which also means "hat" and is a colloquial name for the sign "$\frown$".

Example 2.5.2 For the formula ϕ from Ex. 2.5.1, applying the wary substitution $\widehat{[x \mapsto c]}$ corresponding to the substitution $[x \mapsto c]$, which has turned out to be not denotation-preserving, leads to the following admissible replacement

$$c \doteq c \;\; \rightarrow \;\; \langle c \lhd c+1 \rangle (c \geq x+1)$$

$\square$

The concept of wariness is similar to the concept of admissibility in first-order logic, which says that a substitution σ is admissible for a formula ϕ, if no free variable x of ϕ occurs within the scope of a quantifier binding a variable of σx. Both notions of admissible and wary substitutions strive to ensure that identical symbols still denote the same values after the substitution, provided they did so before. For proving such a behaviour formally, the introduction of a schematic symbol is convenient that combines the treatment of multiple syntactic entities into one notation. Occasionally, this schematic symbol will still be of use during the next chapters as well.

Remark 2.5.5 *The schematic symbol Υ matches all formula and term constructor symbols except quantifiers, modalities and program constructors. Υ also concerns logical constant symbols like $+$, if then else fi, $\wedge$. Likewise, in this context, u is a generalised formal parameter and may as well represent an n-tuple of arguments for $0 \leq n \in \mathbf{N}$.*

Lemma 2.5.6 (Substitution Lemma) *Let σ be an admissible substitution*[27] *on a term (or formula) t, then the substitution principle holds, i.e.*

$$\text{for each interpretation } \ell \text{ for each state } w \;\; val_\ell(w, \sigma t) \;\; = \;\; val_{\sigma^*\ell}(w, t)$$

where $\sigma^*\ell \;:=\; \ell[\mathbf{x} \mapsto \mathrm{val}_\ell(\sigma\mathbf{x})]$ is the semantic modification[28] *(of ℓ)* adjoint *to σ.*

Proof: The substitution σ is a homomorphic[29] continuation of $\sigma|_{\Sigma \cup V}$ to $\mathrm{Fml}(\Sigma \cup V)$. Likewise the valuation ℓ is a homomorphic continuation of the interpretation $\ell|_{\Sigma \cup V}$ to $\mathrm{Fml}(\Sigma \cup V)$. Assume $\sigma = [u \mapsto \sigma(u)]$ is a singleton substitution. Since most parts of this proof only refer to state w, write – by an abuse of notation – $\mathrm{val}_\ell(t)$ or even $\ell(t)$ instead of $val_\ell(w, t)$, whenever appropriate. Note that this abbreviated notation is incompatible with the original notation from Def. 2.3.1 and will only be used in this proof.

[27] Here the substitution σ may simultaneously replace any number of logical variables or – in case of wariness – program variables. There are generalisations of what a substitution can replace for which the substitution lemma still holds.

[28] This semantic modification $\ell[\mathbf{x} \mapsto \mathrm{val}_\ell(\sigma(\mathbf{x}))]$ is meant for all variables $\mathbf{x} \in V$.

[29] In this proof this means: except for quantifiers and modalities.

$$
\begin{tikzcd}
t \arrow[d, mapsto, "\sigma"'] \arrow[dd, bend left=60, mapsto] \\
\sigma t \arrow[d, mapsto, "\ell"'] \quad \sigma^*\ell \\
val_\ell(w,\sigma t) = val_{\sigma^*\ell}(w,t)
\end{tikzcd}
$$

Figure 2.4: Syntactic Substitution vs. Semantic Modification

IA If $t = z \in \Sigma \cup V$ is atomic then there are two cases to consider.

$$
\begin{array}{rl}
\text{I} & z \in \mathrm{dom}(\sigma) \;\Rightarrow\; \ell(\sigma z) = \ell[z \mapsto \mathrm{val}_\ell(\sigma z)](z) = \sigma^*\ell(z) \\
\text{II} & z \notin \mathrm{dom}(\sigma) \;\Rightarrow\; \ell(\sigma z) = \ell(z) = \sigma^*\ell(z)
\end{array}
$$

IS Because σ and ℓ are homomorphic, the conjecture follows as a homomorphic continuation of case IA. More explicitly, consider the case $t = \varUpsilon(u)$ for induction. Then

$$
\begin{array}{rcl}
\ell(\sigma t) & = & \ell\Big(\sigma\big(\varUpsilon(u)\big)\Big) \\
 & \overset{\sigma\,\text{hom.}}{=} & \ell\big((\sigma\varUpsilon)(\sigma u)\big) \\
 & \overset{\ell\,\text{hom.}}{=} & \ell(\sigma\varUpsilon)\big(\ell(\sigma u)\big) \\
 & \overset{\text{IH}}{=} & \sigma^*\ell(\varUpsilon)\big(\sigma^*\ell(u)\big) \\
 & \overset{\sigma^*(\ell)\,\text{hom.}}{=} & \sigma^*\ell\big(\varUpsilon(u)\big) \\
 & = & \sigma^*\ell(t)
\end{array}
$$

$\exists\forall$ There is a similar argument for the case $t = \exists y\,\phi$ of quantifiers. This case makes use of the premise that σ is admissible, which results in y not occurring in any replacements made by σ, especially $y \notin \sigma\big((\Sigma\cup V)\setminus\{y\}\big)$, at least for the relevant variables that actually occur in ϕ. This fact will be denoted more symbolically as $y \notin \sigma\mathbf{x}$. After α-conversion it can further be assumed that $BV(t) \cap \mathrm{dom}(\sigma) = \emptyset$. Then

$$
\begin{array}{rcl}
\ell(\sigma t) & = & \ell\Big(\sigma\big(\exists y\,\phi\big)\Big) \\
 & \overset{\sigma\,\text{hom.}}{=} & \ell\big(\exists y\,\sigma\phi\big) \\
 & \overset{\ell\,\text{“hom.”}}{\iff} & \text{there is } d\ \ell[y \mapsto d](\sigma\phi) \\
 & \overset{\text{IH}}{\iff} & \text{there is } d\ \sigma^*\big(\ell[y \mapsto d]\big)(\phi)
\end{array}
$$

$$\begin{aligned}
&\iff && \text{there is } d\ \ell[y \mapsto d][\mathbf{x} \mapsto \ \mathrm{val}_{\ell[y\mapsto d]}(\sigma\mathbf{x})](\phi) \\
&\iff^{30} && \text{there is } d\ \ell[\mathbf{x} \mapsto \ \mathrm{val}_{\ell[y\mapsto d]}(\sigma\mathbf{x})](\phi) \\
&\overset{\sigma y = y}{\iff} && \text{there is } d\ \ell[\mathbf{x} \mapsto \ \mathrm{val}_{\ell[y\mapsto d]}(\sigma\mathbf{x})][y \mapsto d](\phi) \\
&\overset{y \notin \sigma\mathbf{x}, \text{adm.}}{\iff} && \text{there is } d\ \ell[\mathbf{x} \mapsto \mathrm{val}_{\ell}(\sigma\mathbf{x})][y \mapsto d](\phi) \\
&\iff && \text{there is } d\ (\sigma^*\ell)[y \mapsto d](\phi) \\
&\overset{\sigma^*(\ell) \text{ “hom.”}}{\iff} && \sigma^*\ell\big(\exists y\, \phi\big) \\
&= && \sigma^*\ell(t)
\end{aligned}$$

(I) If $t = \phi = [\alpha]\psi$ then it remains to show the case with $u \in FV(\phi)$. σu is rigid for α by premise $\Rightarrow$

$$\begin{aligned}
& && \mathrm{val}_\ell(s, \sigma\phi) = \mathit{true} \\
&\iff && \text{for each } \ s\rho_\ell(\alpha)s' \ \mathit{true} = \mathrm{val}_\ell(s', \sigma\psi) \overset{\text{IH}}{=} \mathrm{val}_{\ell[u \mapsto \mathrm{val}_\ell(s', \sigma u)]}(s', \psi) \\
&\overset{\text{rigid}}{\iff} && \text{for each } \ s\rho_\ell(\alpha)s' \ \mathit{true} = \mathrm{val}_{\sigma^*\ell}(s', \psi) \\
&\overset{\text{s.b.}}{\iff} && \text{for each } \ s\rho_{\sigma^*\ell}(\alpha)s' \ \mathit{true} = \mathrm{val}_{\sigma^*\ell}(s', \psi) \\
&\iff && \mathrm{val}_{\sigma^*\ell}(s, \phi) = \mathit{true}
\end{aligned}$$

It still remains to show that $\rho_\ell(\sigma\alpha) = \rho_{\sigma^*\ell}(\alpha)$.

(a) If α is of the form $f(s) \lhd t$, then

$$\begin{aligned}
& && w\rho_\ell(\sigma\alpha)w' \\
&\iff && w' = w[f(\mathrm{val}_\ell(\sigma s)) \mapsto \mathrm{val}_\ell(\sigma t)] \\
&\overset{\text{IH}}{\iff} && w' = w[f(\mathrm{val}_{\sigma^*\ell}(s)) \mapsto \mathrm{val}_{\sigma^*\ell}(t)] \\
&\iff && w\rho_{\sigma^*\ell}(\alpha)w'
\end{aligned}$$

(b) $\alpha = \gamma; \delta$, then

$$\begin{aligned}
& && w\rho_\ell(\sigma\alpha)w' \\
&\iff && \text{there is } \ w'' \ w\rho_\ell(\sigma\gamma)w'' \, , \, w''\rho_\ell(\sigma\delta)w' \\
&\overset{\text{IH}}{\iff} && \text{there is } \ w'' \ w\rho_{\sigma^*\ell}(\gamma)w'' \, , \, w''\rho_{\sigma^*\ell}(\delta)w' \\
&\iff && w\rho_{\sigma^*\ell}(\alpha)w'
\end{aligned}$$

[30]Since y also occurs as one of the generic names $\mathbf{x}$, the first modification of y is void because it will be overwritten by the $\ell[\mathbf{x} \mapsto \dots]$ operation. y will still be modified to d.

(c) $\alpha = \texttt{if}(\chi)\,\{\gamma\}\texttt{else}\{\delta\}$, then consider the case that $\mathrm{val}_\ell(\sigma\chi) = \mathit{true}$, which – by induction hypothesis – is equivalent to $\mathrm{val}_{\sigma^*\ell}(\chi) = \mathit{true}$. Then

$$\begin{aligned} & \rho_\ell(\sigma\alpha) \\ = \; & \rho_\ell(\sigma\gamma) \\ \overset{\text{IH}}{=} \; & \rho_{\sigma^*\ell}(\gamma) \\ = \; & \rho_{\sigma^*\ell}(\alpha) \end{aligned}$$

The same argument holds for the case of $\mathrm{val}_\ell(\sigma\chi) = \mathit{false}$ or $\mathrm{val}_{\sigma^*\ell}(\chi) = \mathit{false}$, respectively.

(d) $\alpha = \texttt{while}(\chi)\,\{\gamma\}$ is similar.

■

In order to extend the variable substitution lemma to more general substitutions allowing to replace constants or general terms, Lem. 2.5.6 is used for mediation. Two formulas that emanate from each other by replacement of terms are also related by an intermediate formula involving a new variable instead of the occurrences of the affected terms. From the variable version of the formula, each variant can be reached by ordinary substitution, thereby relating all three variants by a substitution principle.

Proposition 2.5.7 (Generalised Substitution Lemma) *Let σ be an admissible constant*[31] *or variable substitution on a term (or formula) t, then the substitution principle holds, i.e.*

$$\textit{for each interpretation } \ell \textit{ for each state } w \; \mathit{val}_\ell(w, \sigma t) \;=\; \mathit{val}_{\sigma^*\ell}(\sigma^* w, t)$$

where $\sigma^*\ell := \ell[\mathbf{x} \mapsto \mathrm{val}_\ell(\sigma\mathbf{x})]$ and $\sigma^* w := w[\mathbf{x} \mapsto \mathrm{val}_w(\sigma\mathbf{x})]$ are the semantic modifications[32] *adjoint to σ.*

Proof: Assume $\sigma = [c \mapsto s]$ with a constant symbol c and term s. The conjecture is proven by mediation. For this proof, let t_0 denote the variant of t obtained by replacing all occurrences of c by the new variable z. From t_0 the term t can be reached just as well as σt by ordinary variable substitution:

$$\begin{aligned} t &= {t_0}_z^c \\ \sigma t &= t_c^s \\ &= ({t_0}_z^c)[c \mapsto s] \\ &= {t_0}_z^s \end{aligned}$$

[31] i.e. a variable substitution that is further allowed to substitute constant symbols by other terms.

[32] The semantic modifications are meant for all variables $\mathbf{x} \in \Sigma$, respectively $\mathbf{x} \in V$.

Then the valuation of the mediator t_0 relates the valuation of t and σt. Abbreviate $val_\ell(w,s)$ by d, and $val_\ell(w[c \mapsto d],c)$ by e.

$$
\begin{array}{rcl}
val_\ell(w[c \mapsto d], t) & = & val_\ell(w[c \mapsto d], {t_0}^c_z) \\
& \stackrel{2.5.6}{=} & val_{\ell[z\mapsto e]}(w[c \mapsto d], t_0) \\
& = & val_{\ell[z\mapsto d]}(w[c \mapsto d], t_0) \\
& \stackrel{c\notin t_0}{=} & val_{\ell[z\mapsto d]}(w, t_0) \qquad (2.1) \\
& \stackrel{2.5.6}{=} & val_\ell(w, {t_0}^s_z) \\
& = & val_\ell(w, \sigma t)
\end{array}
$$

This concludes the proof. Nevertheless, for possible generalisations let us closely examine the properties of c that have been used during this proof. In case of a compound expression $c = f(u)$ a semantic modification $w[c \mapsto d]$ of c to some value d simply has to be rewritten to $w[f(val_\ell(w,u)) \mapsto d]$ according to Def. 2.3.3. What is important, though, is that $c \notin t_0$ does no longer justify (2.1) in case of compound c (refer to Ex. 2.5.3). The condition under which the above proof generalises to other c reads as follows.

for each interpretation ℓ for each state w

$$val_\ell(w[c \mapsto d], t_0) \quad = \quad val_\ell(w, t_0) \qquad (2.2)$$

with d abbreviating $val_\ell(w,s)$. The abstraction of (2.2) in comparison to (2.1) is the change in notation for the interpretation ℓ, which is only legitimate in this uniform way because $val_\ell(w,s)$ does not depend on the particular interpretation of the new variable z. ■

Example 2.5.3 Let us investigate the futile attempt to receive a similar substitution lemma for general terms. Consider a compound term substitution $[f(s) \mapsto 0]$ on $f(u) > 0 \land f(s) \doteq 0$. Suppose there is some state w that contingently satisfies $\ell, w \vDash s \doteq u$ with d being the value $val_\ell(w,s)$. Then there is the following discrepancy.

$$
\begin{array}{rl}
& val_\ell(w, (f(u) > 0 \land f(s) \doteq 0)^0_{f(s)}) \\
= & val_\ell(w, f(u) > 0 \land 0 \doteq 0) \\
\neq & val_{\ell[f(d)\mapsto 0]}(w, 0 > 0 \land 0 \doteq 0) \\
= & val_{\ell[f(d)\mapsto 0]}(w, f(u) > 0 \land f(s) \doteq 0)
\end{array}
$$

□

Chapter 3

Transformation

3.1 Overview

This section shows that a translation from a dynamic logic of an object-oriented programming language to ODL is possible. As a representative example for the object-oriented source language we choose JavaCardDL (Beckert, 2000; Ahrendt *et al.*, 2004; JavaCard, 2004). The translation will take place on several layers. On the type layer, a representation of class, field and method types has to be found within ODL, which contains all information necessary for later type dependent statements. On the code layer, a single block of source code in JavaCardDL has to be translated into a piece of program in ODL, which has the same effect. Thus, the translation has to preserve the semantics, and – in order to avoid unnecessary encoding – also preserve structure as much as possible. During the discussion of this section, it will be investigated why ODL has its specific structure, what are intrinsic qualities and what contingent choices.

3.2 Object-Oriented Features

Object-oriented programming languages provide a rich set of features or qualities. This section briefly examines which features are non-essential to object-orientation from a logical perspective. The next sections will then deal with those features that demand more attention in their translation in more detail.

Amongst the most promoted features of object-oriented programming, we find aspects of pure software-engineering effect. As such, the coupling of state and behaviour, encapsulation and information hiding are very important on the one hand. But on the other hand, they are only very soft characteristics of mere architectural importance, and their presence imposes no semantical effect. Except perhaps for a minor subtlety with information hiding, which is strongly related to visibility constraints: accessibility modifiers affect the scope of symbols, and thereby the denotational meaning of symbols in some context. Modifiers like `public`/`protected` account to static information during the compilation, only. Therefore they can safely be ignored after a successful compiler run, in which the semantical name analysis phase can impose unique naming conventions on all variables, such that variables for physically distinct entities do not share the same name anymore. Then scoping and visibility concerns have already been resolved and do not require any further consideration during the semantics and inference of the logic. Thus, their removal is a mere simplification of which the validity can be assured by simple compiler technology, but not a crucial one. For these reasons, ODL can ignore the three software-engineering qualities: coupling of state and behaviour, encapsulation and information hiding.

Next, there are more contingent features of object-oriented programming languages. Inner classes are a non-essential recent addition to programming languages like JAVA and C#. They are not yet supported by C++ or SIMULA. As can be anticipated from the recency of this change, inner classes do not account for an essential characteristic of object-orientation. Rather they provide implementational and scoping simplifications. Further there is a very straightforward translation flattening the class containment hierarchy such that inner classes are reduced to ordinary outer classes with explicit associations. Likewise, field overriding or hiding and shadowing is non-essential, and several coding conventions even discourage the use of field overwriting in favour of adequate accessor methods.

The UML standard (Rumbaugh *et al.*, 1998; Rumbaugh *et al.*, 1999) promotes associations as a central modelling concept of object-orientation. Yet, even most contemporary object-oriented programming languages do not have dedicated language constructs for associations but rely on a representation with ordinary attributes within the implementation (Balzert, 2001). Thus, ODL can employ the same representation and does not need to take special care about associations. Although a few languages like C# have built-in language constructs for events, most programming languages like JAVA do not, from which can be concluded that they do not constitute a central aspect of object-orientation. Indeed, rather simple syntactic transformations reduce dedicated events to standard programming language constructs. Most features thus have been unveiled as mere syntactic sugar covering a simple code translation.

Side-effects would impose strict evaluation order constraints and several other complications on the calculus, which is why we want to avoid side-effects within ODL for the sake of simplicity. Translating an expression evaluation involving side-effects to a sequence of statements that are free of side-effects is possible and straightforward anyway, as the KeY inference rules demonstrate.

Exceptions and other reasons for abrupt completion, like loop break, continue or intermediate return statements, complicate almost all inference rules dealing with program blocks or expression evaluation. Therefore, in the spirit of simplicity ODL prefers to abstain from built-in exception handling. Even more so as the presence of exceptions would further necessitate a treatment of partiality, i.e. some expression evaluations could fail[1] and still have to be completed regularly in some meaningful but exceptional way. Such aspects of partial logic usually lead to more special cases, checks and case distinctions within the semantics and calculus.

As will be seen in the next sections, there are some more essential features of object-orientation, which can nevertheless be emulated very naturally within the logic ODL without adding additional syntactic features to the language. Object creation and ad-hoc polymorphism in the form of dynamic dispatch will turn out to belong to this class. Further, in contrast to the type inheritance that spans the subtype hierarchy, implementation inheritance is a mere coding productivity device and can be emulated by delegation or by source code replication in a rather simple way. At the expense of additional invocation indirections, delegation is simpler than source code copying because it avoids renaming and scoping subtleties. Copying is also less convenient because it leads to a tremendous amount of source code duplication during the preprocessing transformation. Thus, all those features have been unveiled – at least from a logical perspective – to reduce to bare syntactic sugar.

Amongst those features that ODL has to keep in order to maintain a proper object-oriented feeling, field access and subtyping will be found. Although the virtues of field access are hidden in ordinary function symbols, they constitute a tremendously important aspect to (imperative) object-orientation: function symbols of a modifiable interpretation in order to permit write access to the state of an object. In conjunction with dynamic type information, subtyping is essential to retain the flavour of "object = state + behaviour", since dynamic typing permits to emulate ad-hoc polymorphism. In much the same respect in which field access provides objects with a unique state, dynamic typing provides a unique behaviour for each object. Furthermore, the fact that objects can be distinguished even in case they contingently happen to share the same values for all their fields ensures that

[1]For example due to a NullPointerException raised because of an intermediate evaluation of $t.a$ in a context where t happens to be `null`.

objects have a unique identity.

3.3 Type Transformation

The relevant type information that is subject to translation essentially consists of the class hierarchy. For this purpose, all that classes reduce to is their subtype relation, and the type signatures of all their methods and fields including information about overwriting. Furthermore, for reasons of a simpler presentation assume field overwriting has been resolved via method overwriting of corresponding accessor methods. Field hiding is assumed to be resolved by renaming as well. This is not a strict requirement, but as practical applications barely use field overwriting without accessor methods anyway, the technical details of a finer treatment do not legitimate the effort.

The type translation from a source language like JavaCardDL to ODL is immediate. The class hierarchy in JavaCardDL directly carries over to the subtype relation of ODL with the lattice completion of Ex. 2.2.1. A member field $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ of class ζ is represented as a non-rigid function $f : \zeta \times \sigma_1 \times \ldots \times \sigma_n \to \tau$, which stores at position $(o, a_1, \ldots, a_n)$ the value that the field f of object o has at position $(a_1, \ldots, a_n)$. Therefore, instead of the familiar object access operator in a JavaCardDL term like $t.a$, the notational variant $a(t)$ is used. Likewise instead of an array access $t.a[i]$, the term $a(t, i)$ is used , resp. $b(s, i, j)$ instead of $s.b[i][j]$. A method $m : \sigma_1 \times \ldots \times \sigma_n \to \tau$ of class ζ is represented as a procedure $m : \zeta \times \sigma_1 \times \ldots \times \sigma_n \to \tau$, which will be called in ODL with arguments $(o, a_1, \ldots, a_n)$ in case of an invocation of m on object o with arguments $(a_1, \ldots, a_n)$ in JavaCardDL like $o.m(a_1, \ldots, a_n)$.

Although the most apparent change of object-oriented programming in comparison to classical imperative programming – the object access operator "dot", particularly in conjunction with method calls – disappears in ODL, this circumstance just has to be considered as a notational variation favoured in the spirit of a maximum simplicity of concepts. Admittedly, keeping the standard dot access operator would maintain a more immediate object-oriented flavour from a naïve position. But with this syntactic device only disguising its root in ordinary function application, the classical function notation is preferred for simplicity. Within practical examples comparable to Java, the more convenient notation $o.x$ is often used instead of the more uncommon $x(o)$, though.

What still needs to be resolved in case of the method call is the virtual dispatch according to the run-time type information[2], which will be dealt

[2]Usually this involves only run-time type information about the invocation object o,

with in §3.4.4 about code transformation.

3.4 Code Transformation

This section takes care of the translation of statements and pieces of JavaCardDL program code.

Apart from resolution of side-effects during expression evaluation, which has to be performed throughout the translation, while-loops and if-statements remain unchanged[3]. Assignments will be translated directly into singleton updates.

The features to be taken care of include side-effects during expression evaluation, order of operand evaluation, object creation, dynamic dispatch of method calls, built-in operators, abrupt completion due to jumps or exceptions, exception handling and raising.

Despite their practical relevance, aspects of multi-threading and concurrency (Apt & Olderog, 1997) or machine-size floating-point arithmetic, pointer-arithmetic, as well as meta-programming like reflection or dynamic class loading will not be considered in this thesis.

3.4.1 Side-Effects & Evaluation Order

Example 3.4.1 The order of operand evaluation can be significant in the presence of side-effects during expression evaluation. For this purpose consider the following program.

```
public class Weird {
  private int x = 7;
  private int incx() {
    x = x+1;
    return x;
  }

  private int decx() {
    x = x-1;
    return x;
  }
```

but some programming languages provide multi-dimensional virtual dispatch.

[3] with the exception of standard normalisation techniques for getting rid of tail check loops ⌜**do** S **while**(a)⌝ and iteration loops ⌜**for** (i=1; i<10; i++)⌝. They can be found in any standard textbook about compiler construction.

```
    public int result() {
      // equivalent to r = (++x) * (-\,-x);
      int r = incx() * decx();
      return r;
    }
  }
```

After executing ⌜**new** Weird().result()⌝, the resulting value depends on the order of *operand* evaluation, not just the *operator* evaluation order. In case of left-to-right evaluation `r` will evaluate to 56, in case of right-to-left evaluation `r` is 42. □

Although the semantics of side-effects during expression evaluation ought not have much practical impact due to the massive readability limitations of substantially side-effecting programs, the transformation has to present a solution for programs that rely on side-effects, nevertheless.

Side-effects would constitute a serious inconvenience for the calculus[4], for during application of the inference rules, all expression and statement evaluations have to be performed in the fixed sequential order imposed by the evaluation order semantics. As the example above shows, the calculus could contradict language semantics otherwise. Especially troublesome is this situation for the programming language C++, which leaves order of evaluation mainly undefined. Verification therefore has to consider all possible orders of evaluation for correctness proofs.[5]

Whatever choice ODL takes for the order of operand evaluation, any source language conventions of operand evaluation order can be realised. Therefore, ODL could safely fix strict left-to-right operand evaluation order.

Example 3.4.2 Assume JavaCardDL uses right-to-left operand evaluation order, but ODL left-to-right. Then translate a JavaCardDL program like

```
r = m(x) * g(x);
```

into the following ODL program

```
int t2 = g(x);
int t1 = m(x);
r = t1 * t2;
```

[4] Side-effects are further inconvenient for language semantics.

[5] Although this tremendous amount can be reduced by structural composition of "bivalent" proofs about the insensitivity of evaluation-orders, which means that whenever it can be proven about the immediate operands of each operator that the result is irrespective of the left-to-right or right-to-left evaluation-order, the value of the whole term is independent from evaluation-order.

Because of the sequential composition operator having a left-to-right evaluation order in ODL, the above program possesses the intended order of side-effect.[6] Obviously, such a translation is necessary in the presence of side-effects. □

What the JavaCardDL to ODL translation has to achieve, is to dispose of all side-effects in an order respecting the evaluation order constraints of the source language. As the discussion above indicates, this is possible. In particular, the translation from Ex. 3.4.2 already achieves this effect as a byproduct. Thereafter, the particular choice of evaluation order for ODL does not matter anymore.

3.4.2 Object Creation

Object creation has to ensure three things. First, dynamic type checks have to be possible on the newly created objects, which implies that the dynamic[7] typing information has to be stored somehow and in a fashion compatible with the dynamic type-check `instanceof` rules. Second, object identity has to be established, i.e. two objects created by two distinct invocations of `new` have to be understood as different objects. And third, object creation should maintain the current extension of a class. This means maintenance of an (implicit) set of the objects created so far by program statement execution, in order to support varying domain semantics by relativising quantification in the ODL constant domain setting. For these purposes, new objects are represented as terms of the form $\mathtt{obj}_{\mathtt{C}}(n)$. Keeping the dynamic type information `C` in this term solves the first problem. The second problem is solved by increasing the object identifier n for each[8] invocation of `new`. The sequential linear order of $\mathbf{N}$ solves the third problem.

Object creation can be treated, for example, using the following translation scheme. An occurrence of $\langle c \lhd \mathtt{new}\,\mathtt{C}() \rangle$ in JavaCardDL is translated to

[6]Notice the difference between operator evaluation order (which is rather crucial) and operand evaluation order (which can be arranged arbitrarily with the above construction).

[7]Despite the static typing of ODL, dynamic type information is necessary because the actual dynamic type of a constant symbol c declared of type `C` can be any subtype $\mathtt{D} \leq \mathtt{C}$ as well. Furthermore, this dynamic type can depend on the previous program statements in an undecidable way.

[8]This would constitute a reason for a global `next` counter. A global counter would ensure object identity throughout all classes without having to deal with questions of class equality in the calculus. However, global counters unnecessarily complicate the maintenance of the current extension of a particular class `C`. Because of which we prefer a whole variety of counters that are local to their class `C`.

the ODL update

$$c \lhd \mathtt{obj}_{\mathtt{C}}(\mathtt{next}_{\mathtt{C}}),$$
$$\mathtt{next}_{\mathtt{C}} \lhd \mathtt{next}_{\mathtt{C}} + 1$$

With this translation, a dynamic type check in a term like $t\,\mathtt{instanceof}\,\mathtt{C}$ is possible according to the dynamic type information $\mathtt{C}$ stored in all newly created objects $\mathtt{obj}_{\mathtt{C}}(n)$.[9]

The linear increment of $\mathtt{next}_{\mathtt{C}}$ provides the proper existence information for relativising quantification to varying domain semantics in the ODL constant domain setting. A quantification like $\forall x\!:\!C\ \phi$ in varying domain semantics[10] corresponds to the following relativised quantification in the ODL constant domain semantics.

$$\forall n\,(n < \mathtt{next}_{\mathtt{C}} \to \phi_x^{\mathtt{obj}_{\mathtt{C}}(n)}) \tag{3.1}$$

The most important property to establish is that newly created objects are never equal to other objects. This object identity will be maintained by incrementing the $\mathtt{next}_{\mathtt{C}}$ counter. Since the object enumerator $\mathtt{obj}_{\mathtt{C}}$ is injective and guaranteed to produce results from disjoint sets of objects (Def. 2.3.1), this counter increase is sufficient to produce distinct objects at each creation.

Hidden in this treatment of object identity lies one subtlety, though. Modalities in different formulas of the same sequent will come up with the same object identifiers for different incarnations of $\mathtt{new}\,\mathtt{C}()$. Therefore, one is likely to suspect conflicts resulting from naming different entities from different origins identically. In the following example, the same $\mathtt{obj}_{\mathtt{C}}(1)$ term results from two different invocations of $\mathtt{new}\,\mathtt{C}()$, about which one has to make sure that object identity is not endangered. The sequent calculus notation will be introduced formally in Chapt. 4.

$$\frac{a(\mathtt{obj}_{\mathtt{C}}(1)) \doteq 0 \land z \doteq \mathtt{obj}_{\mathtt{C}}(1) \ \vdash a(\mathtt{obj}_{\mathtt{C}}(1)) \doteq 0 \land z \doteq \mathtt{obj}_{\mathtt{C}}(1)}{\langle c \lhd \mathtt{new}\,\mathtt{C}()\rangle(a(c) \doteq 0 \land z \doteq c) \ \vdash \langle d \lhd \mathtt{new}\,\mathtt{C}()\rangle(a(d) \doteq 0 \land z \doteq d)}$$

Yet, this actually corresponds to the semantics. Reasoning about two propositionally connected different formulas like in $\langle\alpha\rangle\phi \lor \langle\gamma\rangle\psi$ amounts to assessing two *a priori* independent ways of how program states could evolve from the current state, either according to the algorithm α or according to the program γ. On some particular level of modality, each occurrence of a modality represents a different trace of how the state could evolve, with all underlying programs started in the same initial state. In the example above, both

[9] There will be a discussion with some more clarifications on this topic when the calculus has been introduced in §4.2.4.

[10] i.e. with quantification restricted to range over the set of objects that really have already been created with an explicit invocation of `new`.

programs should even behave equally and thus also come up with the same results, for both are equivalent apart from variable renaming. With different object identifiers for the two different occurrences of a program in the sequent, the proof would not succeed as it should.

More technically speaking, two independent[11] invocations of two programs on the same machine state will (usually) use the same locations in memory for their next allocation.

In comparison to alternative approaches of dealing with object creation, the $\mathtt{obj}_\mathtt{C}$ and $\mathtt{next}_\mathtt{C}$ representation has some advantages. The formula (3.1) for relativising possibilist quantification (Fitting & Mendelsohn, 1998) in constant domain to actualist quantification in varying domain semantics is mere first-order and does not introduce proper dynamic-logic aspects. For example, explicit loop traversal in dynamic-logic would be unavoidable to reproduce the current extension in a list-based representation. By the nature of the counter approach, it is further provable that at all states of program execution there will always be only a finite number of currently created objects, never an infinite amount. Those created objects will also be kept consecutively in memory.

3.4.3 Garbage Collection

In the presence of a garbage-collection mechanism the simple ODL treatment of object creation is no longer a direct model of the real machine behaviour. The problem with garbage collection is that JAVA object allocation will no longer return $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C})$ with $\mathtt{next}_\mathtt{C}$ increasing linearly. Instead by garbage collection and recycling the memory of an old object, say $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$ allocation of a new object may result in the very same memory location respectively $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$. Then the ODL linear increment model is not always faithful. However, the defining conditions of sufficiently wise garbage collection imply that by the time object allocation has revived $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$ for its second use, no other (life) object can have a reference to the old $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$. Thus, traversing the object reference graph from any life object will never lead to $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$. From this perspective, as far as the behaviour is concerned that can be observed by execution of program statements, both $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C})$ and $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C}-17)$ are indistinguishable by any other properties than equality.[12] The fact that $\mathtt{obj}_\mathtt{C}(\mathtt{next}_\mathtt{C})$

[11] Think of this as two programs run on two separate machines, which share the same identical initial memory state.

[12] At least when assuming the common normalisation that object allocation is always followed by a period of clearing all attributes of the newly allocated object to 0 in order to abstract program behaviour from the old memory contents. Even though we have not

and $\texttt{obj}_{\texttt{C}}(\texttt{next}_{\texttt{C}} - 17)$ cannot be distinguished by execution of ODL program statements is also due to the prohibition of reference comparison and pointer arithmetics like in C++.

This meta-reasoning argument shows that – due to the absence of pointer arithmetics – program behaviour cannot depend on the particular memory location returned by the memory allocator, because of which any such object will do. This includes cleared old objects that have been revived by garbage collection, because they cannot have distinguishing object references from outside when following any references of life objects. Therefore simply assuming that the ordinary $\texttt{obj}_{\texttt{C}}(\texttt{next}_{\texttt{C}})$ is allocated instead of any garbage collected old object cannot make a difference. Since the ODL logic does not incorporate resource constraints like limited memory, the ODL calculus and semantics can safely stick to allocating $\texttt{obj}_{\texttt{C}}(\texttt{next}_{\texttt{C}})$ without any effect on the program or loss of verification power. Regardless of the final object reallocation behaviour, implicit destructor invocations have to be dealt with for a translation from programming languages with both garbage collection and destructors.

3.4.4 Dynamic Dispatch

In case of finite subtypes[13], dynamic dispatch of method calls can be reduced to ordinary procedure calls. If ODL had function pointers alias formal procedures, dynamic dispatch could also be resolved for arbitrary type hierarchies with the help of the usual method dispatch tables (Wilhelm & Maurer, 1997). However, ODL prefers to use type cascades for reasons of conceptual simplicity and presentation.

Example 3.4.3 Dynamic dispatch can be resolved by a simple pattern. In general, dynamic dispatch occurs in situations like the one sketched in the following JavaCardDL program.

```
class B {
  public int m(int i) {
    return i + 1;
  }
}
class C extends B {
```

officially demanded attribute clearing in ODL this would be an obvious refinement.

[13]Finite subtypes means that for any type τ there are finitely many subtypes of τ. This does not necessarily imply a finite number of types, since there could still be an infinite number of roots in the class hierarchy. However, for reasons of simplicity assuming a finite type hierarchy is preferable.

```
  public int m(int i) {
    return i * 2;
  }
}
class D extends B {
  public int m(int i) {
    return i + 4;
  }
}
class E extends D {
  public int m(int i) {
    return i - 1;
  }
}
public class Main {
  public int main(B b) {
    return b.m(10);
  }
}
```

In this case, for instance, the methods `m` that are subject to overriding can be renamed to *classname*_`m`. Furthermore, a dedicated dispatch method for indirection is introduced instead of the original method definition. This method performs dynamic dispatch with an if cascade of dynamic type checks. The hierarchical order of type checks needs to be bottom-up to establish flat one-dimensional checks instead of nested if cascades.

```
class B {
  /**
   * substitute for invocation of b.m(i)
   */
  public static final int m(B b, int i) {
    if (b instanceof E) {
      return ((E)b).E_m(i);
    } else if (b instanceof D) {
      // it is decisive, that this check
      // occurs after that for E
      return ((D)b).D_m(i);
    } else if (b instanceof C) {
      return ((C)b).C_m(i);
    } else if (b instanceof B) {
      return ((B)b).B_m(i);
```

```
      } else {
        // cannot occur due to
        // complete type hierarchy
      }
    }
    public final int B_m(int i) {
      return i + 1;
    }
  }
  class C extends B {
    public final int C_m(int i) {
      return i * 2;
    }
  }
  class D extends B {
    public final int D_m(int i) {
      return i + 4;
    }
  }
  class E extends D {
    public final int E_m(int i) {
      return i - 1;
    }
  }
  public class Main {
    public int main(B b) {
      return B.m(b, 10);
    }
  }
```

□

In ODL, the cast operations are assumed unchecked, with this responsibility transferred to explicit dynamic type checks via `instanceof`. Hence, cast operations do not produce any visible effect at all, because of which they are assumed to have not more than mere documentation purpose.

Resolving ad-hoc polymorphism by explicit type-check cascades in the above way also leads to some drawbacks in conjunction with encapsulation. As far as the logic itself is concerned, encapsulation does not provide any semantics and type-check cascades do not impose theoretical problems. However, from a practical theorem prover perspective, encapsulation provides pragmatical advantages. In particular, the above resolution of ad-hoc polymorphism leads to non-modular and non-local proofs. For example, the addition of a subclass invalidates the whole proof, thereby necessitating a similar repetition of all proofs about all programs that possibly invoke methods of the new class. This is due to the fact that the underlying type-check cascade has changed as a result of altering the class hierarchy by adding a new class. From a pragmatic perspective, verification systems would need encapsulation in the form of proof components that allow the assembly of a global proof from modular local proofs, which moreover remain rather stable over program evolution or specification adjustment. Instead, the presence of a proof reuse mechanism (Beckert & Klebanov, 2004) in KeY should alleviate the consequences of this type-check cascade approach to dynamic dispatch. A proof reuse mechanism can track the changes of a new proof obligation in comparison to an older proof attempt and replicate elder inferences for the "equivalent" part. This could cope with the additional type-case statement resulting from the addition of a new class in a more flexible way.

3.4.5 Abrupt Completion

Reasons for abrupt completion are circumstances in which execution leaves sequential composition order[14]. In JAVA, exception throwing, return statements, breaks and continues constitute reasons for abrupt completion. In all those cases, the next statement to execute is not generally determined context-free, but depends on global structural properties of the program. To some respect, exceptions form the most general case of reasons for abrupt completion, because all other reasons can be reduced to exceptions by simple program transformations.

Example 3.4.4 Take a look at the following JAVA program with an intermediate return statement.

```
int m() {
  int c = 0;
  for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
      c = c + (1 << i);
    } else if (i % 2 == 1) {
      c = c - (1 << i);
    }
```

[14] More precisely, wherever execution continuation cannot be expressed by a context-free Chomsky-2 grammar (in an adequately simple non-transitive way without dependency on dynamic properties of the program).

```
      if (c > 10) {
        return 10;
      }
    }
    return c;
  }
```

Intermediate return statements as occurring in the above program, can be reduced to exception throwing as follows.

```
public class Return extends Exception {
  public final int returnValue;
  public Return(int value) {
    this.returnValue = value;
  }
}

int m() {
  int c = 0;
  try {
    for (int i = 0; i < 10; i++) {
      if (i % 2 == 0) {
        c = c + (1 << i);
      } else if (i % 2 == 1) {
        c = c - (1 << i);
      }
      if (c > 10) {
        throw new Return(10);
      }
    }
    throw new Return(c);
  }
  catch (Return r) {
    return r.returnValue;
  }
}
```

The final return statement is left as the last statement of the method for clarity. If this last return statement should be avoided as well, the `catch` statement for the `Return` would have to be moved further up the invocation trace to the caller. □

Example 3.4.5 Intermediate break statements experience almost the same treatment as intermediate return statements.

```
int m() {
  int c = 0;
  loop1: for (int i = 0; i < 10; i++) {
    int power = 1;
    for (int j = 0; j < i; j++) {
      power = 2 * power;
      if (power > 20) {
        break loop1;
      }
    }
    c = c + power;
  }
  return c;
}
```

Loop breaking statements as in the above program can be reduced to the raising of exceptions as follows.

```
public class Loop1Break extends Exception {
  public Loop1Break() {}
}
int m() {
  int c = 0;
  try {
    for (int i = 0; i < 10; i++) {
      int power = 1;
      for (int j = 0; j < i; j++) {
        power = 2 * power;
        if (power > 20) {
          throw new Loop1Break();
        }
      }
      c = c + power;
    }
  }
  catch (Loop1Break b) {}
  return c;
}
```

□

Example 3.4.6 Consider a program with intermediate continuation statements.

```
int m() {
  int c = 0;
  loop1: for (int i = 0; i < 10; i++) {
    int power = 1;
    for (int j = 0; j < i; j++) {
      power = 2 * power;
      if (power == 15) {
        continue loop1;
      }
    }
    c = c + power;
  }
  return c;
}
```

Loop continuation statements as in the above program can be reduced to the raising of exceptions as follows.

```
public class Loop1Continue extends Exception {
  public Loop1Continue() {}
}
int m() {
  int c = 0;
  for (int i = 0; i < 10; i++) {
    try {
      int power = 1;
      for (int j = 0; j < i; j++) {
        power = 2 * power;
        if (power == 15) {
          throw new Loop1Continue();
        }
      }
      c = c + power;
    }
    catch (Loop1Continue cont) {}
  }
  return c;
}
```

□

3.4.6 Exception Handling

Exceptions are not part of ODL, but have to be emulated for an adequate treatment of JavaCardDL exceptions and – by the transformation in §3.4.5 – other reasons for abrupt completion. On the level of ODL, exceptions can either be treated intrinsically within the logic or by preprocessing program transformation. The intrinsic treatment involves appropriate inference rules that pass exceptions up the call stack to the first matching catch clause (refer to (Beckert & Sasse, 2001) for a treatise on exception handling). In the case of immediate program transformation, instead, exception handling will appear explicitly within the program in the form of flags.

Example 3.4.7 Consider the following integer division program.

```
try {
    // perform integer division z = x / y
    z = 0;
    while (x >= y) {
        int old_x = x;
        x = x - y;
        if (x == old_x) {
            throw new DivisionByZeroEx(x, y);
        }
        z = z + 1;
    }
    // use result z for further computation
}
catch (DivisionByZeroEx raised) {
    // handle division by zero somehow
}
catch (OutOfMemoryEx raised) {
    // handle memory overflow somehow
}
```

It can be transformed automatically into a program that does not use exception raising and handling anymore. Still it makes use of the class hierarchy of exceptions, which is just a question of convenience[15].

```
// perform integer division z = x / y
z = 0;
Exception raised = null;
```

[15]Of course, any other class hierarchy duplicating the properties of the user-defined exceptions would suffice.

```
while (raised == null && x >= y) {
    int old_x = x;
    x = x - y;
    if (x == old_x) {
        raised = new DivisionByZeroEx(x, y);
    }
    if (raised == null) {
        z = z + 1;
    }
}
if (raised == null) {
    // use result z for further computation
} else {
    // catching exceptions
    if (raised instanceof DivisionByZeroEx) {
        // handle division by zero somehow
    } else if (raised instanceof OutOfMemoryEx) {
        // handle memory overflow somehow
    } else {
        // continue further up the execution
        // trace by returning raised
    }
}
```

Notice that, contrary to several other transformations, the exception handling program transformation affects most surrounding statements as well. □

Of course, the program transformation approach of exception handling leads to rather unreadable programs, which is why most modern programming languages do incorporate a concept of exceptions. Still this transformation allows uncovering exceptions as non-essential to object-orientation, and permits excluding dedicated exception constructs from ODL without loss of generality. Reintegrating the exception handling of (Beckert & Sasse, 2001) into ODL is straightforward, at least as long as exceptions are restricted to be raised from top-level statements rather than during ordinary expression evaluation.

The main advantage of banning exceptions from ODL is that all expressions have a defined value without the need for a treatment of partiality in the logic. For example, the truth-value of an expression like $\texttt{null}.a \doteq 5$ is neither bound to be *true* nor *false*, *a priori*. Still such an expression needs to receive a well-defined semantics. This problem gets worse for assignments like $\texttt{null}.a \lhd 17$, which requires a well-defined manner of execution as well.

Then the logic ODL would have to promote the exceptional case of values being `null` throughout the inductive valuation. Moreover, `null` would need a proper exceptional treatment in the inference rules. Banning exceptions from ODL alleviates all those technical complications at the cost of more explicit preprocessing transformations.

3.4.7 Built-In Operators

An extension of the ODL logic by built-in operators of the source language is straightforward, provided that there is a suitable axiomatisation of the operator semantics. The most important case for practical program verification is modular integer arithmetics. For simplicity ODL assumes the mathematical notions, instead of the peculiarities of JavaCardDL. See (Beckert & Schlager, 2004; Schlager, 2002) for more information on this topic. Depending on the programmer's intention there are several possibilities for proving statements about programs involving modular arithmetic.

1. The program is intended to work within a range of integers that is assumed to keep away from the machine size by presupposition. Under this assumption, proving properties of programs does not need any special treatment of modular arithmetic.

2. The effect of the program is constrained in such a way that the limit on input values prohibits any overflow during program execution. Then the special treatment of machine integer arithmetic will amount to a proof that no intermediate value ever exceeds, say, 2^{64}, which is possible within ODL without extension.

3. The program is designed for correct behaviour even in the case of modular overflow or it specially exploits the overflow effects.

Only the last case requires dedicated modular integer arithmetics treatment. One way to achieve this would, of course, be to use an explicit embedding of the, say 8-bit, machine-size integers into the unbounded data type `nat` of natural numbers. A modular multiplication $c = a \cdot b$ of two machine-sized numbers will then use the following translation.

$$\begin{array}{l} c' \lhd a \cdot b; \\ \texttt{while}(c' \geq 256)\,\{c' \lhd c' - 256\}; \\ c \lhd c' \end{array}$$

A better approach, though, is to define a modulus operator *mod* within ODL and use it to perform modular arithmetic.[16]

$$c \lhd a \cdot b \mod 256$$

Adding the modulus operator does not lead to a proper extension of ODL, since *mod* can be characterised by the following defining formula.

$$r \doteq a \; mod \; n \quad \leftrightarrow \quad \exists z\mathpunct{:}\mathtt{nat} \;\; a \doteq z \cdot n + r \land r < n$$

With this operator, the definition of a dedicated modular arithmetic addition operator via $a *_{8-bit} b \;:=\; a \cdot b \; mod \; 256$ is possible. However, keeping the *mod* operator explicit could be of advantage in order to simplify proofs by exploiting homomorphic properties of modulo mappings. In comparison to

$$a +_{8-bit} b *_{8-bit} (c +_{8-bit} d)$$

which expands to

$$a + \big(b \cdot (c + d \; mod \; 256) \; mod \; 256\big) \; mod \; 256$$

A simplification to this term can be achieved still by using the property that $x \mapsto x \; mod \; 256$ is a homomorphism of rings, thereby saving a lot of intermediate modular arithmetic inferences.

$$a + b \cdot (c + d) \; mod \; 256$$

In the remainder of this thesis further assume a corresponding built-in treatment of addition and multiplication of concrete natural number literals like $3 + 4$ to 7, for convenience. Converting such built-in rules into the formalism of ordinary inference rules is straightforward.

3.5 Discussion

After we have seen that it is possible to reduce programs of other object-oriented programming languages like Java to ODL, let us continue with a discussion of the particular structure of ODL. As can be seen from the literature, there are other approaches to minimal object-oriented programming languages conceivable as well.

[16]The lax notation $a \cdot b \; mod \; 256$ means $(a \cdot b) \; mod \; 256$, the modulo operator applied to the two arguments $a \cdot b$ and 256.

Most features of modern object-oriented programming languages, in fact, turn out to be orthogonal to the concepts of object-orientation. The question of side-effects and abrupt completion is independent of object-orientation. They are an issue in functional and declarative programming languages, too. PROLOG, for example, allows side-effecting evaluation, while MERCURY (Somogyi *et al.*, 1995; Dowd *et al.*, 2000) prohibits side-effects. MERCURY and ISO PROLOG permit exception handling, whereas other PROLOG dialects do not provide such techniques. HASKELL is a pure functional language without side-effects, while LISP itself is impure, i.e. allows side-effecting expression evaluation. All those orthogonal aspects of the programming language have been removed from ODL since we intend to study object-orientation in isolation, albeit on top of the standard control structures of WHILE.

Cast operations are assumed to perform no type checks, because the `instanceof` predicate already is responsible for type checking. Thus, casts do not contribute anything to the flavour of object-orientation. In most cases the presence of casts only hints at lacking parametric genericity, anyway.

Object creation is in principle very well part of object-orientation and would thus bring along enough justification to be included into an object-oriented language. The simplicity of the translation and the fact that the resulting axiomatisation in §4.2.4 is totally immediate, militate for the exclusion from ODL, though. Further advocacy for this exclusion originates from the highly successful practical performance achieved with the ODL approach as will become clear in §4.3.

As far as dynamic dispatch is concerned, it has a straightforward emulation with dynamic type checks according to §3.4.4. Thus, there is no pressing demand to include native dynamic dispatch into ODL. However, to a certain extent, this is a matter of taste. The converse emulation of dynamic type checks with dynamic dispatch is possible as well, although at the expense of requiring an encoding of class types into objects.[17] We opt for the inclusion of the simpler concept of dynamic typing information and an emulation of dynamic dispatch, rather than the other way around.

Assignment to complex expressions or change of the interpretation of function symbols, cannot be removed from ODL without losing the operational basis of imperative object-oriented programming, which permits the change of structured data and dynamically typed. Of course, this pertains to a schematological notion without arbitrary artificial coding.

[17]cf. "meta"-objects of type `java.lang.Class` in JAVA.

Chapter 4

Proof Theory

4.1 Overview

With this chapter presenting a program verification calculus for ODL, it constitutes the practical mechanism for verifying whether a formula is true about a program or false. In order to show that the proof mechanism cannot produce false answers but always comes to the right conclusions §4.4 establishes the soundness proof. The complementary statement that a verification system based on the ODL calculus makes use of the computing capabilities available to Turing machines on the highest possible degree is contained in §4.5. This vague statement will also be made formally precise via the classical notion of relative completeness in §4.5.

4.2 Calculus for ODL

In this section we present a calculus for proving ODL formulas. A calculus is the foundation of a program verification system and constitutes the central basis of an implementation as a theorem prover.

Definition 4.2.1 (Sequents) *For finite sets* Γ, Δ *of formulas, a* sequent *is defined as*

$$\Gamma \ \vdash \Delta \quad := \quad \bigwedge_{\phi \in \Gamma} \phi \to \bigvee_{\psi \in \Delta} \psi$$

Γ *is called the* antecedent *and* Δ *the* succedent *of the sequent.*

The constituents of sequents being considered as sets effectuates that the multiplicity and order of occurrences of the formulas is irrelevant.

The ODL calculus consists of rules R1 to R47 presented in figures 4.1 to 4.7 on pages 70–74 and 79–80.

Definition 4.2.2 (Derivation & Inference) *Let* $\Phi, \Psi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *be (finite*[1]*) sets of formulas.* Ψ *is* derivable *from* Φ*, if there is* $n \in \mathbf{N}$ *and there is* $\Phi = \Phi_0,\ \Phi_1,\ \Phi_2, \ldots, \Phi_n = \Psi \subseteq \mathrm{Fml}(\Sigma \cup V)$ *such that for each* $i < n$ *for each* $\phi \in \Phi_{i+1}$ *there is* $P_1, \ldots, P_n \in \Phi_i$ *with*

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

is an instance[2] *of an inference rule in the* ODL *calculus.*
Short notation: $\Phi \vdash \Psi$*. In case of* $\Psi = \{\psi\}$ *write* $\Phi \vdash \psi$ *instead of* $\Phi \vdash \{\psi\}$*. Further,* $\vdash \Psi$ *is an abbreviation for* $\emptyset \vdash \Psi$*.*

Although soundness itself will be proven in §4.4, the notion of soundness already plays a role in the remainder of this discourse. Soundness establishes one connexion between syntactical inference and semantical consequence. Since there are two notions of consequence there will also be two notions of soundness.

Definition 4.2.3 (Soundness) *An inference rule*

$$\frac{C_1 \quad \ldots \quad C_n}{D}$$

is called sound wrt. $\vDash_l$ *, if* $C_1, \ldots, C_n \vDash_l D$*. Similarly, the inference rule is called sound wrt.* $\vDash_g$*, if* $C_1, \ldots, C_n \vDash_g D$*.*

The next result allows to generalise top-level inference rules to modal situations. This is especially useful for pragmatic reasons in order to shortcut derivations by applying inference rules prior to unfolding all formulas by the update application mechanism. The conjecture demonstrates that soundness is not disturbed by performing inferences inside (some) nested formulas. Any inference rule can also be applied within an update prefix, with the update prefix spreading to each formula involved in the original inference rule.

Proposition 4.2.4 (Contextual Lifting) *Provided that* Γ, Δ *are not both empty,*

$$\frac{\Gamma' \ \vdash \Delta'}{\Gamma \ \vdash \Delta} \ \textit{sound wrt.} \vDash_l \quad \Rightarrow \quad \frac{[\alpha]\Gamma' \ \vdash \langle\alpha\rangle\Delta'}{[\alpha]\Gamma \ \vdash \langle\alpha\rangle\Delta} \ \textit{sound wrt.} \vDash_l$$

[1]This set is assumed finite for simplicity. In case of compact or finitary consequence relations, this imposes no restriction.

[2]i.e. schematic variables of the inference rule schema experience arbitrary instantiations in order to match the present case.

Especially, in case of deterministic terminating programs like updates or cascades of conditional updates this statement can be refined as follows and generalised to $n \geq 1$ premises.

$$\frac{\Gamma' \ \vdash \Delta'}{\Gamma \ \vdash \Delta} \ \textit{sound wrt.} \models_l \quad \Rightarrow \quad \frac{\langle\alpha\rangle\Gamma' \ \vdash \langle\alpha\rangle\Delta'}{\langle\alpha\rangle\Gamma \ \vdash \langle\alpha\rangle\Delta} \ \textit{sound wrt.} \models_l$$

Proof: For a sequent context Δ of the form $\phi_1, \phi_2, \ldots, \phi_n$ the symbolic notation $\langle\alpha\rangle\Delta$ is an abbreviation for $\langle\alpha\rangle\phi_1, \ldots, \langle\alpha\rangle\phi_n$ rather than $\langle\alpha\rangle(\phi_1 \vee \cdots \vee \phi_n)$, here. Similarly, $[\alpha]\Delta$ abbreviates $[\alpha]\phi_1, \ldots, [\alpha]\phi_n$.

1. First, we show the conjecture for the case of empty antecedents $\Gamma = \Gamma' = \emptyset$. Let w be any state of some interpretation ℓ, then assume $\ell, w \models \langle\alpha\rangle\Delta'$. From this we can conclude that for some w' with $w\rho_\ell(\alpha)w'$ it is $\ell, w' \models \Delta' \overset{\text{sound}}{\Rightarrow} \ell, w' \models \Delta \overset{(!)}{\Rightarrow} \ell, w \models \langle\alpha\rangle\Delta$.

2. In case of arbitrary antecedents Γ, Γ', from the soundness of

$$\frac{\Gamma' \ \vdash \Delta'}{\Gamma \ \vdash \Delta}$$

the conclusion can be drawn by the R1 duality that the following inference rule is sound, too.

$$\frac{\vdash \neg\Gamma', \Delta'}{\vdash \neg\Gamma, \Delta}$$

Then case 1 allows to deduce the soundness of

$$\frac{\vdash \langle\alpha\rangle\neg\Gamma', \langle\alpha\rangle\Delta'}{\vdash \langle\alpha\rangle\neg\Gamma, \langle\alpha\rangle\Delta}$$

The R1 duality and the $\neg\langle\alpha\rangle\neg\phi \equiv [\alpha]\phi$ duality allow to conclude the soundness of

$$\frac{[\alpha]\Gamma' \ \vdash \langle\alpha\rangle\Delta'}{[\alpha]\Gamma \ \vdash \langle\alpha\rangle\Delta}$$

The second conjecture is a direct consequence of the fact that deterministic terminating programs satisfy $\langle\alpha\rangle\phi \equiv [\alpha]\phi$. ■

The next result allows to apply inference rules in sequent context, i.e. whenever there is a match of a sound inference rule to a part of a sequent, it can be applied to the whole sequent as well, leaving the additional context formulas unchanged by the inference application.

Proposition 4.2.5 ("Context-free inference")

$$\frac{\Phi_1 \ \vdash \Psi_1 \quad \dots \quad \Phi_n \ \vdash \Psi_n}{\Phi \ \vdash \Psi}$$

is sound wrt. $\vDash_l$ *then the following inference rule is sound wrt.* $\vDash_l$

$$\frac{\Gamma, \Phi_1 \ \vdash \Psi_1, \Delta \quad \dots \quad \Gamma, \Phi_n \ \vdash \Psi_n, \Delta}{\Gamma, \Phi \ \vdash \Psi, \Delta}$$

Proof: Provided that $\Phi_1 \ \vdash \Psi_1, \dots, \Phi_n \ \vdash \Psi_n \ \vDash \ \Phi \ \vdash \Psi$, we have to show

$$\Gamma, \Phi_1 \ \vdash \Psi_1, \Delta, \dots, \Gamma, \Phi_n \ \vdash \Psi_n, \Delta \ \vDash_l \ \Gamma, \Phi \ \vdash \Psi, \Delta$$

Abbreviate the formula $\Phi_i \ \vdash \Psi_i$ by χ_i and $\Phi \ \vdash \Psi$ by χ. Likewise, combine Γ and Δ into one new set of formulas $\neg\Delta, \Gamma$, which will be called Δ again for simplicity. Then on the basis that $\chi_1, \dots, \chi_n \vDash_l \chi$ the above conjecture simplifies notationally to

$$\chi_1 \vee \Delta, \dots, \chi_n \vee \Delta \ \vDash_l \ \chi \vee \Delta$$

Let w be an arbitrary state of some interpretation ℓ with $\ell, w \vDash \chi_i \vee \Delta$ for each i, then there are two possible cases.

(I) $\ell, w \vDash \Delta \Rightarrow \ell, w \vDash \chi \vee \Delta$.

(II) $\ell, w \nvDash \Delta \Rightarrow$ for each i it is $\ell, w \vDash \chi_i$, from which the soundness premise allows to conclude that $\ell, w \vDash \chi \Rightarrow \ell, w \vDash \chi \vee \Delta$.

■

The inference rules in Fig. 4.1-4.6 (*excluding* those in Fig. 4.7) are sound wrt. $\vDash_l$, because of which Prop. 4.2.5 and Prop. 4.2.4 permit adding sequent context and update prefix to the inference.

Example 4.2.1 (Inference in Context) Consider, for instance, the R23 inference rule, which deals with if-statements.

$$\frac{e \ \vdash \langle\alpha\rangle A \quad \neg e \ \vdash \langle\gamma\rangle A}{\vdash \langle \texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\}\rangle A}$$

Since R23 is sound wrt. $\vDash_l$, Prop. 4.2.4 allows to add to the inference rule an update prefix $\langle\mathcal{U}\rangle$ consisting of a sequence of updates (or one simultaneous update) as follows, without losing soundness.

$$\frac{\langle\mathcal{U}\rangle e \ \vdash \langle\mathcal{U}\rangle\langle\alpha\rangle A \quad \langle\mathcal{U}\rangle\neg e \ \vdash \langle\mathcal{U}\rangle\langle\gamma\rangle A}{\vdash \langle\mathcal{U}\rangle\langle \texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\}\rangle A}$$

Further, since R23 is sound wrt. $\vDash_l$, Prop. 4.2.5 allows to add a sequent context without losing soundness. Adding the sequent context Γ, Δ leads to the following.

$$\frac{\Gamma, \langle\mathcal{U}\rangle e \ \vdash \langle\mathcal{U}\rangle\langle\alpha\rangle A, \Delta \quad \Gamma, \langle\mathcal{U}\rangle\neg e \ \vdash \langle\mathcal{U}\rangle\langle\gamma\rangle A, \Delta}{\Gamma \ \vdash \langle\mathcal{U}\rangle\langle \mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\gamma\}\rangle A, \Delta}$$

Finally, the original inference rule R23 has been amended to fit to a situation like

$$x > 7 \ \vdash \langle c{\lhd}5\rangle\langle \mathtt{if}(x > c+1)\,\{c{\lhd}1\}\mathtt{else}\{c{\lhd}2\}\rangle c \doteq 1$$

Due to the omnipresent need for context addition capabilities, we do not formally distinguish the original inference rule R23 from the more general variant obtained from Prop. 4.2.4 and Prop. 4.2.5, but assume an implicit identification, instead. $\square$

4.2.1 First-Order

For propositional logic standard inference rules of sequent calculus are listed in Fig. 4.1. Similarly, Fig. 4.2 contains a list of standard inference rules (Schmitt, 2003) for first-order logic, plus an integer induction scheme. Note that by definition of substitutions in 2.5.1, R16 and R20 also apply equations within programs in modalities.

Due to the duality of negative formulas in the antecedent and positive formulas in the succedent of a sequent[3], operators that satisfy a duality relation like $\exists x\,\phi \equiv \neg\forall x\,\neg\phi$ or $\langle\alpha\rangle\phi \equiv \neg[\alpha]\neg\phi$, allow rule abbreviations of the following kind. Once inference rules for both dual operators have been specified for succedent occurrences, duality allows to derive the corresponding rules for antecedent occurrence.[4] Therefore, in case of dual operators, we usually specify only the rules for succedent occurrence.

The propositional and quantifier rules in Fig. 4.1 and Fig. 4.2 have the effect of transforming more complex formulas occurring in the sequents to simpler "modal atoms" of the form $\langle\alpha\rangle\phi$ or $[\alpha]\phi$.

4.2.2 Program-Transformation

Fig. 4.3 contains inference rules that transform programs into logical formulas or into other programs within the modalities.

[3]cf. $\neg$ *left* (R1), $\neg$ *right* (R7).

[4]Similarly, rules for antecedent and succedent occurrence of one operator allow to derive corresponding rules for the dual operator.

(R1) $\neg$ left

$$\frac{\vdash A}{\neg A \vdash}$$

(R2) $\wedge$ left

$$\frac{A, B \vdash}{A \wedge B \vdash}$$

(R3) $\vee$ left

$$\frac{A \vdash \quad B \vdash}{A \vee B \vdash}$$

(R4) $\rightarrow$ left

$$\frac{\vdash A \quad B \vdash}{A \rightarrow B \vdash}$$

(R5) cut

$$\frac{A \vdash \quad \vdash A}{\vdash}$$

(R6) weakening (left)

$$\frac{\vdash}{A \vdash}$$

(R7) $\neg$ right

$$\frac{A \vdash}{\vdash \neg A}$$

(R8) $\wedge$ right

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B}$$

(R9) $\vee$ right

$$\frac{\vdash A, B}{\vdash A \vee B}$$

(R10) $\rightarrow$ right

$$\frac{A \vdash B}{\vdash A \rightarrow B}$$

(R11) axiom

$$\frac{}{A \vdash A}$$

(R12) weakening (right)

$$\frac{\vdash}{\vdash A}$$

Figure 4.1: Propositional Inference Rules

(R13) $\forall$ left

$$\frac{A^t_x, \forall x\, A \ \vdash}{\forall x\, A \ \vdash}$$

(R14) $\exists$ left

$$\frac{A^X_x \ \vdash}{\exists x\, A \ \vdash}$$

(R15) induction

$$\frac{\vdash \phi(0) \quad \phi(n) \ \vdash \phi(n+1)}{\vdash \forall n\, \phi(n)}$$

$\Leftarrow n$ new variable

(R16) $\doteq$ subst

$$\frac{\Gamma^t_s, s \doteq t \ \vdash \Delta^t_s}{\Gamma, s \doteq t \ \vdash \Delta}$$

(R17) $\forall$ right

$$\frac{\vdash A^X_x}{\vdash \forall x\, A}$$

(R18) $\exists$ right

$$\frac{\vdash A^t_x, \exists x\, A}{\vdash \exists x\, A}$$

(R19) $\doteq$ reflexive

$$\frac{}{\vdash t \doteq t}$$

(R20) $\doteq$ subst

$$\frac{\Gamma^t_s, t \doteq s \ \vdash \Delta^t_s}{\Gamma, t \doteq s \ \vdash \Delta}$$

Figure 4.2: First-Order Inference Rules. t is a term, X is a new logical variable in the sequent, and all substitutions are admissible.

The inference rules translating program statements to logic directly, relate the meaning of programs and formulas and are common to dynamic logic. Except for updates, the Fig. 4.3 handles all program statements. The treatment that mere program-transformation rules can offer for loops is very limited, though. The corresponding unwind rules R25, R26 only allow a finite treatment of loops. Still, they are of much use in inductions about loops.

In conjunction with the propositional and quantifier rules, the program transformation rules transform formulas to one the following forms.

- $\langle \mathcal{U} \rangle \phi$
- $[\mathcal{U}] \phi$
- $\langle \mathcal{U}; \texttt{while}(e)\, \{\alpha\} \rangle \phi$ or
- $[\mathcal{U}; \texttt{while}(e)\, \{\alpha\}] \phi$

4.2.3 Term Rewriting

This section presents the inference rules that match on terms instead of formulas. Therefor, a rewrite relation will be defined on terms, which states

(R21) composition

$$\frac{\vdash \langle\alpha\rangle\langle\gamma\rangle A}{\vdash \langle\alpha;\gamma\rangle A}$$

(R22) composition

$$\frac{\vdash [\alpha][\gamma]A}{\vdash [\alpha;\gamma]A}$$

(R23) branch

$$\frac{e\ \vdash \langle\alpha\rangle A \quad \neg e\ \vdash \langle\gamma\rangle A}{\vdash \langle \mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\gamma\}\rangle A}$$

(R24) branch

$$\frac{e\ \vdash [\alpha]A \quad \neg e\ \vdash [\gamma]A}{\vdash [\mathtt{if}(e)\,\{\alpha\}\mathtt{else}\{\gamma\}]A}$$

(R25) loop unwind

$$\frac{\vdash \langle \mathtt{if}(e)\,\{\alpha;\mathtt{while}(e)\,\{\alpha\}\}\rangle A}{\vdash \langle \mathtt{while}(e)\,\{\alpha\}\rangle A}$$

(R26) loop unwind

$$\frac{\vdash [\mathtt{if}(e)\,\{\alpha;\mathtt{while}(e)\,\{\alpha\}\}]A}{\vdash [\mathtt{while}(e)\,\{\alpha\}]A}$$

Figure 4.3: Program-Transformation Inference Rules

(R27) update (match)

$$\langle f(s) \lhd t \rangle f(u) \rightsquigarrow \mathit{if}\, s \doteq \langle f(s) \lhd t \rangle u \;\mathit{then}\, t \;\mathit{else}\, f\big(\langle f(s) \lhd t \rangle u\big) \;\mathit{fi}$$

(R28) update (promote)

$$\langle f(s) \lhd t \rangle\, \Upsilon(u) \rightsquigarrow \Upsilon\big(\langle f(s) \lhd t \rangle u\big)$$

$\Leftarrow f \neq \Upsilon \in \Sigma$

(R29) update ($\forall$)

$$\langle \mathcal{U} \rangle \forall x\, \phi \rightsquigarrow \forall x\, \langle \mathcal{U} \rangle \phi$$

$\Leftarrow x$ not in $FV(\mathcal{U})$

(R30) update (match)

$$\langle \mathcal{U} \rangle f(u) \rightsquigarrow \mathit{if}\, s_{i_r} \doteq \langle \mathcal{U} \rangle u \;\mathit{then}\, t_{i_r} \;\mathit{else} \ldots \mathit{if}\, s_{i_1} \doteq \langle \mathcal{U} \rangle u \;\mathit{then}\, t_{i_1} \;\mathit{else}\, f(\langle \mathcal{U} \rangle u) \;\mathit{fi} \;\mathit{fi}$$

$\Leftarrow (i_1, \ldots, i_r) = (i \;:\; f_i = f)$

(R31) update (distinct)

$$\langle \mathcal{U} \rangle\, \Upsilon(u) \rightsquigarrow \Upsilon\big(\langle \mathcal{U} \rangle u\big)$$

$\Leftarrow \mathcal{U}$ contains no updates to $\Upsilon \in \Sigma$

(R32) update merge

$$\langle \mathcal{U} \rangle \langle \mathcal{U}' \rangle \phi \rightsquigarrow \langle \mathcal{U}, f'_1(\langle \mathcal{U} \rangle s'_1) \lhd \langle \mathcal{U} \rangle t'_1, \ldots, f'_m(\langle \mathcal{U} \rangle s'_m) \lhd \langle \mathcal{U} \rangle t'_m \rangle$$

(R33) update determinism

$$[\mathcal{U}] \rightsquigarrow \langle \mathcal{U} \rangle$$

(R34) update (on formula)

$$\langle f(s) \lhd t \rangle p(u) \rightsquigarrow p\big(\langle f(s) \lhd t \rangle u\big)$$

$\Leftarrow p \in \Sigma$ predicate

Figure 4.4: Term Rewrite Rules. Here, $\langle \mathcal{U} \rangle$ is a short notation for the (simultaneous) parallel update $\langle f_1(s_1) \lhd t_1, \ldots, f_n(s_n) \lhd t_n \rangle$, and $\langle \mathcal{U}' \rangle$ short for $\langle f'_1(s'_1) \lhd t'_1, \ldots, f'_m(s'_m) \lhd t'_m \rangle$. The free variables of an update are defined as expected $FV(\langle f_1(s_1) \lhd t_1 \ldots f_n(s_n) \lhd t_n \rangle) := FV(\{s_1, \ldots, s_n, t_1, \ldots, t_n\})$. Further, the notation $(i_1, \ldots, i_r) = (i \;:\; f_i = f)$ selects all indices of top-level function symbol f without changing the order.

(R35) term rewrite (left)

$$\frac{\phi(t) \ \vdash}{\phi(s) \ \vdash}$$

$\Leftarrow (s \ \rightsquigarrow \ t)$ holds

(R36) term rewrite (right)

$$\frac{\vdash \phi(t)}{\vdash \phi(s)}$$

$\Leftarrow (s \ \rightsquigarrow \ t)$ holds

(R37) conditional term split (left)

$$\frac{(e \to \phi(s)) \land (\neg e \to \phi(t)) \ \vdash}{\phi(\mathit{if}\, e\ \mathit{then}\, s\ \mathit{else}\, t\ \mathit{fi}) \ \vdash}$$

(R38) conditional term split (right)

$$\frac{\vdash (e \to \phi(s)) \land (\neg e \to \phi(t))}{\vdash \phi(\mathit{if}\, e\ \mathit{then}\, s\ \mathit{else}\, t\ \mathit{fi})}$$

Figure 4.5: Rewrite Application Inference Rules: $\phi(s)$ is the formula obtained from ϕ by an implicit admissible substitution $\phi(s) = \phi_z^s$. Usually, uniform replacement is assumed but not necessary. By convention, the term rewrite rules are further subject to the constraint that s really occurs within $\phi(s)$, otherwise no proper inference rule application of any visible effect could happen anyway.

what term rewrites are allowed throughout the formulas. The *term rewrite* relation $\rightsquigarrow$ is defined by the rules in Fig. 4.4. Whenever the term rewrite relation $s \rightsquigarrow t$ holds, a rewrite is possible within any formula, thereby replacing s by t, which is what the rules in Fig. 4.5 specify.

Update rules are a peculiarity of ODL. Basically, for applying updates to terms and formulas, there is rule R27 for the case of a match, i.e. potential changes to (the interpretation of) the specific term at hand. Rule R28 handles the case of an unquestionable mismatch. In the latter case of distinct top-level symbols, the change in the interpretation of f cannot affect Υ, but still the argument u might experience a change. Thus, the effect of the update has to be promoted to the arguments. Due to the nature of the schematic symbol Υ, the rules R27, R28 and R29 completely define the effect that an update has on an expression. Proper parallel updates experience a very similar treatment in R30 and R31.

Updates stop in the face of modalities, i.e. except for R32, $\langle f(s) \lhd t\rangle\langle\alpha\rangle\phi$ or $\langle f(s) \lhd t\rangle[\alpha]\phi$ remain unchanged by the rules of Fig. 4.4. A special case of R28 is a rule for atomic symbols. It leads to updates vanishing when they obviously have no more effects.

$$\langle f(s) \lhd t\rangle\, \Upsilon \;\rightsquigarrow\; \Upsilon \Leftarrow \Upsilon \in \Sigma \cup V$$

In the spirit of simultaneous parallel updates, an important goal for practical purposes is the merging of multiple single updates into one simultaneous update. In particular. when thinking of Java as source language, it is obvious that when the power of simultaneous updates should be used, they have to be constructed by merging, since only assignments equivalent to single updates really occur in the program source. Thus, for simultaneous updates to take their advantages over single updates, inference rules have to combine sequences of single updates into parallel updates.

The R33 rule is just a formal trick. By definition of the semantics, updates are deterministic and guaranteed to terminate. For such programs, $\langle\rangle$ and $[]$ express the same statement. Therefore all update handling rules for $\langle\rangle$ and $[]$ are identical. The R33 rule achieves this without duplication of inference rules and without having to introduce schematic modality matching concepts that allow to formulate inference rules irrespective of the particular type of modality.

Essentially, the R38 rule matches $\mathit{if}\, e\; \mathit{then}\, s\; \mathit{else}\, t\; \mathit{fi}$ terms in expression context and rewrites them with s resp. t in the same context. Additionally it adds new logical operators on top-level. This rule has enough potential for smaller performance improvements. If, for example, e already appears on top-level of the sequent in Γ or Δ, unnecessary branches resulting from $(e \rightarrow \phi(s)) \wedge (\neg e \rightarrow \phi(t))$ can be avoided at this stage. This shortcut optimisation pays in the context of update application when e is an equation that already occurs on top-level due to an earlier case distinction affirming or contradicting the case e. R51 will improve on this issue.

We formally define the process of evaluation by repeated rewrite as follows and continue to establish a connexion between a parallel update and equivalent singleton updates. This connexion is stated as a partial confluence property, which is just as strong as required for concise soundness and completeness proofs.

Definition 4.2.6 (Evaluation by Fixed-Point Rewriting) *The relation* $\rightsquigarrow^*$ *is the* fixed-point rewrite *version of* $\rightsquigarrow$ *.* $s \rightsquigarrow^* t$ *holds if and only if there is a finite sequence* $s = s_0, s_1, \ldots, s_n = t \in \mathrm{Trm}(\Sigma \cup V)$ *such that for each* i $s_i \rightsquigarrow s_{i+1}$ *holds and there is no* s_{n+1} *with* $s_n \rightsquigarrow s_{n+1}$*. In this case, we say that* s evaluates to t *(by rewrite).*

Lemma 4.2.7 (Confluence of Parallelised Updates) *"Applying merged updates equals successively applying singleton updates." More formally: assume there are (singleton) updates* $\mathcal{U}_1, \ldots, \mathcal{U}_n$*, which merge by the R32 rule into the parallel update* $\mathcal{U}$*. Then if*[5] $\langle\mathcal{U}_1\rangle\Big(\langle\mathcal{U}_2\rangle\big(\ldots(\langle\mathcal{U}_n\rangle t)\big)\Big)$ *evaluates to* t^**, then so does* $\langle\mathcal{U}\rangle t$ *and vice versa.*

Proof: It is sufficient to consider the case $n = 2$ as the essential case of a structural induction on t.

$$
\begin{aligned}
& \langle\underbrace{f(s_1)\lhd t_1}_{\mathcal{U}_1}\rangle\langle\underbrace{f(s_2)\lhd t_2}_{\mathcal{U}_2}\rangle f(u) \\
\rightsquigarrow\ & \langle\mathcal{U}_1\rangle \mathit{if}\, s_2 \doteq \langle\mathcal{U}_2\rangle u \;\mathit{then}\; t_2 \;\mathit{else}\; f(\langle\mathcal{U}_2\rangle u) \;\mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, \langle\mathcal{U}_1\rangle s_2 \doteq \langle\mathcal{U}_1\rangle\langle\mathcal{U}_2\rangle u \;\mathit{then}\; \langle\mathcal{U}_1\rangle t_2 \;\mathit{else}\; \langle\mathcal{U}_1\rangle f(\langle\mathcal{U}_2\rangle u) \;\mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, \langle\mathcal{U}_1\rangle s_2 \doteq \langle\mathcal{U}\rangle u \;\mathit{then} \\
& \qquad \langle\mathcal{U}_1\rangle t_2 \\
& \mathit{else} \\
& \qquad \mathit{if}\, s_1 \doteq \langle\mathcal{U}_1\rangle\langle\mathcal{U}_2\rangle u \;\mathit{then}\; t_1 \;\mathit{else}\; f(\langle\mathcal{U}_1\rangle\langle\mathcal{U}_2\rangle u) \;\mathit{fi} \\
& \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, \langle\mathcal{U}_1\rangle s_2 \doteq \langle\mathcal{U}\rangle u \;\mathit{then}
\end{aligned}
$$

[5]Here, the notation $\langle\mathcal{U}_1\rangle\Big(\langle\mathcal{U}_2\rangle\big(\ldots(\langle\mathcal{U}_n\rangle t)\big)\Big)$ suggestively expresses that rewrite first occurs to $\langle\mathcal{U}_n\rangle t$, completely, i.e. $\langle\mathcal{U}_n\rangle t \rightsquigarrow^* r$ and, when no more $\rightsquigarrow$ relations hold for the result r, then evaluation continues with $\langle\mathcal{U}_{n-1}\rangle r$ etc. In other words, no *update merge* (R32) application is allowed in between.

$$
\begin{array}{l}
\quad\quad \langle\mathcal{U}_1\rangle t_2 \\
\quad \mathit{else} \\
\quad\quad \mathit{if}\, s_1 \doteq \langle\mathcal{U}\rangle u \;\mathit{then}\, t_1 \;\mathit{else}\, f(\langle\mathcal{U}\rangle u)\;\mathit{fi} \\
\quad \mathit{fi}
\end{array}
$$

On the other hand the application of concatenated (in contrast to merged) updates $\mathcal{U}_1, \mathcal{U}_2$ would lead to the following.

$$
\begin{array}{l}
\underbrace{\langle f(s_1) \lhd t_1, f(s_2) \lhd t_2\rangle}_{\mathcal{U}'} f(u) \\
\rightsquigarrow \mathit{if}\, s_2 \doteq \langle\mathcal{U}'\rangle u \;\mathit{then}\, t_2 \;\mathit{else}\; \mathit{if}\, s_1 \doteq \langle\mathcal{U}'\rangle u \;\mathit{then}\, t_1 \;\mathit{else}\, f(\langle\mathcal{U}'\rangle u)\;\mathit{fi}\;\mathit{fi}
\end{array}
$$

From which one can conclude that the only difference between successive application $\langle\mathcal{U}_1\rangle(\langle\mathcal{U}_2\rangle t)$ and concatenated application $\mathcal{U}'$ are the extra updates of s_2, t_2. The R32 rule just ensures that this update to s_2, t_2 happens. Thus, $\langle\mathcal{U}_1\rangle(\langle\mathcal{U}_2\rangle t)$ equals $\langle\mathcal{U}\rangle t$. ■

Lem. 4.2.7 allows to ignore parallel updates in completeness and soundness proofs. This has the advantage of a simpler presentation and formulation of the relevant calculus part, and – by consequence – of inductive proofs about the calculus.

As Lem. 4.2.7 shows, merging updates is not necessary from a completeness point of view. Pragmatic aspects suggest merging updates to reconcile, combine and simplify the effects of a sequence of multiple updates in advance, prior to spreading the effect over the term structure.

From a practical point of view, the R27 rule still has a disadvantage. Formally, the update application to u duplicates because $\langle f(s) \lhd t\rangle u$ occurs twice. Thus, a very obstinate application of the inference rules would have to repeat the same term rewrites for both occurrences. Of course this is, in fact, unnecessary. Instead, one could first evaluate $\langle f(s) \lhd t\rangle u \;\rightsquigarrow^*\; u'$ and then use the result u' as in $\mathit{if}\, s \doteq u' \;\mathit{then}\, t \;\mathit{else}\, f(u')\;\mathit{fi}$ to save term rewrites. Formally, this could be expressed in the calculus by λ-abstraction with eager evaluation treatment rules as:

$$
\langle f(s) \lhd t\rangle f(u) \;\rightsquigarrow\; \Big(\underbrace{\lambda z. \mathit{if}\, s \doteq z \;\mathit{then}\, t \;\mathit{else}\, f(z)\;\mathit{fi}}_{\text{“}\langle f(s) \lhd t\rangle f\text{''}}\Big)\big(\langle f(s) \lhd t\rangle u\big)
$$

However, mere jungle rewriting techniques, which promote rewrite changes between syntactically identical subterms immediately, are also sufficient to avoid this update duplication problem. Therefore, we consider this question as a matter of implementation, not a matter of calculus.

Applying updates finally does not lead to an impasse but always retains a closable proof if it had been closeable beforehand, though applying updates may not be necessary for closing some goals. However, what is more important is to find an adequate balance between not branching to early via conditional term split and not branching at all. Even though case distinction is necessary in the general case, determining when to defer its proof branching is a very complicated problem. The separation of update proof rules into update application, conditional terms and conditional term split rules clarifies this distinction and is necessary at least to some extent for the definition of more sophisticated update application heuristics.

In conjunction with the propositional, quantifier and program logic transformation rules, the term transformation rules transform formulas to the form $\langle\mathcal{U};\mathtt{while}(e)\,\{\alpha\}\rangle\phi$ or $[\mathcal{U};\mathtt{while}(e)\,\{\alpha\}]\phi$. Even though it seems that what is missing here is a rule for the application of, say, the rigid part of $\mathcal{U}$ to a loop at first sight, such an inference rule does not really improve the theorem prover. This is the reason for the fact that loop handling finally works by one of the R45, R25 inference rules, which – in the particular ODL calculus – experience no advantage of a more detailed structure of the loop. Therefore update application can just as well wait patiently until the other inference rules tackle or unfold the loop body.

4.2.4 Dynamics

Fig. 4.6 collects all inference rules that are special or characteristic for the dynamic logic ODL in addition to the overall update mechanism.

For the *instanceof* (R39) rule to work, the $\doteq$ *subst* (R16) inference rule is required. It allows to replace terms by equal terms until the original object creation expression has been found. Those $\mathtt{obj}_{\mathtt{C}}(n)$ expressions contain the dynamic type, which allows to decide the truth of an instanceof formula. If rewrite by equality does not lead to an $\mathtt{obj}_{\mathtt{C}}(n)$ expression, then the truth-value of `instanceof` is unknown because the context simply does not contain enough information about the dynamic type of the term at hand. Then the proof must continue closing by other means or wait until enough type information has been deduced from the context. Of course, it is straightforward to add inference rules that can decide $t\,\mathtt{instanceof}\,\mathtt{C}$ occurrences according to the mere static type of t.

As a notational simplification, consider the following abbreviation.

Definition 4.2.8

$$x \doteq \mathtt{obj}_{\,\leq\,\mathtt{C}}(n) \quad := \quad \bigvee_{\sigma\leq\mathtt{C}} x \doteq \mathtt{obj}_{\sigma}(n)$$

(R39) instanceof

$$\Leftarrow \mathtt{A} \leq \mathtt{C} \qquad \frac{}{\vdash \mathtt{obj}_{\mathtt{A}}(n)\, \mathtt{instanceof}\, \mathtt{C}}$$

(R40) instanceof

$$\Leftarrow \mathtt{A} \not\leq \mathtt{C} \qquad \frac{}{\mathtt{obj}_{\mathtt{A}}(n)\, \mathtt{instanceof}\, \mathtt{C} \ \vdash}$$

(R41) `new` identity

$$\frac{}{\vdash \forall i\!:\!\mathtt{nat}\ \forall j\!:\!\mathtt{nat}\ (\mathtt{obj}_{\mathtt{C}}(i) \doteq \mathtt{obj}_{\mathtt{C}}(j) \rightarrow i \doteq j)}$$

(R42) `new` disjoint identity

$$\Leftarrow \mathtt{C} \neq \mathtt{D} \qquad \frac{}{\mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{D}}(m) \ \vdash}$$

(R43) `new` generated

$$\frac{}{\vdash \forall o\!:\!C\ \exists n\!:\!\mathtt{nat}\ o \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)}$$

(R44) $\exists$ generalisation

$$\frac{A \ \vdash B}{\exists x\, A \ \vdash \exists x\, B}$$

Figure 4.6: Dynamic Inference Rules

(R45) loop induction right

$$\frac{\Gamma \vdash \langle\mathcal{U}\rangle p, \Delta \quad p, e \vdash [\alpha]p \quad p, \neg e \vdash A}{\Gamma \vdash \langle\mathcal{U}\rangle[\mathtt{while}(e)\,\{\alpha\}]A, \Delta}$$

(R46) $\langle\rangle$ generalisation

$$\frac{A \vdash B}{\langle\alpha\rangle A \vdash \langle\alpha\rangle B}$$

(R47) $[]$ generalisation

$$\frac{A \vdash B}{[\alpha]A \vdash [\alpha]B}$$

Figure 4.7: Global Inference Rules. These inference rules are only sound wrt. $\vDash_g$, because of which Prop. 4.2.5 and Prop. 4.2.4 do not permit adding context to these rules.

Where $\vDash \mathtt{obj}_\perp(n) \doteq \mathtt{null}$ *for simplicity.*

The `new` *generated* (R43) rule expresses that *all* objects can be generated by new object creation expressions, i.e. every object that exists can be created at some time.[6]

4.2.5 Clash Semantics Interaction

In §2.3.6 different semantics for parallel updates in the presence of clashes have been mentioned. This section presents a discussion of their interaction with the calculus.

The inference rules R32 and R30 constitute the impact of the choice of clash semantics. In general, proving statements about the a program requires

[6]Philosophically speaking, all existent things must have had a time of creation, and no object exists intrinsically all times. Contrary to the philosophical quarrels that arise from such a statement, in the context of program execution, it has a plausible justification. Programs have been started at some time, and each object that has been loaded into memory is the result of a corresponding object construction statement. Thus, the memory does not contain "surprises" like spontaneously manifesting objects. Therefore, everything that can be proven about all objects that can ever be created by the program holds for all existent objects. Since – unlike this intuitive reasoning – the ODL constant domain semantics assumes that all objects are initially part of the domain, this ought to be put in another way: Everything that can be proven about all created objects is true about all objects existing at some time.

a case distinction.

$$f(s) \lhd t;$$
$$f(s') \lhd t'$$

The above program α comprises an inherent case distinction because its effect on a state depends on whether or not s and s' evaluate to the same location, which cannot be determined in general. Somehow, a calculus has to decode the effect of α to "f changes at s' to t' (bearing in mind that t' gets evaluated in a state where the change of f at s to t already has been performed). If, furthermore, s and s' evaluate to distinct locations, then f also changes at s to t."

Therefore, at some time during the application or merging of the two updates above, this case distinction has to be dealt with. One natural choice would consist of merging the two updates into a joint parallel update. An alternative would be to defer this distinction just until the application of the merged update to terms.

Lock, skip, nondeterministic and arbitrary clash semantics need case distinctions for update merging. However, last-win semantics performs case distinctions during update application.

The major disadvantage of case distinctions during the time of merging is that they lead to early case distinctions (leading to replicate parts of the proof in multiple goals) and tends to produce unnecessary case distinctions, whenever the distinction later turns out not to make a difference for the particular formulas at hand.

Example 4.2.2 Proving whether a formula ϕ is true after the execution of program α from above requires a case distinction. When this distinction is made at the time of merging the two updates into one, the proof splits very early:[7]

$$\begin{aligned}
&\langle f(s) \lhd t; f(s') \lhd t' \rangle \phi \\
\Rightarrow\ &\langle f(s) \lhd t \rangle \langle f(s') \lhd t' \rangle \phi \\
\Rightarrow\ &\big(s \doteq s' \rightarrow \langle f(s') \lhd t' \rangle \phi\big) \land \\
&\big(s \neq s' \rightarrow \langle f(s) \lhd t, f(s') \lhd t' \rangle \phi\big)
\end{aligned}$$

This leads to two proof branches containing the same formula ϕ. If ϕ itself needs many inferences until closing goals is possible, then this computationally expensive work has to be repeated twice, once on each branch.

[7]For simplicity of presentation, assume that s', t' are immune to changes of $f(s)$, i.e. $f(s)$ is semantically rigid for the program $\langle f(s) \lhd t \rangle$ (Platzer, 2004).

Furthermore, after the effect of the two distinct updates has been promoted into ϕ, the remains of ϕ will not be recognised as having the same origin by most inter-goal proof sharing mechanisms[8], and double work is almost unavoidable.

Even worse wasting occurs, if after some further processing ϕ turns out to be non-sensitive to the change of f at s when f changes at s' to t' anyway, i.e. $\langle f(s')\lhd t'\rangle\phi$ is semantically rigid for the program $\langle f(s)\lhd t\rangle$. Then the whole case distinction and all subsequent proof steps repeated on the second branch have been void. Consider, for example a formula ϕ that can be reduced to $f(s') \doteq t'$ after some processing. $\square$

The disadvantage of case distinctions at the time of update application is that those case distinctions multiply with the update distribution through all subexpressions. The simultaneous replacement option in the R38 rule alleviates the effects of high case distinction multiplication, though, since it integrates all identical case distinctions into a single branch, regardless of the number of places in a formula at which a case distinction is made.

Example 4.2.3 The most general case of an update on a nested term with possible effects at all levels is examined in this example. Since only one function symbol occurs, all occurrences of f are possibly subject to change their value under the update, which leads to a maximum amount of case distinctions.

$$
\begin{aligned}
& \langle f(s)\lhd r\rangle f(f(o)) \\
\rightsquigarrow\ & \mathit{if}\, s \doteq {\scriptstyle \langle f(s)\lhd r\rangle f(o)}\ \mathit{then}\, r\ \mathit{else}\, f\big(\langle f(s)\lhd r\rangle f(o)\big)\ \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, s \doteq {\scriptstyle \langle f(s)\lhd r\rangle f(o)}\ \mathit{then} \\
& \qquad r \\
& \mathit{else} \\
& \qquad f\big(\mathit{if}\, s \doteq \langle f(s)\lhd r\rangle o\ \mathit{then}\, r\ \mathit{else}\, f(\langle f(s)\lhd r\rangle o)\ \mathit{fi}\big) \\
& \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, s \doteq \big({\scriptstyle \mathit{if}\, s \doteq \langle f(s)\lhd r\rangle o\ \mathit{then}\, r\ \mathit{else}\, f(\langle f(s)\lhd r\rangle o)\ \mathit{fi}}\big)\ \mathit{then} \\
& \qquad r \\
& \mathit{else} \\
& \qquad f\big(\mathit{if}\, s \doteq \langle f(s)\lhd r\rangle o\ \mathit{then}\, r\ \mathit{else}\, f(\langle f(s)\lhd r\rangle o)\ \mathit{fi}\big) \\
& \mathit{fi}
\end{aligned}
$$

[8]Inferences on identical subterms of different branches can be performed simultaneously in joint effort, provided that the processing does not contain (non-confluent) incompatible choices in between. In an appropriate inter-goal directed acyclic graph representation, this saves term matching and rule application cost.

$$
\begin{aligned}
\rightsquigarrow\ & \mathit{if}\, s \doteq \big(\scriptstyle \mathit{if}\, s \doteq o\, \mathit{then}\, r\, \mathit{else}\, f(\langle f(s) \lhd r\rangle o)\, \mathit{fi}\big)\ \mathit{then} \\
& \qquad r \\
& \mathit{else} \\
& \qquad f\big(\mathit{if}\, s \doteq o\, \mathit{then}\, r\, \mathit{else}\, f(\langle f(s) \lhd r\rangle o)\, \mathit{fi}\big) \\
& \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, s \doteq \big(\scriptstyle \mathit{if}\, s \doteq o\, \mathit{then}\, r\, \mathit{else}\, f(o)\, \mathit{fi}\big)\ \mathit{then} \\
& \qquad r \\
& \mathit{else} \\
& \qquad f\big(\mathit{if}\, s \doteq o\, \mathit{then}\, r\, \mathit{else}\, f(o)\, \mathit{fi}\big) \\
& \mathit{fi} \\
\text{“}\rightsquigarrow\text{”}\ & \big(s \doteq o \to \mathit{if}\, s \doteq r\, \mathit{then}\, r\, \mathit{else}\, f(r)\, \mathit{fi}\big) \land \\
& \big(s \neq o \to \mathit{if}\, s \doteq f(o)\, \mathit{then}\, r\, \mathit{else}\, f(f(o))\, \mathit{fi}\big) \\
\text{“}\rightsquigarrow\text{”}\ & \big(s \doteq o \land s \doteq r \to r\big) \land \\
& \big(s \doteq o \land s \neq r \to f(r)\big) \land \\
& \big(s \neq o \land s \doteq f(o) \to r\big) \land \\
& \big(s \neq o \land s \neq f(o) \to f(f(o))\big)
\end{aligned}
$$

The two nested occurrences of f lead to the worst case of 2^2 final case distinctions, not considering any knowledge about s, r, o from the context. The best case of update application is the reduction $\langle f(s) \lhd s\rangle f(f(s)) \rightsquigarrow s$ as presented in a detailed proof in Ex. B.1. □

Example 4.2.4 There is a very similar example in which case distinctions can be saved nevertheless, because of the known syntactic equality of some subterms occurring during the rewrite process.

$$
\begin{aligned}
& \langle f(s) \lhd r\rangle f(f(r)) \\
\rightsquigarrow\ & \mathit{if}\, s \doteq {\scriptstyle \langle f(s) \lhd r\rangle f(r)}\ \mathit{then}\, r\, \mathit{else}\, f\big(\langle f(s) \lhd r\rangle f(r)\big)\, \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, s \doteq {\scriptstyle \langle f(s) \lhd r\rangle f(r)}\ \mathit{then} \\
& \qquad r \\
& \mathit{else} \\
& \qquad f\big(\mathit{if}\, s \doteq \langle f(s) \lhd r\rangle r\ \mathit{then}\, r\, \mathit{else}\, f(\langle f(s) \lhd r\rangle r)\, \mathit{fi}\big) \\
& \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, s \doteq \big(\scriptstyle \mathit{if}\, s \doteq \langle f(s) \lhd r\rangle r\ \mathit{then}\, r\, \mathit{else}\, f(\langle f(s) \lhd r\rangle r)\, \mathit{fi}\big)\ \mathit{then} \\
& \qquad r \\
& \mathit{else}
\end{aligned}
$$

$$
\begin{aligned}
&\quad\quad f\big(\mathit{if}\, s \doteq \langle f(s) \lhd r\rangle r \;\mathit{then}\, r \;\mathit{else}\, f(\langle f(s) \lhd r\rangle r)\;\mathit{fi}\big)\\
&\mathit{fi}\\
\rightsquigarrow\; &\mathit{if}\, s \doteq \big(\scriptstyle \mathit{if}\, s \doteq r \;\mathit{then}\, r \;\mathit{else}\, f(\langle f(s) \lhd r\rangle r)\;\mathit{fi}\displaystyle\big)\;\mathit{then}\\
&\quad\quad r\\
&\mathit{else}\\
&\quad\quad f\big(\mathit{if}\, s \doteq r \;\mathit{then}\, r \;\mathit{else}\, f(\langle f(s) \lhd r\rangle r)\;\mathit{fi}\big)\\
&\mathit{fi}\\
\rightsquigarrow\; &\mathit{if}\, s \doteq \big(\scriptstyle \mathit{if}\, s \doteq r \;\mathit{then}\, r \;\mathit{else}\, f(r)\;\mathit{fi}\displaystyle\big)\;\mathit{then}\\
&\quad\quad r\\
&\mathit{else}\\
&\quad\quad f\big(\mathit{if}\, s \doteq r \;\mathit{then}\, r \;\mathit{else}\, f(r)\;\mathit{fi}\big)\\
&\mathit{fi}\\
\text{“}\rightsquigarrow\text{”}\; &\big(s \doteq r \to \mathit{if}\, s \doteq r \;\mathit{then}\, r \;\mathit{else}\, f(r)\;\mathit{fi}\big)\land\\
&\big(s \neq r \to \mathit{if}\, s \doteq f(r) \;\mathit{then}\, r \;\mathit{else}\, f(f(r))\;\mathit{fi}\big)\\
\text{“}\rightsquigarrow\text{”}\; &\big(s \doteq r \land s \doteq r \to r\big)\land\\
&\big(s \doteq r \land s \neq r \to f(r)\big)\land\\
&\big(s \neq r \land s \doteq f(r) \to r\big)\land\\
&\big(s \neq r \land s \neq f(r) \to f(f(r))\big)\\
\text{“}\rightsquigarrow\text{”}\; &\big(s \doteq r \to r\big)\land\\
&\big(s \neq r \land s \doteq f(r) \to r\big)\land\\
&\big(s \neq r \land s \neq f(r) \to f(f(r))\big)
\end{aligned}
$$

As apparent from the effect of this example, the *update (match)* (R27) and *update (promote)* (R28) rules essentially perform bottom-up term rewriting, i.e. they start to determine the effect that an update has on the innermost term and proceed to promote the effect of the update to the top-level symbols. This bottom-up rewriting should not trespass intermediate updates, however. Therefore, we choose to avoid technical difficulties in the formulation of the R27 rule and accept the formal duplication of the subterm $\langle U\rangle u$ during application of the term rewrite rules.

The case $s \doteq r \land s \neq r$ of the above example can further be prevented in advance by R51.

Likewise reasoning as in the case of $\langle f(s) \lhd r\rangle f(f(r))$ applies in the case $s \doteq o \;\vdash\; \langle f(s) \lhd r\rangle f(f(s))$ but for reasons of semantical instead of purely syntactical equality. $\square$

Since – with some care during conditional term branching – the disadvantages of early branching outbalance the disadvantages of multiplying reasons for case distinction by update promotion, ODL uses last-win semantics for clashes. Last-win semantics may seem odd at first sight, but it allows a very simple formulation of merging and seems to have the best trade-off as far as the matters of branching are concerned. Of course this is at the expense of a very technical *update (match)* (R30) for simultaneous updates. Unfortunately, like some other clash semantics, last-win complicates update equality because the order of modifications in an update may be important.

Example 4.2.5 The following formula is untrue in general for last-win semantics.

$$\langle f(s) \lhd t, f(s') \lhd t' \rangle \phi \to \langle f(s') \lhd t', f(s) \lhd t \rangle \phi$$

In other words, closing by update equality[9] relative to a formula ϕ often requires a lot of case distinctions to be made. In the above example, distinguishing between the cases $s \doteq s'$ and $s \neq s'$ would be necessary. Except for special cases of ϕ, closing the above conjecture is only possible in the two cases $s \neq s'$ and $s \doteq s' \wedge t \doteq t'$. Yet, clash semantics that make case distinctions at the time of merging have no advantage either, because they will already have produced precisely those two branches earlier. □

4.2.6 Supplementary

Fig. 4.8 presents optional inference rules that can be emulated by a complete calculus. Nevertheless practical evidence encourages using them for reasons of feasible theorem proving.

Proposition 4.2.9 *Modus ponens is a derived inference rule.*

(R55) *modus ponens*

$$\frac{}{\phi, \phi \to \psi \ \vdash \psi}$$

Proof: The derivation can be achieved by R4 and R11 as follows.

$$\frac{\dfrac{}{\phi \ \vdash \phi, \psi} \quad \dfrac{}{\phi, \psi \ \vdash \psi}}{\phi, \phi \to \psi \ \vdash \psi}$$

■

[9]without the usual update promotion, cuts or induction.

(R48) update occurrence

$$\langle \mathcal{U} \rangle \phi \ \rightsquigarrow \ \langle \mathcal{U}' \rangle \phi$$

$\Leftarrow \mathcal{U}'$ equals $\mathcal{U}$ with locations not occurring admissibly in ϕ removed, i.e. remove $f(s) \lhd t$ iff $\phi_f^{f'} = \phi$ but never remove updates to $\texttt{next}_{\texttt{C}}$.

(R49) update deletion

$$\langle \ldots, f(s) \lhd t, \ldots, f(s) \lhd t' \rangle \ \rightsquigarrow \ \langle \ldots, f(s) \lhd t' \rangle$$

(R50) update no-op

$$\langle \mathcal{U}, f(s) \lhd f(s), \ldots, \rangle \ \rightsquigarrow \ \langle \mathcal{U}, \ldots \rangle$$

$\Leftarrow f(s') \lhd t'$ not in $\mathcal{U}$

(R51) conditional term known

$$\frac{e \ \vdash \phi(s)}{e \ \vdash \phi(\mathit{if}\, e \ \mathit{then}\, s \ \mathit{else}\, t \ \mathit{fi})}$$

(R52) conditional term known

$$\frac{\vdash e, \phi(t)}{\vdash e, \phi(\mathit{if}\, e \ \mathit{then}\, s \ \mathit{else}\, t \ \mathit{fi})}$$

(R53) conditional reconcile

$\mathit{if}\, \chi \ \mathit{then}\, s \ \mathit{else}\, \phi(\mathit{if}\, \chi \ \mathit{then}\, s' \ \mathit{else}\, t \ \mathit{fi}) \ \mathit{fi} \ \rightsquigarrow$
$\mathit{if}\, \chi \ \mathit{then}\, s \ \mathit{else}\, \phi(t) \ \mathit{fi}$

(R54) `new` identity

$$\frac{}{n \neq m \ \vdash \texttt{obj}_{\texttt{C}}(n) \neq \texttt{obj}_{\texttt{C}}(m)}$$

(R55) *modus ponens*

$$\frac{}{\phi, \phi \rightarrow \psi \ \vdash \psi}$$

Figure 4.8: Optional Inference Rules

Proposition 4.2.10 *The following inference rule is a derived rule*

(R54) $\mathtt{new}$ identity

$$\frac{}{n \neq m \;\vdash \mathtt{obj}_{\mathtt{C}}(n) \neq \mathtt{obj}_{\mathtt{C}}(m)}$$

Proof: The inference rules derives after inserting the axiom

$$A \quad := \quad \forall i{:}\mathtt{nat}\ \forall j{:}\mathtt{nat}\ (\mathtt{obj}_{\mathtt{C}}(i) \doteq \mathtt{obj}_{\mathtt{C}}(j) \rightarrow i \doteq j)$$

from R41 into the antecedent with the help of Prop. A.2.2. The proof can finally be concluded with R55.

$$\begin{array}{c}
\hline
\mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{C}}(m) \rightarrow n \doteq m,\ \mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{C}}(m)\ \vdash n \doteq m \\
\hline
A,\ \mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{C}}(m)\ \vdash n \doteq m \\
\hline
A,\ n \neq m\ \vdash \mathtt{obj}_{\mathtt{C}}(n) \neq \mathtt{obj}_{\mathtt{C}}(m) \\
\hline
n \neq m\ \vdash \mathtt{obj}_{\mathtt{C}}(n) \neq \mathtt{obj}_{\mathtt{C}}(m)
\end{array}$$

■

Proposition 4.2.11 *The following inference rule is a derived rule*

$$\frac{}{\vdash \forall x{:}C\ \ \phi\ \leftrightarrow\ \forall n{:}\mathtt{nat}\ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \rightarrow \phi)}$$

Proof: The derivation of the equivalence will be split into two implications. The first implication is immediate and does not even need the axiom.

$$\begin{array}{c}
\hline
\forall x{:}C\ \ \phi, \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)}, x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N)\ \vdash \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)} \\
\hline
\forall x{:}C\ \ \phi, \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)}, x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N)\ \vdash \phi \\
\hline
\forall x{:}C\ \ \phi, x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N)\ \vdash \phi \\
\hline
\forall x{:}C\ \ \phi\ \vdash x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N) \rightarrow \phi \\
\hline
\forall x{:}C\ \ \phi\ \vdash \forall n{:}\mathtt{nat}\ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \rightarrow \phi) \\
\hline
\vdash \forall x{:}C\ \ \phi\ \rightarrow\ \forall n{:}\mathtt{nat}\ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \rightarrow \phi)
\end{array}$$

In this derivation the last two inferences of universal quantifier resolution and equality application are not literally possible. This is because we have omitted the splitting according to the disjunctive definition of $x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N)$. However, splitting only leads to a finite number of branches containing $\mathtt{obj}_{\mathtt{D}}(N)$ instead of $\mathtt{obj}_{\leq \mathtt{C}}(N)$ and completing like the above derivation.

The derivation of the other implication starts with a R5 to introduce the axiom $A := \forall o{:}\mathtt{C}\ \exists n{:}\mathtt{nat}\ o \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)$, which results from R43, into the antecedent according to Prop. A.2.2.

$$\begin{array}{c}
\hline
x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N), x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N) \to \phi \ \vdash \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)} \\
\hline
\exists n{:}\mathtt{nat}\ \ x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n), x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N) \to \phi \ \vdash \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)} \\
\hline
A, X \doteq \mathtt{obj}_{\leq \mathtt{C}}(N), x \doteq \mathtt{obj}_{\leq \mathtt{C}}(N) \to \phi \ \vdash \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)} \\
\hline
A, X \doteq \mathtt{obj}_{\leq \mathtt{C}}(N), \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \vdash \phi_x^{\mathtt{obj}_{\leq \mathtt{C}}(N)} \\
\hline
A, X \doteq \mathtt{obj}_{\leq \mathtt{C}}(N), \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \vdash \phi_x^{X} \\
\hline
A, \exists n{:}\mathtt{nat}\ \ X \doteq \mathtt{obj}_{\leq \mathtt{C}}(n), \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \vdash \phi_x^{X} \\
\hline
A, \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \vdash \phi_x^{X} \\
\hline
A, \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \vdash \forall x{:}C\ \ \phi \\
\hline
A \ \vdash \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \to \ \forall x{:}C\ \ \phi \\
\hline
\vdash \forall n{:}\mathtt{nat}\ \ (x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi) \ \to \ \forall x{:}C\ \ \phi
\end{array}$$

For notational reasons, quantifier inference rules have been applied with weakening whenever one instance is sufficient during the proof. The proof finally closes by a combination of R16 and R55. Still, for the substitution $[x \mapsto \mathtt{obj}_{\leq \mathtt{C}}(N)]$ to be well-defined, we need the same branching into cases where $\mathtt{obj}_{\mathtt{D}}(N)$ occurs instead of $\mathtt{obj}_{\leq \mathtt{C}}(N)$. Again, this branching has been left out for ease of notation. ∎

Example 4.2.6 For the `new` *generated* (R43) rule, there are several alternatives. For example, with the `new` *generated* (R43) inference rule, the following induction scheme for the generated terms derives from the integer induction scheme.

$$\frac{\vdash x \doteq \mathtt{obj}_{\leq \mathtt{C}}(0) \to \phi \quad x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \to \phi \ \vdash x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n+1) \to \phi}{\vdash \forall x{:}C\ \ \phi}$$

where n is a new rigid constant. Formal derivations of alternative inference rules for generated semantics in object creation look as follows, with an adequate generalisation to $\mathtt{obj}_{\leq \mathtt{C}}(n)$ instead of $\mathtt{obj}_{\mathtt{C}}(n)$ by branching.

$$\frac{\dfrac{\forall n{:}\mathtt{nat}\ \ \phi(\mathtt{obj}_{\mathtt{C}}(n)) \to \forall o{:}C\ \ \phi(o) \ \vdash}{\forall o{:}C\ \ \phi(o) \leftrightarrow \forall n{:}\mathtt{nat}\ \ \phi(\mathtt{obj}_{\mathtt{C}}(n)) \ \vdash} \qquad \dfrac{}{\vdash \forall o{:}C\ \ \phi(o) \leftrightarrow \forall n{:}\mathtt{nat}\ \ \phi(\mathtt{obj}_{\mathtt{C}}(n))}}{\vdash}$$

$\square$

4.3 Verification Examples

In this section a look is taken at some prototypical verification examples and the effect that ODL inference rules have on the proof. Each example highlights a particular aspect of object-oriented programs deserving special attention in ODL. In order to retain readable proofs the size of the example programs is kept as compact as possible. Larger examples would require a linear notation that is less fragile for longer proofs and higher levels of branching, but also less intuitive.

To overcome space limitations in the formal notation the next remark introduces a compact notation, which permits to abbreviate program statements occurring within modalities of the examples.

Remark 4.3.1 (Label Abbreviation Context) *In verification examples, program statements will be abbreviated by some unique prefix, i.e. in a program where only one statement begins with* $\ulcorner b \lhd \urcorner$ *only write* $\ulcorner b \lhd \urcorner$ *instead of the statement* $b \lhd \mathtt{new}\,\mathtt{C}()$*. Further the ellipsis* $\ulcorner \ldots \urcorner$ *abbreviates the right context of all successor statements.*

For example, the subprogram $c \lhd \mathtt{new}\,\mathtt{C}(); c.x \lhd b.x + 2$ of the program in Ex. 4.3.4 is abbreviated as $\ulcorner c \lhd \ldots \urcorner$ where the label $\ulcorner c \lhd \urcorner$ uniquely identifies the starting statement of the program fragment.

Example 4.3.1 Continuing Ex. 2.4.1, consider the following simple ODL program α.

$$\mathtt{if}(2 \mid c)\,\{c \lhd c + 2\}\mathtt{else}\{c \lhd c + 1\}$$

It is to be investigated why the following formula is a correct specification of α.

$$c_0 \doteq c \;\rightarrow\; \langle \alpha \rangle c \doteq 2 \cdot (c_0 \div 2) + 2$$

When abbreviating $c \doteq 2 \cdot (c_0 \div 2) + 2$ by ϕ and $\neg(x \mid y)$ by $x \nmid y$, the correctness proof looks as follows.

$$\dfrac{\dfrac{\dfrac{\overline{2 \mid c_0 \;\vdash c_0 + 2 \doteq 2 \cdot (c_0 \div 2) + 2}}{2 \mid c, c_0 \doteq c \;\vdash c + 2 \doteq 2 \cdot (c_0 \div 2) + 2}}{2 \mid c, c_0 \doteq c \;\vdash \langle c \lhd c + 2 \rangle \phi} \qquad \dfrac{\dfrac{\overline{2 \nmid c_0 \;\vdash c_0 + 1 \doteq 2 \cdot (c_0 \div 2) + 2}}{2 \nmid c, c_0 \doteq c \;\vdash c + 1 \doteq 2 \cdot (c_0 \div 2) + 2}}{2 \nmid c, c_0 \doteq c \;\vdash \langle c \lhd c + 1 \rangle \phi}}{c_0 \doteq c \;\vdash \langle \mathtt{if}(2 \mid c) \ldots \rangle c \doteq 2 \cdot (c_0 \div 2) + 2}$$

After application of R16, the equation has been hidden by R6 to improve readability. Finally the proof closes by elementary arithmetic. The second inference involves auxiliary rewrite computations for update application. Abbreviating $\langle c \lhd c+1 \rangle$ by $\langle \mathcal{U} \rangle$, one of those update computations looks – in full circumstantiality – as follows.

$$\begin{aligned}
&\langle \mathcal{U} \rangle (c \doteq 2 \cdot (c_0 \div 2) + 2) \\
\rightsquigarrow\ & (\langle \mathcal{U} \rangle c) \doteq (\langle \mathcal{U} \rangle (2 \cdot (c_0 \div 2) + 2)) \\
\rightsquigarrow\ & c + 1 \doteq \langle \mathcal{U} \rangle (2 \cdot (c_0 \div 2)) + \langle \mathcal{U} \rangle 2 \\
\rightsquigarrow\ & c + 1 \doteq \langle \mathcal{U} \rangle 2 \cdot \langle \mathcal{U} \rangle (c_0 \div 2) + 2 \\
\rightsquigarrow\ & c + 1 \doteq 2 \cdot (\langle \mathcal{U} \rangle c_0 \div \langle \mathcal{U} \rangle 2) + 2 \\
\rightsquigarrow\ & c + 1 \doteq 2 \cdot (c_0 \div 2) + 2
\end{aligned}$$

□

Example 4.3.2 (Update Sequence) To examine the effect of updates sequences, consider the following conjecture about a program involving a sequence of updates.

$$\begin{aligned}
&b \neq c \rightarrow \\
&\langle b.x \lhd b.x + 1; \\
&\ b.x \lhd b.x + 3; \\
&\ c.x \lhd b.x + 2; \\
&\ c.x \lhd b.x + 5 \rangle c.x \doteq b.x + 5
\end{aligned}$$

In order to isolate other effects like object creation from the update sequence aspect, this conjecture is subject to a premise that the references involved denote distinct objects. When abbreviating $c.x \doteq b.x + 5$ by ϕ the proof of the above conjecture looks as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash b.x + 9 \doteq b.x + 4 + 5}}{\vdash \langle b.x \lhd b.x + 4, c.x \lhd b.x + 9 \rangle \phi}}{\vdash \langle b.x \lhd b.x + 4, c.x \lhd b.x + 2 \rangle \langle c.x \lhd b.x + 5 \ldots \rangle \phi}}{\vdash \langle b.x \lhd b.x + 4 \rangle \langle c.x \lhd b.x + 2 \ldots \rangle \phi}}{\vdash \langle b.x \lhd b.x + 1 \rangle \langle b.x \lhd b.x + 3 \ldots \rangle \phi}}{\vdash \langle b.x \lhd b.x + 1 \ldots \rangle c.x \doteq b.x + 5}$$

This proof has been concluded with an update application rewrite using the assumed premise that $b \neq c$, which is important for the proof to succeed.

$$\langle \mathcal{U} \rangle \quad := \quad \langle b.x \lhd b.x + 4, c.x \lhd b.x + 9 \rangle$$

$$
\begin{aligned}
& \langle\mathcal{U}\rangle(b.x) \\
\rightsquigarrow\ & \mathit{if}\, c \doteq \langle\mathcal{U}\rangle b\ \mathit{then}\, b.x + 9\ \mathit{else}\, b.x + 4\ \mathit{fi} \\
\rightsquigarrow\ & \mathit{if}\, \mathit{false}\ \mathit{then}\, b.x + 9\ \mathit{else}\, b.x + 4\ \mathit{fi} \\
\rightsquigarrow\ & b.x + 4
\end{aligned}
$$

Table 4.1: Measurements for the Update Sequence Ex. 4.3.2, cf. Chapt. C

Calculus	Inferences	Branches	Duration
*i*ODL	20	1	0.1s
*i*ODL +mo	24	1	0s
JavaCardDL nomo	82	1	0.3s
JavaCardDL mo	86	1	0.3s

The measurements in Tab. 4.1 demonstrate that – as expected – the ODL treatment of updates produces less overhead than JavaCardDL, in this case by a factor of 4. The zero seconds for *i*ODL +mo[10] is presumably an artefact of the imprecise time measuring technique. By the nature of *i*ODL and the automatic *i*ODL theorem prover, the *i*ODL measurements usually report more inferences than the ODL proof. This is also due to the fact that the automatic theorem prover does not attempt to find the shortest possible proof but is content to produce some proof as quick as possible. □

Example 4.3.3 (XviD Motion Compensation Sequence) In order to investigate the scalability of the update approach, consider the conjecture below about an excerpt of the XviD (XviD, 2004) motion compensation module. This excerpt, α, has been regarded as a representative XviD function for the purposes of generating IA32/SSE2 executions in (Hack *et al.*, 2004). It demonstrates the effects of a long sequence of updates to the prover performance.

$$
\begin{aligned}
& sse = 0; \\
& sse = sse + (b(0) - c(0)) \cdot (b(0) - c(0)); \\
& sse = sse + (b(1) - c(1)) \cdot (b(1) - c(1)); \\
& sse = sse + (b(2) - c(2)) \cdot (b(2) - c(2)); \\
& sse = sse + (b(3) - c(3)) \cdot (b(3) - c(3)); \\
& sse = sse + (b(4) - c(4)) \cdot (b(4) - c(4)); \\
& sse = sse + (b(5) - c(5)) \cdot (b(5) - c(5)); \\
& sse = sse + (b(6) - c(6)) \cdot (b(6) - c(6)); \\
& sse = sse + (b(7) - c(7)) \cdot (b(7) - c(7))
\end{aligned}
$$

[10]Cf. Chapt. C for a description of the merge optimisation variant "mo".

$$b.length > 7 \land c.length > 7 \to \langle\alpha\rangle sse \doteq \sum_{i=0}^{7}(b(i) - c(i)) \cdot (b(i) - c(i))$$

Table 4.2: Measurements for excerpt of XviD Motion Compensation Sequence Ex. 4.3.3, cf. Chapt. C. The column vc specifies whether explicit field access validation checks have been added.

vc	Calculus	Inferences	Branches	Duration
no	iODL	80	1	1s
no	iODL +mo	80	1	1.1s
no	JavaCardDL nomo	1976	33	46s
no	JavaCardDL mo	1987	33	46.6s
yes	iODL	105	3	1.2s
yes	JavaCardDL nomo	2046	35	46.6s

The measurements in Tab. 4.2 reveal a factor of 46 in speed and as much as 33 in branching for the simplified iODL calculus in comparison to JavaCardDL. To some extent, this tremendous performance improvement can, of course, be explained by the finer JavaCardDL treatment of array access. When considering the variant `if`$(b.length > 7 \land c.length > 7)\,\{\alpha\}$ with field access validation checks, the proportion hardly changes, though, which demonstrates a serious potential for optimisations by combining repeated field access validation checks. From this example, opting for a separate and more global treatment of field access validation using standard static analysis technology from compiler construction whenever possible seems favourable. Incorporating compiler construction technology into the proving process is straightforward for the modular iODL approach but could turn out to be comparably cumbersome for JavaCardDL. □

Example 4.3.4 (Object Creation) For a program involving object creation, consider the following conjecture.

$$\begin{array}{l}\langle b \lhd \mathtt{new\,C}();\\ \;b.x \lhd b.x + 1;\\ \;c \lhd \mathtt{new\,C}();\\ \;c.x \lhd b.x + 2\rangle c.x \doteq b.x + 2\end{array}$$

During the proof it is essential to infer that b and c are distinct non-aliased objects by nature of their different sources of creation. This is important for the attribute modification update $c.x \lhd b.x + 2$ in order to discover that it has no effect on $b.x$, since otherwise the conjecture $c.x \doteq b.x + 2$ would be false.

For the formal proof, let ϕ abbreviate $c.x \doteq b.x + 2$. Let n abbreviate $\mathtt{next}_{\mathtt{C}}$ and $o(z)$ abbreviate $\mathtt{obj}_{\mathtt{C}}(z)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\vdash o(n).x + 3 \doteq o(n).x + 1 + 2}}{\vdash \langle b \triangleleft o(n), c \triangleleft o(n+1), o(n).x \triangleleft o(n).x + 1, o(n+1).x \triangleleft o(n).x + 1 + 2\rangle \phi}}{\vdash \langle n \triangleleft n + 2, b \triangleleft o(n), c \triangleleft o(n+1), o(n).x \triangleleft o(n).x + 1\rangle \langle c.x \triangleleft \ldots\rangle \phi}}{\vdash \langle n \triangleleft n + 1, b \triangleleft o(n), o(n).x \triangleleft o(n).x + 1\rangle \langle c \triangleleft \ldots\rangle \phi}}{\vdash \langle n \triangleleft n + 1, b \triangleleft o(n)\rangle \langle b.x \triangleleft \ldots\rangle \phi}}{\vdash \langle b \triangleleft \ldots\rangle c.x \doteq b.x + 2}$$

The last inference involves two auxiliary rewrite computations for update application. Abbreviate with $\langle \mathcal{U} \rangle$ the following update.

$$\langle b \triangleleft o(n), c \triangleleft o(n+1), o(n).x \triangleleft o(n).x + 1, o(n+1).x \triangleleft o(n).x + 3\rangle$$

What is important to determine here is that the update to $o(n+1).x$ cannot have an effect on the value of $b.x$ because b and $o(n+1)$ are no aliases. R41 establishes this information about aliasing because of $o(n+1) \neq o(n)$ and conclude that

$$\begin{array}{ll}
 & \langle \mathcal{U} \rangle (b.x) \\
\rightsquigarrow & \mathit{if}\, o(n+1) \doteq \langle \mathcal{U} \rangle b \;\mathit{then} \\
 & \quad\quad o(n).x + 3 \\
 & \mathit{else} \\
 & \quad\quad \mathit{if}\, o(n) \doteq \langle \mathcal{U} \rangle b \;\mathit{then}\, o(n).x + 1 \;\mathit{else}\, b.x \;\mathit{fi} \\
 & \mathit{fi} \\
\rightsquigarrow & \mathit{if}\, o(n+1) \doteq o(n) \;\mathit{then} \\
 & \quad\quad o(n).x + 3 \\
 & \mathit{else} \\
 & \quad\quad \mathit{if}\, o(n) \doteq o(n) \;\mathit{then}\, o(n).x + 1 \;\mathit{else}\, b.x \;\mathit{fi} \\
 & \mathit{fi} \\
\rightsquigarrow & \mathit{if}\, \mathit{false} \;\mathit{then}\, o(n).x + 3 \;\mathit{else}\, o(n).x + 1 \;\mathit{fi} \\
\rightsquigarrow & o(n).x + 1
\end{array}$$

A similar update application shows

$$\begin{array}{ll}
 & \langle \mathcal{U} \rangle (c.x) \\
\rightsquigarrow & \mathit{if}\, o(n+1) \doteq \langle \mathcal{U} \rangle c \;\mathit{then}\, o(n).x + 3 \;\mathit{else}\, \ldots \;\mathit{fi} \\
\rightsquigarrow & \mathit{if}\, o(n+1) \doteq o(n+1) \;\mathit{then}\, o(n).x + 3 \;\mathit{else}\, \ldots \;\mathit{fi} \\
\rightsquigarrow & \mathit{if}\, \mathit{true} \;\mathit{then}\, o(n).x + 3 \;\mathit{else}\, \ldots \;\mathit{fi} \\
\rightsquigarrow & o(n).x + 3
\end{array}$$

Table 4.3: Measurements for the Object Creation Ex. 4.3.4, cf. Chapt. C

Calculus	Inferences	Branches	Duration
iODL	27	2	0.1s
iODL +mo	33	2	0.1s
JavaCardDL nomo	1359	44	4.9s
JavaCardDL mo	1515	63	5.4s

As the measurements in Tab. 4.3 demonstrate impressively, the treatment of object creation by object enumerators is by far superior to the JavaCardDL approach. Factors of about 50 in both number of inferences and runtime clearly illustrate this fact. And factors of 20 to 30 in branching promise an even higher degree in scalability. Imagine the effect of an exceedingly complex program starting with α. If by the JavaCardDL treatment, after a partial proof of the above program α 20 branches remain with the complex program, then the prover is bound to lose much time analysing the program on several branches. By the ODL object enumerator approach, though, at most two branches of the proof require a further analysis of the complex program. □

Example 4.3.5 (Transitive Object Creation)

$$\begin{array}{l}\langle b \lhd \texttt{new}\,\texttt{C}();\\ \ b.x \lhd b.x + 1;\\ \ c \lhd \texttt{new}\,\texttt{C}();\\ \ c.x \lhd b.x + 2;\\ \ d \lhd \texttt{new}\,\texttt{C}();\\ \ d.x \lhd c.x + 3\rangle d.x \doteq b.x + 5\end{array}$$

Table 4.4: Measurements for the Transitive Object Creation Ex. 4.3.5, cf. Chapt. C

Calculus	Inferences	Branches	Duration
iODL	44	3	0.1s
JavaCardDL nomo	> 4301	> 151 (3 open)	> 17.8s

In comparison to Ex. 4.3.4, when scaling the number of object creations involved from two to three, the advantage of the ODL object enumerator representation over list-based representation of object creation is even more drastic, as Tab. 4.4 shows. The inability of KeY for finally proving the example is due to a known problem with nonadjacent object creation, though. Yet, there also is a fundamental limitation with the list-based representation. Comparing two objects for inequality involves traversing the list of created objects and proving that they occur at different positions. In KeY for JAVA-CARDDL, this essentially amounts to traversing the creation list from one object and proving that the one created later will be found after some finite number of list traversions, which is an inherently dynamic property. Even intuitively, stating about 13 and 29 that they are different numbers, because of which bijections map them to distinct objects, is simpler than expressing a circumstance like

> If, after traversing the object generation chain 13 times to reach object A, the next attribute of the creation list is followed another 16 times, object B will be found, from which their inequality can be deduced – by the noncircular nature of the object list and the unicity of its content.

The common concept underlying both the arithmetic ODL approach to object creation and a list-based representation consists of a device for *generating* new objects from the preexisting ones. In the case of object enumerators those new objects will be generated – indirectly – from the constant 0 and the successor function on natural numbers, while list-based representations utilise the head of the chain of generatable objects and the list link for the same purpose. Even though at this level, there is no difference in essence, there is one of representation. Contrary to lists, natural numbers permit a more elaborate representation than just successor cascades, and the standard prover capabilities are usually far more mature for arithmetics. Moreover, natural numbers possess the most basic generators of infinite sets. □

Example 4.3.6 (XviD Motion Compensation Function) So to say, as a "wild type" or real-world example, consider a conjecture about a complete function from the XviD (XviD, 2004) motion compensation module. The ODL program α further below is a rough translation of the `sse8_16bit_c` function in the `motion/sad.c` source code file and surrounds the program from Ex. 4.3.3 by a loop with stride, i.e. non-adjacent traversal through memory, which has dramatic effects on the performance of caching and pipelines. Depending on very strict hardware parameters, a stride can improve or degrade overall performance.

$$b.length > 63 \land c.length > 63 \rightarrow \langle\alpha\rangle sse \doteq \sum_{i=0}^{63}(b(i) - c(i)) \cdot (b(i) - c(i))$$

$$
\begin{array}{l}
sse = 0; \\
\texttt{int}\ n = 8; \\
\texttt{int}\ stride = 8; \\
\texttt{int}\ i = 0; \\
k = 0; \\
\texttt{while}(k < n)\,\{ \\
\quad sse = sse + (b[i+0] - c[i+0]) \cdot (b[i+0] - c[i+0]); \\
\quad sse = sse + (b[i+1] - c[i+1]) \cdot (b[i+1] - c[i+1]); \\
\quad sse = sse + (b[i+2] - c[i+2]) \cdot (b[i+2] - c[i+2]); \\
\quad sse = sse + (b[i+3] - c[i+3]) \cdot (b[i+3] - c[i+3]); \\
\quad sse = sse + (b[i+4] - c[i+4]) \cdot (b[i+4] - c[i+4]); \\
\quad sse = sse + (b[i+5] - c[i+5]) \cdot (b[i+5] - c[i+5]); \\
\quad sse = sse + (b[i+6] - c[i+6]) \cdot (b[i+6] - c[i+6]); \\
\quad sse = sse + (b[i+7] - c[i+7]) \cdot (b[i+7] - c[i+7]); \\
\quad i = i + stride; \\
\quad k = k + 1 \\
\}
\end{array}
$$

To improve readability of the expanded version of the abbreviation of the sum, a modality has been used. Furthermore, this supports a generalisation to arbitrary lengths by induction, which is not difficult. Finally, a modality with the following program essentially compares the results of a program with stride to one without stride. Regardless of the performance gains, if the introduction of a stride altered the result of the computation, its use would be disastrous.

$$
\begin{array}{l}
sum = 0; \\
k = 0; \\
\texttt{while}(k < 64)\,\{ \\
\quad sum = sum + (b[k] - c[k]) \cdot (b[k] - c[k]); \\
\quad k = k + 1 \\
\}
\end{array}
$$

With the above auxiliary specification program γ, the mechanical proof uses a slightly different specification:

$$b.length > 63 \land c.length > 63 \land \langle\gamma\rangle v \doteq sum \rightarrow \langle\alpha\rangle sse \doteq sum$$

Tab. 4.5 contains performance measurements for different values of n and corresponding adaptations of the other values. A comparison of the performance displays factors of about 35 in speed and 20 in number of inferences, independent of n. □

Table 4.5: Measurements for a function in XviD Motion Compensation Ex. 4.3.6, cf. Chapt. C

n	Calculus	Inferences	Branches	Duration
2	iODL	719	21	15s
2	JavaCardDL nomo	14867	282	543.2s
3	iODL	1188	30	40.2s
3	JavaCardDL nomo	22600	425	1373.2s
8	iODL	4689	75	1569.7s
8	JavaCardDL nomo	stack overflow		

Example 4.3.7 (Parametric Loop) Consider the following program, α, with a very simple loop of an effect depending on the particular value of the natural number variable n.

$$\begin{array}{l} s \lhd 0; \\ i \lhd 0; \\ \texttt{while}(i < n)\,\{ \\ \quad s \lhd s + i; \\ \quad i \lhd i + 1 \\ \} \end{array}$$

The following formula is a correct specification of α.

$$[\alpha] s \doteq n \cdot (n-1) \div 2$$

When abbreviating $s \doteq n \cdot (n-1) \div 2$ by ϕ, the conjecture can be proven as follows, with $i \leq n \wedge s \doteq i \cdot (i-1) \div 2$ as invariant p for proving the behaviour of the loop.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 0 \leq 0 \wedge 0 \doteq 0 \div 2}}{\vdash \langle s \lhd 0, i \lhd 0 \rangle p} \qquad \dfrac{}{p, i < n \ \vdash [s \lhd s + i \ldots] p} \qquad \dfrac{}{p, \neg(i < n) \ \vdash \phi}}{\vdash \langle s \lhd 0, i \lhd 0 \rangle [\texttt{while}(i < n)\,\{\} \ldots] \phi}}{\vdash \langle s \lhd 0 \rangle [i \lhd \ldots] \phi}}{\vdash [s \lhd \ldots] \phi}$$

The right branch can be concluded by R16 after a *cut* (R5) of the equation $i \doteq n$, as deduced from the formulas $i \leq n$ and $\neg(i < n)$. The remaining middle branch can be closed by the following reasoning with the help of R27, R16 and simple arithmetics.

$$\begin{array}{c} \hline p, i < n \ \vdash i + 1 \leq n \ \wedge \ i \cdot (i-1) \div 2 + i \doteq (i+1) \cdot i \div 2 \\ \hline p, i < n \ \vdash i + 1 \leq n \ \wedge \ s + i \doteq (i+1) \cdot i \div 2 \\ \hline p, i < n \ \vdash \langle s \lhd s + i, i \lhd i + 1 \rangle p \\ \hline p, i < n \ \vdash \langle s \lhd s + i \rangle [i \lhd i + 1 \ldots] p \\ \hline p, i < n \ \vdash [s \lhd s + i \ldots] p \end{array}$$

The required arithmetic reasoning involves one mathematical fact.

$$i \cdot (i-1) \div 2 + i = i \cdot (i-1) \div 2 + i \cdot 2 \div 2 = (i+1) \cdot i \div 2$$

□

Example 4.3.8 (Complicated Loop) Contrary to exception handling approaches like the one in (Beckert & Sasse, 2001), the ODL emphasis on simple structure facilitates inclusion of the *composition* (R22) rule. This rule would be incorrect in the presence of exception handling environments. Consider the following more involved program α.

$$\begin{array}{l} \texttt{while}(c.x < d.x)\,\{ \\ \quad c.x \lhd c.x + 1; \\ \quad c \lhd c.next; \\ \quad d.x \lhd d.x - 1; \\ \quad d \lhd d.previous \\ \}; \\ c.x \lhd 1 + 2 + 3 \end{array}$$

Proving general statements about α is tremendously complicated, and almost always needs an induction to treat the loop, which also has an effect that is slightly difficult to describe. Fully automatic treatment of α is difficult. Yet, consider the simple conjecture $[\alpha]c.x \doteq 6$, which has a very simple proof using *composition* (R22). Moreover, this proof has been found fully automatically almost instantly.[11]

$$\begin{array}{c} \hline \vdash \mathit{true} \\ \hline \vdash [\texttt{while}(c.x < d.x)\,\{\ldots\}]\mathit{true} \\ \hline \vdash [\texttt{while}(c.x < d.x)\,\{\ldots\}]6 \doteq 6 \\ \hline \vdash [\texttt{while}(c.x < d.x)\,\{\ldots\}][c.x \lhd 1 + 2 + 3]c.x \doteq 6 \\ \hline \vdash [\alpha]c.x \doteq 6 \end{array}$$

[11] Using the derived modal tautology $[\alpha]\mathit{true} \equiv \mathit{true}$

Of course, proving this conjecture would also be possible within the JavaCardDL calculus. However, because JavaCardDL has no composition rule the prover has to treat the loop first. On account of the unbounded nature of the loop in α, this is only possible by explicit induction. But fully automatic treatment of inductions by R45 is hindered due to the choice of the right induction hypothesis. In this case, a choice of *true* would suffice after which the proof can still be closed automatically in a straightforward way.

Table 4.6: Measurements for the Complicated Loop Ex. 4.3.8, cf. Chapt. C

Calculus	Inferences	Branches	Duration
*i*ODL	12	1	0s
JavaCardDL nomo	∞	∞	∞
JavaCardDL mo	∞	∞	∞

Generalisation of this example for more complex loops and successor statements that need some particular but not all information about the program state reached after the loop is possible. Then the *composition* (R22) rule helps to direct the attention to the loop information really needed in the remaining conjecture about the successor statements. In some sense, it allows to choose a behaviour comparable to the wp-calculus (Dijkstra, 1976) by isolating loops in order to reconcile the remaining conjectures about them. Those conjectures can be focused on what is required by the successor statements and postcondition. □

4.4 Soundness

This section will establish the soundness proof for the ODL calculus. Soundness ensures the purely syntactical calculus to respect the actual language semantics. Soundness guarantees that all statements derivable from valid premises are really true, thereby relating syntax and truth. Unsound calculi do not (really) qualify for program verification systems, since we cannot trust their statements about programs, regardless of how much evidence the proof system is able to exhibit in support of the statement. Unsound proof systems would purport that defective programs have been verified as correct, leaving bugs in "verified" software. To prevent this, consider the following proof that the ODL calculus is sound.

Theorem 1 (Soundness) *The* ODL *calculus consisting of the inference rules presented in §4.2 is* sound, *i.e. for each* $\Gamma \subseteq \mathrm{Fml}(\Sigma \cup V)$ *for each* $\phi \in$

$\mathrm{Fml}(\Sigma \cup V)$

$$\Gamma \vdash \phi \quad \textit{implies} \quad \Gamma \vDash_g \phi$$

Proof: The proof constitutes of soundness proofs for each individual inference rule. For each inference rule we are bound to show that in whatever state and interpretation the premises of the inference rule holds, the conclusion holds as well. The inference rule

$$\frac{C_1 \quad \dots \quad C_n}{D}$$

is sound if and only if $C_1, \dots, C_n \vDash_g D$, which involves an implicit universal treatment of free variables. In fact, according to Rem. 2.3.15 it is also sufficient to prove a stronger statement of local consequence with respect to variables, which will be preferred whenever possible. The following sections contain those individual soundness proofs. ■

4.4.1 First-Order

The propositional and first-order inference rules are standard. Their soundness thus follows from the classical proofs since ODL does not change the semantics of propositional or first-order connectives. Further, constant domain semantics ensures that object creation does not interfere with the soundness proofs for quantifiers.

Proposition 4.4.1 *The quantifier inference rules are sound.*

Proof:

R18 Assuming for some state w of an interpretation ℓ that $\ell, w \vDash A_x^t$, we have to show that $\ell, w \vDash \exists x\, A$ holds as well. Using t as a "witness" for the circumstance that A is holding of something, the proof can be completed with the following argument. Since $\sigma := [x \mapsto t]$ must be admissible for the inference rule to be applicable, Lem. 2.5.6 allows to conclude that $\mathit{true} = \mathit{val}_\ell(w, A_x^t) = \mathit{val}_{\sigma^* \ell}(w, A)$, because of which $\ell, w \vDash \exists x\, A$ holds as expected.

R17 This proof relies on the fact that – due to its implicit universal nature – the new variable X allows for arbitrary interpretations. By definition of $\vDash$ in 2.3.10-2.3.11, free variables are implicitly treated universally[12].

[12]Equivalently, they experience an explicit universal closure.

Finally, thus, R17 just extends the range of the universal quantifier binding x to the (implicit) top-level and ensures by variable renaming that this does not cause conflicts with other parts of the sequent.

■

Likewise, with the ODL program execution structure following the execution order of WHILE, the standard inference rules for dynamic logic in §4.2.2 are sound as the classical proofs reveal. Neither reasons for abrupt completion nor labelled loops necessitate non-standard inference rules.[13]

Proposition 4.4.2 *The following inference rule is sound*

$$\text{(R16)} \quad \doteq \text{subst}$$
$$\frac{\Gamma^t_s, s \doteq t \ \vdash \Delta^t_s}{\Gamma, s \doteq t \ \vdash \Delta}$$

Proof: This conjecture is a generalised consequence of Prop. 2.5.7. For a proof note that (2.2) holds since by premise $val_\ell(w, s) = val_\ell(w, t)$. ■

Example 4.4.1 (Non-wary substitutive) Let us investigate what happens not paying attention to the wariness constraint in the $\doteq$ *subst* (R16) rule. Consider a formula Δ for which the substitution $[z \mapsto \mathtt{x}]$ would be non-admissible because of the trespassing of x beyond the update to x. x is a (non-rigid) program variable, and z a (rigid) logical variable. Application of the $\doteq$ *subst* (R16) inference rule under disregard of wariness would lead to:

$$\frac{\mathtt{x} \doteq 1, z \doteq \mathtt{x} \ \vdash \mathtt{x} > 0 \land \langle x \lhd x + 1 \rangle \mathtt{x} > 1}{z \doteq 1, z \doteq \mathtt{x} \ \vdash z > 0 \land \langle x \lhd x + 1 \rangle z > 1}$$

In case of $val_\ell(w, z) = val_\ell(w, \mathtt{x}) = 1$, the premise is true while the consequence is false. Put this in contrast to an inference respecting wariness, which would lead to the following sound consequence.

$$\frac{\mathtt{x} \doteq 1, z \doteq \mathtt{x} \ \vdash \mathtt{x} > 0 \land \langle x \lhd x + 1 \rangle z > 1}{z \doteq 1, z \doteq \mathtt{x} \ \vdash z > 0 \land \langle x \lhd x + 1 \rangle z > 1}$$

This shows that without wariness the $\doteq$ *subst* (R16) inference rule would be unsound.

[13]Refer to (Beckert & Sasse, 2001) for the more complicated inference rules in the presence of exception handling in JAVACARDDL.

When ignoring wariness constraints for the replacement term t, inference becomes unsound in a similar way. $\mathtt{x}$ is a program variable, and t a term. Without wariness the following unsound inference would become possible.

$$\frac{t \doteq 1, \mathtt{x} \doteq t \ \vdash t > 0 \land \langle x \lhd t - 1\rangle t > 0}{\mathtt{x} \doteq 1, \mathtt{x} \doteq t \ \vdash \mathtt{x} > 0 \land \langle x \lhd x - 1\rangle \mathtt{x} > 0}$$

When wariness is taken into account the following sound inference is taken instead.

$$\frac{t \doteq 1, \mathtt{x} \doteq t \ \vdash t > 0 \land \langle x \lhd t - 1\rangle \mathtt{x} > 0}{\mathtt{x} \doteq 1, \mathtt{x} \doteq t \ \vdash \mathtt{x} > 0 \land \langle x \lhd x - 1\rangle \mathtt{x} > 0}$$

□

Proposition 4.4.3 *The following inference rule is sound*

(R15) induction

$$\frac{\vdash \phi(0) \quad \phi(n) \ \vdash \phi(n+1)}{\vdash \forall n\, \phi(n)}$$

$\Leftarrow n$ new variable

Proof: The soundness of this inference rule essentially amounts to the induction principle of natural numbers:

$$\phi(0) \land \forall n\, (\phi(n) \to \phi(n+1)) \quad \to \quad \forall n\, \phi(n)$$

The introduction of a new free variable n is just another way of expressing universal quantification on sequent level instead of on formula level. ■

4.4.2 Term Rewriting

Proposition 4.4.4 *The following inference rules are sound*

(R37) conditional term split (left)

$$\frac{(e \to \phi(s)) \land (\neg e \to \phi(t)) \ \vdash}{\phi(\mathit{if}\, e\ \mathit{then}\, s\ \mathit{else}\, t\ \mathit{fi}) \ \vdash}$$

(R38) conditional term split (right)

$$\frac{\vdash (e \to \phi(s)) \land (\neg e \to \phi(t))}{\vdash \phi(\mathit{if}\, e\ \mathit{then}\, s\ \mathit{else}\, t\ \mathit{fi})}$$

Proof: Let s be any state in any interpretation ℓ. We have to show that whenever $\ell, s \vDash (e \rightarrow \phi(r)) \wedge (\neg e \rightarrow \phi(t))$ holds, so does

$$\ell, s \vDash \phi(\mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi})$$

As a stronger statement this proof even shows local equivalence, which establishes the soundness of both inference rules.

$$(e \rightarrow \phi(r)) \wedge (\neg e \rightarrow \phi(t)) \;\equiv\; \phi(\mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi})$$

The proof follows an induction on the structure of $\phi(u)$.

(I) $\phi(u) = f(u)$ with a predicate symbol $f \in \Sigma$ is the essential case[14]

$$\begin{aligned}
& \mathit{val}_\ell(w, \phi(\mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi})) \\
= \;& \mathit{val}_\ell(w, f)\big(\mathit{val}_\ell(w, \mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi})\big) \\
= \;& \mathit{val}_\ell(w, f)\left(\left\{ \begin{array}{ll} \mathit{val}_\ell(w, r) & \Leftarrow \mathit{val}_\ell(w, e) = \mathit{true} \\ \mathit{val}_\ell(w, t) & \Leftarrow \mathit{val}_\ell(w, e) = \mathit{false} \end{array} \right\}\right) \\
= \;& \left\{ \begin{array}{ll} \mathit{val}_\ell(w, f)\big(\mathit{val}_\ell(w, r)\big) & \Leftarrow \mathit{val}_\ell(w, e) = \mathit{true} \\ \mathit{val}_\ell(w, f)\big(\mathit{val}_\ell(w, t)\big) & \Leftarrow \mathit{val}_\ell(w, e) = \mathit{false} \end{array} \right\} \\
= \;& \mathit{val}_\ell(w, (e \rightarrow f(r)) \wedge (\neg e \rightarrow f(t))) \\
= \;& \mathit{val}_\ell(w, (e \rightarrow \phi(r)) \wedge (\neg e \rightarrow \phi(t)))
\end{aligned}$$

Likewise reasoning concludes the cases for propositional connectives.

(II) $\phi(u) = \forall x\, \psi(u)$. By premise, the substitution $[u \mapsto \mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi}]$ is first-order admissible for $\phi(u)$, implying that $x \notin FV(e)$. Then

$$\begin{aligned}
\phi(\mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi}) \quad &= \quad \forall x\, \psi(\mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi}) \\
&\overset{\text{IH}}{\equiv} \quad \forall x\, \big((e \rightarrow \psi(r)) \wedge (\neg e \rightarrow \psi(t))\big) \\
&\equiv \quad \forall x\, (e \rightarrow \psi(r)) \;\wedge\; \forall x\, (\neg e \rightarrow \psi(t)) \\
&\overset{x \notin FV(e)}{\equiv} \quad (e \rightarrow \forall x\, \psi(r) \;\wedge\; (\neg e \rightarrow \forall x\, \psi(t))
\end{aligned}$$

(III) $\phi(u) = [\alpha]\psi(u)$. By premise, the substitution $[u \mapsto \mathit{if}\, e\ \mathit{then}\, r\ \mathit{else}\, t\ \mathit{fi}]$ is

[14] Cases like $\phi(u) = f(u, t')$ or with function symbols f instead of predicate symbols follow the same argument.

denotation-preserving for $\phi(u) \Rightarrow e$ is rigid for α. Then

$$
\begin{array}{cl}
 & \phi(\mathit{if}\, e\, \mathit{then}\, r\, \mathit{else}\, t\, \mathit{fi}) \\
= & [\alpha]\psi(\mathit{if}\, e\, \mathit{then}\, r\, \mathit{else}\, t\, \mathit{fi}) \\
\overset{\text{IH}}{\equiv} & [\alpha]\big((e \to \psi(r)) \land (\neg e \to \psi(t))\big) \\
\equiv & [\alpha](e \to \psi(r)) \ \land \ [\alpha](\neg e \to \psi(t)) \\
\overset{\text{cut}}{\equiv} & \big(e \ \to \ [\alpha](e \to \psi(r)) \land [\alpha](\neg e \to \psi(t))\big) \\
 & \land\big(\neg e \ \to \ [\alpha](e \to \psi(r)) \land [\alpha](\neg e \to \psi(t))\big) \\
\overset{e \text{ rigid for } \alpha}{\equiv} & \big(e \ \to \ [\alpha](\mathit{true} \to \psi(r)) \land [\alpha](\mathit{false} \to \psi(t))\big) \\
 & \land\big(\neg e \ \to \ [\alpha](\mathit{false} \to \psi(r)) \land [\alpha](\mathit{true} \to \psi(t))\big) \\
\equiv & \big(e \ \to \ [\alpha]\psi(r) \land [\alpha]\mathit{true}\big) \\
 & \land\big(\neg e \ \to \ [\alpha]\mathit{true} \land [\alpha]\psi(t)\big) \\
\equiv & (e \to [\alpha]\psi(r)) \ \land \ (\neg e \to [\alpha]\psi(t))
\end{array}
$$

The remaining cases are similar. ■

Without wariness, the *conditional term split (right)* (R38) rule would be unsound as the following example demonstrates.

Example 4.4.2 (Non-wary conditional term split) Consider a formula $\phi(u)$, for which the implicit[15] substitution $[u \mapsto \mathit{if}\, \mathtt{x} > 0\, \mathit{then}\, a\, \mathit{else}\, b\, \mathit{fi}]$ would be non-admissible due to the trespassing of `x` beyond the update to `x`.

$$\phi(u) \quad := \quad p(\langle x \lhd 0 \rangle u)$$

Without wariness the following inference is unsound, with a, b being constant symbols and `x` a program variable.

$$\frac{\mathtt{x} \doteq 1 \ \vdash (\mathtt{x} > 0 \to \phi(a)) \land (\mathtt{x} \leq 0 \to \phi(b))}{\mathtt{x} \doteq 1 \ \vdash \phi(\mathit{if}\, \mathtt{x} > 0\, \mathit{then}\, a\, \mathit{else}\, b\, \mathit{fi})}$$

At this stage we need to look for a model ℓ, s of the premise that is no model of the consequence . More precisely, we construct a model ℓ, s of the premise $(\mathtt{x} > 0 \to p(\langle x \lhd 0 \rangle a)) \land (\mathtt{x} \leq 0 \to p(\langle x \lhd 0 \rangle b))$ that is no model of the consequence $p(\langle x \lhd 0 \rangle \mathit{if}\, \mathtt{x} > 0\, \mathit{then}\, a\, \mathit{else}\, b\, \mathit{fi})$ and satisfies the antecedent formula $\mathtt{x} \doteq 1$. Let $\ell \models p(a)$, $\ell \not\models p(b)$, $\ell, s \models \mathtt{x} \doteq 1$. On the one hand, $\ell, s \models \mathtt{x} > 0 \land p(\langle x \lhd 0 \rangle a)$ and $\ell, s \models \neg \mathtt{x} \leq 0$ hold. But on the other hand, $\ell, s \not\models p(\langle x \lhd 0 \rangle \mathit{if}\, \mathtt{x} > 0\, \mathit{then}\, a\, \mathit{else}\, b\, \mathit{fi})$, because of $\ell, s[x \mapsto 0] \models \neg(\mathtt{x} > 0) \land \neg p(b)$. □

For proving the soundness of the *term rewrite (right)* (R36) rule we show that term rewrite does not change the semantics.

[15] This substitution is implicit in the notation $\phi(\mathit{if}\, \mathtt{x} > 0\, \mathit{then}\, a\, \mathit{else}\, b\, \mathit{fi})$.

Definition 4.4.5 *A term rewrite rule* $s \rightsquigarrow t$ *is* valuation-preserving, *if all instances* $s \rightsquigarrow t$ *of the rule satisfy for each interpretation* ℓ *and each state* w *thereof that* $val_\ell(w, s) = val_\ell(w, t)$.

The soundness proof of the term rewrite rule splits into a proof of valuation-preservation for the term rewrite system, and into a soundness proof of the term rewrite application inference rule based on the fact that the term rewrite rules are valuation-preserving.

Proposition 4.4.6

$$\begin{array}{l}
\text{(R35)} \quad \text{term rewrite (left)} \\
\qquad\qquad \dfrac{\phi(t) \;\vdash}{\phi(s) \;\vdash} \\
\Leftarrow (s \rightsquigarrow t) \text{ holds} \\
\text{(R36)} \quad \text{term rewrite (right)} \\
\qquad\qquad \dfrac{\vdash \phi(t)}{\vdash \phi(s)} \\
\Leftarrow (s \rightsquigarrow t) \text{ holds}
\end{array}$$

are sound, provided that all rewrite rules are valuation-preserving.

Proof: The proof is an immediate inductive extension of the concept of "valuation-preserving". ■

Proposition 4.4.7 *The following term rewrite rules are valuation-preserving*

$$\begin{array}{l}
\text{(R27)} \quad \text{update (match)} \\
\langle f(s) \lhd t \rangle f(u) \rightsquigarrow \mathit{if}\, s \doteq \langle f(s) \lhd t \rangle u \;\mathit{then}\, t \;\mathit{else}\, f\big(\langle f(s) \lhd t \rangle u\big)\;\mathit{fi} \\
\text{(R28)} \quad \text{update (promote)} \\
\qquad\qquad \langle f(s) \lhd t \rangle\, \Upsilon(u) \rightsquigarrow \Upsilon\big(\langle f(s) \lhd t \rangle u\big) \\
\quad \Leftarrow f \neq \Upsilon \in \Sigma \\
\text{(R29)} \quad \text{update } (\forall) \\
\qquad\qquad \langle \mathcal{U} \rangle \forall x\, \phi \rightsquigarrow \forall x\, \langle \mathcal{U} \rangle \phi \\
\quad \Leftarrow x \text{ not in } FV(\mathcal{U})
\end{array}$$

Proof:

$$
\begin{array}{rl}
& val_\ell(w, \langle f(s) \lhd t\rangle f(u)) \\
= & val_\ell(w', f(u)) \\
= & \begin{cases} val_\ell(w,t) & \Leftarrow val_\ell(w',u) = val_\ell(w,s) \\ val_\ell(w', f(u)) & \Leftarrow val_\ell(w',u) \neq val_\ell(w,s) \end{cases} \\
= & \begin{cases} val_\ell(w,t) & \Leftarrow val_\ell(w',u) = val_\ell(w,s) \\ val_\ell(w,f)\big(val_\ell(w',u)\big) & \Leftarrow val_\ell(w',u) \neq val_\ell(w,s) \end{cases} \\
\stackrel{2.3.8}{=} & \begin{cases} val_\ell(w,t) & \Leftarrow val_\ell(w, \langle\mathcal{U}\rangle u) = val_\ell(w,s) \\ val_\ell(w,f)\big(val_\ell(w, \langle f(s) \lhd t\rangle u)\big) & \Leftarrow val_\ell(w, \langle\mathcal{U}\rangle u) \neq val_\ell(w,s) \end{cases} \\
= & \begin{cases} val_\ell(w,t) & \Leftarrow val_\ell(w, \langle f(s) \lhd t\rangle u) = val_\ell(w,s) \\ val_\ell(w, f(\langle f(s) \lhd t\rangle u)) & \Leftarrow val_\ell(w, \langle f(s) \lhd t\rangle u) \neq val_\ell(w,s) \end{cases} \\
= & val_\ell(w, \mathit{if}\, s \doteq \langle f(s) \lhd t\rangle u \;\mathit{then}\, t \;\mathit{else}\, f\big(\langle f(s) \lhd t\rangle u\big) \;\mathit{fi})
\end{array}
$$

where $w' := w[f(val_\ell(w,s)) \mapsto val_\ell(w,t)]$ is the state reached from w by execution of $f(s) \lhd t$, and $\langle\mathcal{U}\rangle$ abbreviates $\langle f(s) \lhd t\rangle$. ■

4.4.3 Dynamics

Proposition 4.4.8 *The following inference rules are sound*

(R39) instanceof

$$\Leftarrow \mathtt{A} \leq \mathtt{C} \quad \frac{}{\vdash \mathtt{obj}_{\mathtt{A}}(n)\, \mathtt{instanceof}\, \mathtt{C}}$$

(R40) instanceof

$$\Leftarrow \mathtt{A} \not\leq \mathtt{C} \quad \frac{}{\mathtt{obj}_{\mathtt{A}}(n)\, \mathtt{instanceof}\, \mathtt{C}\ \vdash}$$

(R41) `new` identity

$$\frac{}{\vdash \forall i{:}\mathtt{nat}\ \forall j{:}\mathtt{nat}\ (\mathtt{obj}_{\mathtt{C}}(i) \doteq \mathtt{obj}_{\mathtt{C}}(j) \rightarrow i \doteq j)}$$

(R42) `new` disjoint identity

$$\Leftarrow \mathtt{C} \neq \mathtt{D} \quad \frac{}{\mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{D}}(m)\ \vdash}$$

(R43) `new` generated

$$\frac{}{\vdash \forall o{:}C\ \exists n{:}\mathtt{nat}\ o \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)}$$

Proof:

R39- Assuming for some state w of an According to Rem. 2.3.2 $\mathtt{obj}_{\mathtt{A}}(n) \in \ell(A) \subseteq \ell(C)$ for a subclass `A` of `C`. For $\mathtt{A} \not\leq \mathtt{C}$, instead, there are two possibilities. Either $\mathtt{C} \leq \mathtt{A}$, in which case $\mathtt{obj}_{\mathtt{A}}(n) \in O_{\mathtt{A}} \subseteq \ell(A) \setminus \ell(C)$. Or `A` and `C` are incomparable according to the type hierarchy. Then it is $\mathtt{obj}_{\mathtt{A}}(n) \in O_{\mathtt{A}}$ while $O_{\mathtt{A}} \cap \ell(C) = \emptyset$ by construction in Def. 2.3.1. Thus, in either case, $\mathtt{obj}_{\mathtt{A}}(n)\, \mathtt{instanceof}\, \mathtt{C}$ is true if and only if `A` is a subclass of `C`, thereby justifying R39 and R40.[16]

R41- Likewise, the semantics of $\mathtt{obj}_{\mathtt{C}}(n)$ demands injectivity and disjointness, which justifies the R41, R42 rules. This is because $n \neq m \rightarrow \mathtt{obj}_{\mathtt{C}}(n) \neq \mathtt{obj}_{\mathtt{C}}(m)$ is an ODL tautology, and for $\mathtt{C} \neq \mathtt{D}$, $\mathtt{obj}_{\mathtt{C}}(n) \doteq \mathtt{obj}_{\mathtt{D}}(m)$ is always false by Def. 2.3.1.

[16] This also fits to the effects of object creation in §3.4.2, where $\mathtt{obj}_{\mathtt{A}}(n)$ is an object of dynamic type `A` created via `new A()`.

R43 The soundness of the R43 rule is a direct consequence of the fact that the semantics of $\texttt{obj}_{\texttt{C}}(n)$ demands surjectivity.

■

Proposition 4.4.9 *The following inference rule is sound wrt.* $\vDash_g$

$$\begin{array}{c}\text{(R45)}\quad \text{loop induction right}\\ \dfrac{\Gamma\ \vdash \langle\mathcal{U}\rangle p, \Delta \quad p, e\ \vdash [\alpha]p \quad p, \neg e\ \vdash A}{\Gamma\ \vdash \langle\mathcal{U}\rangle[\texttt{while}(e)\,\{\alpha\}]A, \Delta}\end{array}$$

Proof: Proving the conjecture requires to show that

$$\ell \vDash \Gamma\ \vdash \langle\mathcal{U}\rangle[\texttt{while}(e)\,\{\alpha\}]A, \Delta$$

holds in any interpretation ℓ on the basis of the following premises.

(a) $\ell \vDash \Gamma\ \vdash \langle\mathcal{U}\rangle p, \Delta$.

(b) $\ell \vDash p, e\ \vdash [\alpha]p$

(c) $\ell \vDash p, \neg e\ \vdash A$

Assume that the program $\texttt{while}(e)\,\{\alpha\}$ takes precisely n iterations to complete its run when started in some state s of ℓ. If $n = \infty$ the conjecture is trivial by the notion of box modalities. The proof follows an induction over the number n of iterations.

IH If the program $\texttt{while}(e)\,\{\alpha\}$ is started in a state s, in which p holds and from which $\texttt{while}(e)\,\{\alpha\}$ performs $\leq n$ iterations to complete its run, then $\ell, s \vDash [\texttt{while}(e)\,\{\alpha\}]A$.

IA $n = 0 \Rightarrow \ell, s \vDash \neg e$, otherwise the loop would perform at least one iteration. By assumption $\ell, s \vDash p$ holds in the state s, which further implies with (c) that $\ell, s \vDash A \Rightarrow \ell, s \vDash [\texttt{while}(e)\,\{\alpha\}]A$.

IS $n > 0 \Rightarrow \ell, s \vDash e$, since the loop body is entered. The assumption yields $\ell, s \vDash p$. Let s' be some successor state with $s\rho_\ell(\alpha)s'$, from which $\texttt{while}(e)\,\{\alpha\}$ will thus only perform less than n iterations. Furthermore, (b) ensures that $\ell, s' \vDash p$ still holds. By IH this implies $\ell, s' \vDash [\texttt{while}(e)\,\{\alpha\}]A$. Since s' has been arbitrary one can conclude that $\ell, s \vDash [\alpha][\texttt{while}(e)\,\{\alpha\}]A$, respectively $\ell, s \vDash [\texttt{while}(e)\,\{\alpha\}]A$, since $\ell, s \vDash e$ holds.

In an arbitrary state w of ℓ there are two possible cases. Either $\ell, w \vDash \Gamma \vdash \Delta$ already holds, then there is nothing to show because

$$\ell, w \vDash \Gamma \ \vdash \langle\mathcal{U}\rangle[\texttt{while}(e)\,\{\alpha\}]A, \Delta$$

holds more than ever. Or $\Gamma \vdash \Delta$ does not hold, i.e. Γ is true but Δ false, then (a) locally implies $\ell, w \vDash \langle\mathcal{U}\rangle p$. This allows to conclude that there is a state s with $w\rho_\ell(\mathcal{U})s$ and $\ell, s \vDash p$ from which the induction enforces that $\ell, s \vDash [\texttt{while}(e)\,\{\alpha\}]A$. Then in the original state, $\ell, w \vDash \langle\mathcal{U}\rangle[\texttt{while}(e)\,\{\alpha\}]A$ holds as well. ■

Example 4.4.3 R45 is not sound wrt. $\vDash_l$ as the following example shows. Consider the program $\texttt{while}(x < 10)\,\{x \lhd x+1\}$ with the (wrong) postcondition $x < 3$, which also serves as a (false) invariant. Then the premises of R45 are satisfied locally in a state with $val_\ell(w, x) = 0$, but the consequence is still wrong, because the postcondition is finally untrue in the state $w[x \mapsto 10]$. Moreover, the invariant $x < 3$ will already be hurt after 3 iterations. □

4.5 Completeness

This section presents the proof of the central statement of this thesis: the ODL calculus is complete relative to arithmetic.

A colossal property for a program verification system based on the ODL calculus to possess would be the ability to prove *all* true statements about *all* programs. This is – of course – impossible, since ODL contains[17] first-order arithmetic, which is itself already inherently incomplete by the *Unvollständigkeitssatz* of (Gödel, 1931). Therefore, a proof system embracing first-order arithmetic like ODL does cannot ever have a calculus that is both sound and complete. Still the question remains how to compare different calculi for ODL or other (dynamic) logics. From intuitive deliberations, the existence of increasingly incomplete calculi obtained by removing integral inference rules from an intact calculus is apparent. For example, an attempt to remove all inference rules for sequential composition from ODL would certainly lead to a decisively less complete calculus. This is due to the fact that even very simple conjectures would possess no proof in this situation, although this circumstance is completely detached from concerns about the inherent complexity of first-order arithmetic or equivalent means.

Finally, the issue remains to what extent the ODL calculus is complete, bearing in mind that the underlying arithmetic is not. Vaguely speaking, we

[17] The logic ODL permits to formulate statements about its domain of interpretation without involving any dynamic logic modalities at all.

put aside the completeness problems of the domain of interpretation and investigate completeness apart from that. In more elaborate words this amounts to factorising the complexities of first-order arithmetic from the completeness question. The standard notion from literature (Cook, 1978) for completeness modulo an underlying logic is that of relative completeness, which in this case means completeness relative to the domain of computation: first-order arithmetic. Thereby, relative completeness examines the imaginary situation of what would happen to completeness if only arithmetic itself had been complete.

Theorem 2 (Relative Completeness) *The* ODL *calculus consisting of the inference rules presented in §4.2 is* relatively complete *for arithmetical structures*[18]*, i.e. in conjunction with the following oracle inference rule*

(R56) First-Order Oracle

$$\frac{}{\vdash F}$$

$\Leftarrow F \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ is a first-order arithmetic tautology

the following completeness statement is true for each $\phi \in \mathrm{Fml}(\Sigma \cup V)$

$$\textit{if } \ell \vDash \phi \textit{ for each arithmetical structure } \ell \quad \textit{then} \quad \vdash \phi$$

This central theorem will be proven in the remainder of this section with the proof split into several separate smaller conjectures for structural reasons. Once those simpler conjectures have been proven, the proof of Th. 2 will be built in §4.5.5 from the results 4.5.1-4.5.15.

In principle, the proof inductively reduces the conjecture equivalently to first-order logic by Lem. 4.5.4, first. The mere first-order conjecture ϕ' can be proven by R56. Then the proof shows that the dynamic parts can be inferred again inductively from the first-order equivalents, cf. Prop. 4.5.13 and Th. 2. For this purpose, Prop. 4.5.1 and Prop. 4.5.3 facilitate an isolated examination of the single modalities in ϕ and its byproducts.

In the context of completeness investigations simply treat $\langle\alpha\rangle\phi$ as an abbreviation of $\neg[\alpha]\neg\phi$, and $\forall x\, \phi$ as an abbreviation of $\neg\exists x\, \neg\phi$.

[18] Arithmetical structures are those interpretations of a domain having $\mathbf{N}$ as domain for `nat` with the usual interpretations for $0, +, \cdot, <, \doteq$ on integers. More generally speaking, arithmetical structures contain a first-order definable isomorphic copy of $\mathbf{N}$, and first-order definable functions for coding finite sequences of elements of the domains into single elements.

4.5.1 First-Order

The next two results isolate the completeness statements about ODL with respect to smaller fragments of logic: the propositional and the first-order fragment of ODL.

Proposition 4.5.1 *The* ODL *calculus is* propositionally complete*, i.e. all instances of propositional tautologies are derivable and* modus ponens *is an inference rule.*

Proof: Propositional completeness is immediate for ODL because the ODL calculus has been built on top of the sound and complete standard inference rules for first-order logic, which is also complete when restricted to propositional logic reasoning. Moreover, *modus ponens* (R55) is a derived inference rule according to Prop. 4.2.9. ∎

Proposition 4.5.2 *The* ODL *calculus is* complete for first-order formulas*, i.e. for each* $F \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *of first-order logic* $\vDash F \;\; \Rightarrow \vdash F$*.*

Proof: Once more, first-order completeness is immediate for ODL because the ODL calculus is built on top of the sound and complete standard inference rules for first-order logic. ∎

The next result will be used during the completeness proof to export any arguments that have been performed on top-level into modality or quantifier context.

Proposition 4.5.3 *For each* $x \in V, \phi, \psi \in \mathrm{Fml}(\Sigma \cup V)$ *and each* $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ *the generalisation rules hold, i.e.*

1. $\vdash \phi \to \psi \;\; \Rightarrow \vdash \langle\alpha\rangle\phi \to \langle\alpha\rangle\psi$
2. $\vdash \phi \to \psi \;\; \Rightarrow \vdash [\alpha]\phi \to [\alpha]\psi$
3. $\vdash \phi \to \psi \;\; \Rightarrow \vdash \exists x\, \phi \to \exists x\, \psi$

Rule (2) *is called rule of regularity in (Fitting & Mendelsohn, 1998).*

Proof: In order to validate those derivations, ODL immediately contains the generalisation inference rules R44, R46, R47 for $\exists$, $\langle\rangle$ and $[]$. ∎

4.5.2 Expressibility

The next lemma provides an integral component of the completeness proof. It shows the meaning of any ODL dynamic logic formula to be equivalently expressible in first-order logic, when using the coding capabilities of proper arithmetic. Despite all its structural multi-modal extension, the dynamic logic ODL only talks about aspects that might have been expressed in first-order arithmetic as well. Even though dynamic logics massively increase both notational intuition and practical proving power, it is always possible to produce a (rather complicated) first-order formula that utilises the encoding capabilities of natural numbers to express the very same statement. In other words, the expressive power of ODL and first-order arithmetic are on the same level.

For the proof to succeed, the effect of the programs occurring in modalities of a formula have to be represented in first-order logic. The key insight for this expressibility is that the equation $c' \doteq t$ relates the values of the symbols c and c' just like the diamond $\langle c \lhd t \rangle$ relates the prestate value of c to the poststate value of c (in $\langle c \lhd t \rangle \phi(c)$). Formalising the precise correspondence of c' and the poststate value of c adequately is one challenge of the proof. Representing finite sequences[19] of intermediate states passed by during the execution of a loop statement is the other challenge.

An effective expressibility[20] lemma immediately implies the existence of a relatively complete calculus for trivial reasons: A calculus that pursues the transformation of the constructive proof will transform ODL to first-order logic from where completeness relative to first-order arithmetic is evident. Though at the heart of the completeness argument, Lem. 4.5.4 alone does not establish that the particular calculus that we chose is relatively complete.

Lemma 4.5.4 (Expressibility) ODL *is* expressible *in first-order logic, i.e. for each* $\phi \in \mathrm{Fml}(\Sigma \cup V)$ *there is* $\phi' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *of first-order logic such that in every arithmetical structure* ℓ *we have the local equivalence* $\ell \vDash \phi \leftrightarrow \phi'$. *As symbolic notation for this circumstance you often find* ODL $\leq_{\mathbf{N}}$ FOL *or, since the reverse expressibility reduction* FOL $\leq_{\mathbf{N}}$ ODL *holds for trivial reasons,* ODL $\equiv_{\mathbf{N}}$ FOL.

Proof: The proof follows an induction on the structure of the formula ϕ for which it is imperative to find a first-order equivalent ϕ' over arithmetical structures.

IA If ϕ is an atomic first-order formula then $\phi' := \phi$ already is first-order such that nothing has to be proven.[21]

[19] Sequences of arbitrary length.

[20] A logic is effectively expressive (or effectively expressible in first-order logic), if the expressibility lemma further guarantees the computability of the process of constructing a first-order translation. The proof of Lem. 4.5.4 is, in fact, constructive and effective.

[21] Updates on terms constitute only intermediate auxiliary stages in the ODL calculus,

1. $\phi = t\,\texttt{instanceof}\,\texttt{C}$ as a first-order typing relationship can be represented in many-sorted first-order logic with some notation. Also see Rem. 4.5.5.

2. $\phi = \psi(\mathit{if}\, e\, \mathit{then}\, s\, \mathit{else}\, t\, \mathit{fi})$ is equivalent to $\chi \ := \ (e \to \psi(s)) \land (\neg e \to \psi(t))$ according to Prop. 4.4.4. Therefore, the formula expressing the simpler χ in first-order logic also expresses ϕ.[22].

3. $\phi = F \lor G$ then by induction hypothesis there are formulas F', G' such that $\vDash F \leftrightarrow F', \vDash G \leftrightarrow G'$, from which can be concluded by congruence of $\leftrightarrow$ that
$$\vDash \quad \underbrace{F \lor G}_{\phi} \leftrightarrow \underbrace{F' \lor G'}_{\phi'}$$
Likewise reasoning concludes the other propositional connectives or quantifiers.

4. For $\phi = \langle\alpha\rangle\psi$, the proof will be first presented as a sketch, about which the more subtle details will be filled in later on. Let z denote the finite vector of program variable or function symbols occurring in α. It is finite because a program α of finite length can only mention finitely many distinct symbols.[23] Denote by z' a vector of the same length and types as z but with fresh symbols, which are new to ϕ. Following, a substitution $[z \mapsto z']$ is a short notation for the simultaneous substitution of the elements x of z by the corresponding elements x' of z'. Likewise an equality statement like $z \doteq z'$ is to be understood component-wise. A similar component-wise understanding holds for meta-language equality in $w(z) = z'$. With this being stated a selection of $\exists z'\,(\tau_\alpha(z, z') \land {\psi'}_z^{z'})$ as ϕ' immediately raises a problem. The quantifier is higher-order since z' still contains function symbols f' for function symbols f occurring in α. Let us – for the time being – pretend that there was not sign of higher-order involved and resolve those matters in retrospect later on. It remains to show that
$$\vDash \quad \langle\alpha\rangle\psi \ \leftrightarrow \ \exists z'\,(\tau_\alpha(z, z') \land {\psi'}_z^{z'})$$

which is the reason for their minor importance for expressibility and completeness. Further, they can be treated rather similar to updates on formulas.

[22]For instance, $\phi' \ := \ (e \to \psi'(s)) \land (\neg e \to \psi'(t))$

[23]Contrary to WHILE in the case of ODL α could read and modify an unbounded number of memory locations by loops over function access like $f(i)$. Still there only occurs one function symbol f and one program variable i within α then, regardless of their broad extent of accessible memory locations. This finite denotator property would no longer hold for programming languages which allow the following program $\texttt{while}(i < n)\,\{c_i \lhd c_i + 1\}$.

For which it remains to construct the first-order formula $\tau_\alpha(z, z')$ characterising the possible program state transitions from the state characterised by the values z to the state characterised by z' such that for each state w

$$\ell, w \vDash \tau_\alpha(z, z') \quad \iff \quad \text{there is } w' \text{ with } w\rho_\ell(\alpha)w' \; w'(z) = z'$$

Alternatively, taking into account that α does not modify z' because it does not even occur in α, $\tau_\alpha(z, z')$ should satisfy

$$\vDash \quad \tau_\alpha(z, z') \; \leftrightarrow \; \langle\alpha\rangle z \doteq z'$$

In principle, it would be sufficient to show that the state transition relation of programs is computable and conclude that it has a defining first-order arithmetic formula. There are some subtleties with the effect of updates and subtleties with the encoding in loop characterisations, though. Moreover since (Harel, 1979) and (Schlager, 2000) contained invalid completeness proofs even for the basic case without updates, this proof explicitly constructs a characteristic formula $\tau_\alpha(z, z')$ by an effective translation. See Rem. 4.5.6 for an analysis of the reasons and importance of the discrepancy.

(a) $\tau_{x \lhd t}(z, z') \;\; := \;\; (x' \doteq t) \wedge \bigwedge_{x \neq y \in z} y' \doteq y$, where x in the vector z corresponds to x' in z'.

(b) $\tau_{f(s) \lhd t}(z, z') \;\; := \;\; \forall x\, (x \neq s \rightarrow f'(x) \doteq f(x)) \wedge f'(s) \doteq t \wedge \bigwedge_{x \neq y \in z} y' \doteq y$.

(c) $\tau_{\mathtt{if}(\chi)\,\{\gamma\}\mathtt{else}\{\delta\}}(z, z') \;:=\; (\chi' \rightarrow \tau_\gamma(z, z')) \wedge (\neg\chi' \rightarrow \tau_\delta(z, z'))$.

(d) $\tau_{\gamma;\delta}(z, z') \;:=\; \exists z''\, (\tau_\gamma(z, z'') \wedge \tau_\delta(z'', z'))$.

(e) The characterisation of loops is subject to additional function symbols assumed to enrich Σ, which receive their intended behaviour according to the axiomatisation in the formula *Enc*. The inductive construction introduces multiple repeated axiomatisations *Enc*, which is unnecessary on the one hand but harmless on the other.

$$\begin{aligned}
\tau_{\mathtt{while}(\chi)\,\{\gamma\}}(z, z') \quad := \quad & \mathit{Enc} \rightarrow \exists S\!:\!\mathtt{nat}\; \Big(\\
& \quad \mathit{nth}(S, 0) \doteq z \wedge \mathit{nth}(S, \mathit{length}(S)) \doteq z' \\
& \wedge \forall i\!:\!\mathtt{nat}\; \big(i < \mathit{length}(S) \rightarrow \\
& \qquad\quad \tau_\alpha(\mathit{nth}(S, i), \mathit{nth}(S, i+1)) \\
& \qquad \wedge\, {\chi'}_z^{\mathit{nth}(S,i)}\big) \\
& \wedge \neg{\chi'}_z^{z'} \Big)
\end{aligned}$$

For formulas like $nth(S,0) \doteq z$ and arguments to τ_α use the fixed computable bijection $\mathbf{N} \cong \mathbf{N}^{\text{length}(z)}$ to encode finite vectors of variables in one value.

Enc is a formula that characterises encodings of finite sequences into a single natural number.[24] Provided that there is some Gödel encoding of the elements in $\ell(\tau)$ of the base type τ, this is possible by the computable bijection $\mathbf{N} \cong \mathbf{N}^n$. Thus, choose Enc in such a way that the following holds for each state w .

$$\ell, w \models Enc \to nth(S,i) \doteq y \iff \begin{array}{l} \text{there is } a_1,\dots,a_n \in \ell(\tau) \\ val_\ell(w,S) = \text{gödel}((a_1,\dots,a_n))\,, \\ val_\ell(w,i) \le n\,, \\ val_\ell(w,y) = a_{val_\ell(w,i)} \end{array}$$

where gödel is an injection of the finite sequences over $\ell(\tau)$ into $\mathbf{N}$. Further, $nth : \mathtt{nat} \times \mathtt{nat} \to \tau$ is a function symbol for representing sequence encoding. Similarly, $length : \mathtt{nat} \to \mathtt{nat}$ is used as a function symbol to speak about the length of such an encoded sequence. Enc ensures that it satisfies for each state w

$$\ell, w \models Enc \to length(S) \doteq n \iff \begin{array}{l} \text{there is } a_1,\dots,a_n \in \ell(\tau) \\ val_\ell(w,S) = \text{gödel}((a_1,\dots,a_n)) \end{array}$$

as could be characterised by the defining formula

$$length(S) \doteq n \;\leftrightarrow\; \exists s_n\!:\!\tau\;\; nth(S,n) \doteq s_n \land \neg\exists s'\!:\!\tau\;\; nth(S,n+1) \doteq s'$$

This concludes the proof apart from our fraud with higher-order, which still needs to be dissolved. For this purpose second-order quantifiers

[24] The encoding can be achieved with the following axiomatic characterisation of exponentiation, pairing and iterative sequence processing, assuming implicit universal quantification for readability.

$$\begin{array}{rcl} b^0 \doteq 1 & \land & b^{n+1} \doteq b \cdot b^n \\ pair(S,x,y) & \leftrightarrow & S \doteq 2^{\text{gödel}(x)} \cdot 3^{\text{gödel}(y)} \\ nth(S,i) \doteq y & \leftrightarrow & \exists d\!:\!\mathtt{nat}\ \exists S'\!:\!\mathtt{nat}\ \big(pair(S,d,S') \land i \le d \\ & & \quad \land \exists S''\!:\!\mathtt{nat}\ iseqnth(S',S'',i,y)\big) \\ iseqnth(S,S',0,y) & \leftrightarrow & pair(S,\text{gödel}(y),S') \\ iseqnth(S,S',i+1,y) & \leftrightarrow & \exists \tilde{S}\!:\!\mathtt{nat}\ \exists \tilde{y}\!:\!\tau\ \big(iseqnth(S,\tilde{S},i,\tilde{y}) \land pair(\tilde{S},\text{gödel}(y),S')\big) \end{array}$$

have to be reduced to first-order quantifiers although this is, of course, impossible in general. A countability argument also suggests that there is no serious hope to reduce general quantification over the uncountable domain of functions to quantification over the countable domain of, say, natural numbers properly.

Enc is assumed to characterise symbols for access to modifiable non-rigid functions. The term $\mathit{apply}(F_f, x)$ is intended to produce the value of the function f at the position x, thereby considering the accumulated modifications according to the number F_f. The term $\mathit{change}(F_f, x, y)$ should modify the interpretation of f at x to y with regard to the already accumulated modifications according to F_f. *Enc* could achieve this characterisation with the following defining formulas[25]

$$
\begin{aligned}
& y \doteq \mathit{apply}(F_f, x) \\
\leftrightarrow \quad & \\
& \exists i{:}\texttt{nat}\ \ (i \leq \mathit{length}(F_f) \land \mathit{nth}(F_f, i) \doteq (x, y)) \\
& \lor \neg\exists i{:}\texttt{nat}\ \exists y{:}\tau\ \ (i \leq \mathit{length}(F_f) \land \mathit{nth}(F_f, i) \doteq (x, y)) \\
& \quad \land\ y \doteq f(x)
\end{aligned}
$$

and

$$
\begin{aligned}
& F'_f \doteq \mathit{change}(F_f, x, y) \\
\leftrightarrow \quad & \\
& \exists i{:}\texttt{nat}\ \exists y_0{:}\tau\ \ \big(i \leq \mathit{length}(F_f) \land \mathit{nth}(F_f, i) \doteq (x, y_0) \\
& \qquad \land\ \mathit{length}(F'_f) \doteq \mathit{length}(F_f) \\
& \qquad \land\ \mathit{nth}(F'_f, i) \doteq (x, y) \\
& \qquad \land\ \forall j\ (j \leq \mathit{length}(F_f) \land j \neq i \rightarrow \mathit{nth}(F'_f, j) \doteq \mathit{nth}(F_f, j)) \\
& \quad \big) \\
& \lor \neg\exists i{:}\texttt{nat}\ \exists y_0{:}\tau\ \ \big(i \leq \mathit{length}(F_f) \land \mathit{nth}(F_f, i) \doteq (x, y_0)\big) \\
& \quad \land\ \mathit{length}(F'_f) \doteq \mathit{length}(F_f) + 1 \\
& \quad \land\ \mathit{nth}(F'_f, \mathit{length}(F_f) + 1) \doteq (x, y) \\
& \quad \land\ \forall j\ (j \leq \mathit{length}(F_f) \rightarrow \mathit{nth}(F'_f, j) \doteq \mathit{nth}(F_f, j))
\end{aligned}
$$

Since (initial) states are allowed arbitrary interpretations of f one cannot possibly hope to encode arbitrary functions of infinite support into the natural number F_f. Even if computable functions possess a finite computing program, of which the gödelisation could be used for fallback

[25] In this context, $Z \doteq (x, y)$ is short list notation for $\mathit{nth}(Z, 0) \doteq x \land \mathit{nth}(Z, 1) \doteq y \land \mathit{length}(Z) \doteq 2$.

purpose, initial states could just as well start with an uncomputable interpretation for f. Encoding arbitrary members of the uncountable set of uncomputable functions in a natural number F_f is impossible by a countability argument. Representing a complete state in first-order logic is thus, of course, impossible in the general case of states with uncomputable functions. The idea is that irrespective of the (possibly uncomputable) interpretation of the function symbol f the accumulated change to its interpretation as resulting from the execution of program statements can be represented in first-order logic. The $y \doteq f(x)$ formula in the definition of *apply* assures that whenever the accumulated changes to F_f did not modify the interpretation of f at x the original interpretation of f in the current state applies.

The notation F'_f in $F'_f \doteq \mathit{change}(F_f, x, y)$ suggests that after an emulated change to the interpretation of F_f the resulting encoded function F'_f will still fallback to f during an $\mathit{apply}(F'_f, z)$ outside of the (finite) set of locations where the interpretation of F'_f has been modified by *change* invocations.

To put it in a nutshell, we apply the principle of updates on the meta-level again: even though the whole state is generally non-representable in first-order logic, the finite description of the change caused by the execution of program statements is expressible in first-order logic.

Instead of the above "premature" treatment of assignment, in the light of the higher-order quantification problem use the following characterisation of the transition relation for updates.

(b) $$\tau_{f(s) \lhd t} \quad := \quad F'_f \doteq \mathit{change}(F_f, s, t) \land \bigwedge_{F_f \neq x \in z} x' \doteq x$$

This formula presupposes that all occurrences of non-rigid $f(x)$ have been replaced by appropriate $\mathit{apply}(F, x)$ *after* flattening to first-order logic. This means except for the occurrences within the defining conditions of *apply* and *change* in *Enc*, where the subformula $y \doteq f(x)$ remains unchanged.

■

Remark 4.5.5 *As for all other aspects, there is no strict requirement for the first-order logic to employ the* ODL *notation for expressing a typing relationship. Still this is assumed during this thesis for simplicity.*

Example 4.5.1 First-order logic with statically typed quantification that does not have a concept of subtyping is sufficient for expressing ODL by the following additional translation.

$$t \, \texttt{instanceof} \, \texttt{C} \;\leftrightarrow\; \bigvee_{\tau \leq \texttt{C}} \exists x \!:\! \tau \;\; x \doteq t$$

□

The astonishing fact about Lem. 4.5.4 is that it does *not* impose any restrictions on the states or interpretations involved other than there being properly working natural numbers. This circumstance is especially remarkable in the light of the arbitrary nature of functions. Some states could take the liberty of employing even uncomputable functions as an interpretation for function symbols. Although startling at first sight, this is perfectly possible for states from a logical perspective, since there is no need to restrict the set of models to those with properly computable functions, only. And that uncomputable aspects still play an important role is visible, at the latest, from the preoccupation with program verification, which is an uncomputable problem from the bottom.[26]

Another interesting point to be mentioned is that our first proof of Lem. 4.5.4 went by encoding the whole function for a symbol f into a natural number. Since the *apply* and *change* formulas need to be able to decode this information in a computable manner, this directly implies – by Church-Turing Thesis – that it is impossible to express other functions than ordinary computable ones. Thus, the first proof had to limit Lem. 4.5.4 to the case of states with computable functions, only. A global restriction of the concept of models to those interpretations with computable functions, leads to a very unconventional proof theory, some of which is discussed in (Crossley, 1981).

Fortunately, the subtle technique to encode only the change to the function in a natural number rather than the whole function, made it possible to present an unrestricted Lem. 4.5.4. This approach innocently leaves the rest of the evaluation to the state semantics. Within the part of the domain where the interpretation of the function has been changed by the execution of program statements, the natural number encoding is used. Beyond that domain, mentioning the term $f(x)$ just amounts to a denotation of the original value of the function associated to f at the value of x.

Let us – briefly – investigate what the consequences of a weakened Lem. 4.5.4 would be for the whole completeness proof. Tracing the uses during the proof reveals that the whole theory would have to be rewritten to talk about computable states, only. Especially, one would have to find a stronger version

[26]It is not that easy, though, to imagine a situation with an uncomputable function for, say, the link structure of the ⌜next⌝ attribute symbol used for chaining nodes in a linked list implementation.

of both Prop. 4.5.2 and R56, which already guarantee derivability for those formulas that only happen to be true in computable states, even if they are not tautological in uncomputable states.

Remark 4.5.6 *The proof of Lem. 4.5.4 resolves a flaw in the completeness proofs that have been published in (Harel, 1979) and (Schlager, 2000).*

(Harel, 1979) and (Schlager, 2000) contain invalid relative completeness proofs for a subtle reason. Their characterisation for the expressiveness proof of the primitive assignment statement $x \lhd t$ of the classical WHILE programming language is wrong. This remainder of this section closely analyses the ramifications, using the notation from Lem. 4.5.4.

The essential case of modalities in Lem. 4.5.4 is proven by constructing a characteristic first-order formula $\tau_\alpha \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ that characterises all state transitions on behalf of the execution of α. τ_α needs to specify the program state transitions under α from the state characterised by z to the state characterised by z'.

For the simple assignment $x \lhd t$ the proofs of Th. 3.2 p.30 in (Harel, 1979) and Th. 47 p.56 in (Schlager, 2000) chose the following manifest but wrong attempt for a characteristic formula in the expressiveness proof.

$$\tau_{x \lhd t} \;:=\; (x' \doteq t) \tag{4.1}$$

Unfortunately, this choice is substantially flawed as will be illustrated in the following example.

Example 4.5.2 Assuming there is a program α that contains two program variables[27] x, y and thus has a vector z of length 2. For example, consider the program α.

$$\begin{array}{l} x \lhd 7; \\ y \lhd y + 5 \end{array}$$

Then the overall characterising formula τ_α is built from the characterising formulas $\tau_{x \lhd 7}$ resp. $\tau_{y \lhd y+5}$ of the two assignments as follows.

$$\tau_\alpha \;:=\; \exists z'' \, ({\tau_{x \lhd 7}}^{z''}_{z'} \land {\tau_{y \lhd y+5}}^{z''}_{z})$$

More precisely when expanding the vectorial notation z'' and using the explicit notation for substitutions for clarity this equals

$$\exists x'' \, \exists y'' \, (\tau_{x \lhd 7}[x' \mapsto x''][y' \mapsto y''] \land \tau_{y \lhd y+5}[x \mapsto x''][y \mapsto y''])$$

[27] i.e. ODL constant symbols.

When inserting the particular instances for $\tau_{x \lhd 7}$ and $\tau_{y \lhd y+5}$ and performing the substitution this leads to

$$\psi \;:=\; \exists x'' \, \exists y'' \, (x'' \doteq 7 \land y' \doteq y'' + 5)$$

However, ψ does not characterise the transition performed by α. The first-order formula ψ ought to specify the program state transitions from the state characterised by x and y to the state characterised by x' and y'. But both, the new value of x' and of y' are characterised incorrectly. The change of x to x' of the value 7 slips away unnoticed. The internal quantification of x'' restricted to $x'' \doteq 7$ is futile for this purpose, because it does not tell anything about the new value of x'. The formula $\exists y'' \, y' \doteq y'' + 5$ on the other hand only says that y' is the fifth successor of *some* value, not of the value of y in the prestate. ψ thus completely fails to characterise the transition of x as well as y, but almost boils down to the universal state transition relation $\mathrm{dom}(\ell) \times \mathrm{dom}(\ell)$. $\square$

Therefore, the characterisation (4.1) for the assignment statement $x \lhd t$ is incorrect and invalidates the expressiveness and completeness proof in (Harel, 1979) and (Schlager, 2000).

What (4.1) expresses correctly, is the effect that $x \lhd t$ has on the atomic program variable x. However, what it does not even talk about at all is how the assignment $x \lhd t$ relates the values of any other variables y from the prestate to the poststate of the assignment execution. (4.1) acts as if there was no connexion between y and y' at all after an assignment to x has been made, which is wrong.

A solution to the problem is to change the characterising formula for assignments as has been performed in the Lem. 4.5.4 proof presented in this thesis. Consider a program variable vector z of length n as determined by the complete program under investigation. Then even if the current program to characterise, $x \lhd t$, involves less variables (say x, u) we still use the complete vector z of length n. Then the characterising formula can be defined to mention what changes and what remains during the state transition at once.

$$\tau_{x \lhd t} \;:=\; x' \doteq t \land \bigwedge_{x \neq y \in z} y' \doteq y$$

Example 4.5.3 Continuing Ex. 4.5.2, the overall characterising formula of α is

$$\exists x'' \, \exists y'' \, (x'' \doteq 7 \land y'' \doteq y \;\land\; y' \doteq 5 \land x' \doteq x'')$$

which could be simplified by standard quantifier reasoning to

$$x' \doteq 7 \land y' \doteq 5$$

$\square$

4.5.3 Rewrite Completion

This section proves a rewrite termination result. This result says that the term rewrite process caused by update application always comes to completion, which is essential for the completeness proof of assignment treatment.

To emphasise: it is *not* the case that nonterminating rule application would spoil all completeness ambitions. On the contrary, by the intrinsically undecidable nature of the verification problem, (relatively) complete calculi must have rules that – in the general case – can be applied indefinitely. Rather, it is immediately important to provide a finite treatment of updates in order to guarantee that each valid assertion about a complex function assignment can be proven with a finite derivation. An infinite descent of rewrite rule applications would never constitute a proof.

On an intuitive level, a system that bloats formulas during update promotion arbitrarily would not be able to complete many proofs anyway. The formal nature of update promotion, though, spreads the effect of updates exponentially[28] to the subterms and possibly introduces larger terms by case distinctions with conditional terms. It is not at all obvious that this growth will ever come to an end. Therefore, it has to be shown that term rewrite cannot happen indefinitely but that there is always an end to the update application process in the next result.

Lemma 4.5.7 (Noetherian Term Rewritings) *The term rewrite relation* $\rightsquigarrow$ *is* Noetherian*, i.e. repeated term rewrite always terminates.*

Termination is generally a very challenging property to prove. Verifying from scratch that the relation $\rightsquigarrow$ is Noetherian would be a tremendous amount of work. This is due to the *update (match)* (R27) inference rule having two interdependent complexity aspects: number and depth of terms under updates. Even worse, those two aspects behave in an "Ackermanian"[29] way, which means that once the one aspect has been improved by the inference rule, the other aspect is bound to have been impaired. By shifting an update $\langle \mathcal{U} \rangle$ in front of $f(u_1, \ldots, u_n)$ to all its subterms, R27 decreases the depth of the term after $\langle \mathcal{U} \rangle$ but introduces more terms $\langle \mathcal{U} \rangle u_i$ in front of which $\langle \mathcal{U} \rangle$ resides. Moreover, the introduction of the conditional term by R27 further increases the depth of the terms behind any additional update around $\langle \mathcal{U} \rangle f(u_1, \ldots, u_n)$, which again leads to more terms that require update application. What is even worse is that the overall depth of newly introduced update terms does not strictly decrease because the same terms s_i and t of

[28]In the worst case without optimisations.

[29]This notion derives from the Ackermann function that Wilhelm Ackermann (1896 - 1962) has proven to be not primitive recursive, see p.451 in (Cormen *et al.*, 1990).

a positive depth will be unpacked by the update application rule. Deciding whether all those competing aspect changes unite to an overall trend downwards[30] or lead to a circular divergence would be exceedingly nontrivial.

For those reasons, the proof of Lem. 4.5.7 follows a different approach with more involved standard notions on term rewriting. The key idea for proving this lemma is to find a relation $\succ$ that bounds the rewrite relation $\rightsquigarrow$ and is comparably simple to verify as Noetherian. If $\succ$ contains[31] $\rightsquigarrow$, any application of a rewrite must decrease the terms with respect to $\succ$. When $\succ$ is Noetherian, this reduction cannot happen infinitely often. To render this idea more precise and find an appropriate Noetherian relation $\rightsquigarrow$, we have to introduce some technical utilities from (Dershowitz & Plaisted, 2001; Baader & Nipkow, 1998; Dershowitz & Jouannaud, 1990), though.

Definition 4.5.8 *A* rewrite ordering *is an irreflexive and transitive*[32] *relation* $\succ \subseteq \mathrm{Trm}(\Sigma \cup V) \times \mathrm{Trm}(\Sigma \cup V)$ *that satisfies for each* $s, t \in \mathrm{Trm}(\Sigma \cup V)$

(σ)	$s \succ t$	$\Rightarrow$	$\sigma s \succ \sigma t$ *for each substit.* σ	*"stable substitution"*
(mon)	$s \succ t$	$\Rightarrow$	$\phi(s) \succ \phi(t)$ *for each term* $\phi(x)$	*"stable instantiation"*

Remark 4.5.9 *Condition (mon) is equivalent to*

$$s \succ t \quad \Rightarrow \quad \text{for each } f \in \Sigma\ f(\ldots, s, \ldots) \succ f(\ldots, t, \ldots) \quad \text{"monotone"}$$

Rewrite orderings characterise what a term ordering has to fulfil in order for the compatibility with the ordering to inherit from rewrite rules to applications of the rewrite rules. If all rules of a term rewrite relation are compatible with $\succ$, i.e. $\succ$ contains $\rightsquigarrow$, then if $\succ$ is a rewrite ordering, this also holds for all applications of those rules. Applications of rewrite rules happen as instances of the rewrite rule and occur in some context. Instantiation inheritance is handled by (σ), while context inheritance is dealt with by the (mon) condition.

Definition 4.5.10 *A rewrite ordering* $\succ \subseteq \mathrm{Trm}(\Sigma \cup V) \times \mathrm{Trm}(\Sigma \cup V)$ *is a (strict)* simplification ordering *if it satisfies for each* $s \in \mathrm{Trm}(\Sigma \cup V)$

$$(sub) \quad f(\ldots, s, \ldots) \succ s \quad \text{"subterm property"}$$

Theorem 3 (Simplification Ordering) *Every simplification ordering over terms of a finite vocabulary is Noetherian.*

---

[30]With a sufficiently complex notion of which term is "larger", irrespective of its linear size and depth.

[31]in the sense that $\succ \supseteq \rightsquigarrow$, i.e. for each s, t if $s \rightsquigarrow t \Rightarrow s \succ t$.

[32]i.e. it satisfies for each t that $t \succ t$ does not hold, and for each s, t, u $s \succ t$ and $t, u \succ$ then $s \succ u$.

This theorem has a complex proof in (Dershowitz, 1982) based on the deep result of Kruskal's Tree Theorem (Kruskal, 1960; Nash-Williams, 1963). Combining this theorem with another result from the literature (Baader & Nipkow, 1998) allows to find a Noetherian order restraining the term rewrite relation. The vocabulary occurring in a term that experiences rewriting is always finite, regardless of the finitude of the overall signature.

Proposition 4.5.11 *Let $>$ be an ordering on Σ. The* recursive path ordering *(RPO[33]) $\succ$ defined as follows is a simplification ordering.*

(a) $f(s_1, \ldots, s_n) \succ t$ *if there is* i *with* $s_i \succ t$ *or* $s_i = t$.

(b) $f(s_1, \ldots, s_n) \succ g(t_1, \ldots, t_m)$ *if* $f > g$ *and for each* i $f(s_1, \ldots, s_n) \succ t_i$.

(c) $f(s_1, \ldots, s_n) \succ f(t_1, \ldots, t_n)$ *if* $\{\!\{s_1, \ldots, s_n\}\!\} \succ \{\!\{t_1, \ldots, t_n\}\!\}$.

Where the rewrite ordering $\succ$ is extended to multisets M, N according to (Menzel & Schmitt, 2000) as follows.

$$
\begin{aligned}
M \succ N \quad :\Longleftrightarrow \quad & \textit{there is } M^+ \subseteq M,\ N^+ \subseteq N \ \textit{ with } (M \setminus M^+) \cup N^+ = N \\
& \textit{and for each } b \in N^+ \textit{ there is } a \in M^+ \ a \succ b
\end{aligned}
$$

With the above terminology one can attempt to find a recursive path ordering for the proof of Lem. 4.5.7. For the proof to succeed all rewrite rules have to be compatible with the very same ordering. The next result will only demonstrate this compatibility for the most complicated rewrite rule.

Proposition 4.5.12 *The R27 inference rule is compatible with a recursive path ordering.*

Proof: Consider the *update (match)* (R27) rewrite rule in the version relevant for this proof. $\mathcal{U}$ is a short notation for the update $f(s_1, \ldots, s_n) \lhd t$.

$$
\begin{aligned}
& \langle \mathcal{U} \rangle f(u_1, \ldots, u_n) \ \rightsquigarrow \\
& \qquad \mathit{if}\, s_1 \doteq \langle \mathcal{U} \rangle u_1 \wedge \cdots \wedge s_n \doteq \langle \mathcal{U} \rangle u_n \ \mathit{then}\, t \ \mathit{else}\, f(\langle \mathcal{U} \rangle u_1, \ldots, \langle \mathcal{U} \rangle u_n) \ \mathit{fi}
\end{aligned}
$$

Let l abbreviate the left hand side $\langle \mathcal{U} \rangle f(u_1, \ldots, u_n)$, and r abbreviate the right hand side

$$
\mathit{if}\, s_1 \doteq \langle \mathcal{U} \rangle u_1 \wedge \cdots \wedge s_n \doteq \langle \mathcal{U} \rangle u_n \ \mathit{then}\, t \ \mathit{else}\, f(\langle \mathcal{U} \rangle u_1, \ldots, \langle \mathcal{U} \rangle u_n) \ \mathit{fi}
$$

[33] Also referred to as bag ordering.

It remains to show that $l \succ r$. For the purpose of this proof assume that updates formulas like in l have the following abstract syntactic structure[34] in prefix notation.

$$l \;=\; \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;f(u_1,\ldots,u_n)\big)$$

We pick the following (partial) ordering on the vocabulary as a basis of the $\succ$ recursive path ordering:[35]

$$\mathfrak{U} > \textit{if then else fi}, \quad \mathfrak{U} > \doteq, \quad \mathfrak{U} > \wedge, \quad \mathfrak{U} > f$$

$l \succ r$ holds according to (b) of Def. 4.5.11 because $\mathfrak{U} >$ *if then else fi* and

- $\Leftarrow$ $l \succ s_1 \doteq \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_1\big) \wedge \cdots \wedge s_n \doteq \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_n\big)$, which by (b) holds because of $\mathfrak{U} > \wedge$ and
  - $\Leftarrow$ $l \succ s_i \doteq \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_i\big)$, which by (b) holds because of $\mathfrak{U} >\doteq$ and
    - $\Leftarrow$ $l \succ s_i$, which by (a) holds because of
      - $\Leftarrow$ $f(s_1,\ldots,s_n) \succ s_i$ is true by (a).
    - $\Leftarrow$ $l \succ \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_i\big)$. So by (c) it only remains to show that $\{\!\{f(s_1,\ldots,s_n),t,f(u_1,\ldots,u_n)\}\!\} \succ \{\!\{f(s_1,\ldots,s_n),t,u_i\}\!\}$. This in turn holds because of
      - $\Leftarrow$ $f(u_1,\ldots,u_n) \succ u_i$ is true by (a).
- $\Leftarrow$ $l \succ t$ which is true by (a).
- $\Leftarrow$ $l \succ f\Big(\mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_1\big),\ldots,\mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_n\big)\Big)$ which – by (b) holds – because of $\mathfrak{U} > f$ and
  - $\Leftarrow$ $l \succ \mathfrak{U}\big(f(s_1,\ldots,s_n),t,\;u_i\big)$ Thus, by (c) it remains to show that $\{\!\{f(s_1,\ldots,s_n),t,f(u_1,\ldots,u_n)\}\!\} \succ \{\!\{f(s_1,\ldots,s_n),t,u_i\}\!\}$. This in turn holds because of
    - $\Leftarrow$ $f(u_1,\ldots,u_n) \succ u_i$ is true by (a).

■

Along with very similar reasoning for the other rewrite rules, this completes the proof of Lem. 4.5.7. Additionally, for later use note that for a first-order formula $F \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ the fixed-point rewrite $\langle\mathcal{U}\rangle F \;\leadsto^*\; F'$ finally

[34] This precise structure is – to some extent – critical for the success of the proof. For example, ignoring the s_i or t for the abstract syntactic structure would perish the proof.

[35] This whole proof has been produced automatically by a small PROLOG program.

reaches an $F' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ of first-order logic, as can be seen from the fact that, while there is still a remaining update left, it can be rewritten by one of the update rules. Furthermore, any modality occurring is an update since the update rules only introduce new updates and $\langle\mathcal{U}\rangle F$ did not initially contain any other modalities.

For the rewriting continued with an application of one of the update application term rewrite rules R27, R28, R29, implicitly assume that bound α-renaming has been used. Whenever the term rewrite system is about to stop because of the $x \notin FV(\mathcal{U})$ condition of the R29 rewrite rule, α-conversion is used to rename the bound variable x.

4.5.4 Elementary Completeness

The next results prove completeness in an elementary case. Even though having the same formulation as Th. 2 on the surface, those results only express completeness limited to the simpler case of Hoare triples, i.e. pre- and postcondition specifications restricted to first-order logic. In contrast to full ODL, a Hoare specification $F \rightarrow \langle\alpha\rangle G$ cannot contain further modalities within F and G. In comparison to full dynamic logic, first-order Hoare triples tremendously simplify the proof. Further the first-order cases constitute the basis for the final completeness proof for arbitrary formulas of dynamic logic.

Proposition 4.5.13 (Elementary $\langle\rangle$ completeness) *For each program $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ and each $F, G \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ of first-order logic*

$$\vDash F \rightarrow \langle\alpha\rangle G \;\Rightarrow\; \vdash F \rightarrow \langle\alpha\rangle G$$

Proof: From the validity of $\vDash F \rightarrow \langle\alpha\rangle G$ we have to conclude that the calculus can deduce the valid conjecture $\vdash F \rightarrow \langle\alpha\rangle G$ as well. The proof follows an induction of the structure of the program α.

IH As induction hypothesis this proof uses a slightly modified variant: Whenever $\vDash F \rightarrow \langle\alpha\rangle G$ holds for first-order formulas $F, G \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ then $F \vdash \langle\alpha\rangle G$ can be derived. From this the conjecture $\vdash F \rightarrow \langle\alpha\rangle G$ can always be concluded by an application of the R10 inference rule.

1. $\vDash F \rightarrow \underbrace{\langle f(s) \lhd t\rangle}_{\mathcal{U}} G$. By Lem. 4.5.7 there is $\tilde{G}$ with $\langle\mathcal{U}\rangle G \rightsquigarrow^* \tilde{G}$. Moreover the proof of Lem. 4.5.7 shows that $\tilde{G} \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ is first-order. From Prop. 4.4.7 we know that still $\vDash F \rightarrow \tilde{G}$, which is first-order, because of which the ODL calculus can – by assumption of

the notion of relative completeness – also prove $\vdash F \to \tilde{G}$ by R56. Prop. A.2.1 allows to conclude that there also is a derivation of $F \vdash \tilde{G}$. From this the R36 inference rule of the ODL′ calculus allows to continue the derivation to $F \vdash \langle \mathcal{U} \rangle G$.

Here the ODL′ calculus works like the ODL calculus except that term rewrites are restricted to occur according to $\rightsquigarrow^*$ instead of $\rightsquigarrow$. To be more precise, instead of the ODL inference rules R35 and R36, ODL′ contains the following.

$$\text{(R57)} \quad \text{term rewrite (left)} \qquad \frac{\phi(t) \ \vdash}{\phi(s) \ \vdash} \Leftarrow (s \ \rightsquigarrow^* \ t) \text{ holds}$$

$$\text{(R58)} \quad \text{term rewrite (right)} \qquad \frac{\vdash \phi(t)}{\vdash \phi(s)} \Leftarrow (s \ \rightsquigarrow^* \ t) \text{ holds}$$

From Lem. 4.5.7 we know that term rewrites can always be continued to a fixed-point, while Prop. 4.4.7 allows to conclude that the final result will still share the same value. So in a certain sense, ODL′ specialises the ODL calculus to the case that term rewrites always accumulate to one large term rewrite evaluating a single subterm as far as possible. If, by this completeness proof for ODL, we have been able to show ODL′ is relatively complete, then so is ODL even more, because amongst other proofs it allows the emulation of the ODL′ behaviour by successive term rewrites.

2. $\vDash F \to \langle \texttt{if}(\chi)\,\{\gamma\}\texttt{else}\{\delta\}\rangle G$ then according to the semantics in Def. 2.3.8 this also implies that $\vDash F \wedge \chi \to \langle \gamma \rangle G$, resp. $\vDash F \wedge \neg\chi \to \langle \delta \rangle G$. The induction hypothesis allows to conclude $F, \chi \vdash \langle \gamma \rangle G$, resp. $F, \neg\chi \vdash \langle \delta \rangle G$. With the *branch* (R23) inference rule this leads to a derivation of $F \vdash \langle \texttt{if}(\chi)\,\{\gamma\}\texttt{else}\{\delta\}\rangle G$.

3. $\vDash F \to \langle \gamma; \delta \rangle G$ then according to Def. 2.3.8 this also implies that $\vDash F \to \langle \gamma \rangle \langle \delta \rangle G$. By Lem. 4.5.4 there is a first-order formula $G' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ with $\vDash G' \leftrightarrow \langle \delta \rangle G$. From the validity of $\vDash F \to \langle \gamma \rangle G'$ the induction hypothesis allows to conclude

$$F \vdash \langle \gamma \rangle G' \tag{4.2}$$

Because of $\vDash G' \rightarrow \langle\delta\rangle G$ the induction hypothesis allows to conclude $G' \vdash \langle\delta\rangle G$. With an application of $\langle\rangle$ *generalisation* (R46) the derivation can be extended to $\langle\gamma\rangle G' \vdash \langle\gamma\rangle\langle\delta\rangle G$. In combination with (4.2) this allows to derive $F \vdash \langle\gamma\rangle\langle\delta\rangle G$, from which the *composition* (R21) rule allows to derive $F \vdash \langle\gamma;\delta\rangle G$.

4. $\vDash F \rightarrow \langle\mathtt{while}(\chi)\,\{\gamma\}\rangle G$. This case uses a first-order formula Ω expressing the termination of the loop after n iterations in a state satisfying G. It is very similar to the expressibility formula $\tau_{\mathtt{while}(\chi)\,\{\gamma\}}$ from the proof of Lem. 4.5.4, except for the export of the iteration count n.

$$\begin{array}{rcl}\Omega(n) & := & Enc \rightarrow \exists S{:}\mathtt{nat}\ \big(\\ & & \quad nth(S,0) \doteq z \wedge nth(S, length(S)) \doteq z' \\ & & \wedge\ \forall i{:}\mathtt{nat}\ \ (i < length(S) \rightarrow \tau_\alpha(nth(S,i), nth(S,i+1))) \\ & & \wedge\ \forall i{:}\mathtt{nat}\ \ (i < length(S) \rightarrow {\chi'}_z^{nth(S,i)}) \\ & & \wedge\ \neg{\chi'}_z^{z'} \\ & & \wedge\ G_z^{z'} \wedge length(S) \doteq n\big)\end{array}$$

So except for the iteration count export, $\Omega(n)$ and $\tau_{\mathtt{while}(\chi)\,\{\gamma\}}(z,z') \wedge G_z^{z'}$ are equivalent.

$\vDash \Omega(0) \rightarrow G$ and $\vDash \Omega(0) \rightarrow \neg\chi$ are valid. Thus, also $\vDash \Omega(0) \wedge \chi \rightarrow$ *false*. As mere first-order, they are assumed to be derivable by R56. From $\Omega(0)\ \vdash G$ the following formula can be derived by *weakening (left)* (R6).

$$\Omega(0), \neg\chi\ \vdash G \tag{4.3}$$

Likewise from $\Omega(0), \chi\ \vdash$ *false* the *weakening (right)* (R12) inference rule can derive $\Omega(0), \chi\ \vdash \langle\gamma\rangle\langle\mathtt{while}(\chi)\,\{\gamma\}\rangle G$, from which *composition* (R21) can conclude

$$\Omega(0), \chi\ \vdash \langle\gamma; \mathtt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.4}$$

The *branch* (R23) inference rule can combine (4.3) and (4.4) to produce a derivation of $\Omega(0)\ \vdash \langle\mathtt{if}(\chi)\,\{\gamma; \mathtt{while}(\chi)\,\{\gamma\}\}\rangle G$ from which the *loop unwind* (R25) rule derives $\Omega(0)\ \vdash \langle\mathtt{while}(\chi)\,\{\gamma\}\rangle G$ such that the $\rightarrow$ *right* (R10) inference rule can come up with a derivation of the following.

$$\vdash \Omega(0) \rightarrow \langle\mathtt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.5}$$

which lies the anchor of the arithmetic object-level induction.

According to the loop semantics $\vDash \Omega(n+1) \to \langle\gamma\rangle\Omega(n)$ is true, because of which it can be derived by induction hypothesis since γ is less complex. Thus, this provides a derivation for

$$\Omega(n+1) \ \vdash \langle\gamma\rangle\Omega(n) \tag{4.6}$$

The induction hypothesis on the object-level is

$$\Omega(n) \to \langle\texttt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.7}$$

So for the moment let us assume there would be a derivation of

$$\Omega(n) \ \vdash \langle\texttt{while}(\chi)\,\{\gamma\}\rangle G$$

from which $\langle\rangle$ *generalisation* (R46) derives

$$\langle\gamma\rangle\Omega(n) \ \vdash \langle\gamma\rangle\langle\texttt{while}(\chi)\,\{\gamma\}\rangle G$$

which with (4.6) refines to $\Omega(n+1) \ \vdash \langle\gamma\rangle\langle\texttt{while}(\chi)\,\{\gamma\}\rangle G$. Now R21 continues with $\Omega(n+1) \ \vdash \langle\gamma;\texttt{while}(\chi)\,\{\gamma\}\rangle G$, and *weakening (left)* (R6) leads to a derivation

$$\Omega(n+1), \chi \ \vdash \langle\gamma;\texttt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.8}$$

The loop semantics imply $\vDash \Omega(n+1) \to \chi$ alias $\vDash \Omega(n+1) \wedge \neg\chi \to \mathit{false}$, which – as mere first-order – is assumed to be derivable by R56. From $\Omega(n+1), \neg\chi \ \vdash \mathit{false}$ *weakening (right)* (R12) the conclusion can be drawn that

$$\Omega(n+1), \neg\chi \ \vdash G \tag{4.9}$$

The R23 inference rule can combine (4.8) and (4.9) to derive

$$\Omega(n+1) \ \vdash \langle\texttt{if}(\chi)\,\{\gamma;\texttt{while}(\chi)\,\{\gamma\}\}\rangle G$$

which the R25 rule can carry forward to a derivation of

$$\Omega(n+1) \ \vdash \langle\texttt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.10}$$

To put it in a nutshell the above reasoning has constructed a derivation of (4.10) from a sequent variant of (4.7). By Prop. A.2.1 this amounts to a derivation of

$$\Omega(n) \to \langle\texttt{while}(\chi)\,\{\gamma\}\rangle G \ \vdash \Omega(n+1) \to \langle\texttt{while}(\chi)\,\{\gamma\}\rangle G \tag{4.11}$$

The *induction* (R15) inference rule for natural numbers allows to derive from (4.5) and (4.11) the formula

$$\vdash \forall n{:}\mathtt{nat}\ (\Omega(n) \to \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G) \tag{4.12}$$

By premise, $\vDash F \to \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G$ holds, thus also $\vDash F \to \exists n : \mathtt{nat}\ \Omega(n)$, which is first-order and as such assumed derivable by R56. In conjunction with (4.12), quantifier inference rules R14, R13 finally allow to derive $F \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G$ from this as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{\Omega(N), (\Omega(N) \to \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G), F\ \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G}}{\Omega(N), \forall n{:}\mathtt{nat}\ (\Omega(n) \to \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G), F\ \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G}}{\exists n{:}\mathtt{nat}\ \Omega(n), \forall n{:}\mathtt{nat}\ (\Omega(n) \to \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G), F\ \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G}}{\exists n{:}\mathtt{nat}\ \Omega(n), F\ \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G}}{F\ \vdash \langle \mathtt{while}(\chi)\,\{\gamma\}\rangle G}$$

The bottommost two inferences are abbreviated R5 cuts with Prop. A.2.2 according to the derivations already performed above. The topmost situation can be closed by *modus ponens* (R55).

■

Proposition 4.5.14 (Elementary [] completeness) *For each program* $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ *and each* $F, G \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *of first-order logic*

$$\vDash F \to \neg\langle\alpha\rangle G \ \Rightarrow \vdash F \to \neg\langle\alpha\rangle G$$

Or equivalently: for each $\alpha \in \mathrm{Prg}(\Sigma \cup V)$ *for each* $F, G \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *of first-order logic*

$$\vDash F \to [\alpha]G \ \Rightarrow \vdash F \to [\alpha]G$$

Proof: The duality of $\neg\langle\alpha\rangle\neg\psi \equiv [\alpha]G$ establishes the equivalence.

The proof is similar to Prop. 4.5.13. The cases for updates, branching and sequential composition are simple adaptations of the corresponding cases in Prop. 4.5.13. What remains to show is that $F \vdash [\mathtt{while}(\chi)\,\{\gamma\}]G$ can be derived by the *loop induction right* (R45) inference rule. Define the loop invariant as a first-order encoding of the statement that all potential poststates of the loop satisfy G. The formula $\tau_\alpha(z, z')$ stems from the proof of Lem. 4.5.4.

$$p \ := \ Enc \to \forall z'\,(\tau_\alpha(z, z') \to G_z^{z'})$$

Since $F \to p$ and $p \land \neg\chi \to G$ are valid, they are assumed to be derivable as mere first-order by the notion of relative completeness. Thus, R56 provides derivations of

$$F \vdash p \tag{4.13}$$

$$p, \neg\chi \vdash G \tag{4.14}$$

Likewise, $p \land \chi \to [\gamma]p$ is valid and – as γ is of a smaller complexity – derivable by induction hypothesis.

$$p, \chi \vdash [\gamma]p \tag{4.15}$$

From (4.13), (4.15), and (4.14) the *loop induction right* (R45) inference rule allows to conclude the intended derivation. ■

Proposition 4.5.15 *for each* $F \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ *of first-order logic*

$$\vDash F \to t\,\texttt{instanceof}\,\texttt{C} \;\Rightarrow\; \vdash F \to t\,\texttt{instanceof}\,\texttt{C}$$

$$\vDash F \to \neg(t\,\texttt{instanceof}\,\texttt{C}) \;\Rightarrow\; \vdash F \to \neg(t\,\texttt{instanceof}\,\texttt{C})$$

Proof: As a preparation, note that the following correspondence holds in ODL.

$$t\,\texttt{instanceof}\,\texttt{C} \;\equiv\; \exists n\!:\!\texttt{nat}\;\; t \doteq \texttt{obj}_{\leq\texttt{C}}(n) \tag{4.16}$$

If in some state w of an interpretation ℓ, the relation $\ell, w \vDash t\,\texttt{instanceof}\,\texttt{C}$ holds, then due to the circumstance that t is an object of a subtype D of C. Therefore, $val_\ell(w, t) \in O_{\texttt{D}}$ and it holds for some $d \in \mathbf{N}$ by bijectivity of $\texttt{obj}$ according to Def. 2.3.1 that

$$\ell[n \mapsto d], w \vDash\; t \doteq \texttt{obj}_{\texttt{D}}(n)$$

Thus, $\ell, w \vDash \exists n\!:\!\texttt{nat}\;\; t \doteq \texttt{obj}_{\leq\texttt{C}}(n)$ is true. The converse relation is immediate.

If $F \to t\,\texttt{instanceof}\,\texttt{C}$ is a tautology then – by congruence – so is $F \to \exists n : \texttt{nat}\;\; t \doteq \texttt{obj}_{\leq\texttt{C}}(n)$ according to (4.16). Hence, $\vDash F \to \exists n : \texttt{nat}\;\; t \doteq \texttt{obj}_{\leq\texttt{C}}(n)$, which is first-order and thus assumed[36] to be derivable with R56.

[36] Knowing that R41, R42 and R43 immediately characterise the semantical first-order restrictions to $\texttt{obj}$. More precisely, this can be justified as follows. First-order logic without arithmetic is complete. If, instead of first-order bijectivity and disjointness premises for $\texttt{obj}$, inference rules had been added that express the very same meaning then the enhanced first-order logic would still have been complete. Thereafter, the extension of ODL beyond first-order logic can be considered.

A derivation of $\vdash F \rightarrow \exists n : \texttt{nat}\ t \doteq \texttt{obj}_{\leq \texttt{C}}(n)$ can be continued to a derivation of $F \vdash t\,\texttt{instanceof}\,\texttt{C}$ by a *cut* (R5) in the sense of Prop. A.2.2 with the following deduction.

$$\dfrac{\dfrac{\overline{F, t \doteq \texttt{obj}_{\leq \texttt{C}}(N) \ \vdash \texttt{obj}_{\leq \texttt{C}}(N)\,\texttt{instanceof}\,\texttt{C}}}{F, t \doteq \texttt{obj}_{\leq \texttt{C}}(N) \ \vdash t\,\texttt{instanceof}\,\texttt{C}}}{F, \exists n{:}\texttt{nat}\ \ t \doteq \texttt{obj}_{\leq \texttt{C}}(n) \ \vdash t\,\texttt{instanceof}\,\texttt{C}}$$

The proof finally closes by R39.[37]

The derivation of $\vdash F \rightarrow \neg(t\,\texttt{instanceof}\,\texttt{C})$ from the fact that $\models F \rightarrow \neg(t\,\texttt{instanceof}\,\texttt{C})$ is similar, but involves R40. ■

4.5.5 Relative Completeness

Having succeeded with the proofs of the above auxiliary statements it is now possible to finish the proof of the central theorem Th. 2 from p.110 on the basis of the lemmata 4.5.1-4.5.15.

Broadly speaking, the proof proceeds as follows: By propositional recombination, we inductively identify fragments of the formula at hand that have a form qualitatively similar to $\phi_1 \rightarrow \langle\alpha\rangle\phi_2$. Its constituents ψ_i will then be expressed equivalently in first-order logic with the power of arithmetic encoding by Lem. 4.5.4. Finally, the proof relies on Prop. 4.5.13 and Prop. 4.5.14 to resolve the first-order Hoare triple base cases.

Proof: The proof of Th. 2 follows a basic structure analogue to that of Th. 3.1 of §3.1 on p.28 in (Harel, 1979). For a formula ϕ with $\ell \models \phi$ for each arithmetical structure ϕ, it has to be shown that $\vdash \phi$ can be proven within the ODL calculus. A reasonable proof of this conjecture needs a number of preparations to treat the separate aspects of ODL in isolation as much as possible.

1. If ϕ is of the form $\psi(\mathit{if}\, e\ \mathit{then}\, s\ \mathit{else}\, t\ \mathit{fi})$ then (inductively) consider the simpler[38] and – by Prop. 4.4.4 – equivalent formula $(e \rightarrow \psi(s)) \wedge (\neg e \rightarrow \psi(t))$ instead. From this, the deduction of ϕ can be concluded with an application of R38.

[37] Prior to the application of R16, the notational abbreviation $\texttt{obj}_{\leq \texttt{C}}(N)$ has to be resolved into several branches involving $\texttt{obj}_{\texttt{D}}(N)$ for $D \leq \texttt{C}$ instead. Further, in this derivation, the range of effect of $\doteq$ *subst* (R16) has been restricted by Prop. 4.2.5 to prevent performing replacements in F, though this is uncritical to the success of the proof.

[38] Simpler in terms of the number of occurrences of *if then else fi* .

2. For preparation let us remark that one can assume ϕ to be presented in conjunctive normal form. Otherwise let $\tilde{\phi}$ be the conjunctive normal form of ϕ. Provided that there is a derivation of $\vdash \tilde{\phi}$ there is already one of $\vdash \phi$, since $\vDash \phi \leftrightarrow \tilde{\phi} \overset{4.5.1}{\Rightarrow} \vdash \phi \leftrightarrow \tilde{\phi} \overset{\vdash\tilde{\phi}}{\Rightarrow} \vdash \tilde{\phi} \wedge (\tilde{\phi} \rightarrow \phi) \overset{R55}{\Rightarrow} \vdash \phi$.

The remainder of the proof follows an induction on a measure $|\phi|$ with the following definition.

$$|\phi| \quad := \quad \text{number of modalities, quantifiers}^{39}\text{or } \texttt{instanceof} \text{ in } \phi$$

IA $n = 0 \Rightarrow \phi \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ is first-order $\overset{4.5.2}{\Rightarrow} \vdash \phi$.

IS $n > 0$ leaves several cases according to the conjunctive normal form of ϕ.

- $\phi = \phi_1 \wedge \phi_2$, then individually deduce the simpler proofs for $\vdash \phi_1$ and $\vdash \phi_2$, which are combined by R8. Likewise reasoning handles the case $\phi = \neg\phi_1$.
- ϕ is a disjunction and – without loss of generality[40] – has one of the following forms

$$\begin{array}{ccl} \phi_1 & \vee & \langle\alpha\rangle\phi_2 \\ \phi_1 & \vee & [\alpha]\phi_2 \\ \phi_1 & \vee & \exists x\, \phi_2 \\ \phi_1 & \vee & \forall x\, \phi_2 \end{array}$$

  As a unified notation for those four cases use $\phi_1 \vee \rho\phi_2$.

  Without loss of generality, it can further be assumed that $\rho\phi_2$ is not first-order since we could otherwise consider $\rho\phi_2 \vee \phi_1$ instead. If ϕ_1 was first-order as well, then the case $n = 0$ would apply.

  The fact that $\rho\phi_2$ is not first-order allows to conclude $|\phi_2| < |\phi|$ because ϕ_2 already has less modalities or quantifiers. Likewise $|\phi_1| < |\phi|$ because $\rho\phi_2$ contributes one modality or quantifier to $|\phi|$ that is not part of ϕ_1.

  According to Lem. 4.5.4 there are first-order formulas $\phi_1', \phi_2' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ with $\vDash \phi_i \leftrightarrow \phi_i'$ for $i = 1, 2$. Thus, by congruence

[39] In this case quantifiers in front of mere first-order formulas will not be taken into account.

[40] Otherwise use associativity and commutativity to select a different order for the disjunction.

this leads to $\models \phi_1' \vee \rho(\phi_2') \Rightarrow \models \neg\phi_1' \rightarrow \rho(\phi_2')$. Then – depending on the particular case of ρ – Prop. 4.5.13, Prop. 4.5.14 or Prop. 4.5.2 derive

$$\vdash \neg\phi_1' \rightarrow \rho(\phi_2') \tag{4.17}$$

Further $\models \phi_1 \leftrightarrow \phi_1'$ implies $\models \neg\phi_1 \rightarrow \neg\phi_1'$, which, because of $|\phi_1| < |\phi|$, is derivable as well. In combination with (4.17), Prop. 4.5.1 allows to derive from $\vdash \neg\phi_1 \rightarrow \neg\phi_1'$ that

$$\vdash \neg\phi_1 \rightarrow \rho(\phi_2') \tag{4.18}$$

Likewise $\models \phi_2' \leftrightarrow \phi_2$ implies $\models \phi_2' \rightarrow \phi_2$, which is derivable because of $|\phi_2| < |\phi|$. From $\vdash \phi_2' \rightarrow \phi_2$ Prop. 4.5.3 allows to extend the derivation to $\vdash \rho\phi_2' \rightarrow \rho\phi_2$. Finally Prop. 4.5.1 allows to combine the latter with (4.18) to produce a derivation $\vdash \neg\phi_1 \rightarrow \rho\phi_2$ alias $\vdash \phi_1 \vee \rho\phi_2$.

– ϕ is a disjunction and has one of the following forms

$$\begin{array}{lcl} \phi_1 & \vee & t\,\texttt{instanceof}\,\texttt{C} \\ \phi_1 & \vee & \neg(t\,\texttt{instanceof}\,\texttt{C}) \end{array}$$

As a unified notation for those two cases use $\phi_1 \vee \rho\phi_2$.

The fact that $\rho\phi_2$ contributes one `instanceof` to $|\phi|$, which is not part of ϕ_1, entails $|\phi_1| < |\phi|$.

According to the characterisation in Lem. 4.5.4 there is a first-order formula $\phi_1' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ with $\models \phi_1 \leftrightarrow \phi_1'$. Thus, by congruence this leads to $\models \phi_1' \vee \rho\phi_2 \Rightarrow \models \neg\phi_1' \rightarrow \rho\phi_2$. From this, Prop. 4.5.15 derives

$$\vdash \neg\phi_1' \rightarrow \rho\phi_2 \tag{4.19}$$

The rest of the proof works similar to the above case, yet using (4.19) instead of (4.17). $\models \phi_1 \leftrightarrow \phi_1'$ implies $\models \neg\phi_1 \rightarrow \neg\phi_1'$, which, because of $|\phi_1| < |\phi|$, is derivable as well. In combination with (4.19), Prop. 4.5.1 allows to derive from $\vdash \neg\phi_1 \rightarrow \neg\phi_1'$ that $\vdash \neg\phi_1 \rightarrow \rho\phi_2$.

■

All sound dynamic logic inference rules not having been used during the proof of Th. 2 are bound to be derived rules. Even though they are – to some extent – superfluous from a logical perspective, they are important from a pragmatical perspective in order to keep proofs down to a manageable complexity. Finally, Lem. 4.2.7 reveals the ODL proof system relying on merged parallel updates instead of singleton updates to remain relatively complete.

4.5.6 Completeness Lifting

ODL has been proven relatively complete in Th. 2 with the proof extending from §4.5.1 on p.111 to §4.5.4 as far as p.134. The completeness proof further imports some complicated proofs from results in earlier sections. Usually, one would prefer not to repeat such a tremendous effort when moving on to a calculus for another object-oriented programming language.

When confronted with programs of an object-oriented programming language with more native features than ODL, verification can follow either one of the two foreign language approaches drafted in §1.3. The intermediate language translation approach 2 works perfectly well and does not need any adaptation of the completeness proof. The full-custom calculus and language extension approach 3, instead, necessitates a repetition and adaptation of some parts of the completeness proof. We investigate this adaptation and analyse the portability of the completeness proof: the question arises how much effort can be reused for a completeness proof of an extended logic and which parts have to be reconsidered.

The results from §4.5.1 remain valid for extended logics. Lem. 4.5.4 has to be proven again. However, those parts of the structurally inductive proof concerning features of the language that have not been affected by the extension can be reused. Likewise, Prop. 4.5.13 and Prop. 4.5.14 need to be proven again for the new or modified syntactic elements. For changes that affect large parts of the language semantics like side-effecting expression evaluation or partial expression evaluation due to exception raising, this can still be a tremendous amount of work.

For those reasons, the next result provides a criterion that allows to lift the completeness proof for the ODL logic to extended logics with less effort.

Proposition 4.5.16 (Completeness Extension) *Let $L' > L$ be a logic that is an extension of the logic L, i.e. all formulas of L are formulas of L' and within those formulas they share the same tautologies. Let $R' \supset R$ be a calculus of L that contains the calculus R of L. If*

(i) R is complete for L, and

(ii) $R'_e := \{r \in R' : r$ (locally) equivalent$\}$ finally reduces to L, i.e. for each $\phi' \in \mathrm{Fml}_{L'}(\Sigma \cup V)$ there is $\Phi \subseteq \mathrm{Fml}_L(\Sigma \cup V)$ such that $\Phi \vdash \phi'$ with inference rules from R'_e.

then R' is complete for L'.

Proof: Let $\phi' \in \mathrm{Fml}_{L'}(\Sigma \cup V)$ with $\vDash \phi'$, i.e. for each interpretation ℓ $\ell \vDash \phi'$. We are bound to show that $\vdash \phi'$.

Let $\Phi \subseteq \mathrm{Fml}_L(\Sigma \cup V)$ according to (ii). Since the inference rules in R'_e are (locally) equivalent transformations $\ell \vDash \phi'$ implies $\ell \vDash \Phi$ inductively. This argument can be repeated for any ℓ, because of which $\vDash \Phi$ can be concluded. By (i) this implies $\vdash \Phi$. By transitivity of $\vdash$ this can be combined with $\Phi \vdash \phi'$ to prolongate to the desired derivation of $\vdash \phi'$. ■

Note that – contrary to most other transformation situations – confluence is not an issue for the reduction in (ii). It is of no importance at all whether the additional inference rules of R' can agree on one single target translation of the additional concepts or lead to a variety of different translations. The local equivalence ensures that whatever form the particular final translation happens to take, it is an adequate starting point for proving the original statement without losing relative completeness. It is, though, vitally important for the overall argument that the translation process finally terminates.

Corollary 4.5.17 *A calculus for* JavaCardDL *that extends the* ODL *calculus by additional inference rules is relatively complete for* JavaCardDL *if condition* (ii) *holds.*

Proof: The restriction to relative completeness instead of completeness just reflects the addition of the same inference rule *First-Order Oracle* (R56). ■

Now, examine some examples of JavaCardDL inference rules that permit the treatment of additional JavaCard features, which are not part of ODL, without losing soundness or relative completeness.

Example 4.5.4

$$\frac{\vdash \langle \mathcal{U}; \texttt{while}(\chi)\ \{\alpha;\gamma\}\rangle\phi}{\vdash \langle \texttt{for}(\mathcal{U};\chi;\gamma)\{\alpha\}\rangle\phi}$$

This is a locally equivalent[41] inference rule that directly reduces `for` loops to ODL in a single inference. Thus, the additional language feature of `for` loops will be completely dealt with by adding this single inference rule to the ODL calculus. Likewise reasoning concludes the case of `do-while` loops with the following inference rule.

$$\frac{\vdash \langle \alpha; \texttt{while}(\chi)\ \{\alpha\}\rangle\phi}{\vdash \langle \texttt{do}\{\alpha\}\texttt{while}(\chi)\rangle\phi}$$

□

[41] Thus, it would be part of R'_e.

Example 4.5.5 The following inference rule performs the transformation from §3.4.2 on-the-fly.

$$\frac{\vdash \langle b \lhd \mathtt{obj_C}(\mathtt{next_C}), \mathtt{next_C} \lhd \mathtt{next_C} + 1\rangle \phi}{\vdash \langle b \lhd \mathtt{new\,C}()\rangle \phi}$$

is a locally equivalent[42] inference rule that directly reduces to ODL. Thus, all atomic object creation statements will be dealt with adequately by adding this inference rule to the ODL calculus, apart from the fact that it does not yet invoke the constructor. But this is just a matter of openly embedding the constructor body due to the lack of dynamic dispatch for constructor calls. □

Example 4.5.6 Due to the presence of side-effects the order of evaluation is important in JAVA. Therefore, JAVACARDDL has expression evaluation inference rules, which ensure that the calculus respects the JAVA evaluation order. The JAVACARDDL inference rules successively transform expression evaluations that possibly involve side-effecting subexpressions to a sequence of assignments similar to the effect of the preprocessing transformation in §3.4.1. Consider what happens to completeness when adding those inference rules, which are aware of side-effecting expression evaluation. In this example, v_i will always denote atomic program variables, while e_i denotes either compound or atomic expressions. JAVACARDDL contains a multitude of expression evaluation inference rules of the following flavour to treat compound subexpressions.[43]

$$\frac{\vdash \langle v_0 = e_0; v_1 = e_1; \dots v_n = e_n; v_0.a = op(v_1, \dots, v_n)\rangle \phi}{\vdash \langle e_0.a = op(e_1, \dots, e_n)\rangle \phi}$$

$$\frac{\vdash \langle v_0 = e_0; v_1 = v_1 + 1; v_0.a = v_1\rangle \phi}{\vdash \langle e_0.a = \mathtt{++}v_1\rangle \phi}$$

Where op is an arbitrary JAVA operator with arity n and the new variable v_i has the same type as the expression e_i. Those inference rules are locally equivalent transformations.

Contrary to the previous examples those inference rules do not generally reduce formulas to ODL directly but may need considerably more than one

[42]Notice that the soundness of this inference rule is bound to the fact that *update occurrence* (R48) does not remove updates to $\mathtt{next_C}$ as long as an $\mathtt{new\,C}()$ term could still expand to a $\mathtt{next_C}$ by this inference rule.

[43]This example is about JAVA assignments instead of ODL updates. Therefore, in order to avoid confusion the assignment notation $a = b$ is preferred instead of $a \lhd b$ in this context.

inference to do so. Such a situation arises when the compound subexpressions e_i are again subject to side-effects and need further reduction by those rules. Once ordinary ODL expressions have been reached, an assignment $a = b$ would constitute a well-formed update $a \lhd b$.[44] Further, as soon as none of the above mentioned types of inference rules is applicable any more, ODL expressions must have been reached. Thus, in order to show condition (ii) of the completeness extension criterion termination has to be proven.

Proposition 4.5.18 *The transformation performed by the above extended evaluation order inference rules terminates.*

Proof: To prove termination, pick a number $b \in \mathbf{N}$ that is strictly greater than the arities of all JAVA operators plus 1. With this one can define a measure for terms that decreases in a Noetherian way during the transformation.

$$\begin{aligned} |\Upsilon(u_1, \ldots, u_n)| &:= b \cdot \sum_{i=1}^{n} |u_i| \\ |v| &:= 1 \end{aligned}$$

Then it remains to show that the overall measure decreases by each application of the above inference rules. Consider

$$\frac{\vdash \langle v_0 = e_0; v_1 = e_1; \ldots v_n = e_n; v_0.a = op(v_1, \ldots, v_n)\rangle\phi}{\vdash \langle e_0.a = op(e_1, \ldots, e_n)\rangle\phi}$$

Then the overall measure of the conclusion in comparison with the premise is as follows.

$$\begin{aligned} |e_0.a| + |op(e_1, \ldots, e_n)| &= b \cdot |e_0| + b \cdot \sum_{i=1}^{n} |e_i| \\ &= b \cdot \sum_{i=0}^{n} |e_i| \\ &= \sum_{i=0}^{n} |e_i| + (b-1) \cdot \sum_{i=0}^{n} |e_i| \\ &\geq \sum_{i=0}^{n} |e_i| + (b-1) \cdot (n+b) \\ &\geq \sum_{i=0}^{n} |e_i| + (b-1) \cdot b \end{aligned}$$

[44] For simplicity assume such a conversion to happen automatically as soon as possible.

$$
\begin{aligned}
&\overset{b-1>n}{>} && \sum_{i=0}^{n} |e_i| + b \cdot n \\
&= && \sum_{i=0}^{n} |e_i| + |op(v_1, \ldots, v_n)|
\end{aligned}
$$

The first inequality holds since – by premise – not all subexpressions e_i are mere atomic variables, otherwise the program would already constitute an ODL update so that the translation would have terminated.[45] ■

As this proof shows, adding the evaluation order inference rules to ODL still leads to a relatively complete calculus for object-oriented programming languages in the presence of side-effecting expression evaluation. □

[45]Notice that the $|v_i|$ and $|v_0.a|$ occurrences from the left hand sides of the new temporary value assignments do not contribute to $|\cdot|$. This is due to the circumstance that they could provide overhead when most of the e_i are already atomic variables and $n \leq 1$. However, a straightforward optimisation of the transformation rule avoids the copying to the temporary variable v_i for atomic e_i, thereby circumventing the counting problem altogether.

Chapter 5

Extensions

5.1 Object Createdness

Example 5.1.1 (Motivation) Consider the following conjecture ϕ about a program α involving object creation.

$$\begin{array}{l} b \neq \texttt{null} \rightarrow \\ \langle b.x \lhd b.x + 1; \\ \ c \lhd \texttt{new}\,\texttt{C}(); \\ \ c.x \lhd b.x + 2 \rangle c.x \doteq b.x + 2 \end{array}$$

Though astonishing from a programming language point of view, this seemingly basic conjecture cannot be proven. This is due to the fact that the proof vitally depends on the ability to infer that b and c are distinct objects. Otherwise the postcondition $c.x \doteq b.x+2$ cannot ever be true, since $c.x \doteq c.x+2$ is always false.

The Java language semantics, however, ensure for whatever object b refers to, that this object must have been created at some time. Since b already exists initially and prior to executing α, its object must also have been created prior to the allocation of c during α. Since object creation produces distinct objects in distinct invocations, b and c can be inferred with this meta-reasoning to refer to different objects.

Within the ODL calculus from §4.2, proving that b and c differ is impossible, though. Yet this proves to be the desired outcome for a sound calculus, since – for validity – the ODL language semantics permit b and c to draw their initial values from *any* state of any interpretation. Whereas, choosing $val_\ell(w, b) = val_\ell(w, \texttt{obj}_\texttt{C}(5))$ is quite possible for ODL states, even in the worst case of $val_\ell(w, \texttt{next}_\texttt{C}) = 5$, i.e. b refers to the (in w still uncreated) object that *will* be returned on the next invocation of $\texttt{new}\,\texttt{C}()$, which constitutes a counter-example to ϕ.[1] In general, the conjecture ϕ is untrue, but only in states that cannot occur during the execution of JAVA programs anyway.

In C++, referring to arbitrary locations right in the middle of memory that have not been issued by proper object creation before is very well possible with pointer arithmetics. In α, if b has been assigned an arbitrary memory location by pointer arithmetics of some previous program run then b might just happen to refer to the object that will be produced by object creation in $c \lhd \mathtt{new}\,\mathtt{C}()$. Though this coincidence might seem unlikely, this is what program verification is about: proving that programs work under *all* circumstances, not just the sufficiently "regular" ones respected when programming.

Thus, for ODL to provide a faithful representation of both JAVA and C++ programs, referring beyond the limit of $\mathtt{next}_\mathtt{C}$ has to be possible. Still there is a difference in the amount of structural information about the program that needs to be accessible to the prover. In particular, as a C++ translation, conjecture ϕ has to be provably false, while as a JAVA translation[2], ϕ should be true. The most simple way of achieving this is to add additional premises to the specification in the case of JAVA that express proper createdness constraints for all symbols completely under control of the program. □

The question of whether or not symbols occurring in an ODL program can denote uncreated objects is translation dependent. The remainder of this chapter considers the case of an ODL translation with JAVA as source language.

Definition 5.1.1 (Object Createdness) *For a term $t \in \mathrm{Trm}(\Sigma \cup V)_\mathtt{C}$ define an abbreviation for the createdness formula*[3]

$$\mathit{created}(t) \;:=\; \exists n{:}\mathtt{nat}\ \left(n < \mathtt{next}_{\leq \mathtt{C}} \land t \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)\right)$$

A state w of an interpretation ℓ is called created *if all program objects have been created, i.e.*

$$\textit{for each } t \in \mathrm{Trm}(\tilde{\Sigma} \cup \emptyset) \textit{ it is } \ell, w \vDash \mathit{created}(t)$$

Where $\tilde{\Sigma} := \Sigma \setminus \{\mathtt{obj}_\mathtt{C} : C \in \mathrm{TYP}\}$ is the set of ground terms not including $\mathtt{obj}_\mathtt{C}$.[4] *The members of $\mathrm{Trm}(\tilde{\Sigma} \cup \emptyset)$ will be referred to as* program terms.

[1] Remember that the object creation in §3.4.2 ensures that $\mathtt{next}_\mathtt{C}$ always marks the OID that will experience the next creation and is *strictly* larger than the OIDs of any objects that have already been created so far.

[2] This also holds for a translation from MANAGED C++ (Thai & Lam, 2001).

[3] In contrast to the notion of existence, which is immediate in constant domain semantics, createdness enjoys varying domain semantics.

[4] $\mathtt{obj}_\mathtt{C}(1000)$ does not need to be created, of course.

Remark 5.1.2 *Adding the family of premises below to the specification of a program originating from a* Java *translation is sufficient for proving conjectures like that from Ex. 5.1.1.*

$$
\begin{aligned}
&\textit{for each constant symbol } c \in \Sigma \\
&\quad \textit{created}(c) \\
&\textit{for each function symbol } f \in \tilde{\Sigma} \\
&\quad \forall x_1 \ldots \forall x_n \, (\textit{created}(x_1) \wedge \cdots \wedge \textit{created}(x_n)) \rightarrow \textit{created}(f(x_1, \ldots, x_n)))
\end{aligned}
$$

In the case of Ex. 5.1.1, the relevant family of premises is pleasantly small due to the limited amount of symbols occurring in the program. In general, the approach in Rem. 5.1.2 produces unnecessarily large formulas, though. Rather than adding such a large family of premises, ODL pursues a more implicit solution, which exhibits a high performance in practical proving scenarios.

The basic idea underlying this concept is to provide createdness information about program elements on demand rather than on the basis of explicit additions to the specification in advance. Therefor, the family of explicit premises from Rem. 5.1.2 will be replaced by a single schematic inference rule expressing universal createdness: R59 in Fig. 5.1. Although, as the discussion in Ex. 5.1.1 shows, an inference according to R59 is not sound for all states but only for those encountered during the execution of Java programs. Therefore, in this chapter, states are restricted to let program terms denote created objects only, thereby reducing the set of models a specification has to comply with and further enlarging the set of true specifications about ODL programs originating from Java.

Definition 5.1.3 *For the remainder of this chapter, all states are restricted to be created and all programs are restricted to preserve createdness*[5]*, thereby attaining a well-defined semantics. When necessary to avoid confusion, the logic with states and programs restricted as aforementioned is called* ODL$^+$.

Remark 5.1.4 *The requirements of Def. 5.1.3 are met in the case of* Java. *The machine states, in which the* Java Virtual Machine^{TM} *(Lindholm & Yellin, 1999) starts execution of a* Java *program, correspond to created states. Moreover, createdness is a property preserved within states under execution of* ODL *programs translated from* JavaCardDL.

Proof: (Sketch) According to the translation in Chapt. 3 from JavaCardDL, the only access to new objects happens in the form of $\mathtt{obj_C(next_C)}$ in §3.4.2,

[5]i.e. the execution of programs never leaves the set of created states.

where $\mathtt{next}_\mathtt{C}$ increases simultaneously, thereby ensuring that the new denotation refers to a (freshly) created object. A decrease or otherwise manipulation of $\mathtt{next}_\mathtt{C}$ does not occur in the case of JavaCardDL, which ensures that objects once having been created stay created and the interpretation of constant symbols cannot be "invalidated" during a transition. ■

5.2 Calculus Extension

(R59) `new` created

$$\frac{}{\vdash \exists n{:}\mathtt{nat}\ (n < \mathtt{next}_{\leq \mathtt{C}} \land t_\mathtt{C} \doteq \mathtt{obj}_{\leq \mathtt{C}}(n))}$$

(R60) `new` created remark

$$\frac{\langle\mathcal{U}\rangle(t_\mathtt{C} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \land X < \mathtt{next}_{\leq \mathtt{C}}),\ \langle\mathcal{U}\rangle t_\mathtt{C} \doteq \mathtt{obj}_\mathtt{D}(n)\ \vdash}{\langle\mathcal{U}\rangle t_\mathtt{C} \doteq \mathtt{obj}_\mathtt{D}(n)\ \vdash}$$

(R61) `new` created remark

$$\frac{\langle\mathcal{U}\rangle(t_\mathtt{C} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \land X < \mathtt{next}_{\leq \mathtt{C}})\ \vdash \langle\mathcal{U}\rangle t_\mathtt{C} \doteq \mathtt{obj}_\mathtt{D}(n)}{\vdash \langle\mathcal{U}\rangle t_\mathtt{C} \doteq \mathtt{obj}_\mathtt{D}(n)}$$

Figure 5.1: Object Createdness Inference Rules for Java and Managed C++. Here, X is a new variable and $t_\mathtt{C} \in \mathrm{Trm}(\tilde{\Sigma} \cup \emptyset)_\mathtt{C}$ a ground term of static type `C`.

The extended ODL calculus ODL$^+$ consists of the inference rules from Fig. 5.1 in addition to those in §4.2. As a notational simplification, consider the following abbreviation.

Definition 5.2.1

$$n < \mathtt{next}_{\leq \mathtt{C}}\ \land\ x \doteq \mathtt{obj}_{\leq \mathtt{C}}(n) \quad := \quad \bigvee_{\mathtt{D} \leq \mathtt{C}} (x \doteq \mathtt{obj}_\mathtt{D}(n) \land n < \mathtt{next}_\mathtt{D})$$

The `new` *created* (R59) rule expresses that *all* objects are generated by new object creation expressions, i.e. every object that exists must have been created at some time. By the syntactical restriction of $t_\mathtt{C}$, R59 and its descendants do not apply for terms of the form $\mathtt{obj}_\mathtt{C}(s)$. However, discovering about the term $\mathtt{obj}_\mathtt{C}(s)$ that it can be written in the form $\mathtt{obj}_\mathtt{C}(n)$ for some n

is not a considerably remarkable insight to gain. Finally, note that the R60, R61 inference rules are intended to be continued with R16 in order to utilise the new knowledge about the structure of $t_{\mathtt{C}}$.

According to Def. 5.1.3, the set of possible denotations for constant symbols extends[6] with the transition from state to state. Constant and function symbols from $\tilde{\Sigma}$ thus enjoy varying domain semantics in extended ODL. In order to avoid technical subtleties with replacing possibilist quantification by actualist quantification for proper varying domain semantics, variables keep their unrestricted and state-independent domain of possible interpretations, though. Essential for the language to retain overall constant domain semantics and a possibilist-style calculus (Fitting & Mendelsohn, 1998) in this semi-varying domain setting is the restriction of R59 and its descendants to program terms. This phenomenon will be illustrated in the next example.

Example 5.2.1 (Variable Createdness) A (prohibited) application of the R59 inference rule to a variable would be "unfortunate"[7]. Consider a formula with a universal quantification

$$\forall x\!:\!C \;\; \phi(x) \tag{5.1}$$

from which R17 allows to infer $\phi(y)$. Introducing R59 for an illegal $t_{\mathtt{C}} = y$ as a new premise in the antecedent according to Prop. A.2.2 leads to[8]

$$\exists n\!:\!\mathtt{nat} \;\; (n < \mathtt{next}_{\mathtt{C}} \land y \doteq \mathtt{obj}_{\mathtt{C}}(n)) \;\vdash \phi(y)$$

After application of R14 and R16 this yields

$$N < \mathtt{next}_{\mathtt{C}}, y \doteq \mathtt{obj}_{\mathtt{C}}(N) \;\vdash \phi(\mathtt{obj}_{\mathtt{C}}(N)) \tag{5.2}$$

Although both formulas are (implicitly) universally quantified, (5.2) expresses a considerably weaker circumstance than (5.1), since (5.2) is restricted by $N < \mathtt{next}_{\mathtt{C}}$ to objects that have already been created in the current state, while (5.1) extends to all objects that will ever exist at any time. It is simple to add premises to the antecedent, which falsify (5.1) but validate the (ill-)derived (5.2). □

[6]Strictly extending under transitions that increase $\mathtt{next}_{\mathtt{C}}$.

[7]It might even be incorrect, since – in general – it would be possible to derive contradictions. Terms containing variables suffer from the same problem as they are not under complete control of the program.

[8]For simplicity suppressing the subtype hierarchy aspects, here.

Proposition 5.2.2 *The following inference rules are derived rules*

(R60) `new` created remark

$$\frac{\langle\mathcal{U}\rangle(t_{\mathtt{C}} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \wedge X < \mathtt{next}_{\leq \mathtt{C}}),\ \langle\mathcal{U}\rangle t_{\mathtt{C}} \doteq \mathtt{obj}_{\mathtt{D}}(n)\ \vdash}{\langle\mathcal{U}\rangle t_{\mathtt{C}} \doteq \mathtt{obj}_{\mathtt{D}}(n)\ \vdash}$$

(R61) `new` created remark

$$\frac{\langle\mathcal{U}\rangle(t_{\mathtt{C}} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \wedge X < \mathtt{next}_{\leq \mathtt{C}})\ \vdash \langle\mathcal{U}\rangle t_{\mathtt{C}} \doteq \mathtt{obj}_{\mathtt{D}}(n)}{\vdash \langle\mathcal{U}\rangle t_{\mathtt{C}} \doteq \mathtt{obj}_{\mathtt{D}}(n)}$$

Proof: R60 and R61 are special instances of the following inference rules, which will be proven to be derived rules.

$$\frac{\langle\mathcal{U}\rangle(t_{\mathtt{C}} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \wedge X < \mathtt{next}_{\leq \mathtt{C}}),\ \langle\mathcal{U}\rangle\phi(t_{\mathtt{C}})\ \vdash}{\langle\mathcal{U}\rangle\phi(t_{\mathtt{C}})\ \vdash}$$

$$\frac{\langle\mathcal{U}\rangle(t_{\mathtt{C}} \doteq \mathtt{obj}_{\leq \mathtt{C}}(X) \wedge X < \mathtt{next}_{\leq \mathtt{C}})\ \vdash \langle\mathcal{U}\rangle\phi(t_{\mathtt{C}})}{\vdash \langle\mathcal{U}\rangle\phi(t)}$$

Even though the above inference rules are more general than R60 and R61, this more focused application context leads to a higher goal orientation. R60 and R61 only annotate additional knowledge about generatedness when there are first signs of it being of potential use. R60 and R61 considerably extend the degree of automation in the verification system. For soundness, though, this circumstance is of no importance, because of which the proof continues for the more general and less focused variants.

The soundness of the inference rule is not bound to the presence of $\langle\mathcal{U}\rangle\phi(t)$. Instead, applying the inference rule to annotate the generatedness of arbitrary objects is always possible. Yet, this information about generatedness is only helpful in order to produce additional facts about the terms that are actually present in the current proof obligation. Thus, the occurrence constraint of $\langle\mathcal{U}\rangle\phi(t)$ has a character more directed towards the goal rather than being a logical prerequisite. Matching on occurrence of $\langle\mathcal{U}\rangle\phi(t)$ only directs the attention to the generatedness of terms that really could play a role in the proof. This occurrence constraint prevents rule applications that do not make sense in the current scenario.

By the above consideration, $\langle\mathcal{U}\rangle\phi(t)$ will be left out completely, thereby both inference rules can be treated simultaneously. R46 allow to derive the following from axiom R59

$$\vdash \langle\mathcal{U}\rangle\exists n{:}\mathtt{nat}\ (n < \mathtt{next}_{\leq \mathtt{C}} \wedge t \doteq \mathtt{obj}_{\leq \mathtt{C}}(n))$$

With R5 in Prop. A.2.2, this can be inserted into the antecedent, rendering possible the following derivation with the help of R14.

$$\dfrac{\dfrac{\overline{\langle\mathcal{U}\rangle(X < \mathtt{next}_{\leq \mathtt{C}} \wedge t \doteq \mathtt{obj}_{\leq \mathtt{C}}(X)) \ \vdash}}{\exists n{:}\mathtt{nat}\ \langle\mathcal{U}\rangle(n < \mathtt{next}_{\leq \mathtt{C}} \wedge t \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)) \ \vdash}}{\langle\mathcal{U}\rangle\exists n{:}\mathtt{nat}\ (n < \mathtt{next}_{\leq \mathtt{C}} \wedge t \doteq \mathtt{obj}_{\leq \mathtt{C}}(n)) \ \vdash}$$

The inference permuting quantifiers and modalities is crucial, here. An α-renaming ensures that $\langle\mathcal{U}\rangle$ does not contain n. Constant domain semantics are characterised by the conjunction of *Barcan* formulas (Fitting & Mendelsohn, 1998)[9] and *Converse Barcan* formulas:[10]

$$\begin{aligned} \langle\alpha\rangle\exists x\,\phi &\ \rightarrow\ \exists x\,\langle\alpha\rangle\phi \\ \exists x\,\langle\alpha\rangle\phi &\ \rightarrow\ \langle\alpha\rangle\exists x\,\phi \end{aligned}$$

Therefore, having constant domain semantics for variables, the (Converse) Barcan formulas are true in ODL, and – by Th. 2 – can be proven formally. Hence, the quantifier-modality permutation of $\exists n{:}\mathtt{nat}$ and $\langle\mathcal{U}\rangle$ is derivable. ■

As a preparation for a soundness and relative completeness theorem for ODL$^+$ consider the following auxiliary statement.

Remark 5.2.3 *Except for arbitrary applications of cut (R5), the* ODL *calculus only produces subprograms of programs from modalities occurring in the formulas under consideration, which preserve createdness. Furthermore, during the proof of Th. 2, the applications of R5 are limited to first-order cuts resulting from expressibility or mere "structural" cuts that combine individual derivations about subformulas according to Prop. A.2.2.*

With the precautions from Ex. 5.2.1 in mind, soundness and relative completeness of the extended ODL calculus inherit from Th. 1 and 2 for a comparably simple reason, at least from an intuitive perspective. The restriction placed to the states by Def. 5.1.1 is first-order, and equivalent to adding a set of premises to the specification, in which case the ODL calculus is relatively complete by Th. 2. Furthermore, this additional restriction is directly made accessible to the extended ODL calculus by R59, because of which the extended ODL calculus is relatively complete as well. Yet this reasoning contains a minor subtlety: the set of additional premises is infinite. *After* finishing the proof, though, this set can be projected in retrospect

[9] See Prop. 4.9.10 in (Fitting & Mendelsohn, 1998).

[10] Assuming x does not occur in α as is the case for $\langle\mathcal{U}\rangle$.

to the finite amount of instances needed during the proof. Essentially, this observation motivates the proof of the following result.

Theorem 4 (Soundness & Completeness) *With states restricted according to Def. 5.1.3, the extended* ODL^{+} *calculus consisting of the inference rules presented in §4.2 plus R59 is sound and complete relative to first-order arithmetic assuming* relative compactness, *i.e. in conjunction with R56,* ODL^{+} *is complete under the premise that*

$$\begin{array}{l} \textit{If } \chi \textit{ is a consequence of } \Phi \textit{ in first-order arithmetic} \\ \textit{then there is a finite subset } E \subset \Phi \textit{ of which } \chi \textit{ is a consequence} \\ \Leftarrow \chi \in \mathrm{Fml}_{FOL}(\Sigma \cup V), \Phi \subseteq \mathrm{Fml}_{FOL}(\Sigma \cup V) \end{array} \qquad (5.3)$$

Proof: Createdness is characterised by the following set of global premises

$$Q \quad := \quad \{\mathit{created}(t) \ : \ t \in \mathrm{Fml}_{\mathrm{ODL}^{+}}(\tilde{\Sigma} \cup \emptyset)\}$$

Proving in ODL^{+}, which – by Def. 5.1.1 and Def. 2.3.11 – is equivalent to proving in ODL under the additional global premises Q, remains sound. As R59 only derives elements of Q, it is sound as well.

For relative completeness, assume that ϕ is a formula with modalities limited to programs preserving createdness such that

$$\vDash \phi \qquad \text{in created states}^{11} \qquad (5.4)$$

By the correspondence from Def. 5.1.1 between the formula *created*(t) and created states, (5.4) implies according to Def. 2.3.11 that

$$Q \vDash_g \phi \qquad \text{in all states}$$

After defining[12]

$$\Box^{*}Q \quad := \quad \{\underbrace{[\alpha_1]\dots[\alpha_n]}_{k \text{ times}}F \ : \ k \in \mathbf{N}\,,\ F \in Q\,,\ \alpha_i \in \mathrm{Prg}_{\mathrm{ODL}^{+}}(\Sigma \cup \emptyset)\}$$

Lem. 2.3.14 implies $\Box^{*}Q \vDash_l \phi$. To continue the proof as desired, we claim that

$$\text{there is a finite subset } E \subset \Box^{*}Q \ \text{ with } E \vDash_l \phi \qquad (5.5)$$

Once this has been established, Lem. 2.3.13 allows to conclude $\vDash_l E \to \phi$, which implies $\vdash E \to \phi$ by Th. 2. In order to demonstrate that $\vdash \phi$ can be

[11] i.e. in all interpretations ℓ containing only created states

[12] Restricting α to subprograms of a program from a modality occurring in ϕ should be sufficient according to Rem. 5.2.3.

proven in ODL$^+$, it remains to show that E only contains instances that can be derived from R59 via Prop. A.2.2.

All formulas from $\Box^* Q$ have the form $[\alpha_1] \ldots [\alpha_n]\mathit{created}(t)$, thus this also holds for those remaining in E. Moreover, they can be obtained from the following derivation involving several applications of R47 and one of R59.

$$\cfrac{\cfrac{\cfrac{\overline{\vdash \mathit{created}(t)}}{\vdash [\alpha_n]\mathit{created}(t)}}{\vdots}}{\vdash [\alpha_1]\ldots[\alpha_n]\mathit{created}(t)}$$

This concludes the proof apart from claim (5.5). If (5.5) does not apply, then $\Box^* Q$ and ϕ can be expressed equivalently in first-order logic as $(\Box^* Q)' \subseteq \mathrm{Fml}_{FOL}(\Sigma \cup V)$ and $\phi' \in \mathrm{Fml}_{FOL}(\Sigma \cup V)$ according to Lem. 4.5.4. Since Lem. 4.5.4 further establishes local equivalence, which is a congruence relation, $(\Box^* Q)' \vDash_l \phi'$ still holds. In this situation, (5.3) implies the existence of a finite first-order subset $E' \subset (\Box^* Q)'$ with $E' \vDash_l \phi'$. Since, again, Lem. 4.5.4 establishes the congruence of local equivalence, this ensures $E' \vDash_l \phi$. Continuing the argument with E' instead of E concludes the proof, if only each $\psi' \in E'$ can be derived in ODL$^+$. This, in turn, holds because – by the above deliberations – the formula $\phi \in E$, which has been expressed in first-order logic as ϕ' by Lem. 4.5.4, is derivable in ODL$^+$. Since $\vDash \psi \rightarrow \psi'$ holds in ODL, Th. 2 ensures that $\vdash \psi \rightarrow \psi'$ can be inferred in both ODL and ODL$^+$, $\vdash \psi \rightarrow \psi'$ and $\vdash \psi$ combine to a derivation of $\vdash \psi'$ by R55. ■

Tracking down the uses of the additional relative compactness assumption (5.3) of Th. 4 in comparison to Th. 2 leads to the conjecture (5.5). A formula for which (5.5) does not hold necessarily requires an infinite amount of premises. Obtaining a finite proof in any calculus for a formula that inherently depends on the presence of infinitely many premises is hopeless, since even mentioning all those required premises produces an infinite proof attempt. From this perspective, a stronger notion of completeness than completeness relative to first-order arithmetic assuming relative compactness seems unlikely. What the notion of completeness assuming relative compactness attempts to formalise is that all deficiencies of a calculus for an inherently incomplete dynamic logic like ODL originate from corresponding shortcomings of basic first-order arithmetic. Therefore, ODL does not introduce any additional qualitative obstacles complexities. Apart from the logic imposing global premises – Th. 4 does not seriously differ in qualitative character from Th. 2. This circumstance suggests that not all possibilities can be eliminated of the premise (5.3) being an artefact of the particular choice of proof, though.

In fact, as will be sketched in the next corollary, the relative compactness assumption can be removed from Th. 4 without loss of generality.

Corollary 5.2.4 (Relative Compactness Assumption) ODL^+ *is relatively complete.*

Proof: It remains to show that the relative compactness assumption can be removed from Th. 4 by adjusting E "appropriately".

Consider some formula ϕ satisfying (5.4). Denote by A_0 the finite set of all updates occurring as subprograms within modalities of ϕ. Similarly, let T the finite set of all program terms occurring in ϕ. From an intuitive perspective, only createdness about terms in T should play a role during a proof about ϕ, although those terms can be subject to arbitrary nesting of updates from A_0. This intuition about what might matter to prove ϕ can be rendered more precise with the "structural" behaviour of the ODL^+ calculus circumscribed in Rem. 5.2.3. Instead of $\Box^* Q$ from the proof of Th. 4, any other means of describing that only the terms constructed by nesting updates from U in front of terms from T should be sufficient. Still, an arbitrary amount of updates from U in front of the finite set T still leads to an infinite number of terms replacing $\Box^* Q$.

In this situation, the built-in power of dynamic logic comes in handy. Remember from the proof of Lem. 4.5.4 the existence of a formula nth for sequence[13] encoding, which can be programmed in ODL using the computable bijection $\mathbf{N} \cong \mathbf{N}^2$. Within ODL programs, use the notation $d \triangleleft nth(c, i)$ as a macro call to this function resolved via open embedding. With $A_0 = \{\alpha_1, \ldots, \alpha_r\}$ being the updates relevant to ϕ, consider the following formula $\mathcal{R}$ for characterising update sequences of ϕ.[14]

$$\begin{aligned} \forall Z\!:\!\texttt{nat}\ \big(c \doteq Z \rightarrow [\texttt{while}(c \neq 0)\,\{ & \\ & d \triangleleft nth(c,1); \\ & c \triangleleft nth(c,2); \\ & \texttt{if}(d \doteq 1)\,\{\alpha_1\}; \\ & \vdots \\ & \texttt{if}(d \doteq r)\,\{\alpha_r\}; \\ \}]\ & \bigwedge_{t \in T} \mathit{created}(t)\big) \end{aligned}$$

[13] A pairing function derived from the predicate *pair* as occurring in footnote 24 during the proof would be sufficient for the current context.

[14] c, d are new constant symbols and Z is a new variable.

$\mathcal{R}$ is a very careful approximation of all sequences of updates that might be performed during the execution of any programs occurring in ϕ. All ODL programs consist of some control-structure surrounding updates as atomic programs. Those updates are the only program statements that perform a state change by themselves. Depending on the particular value of Z, the loop body in program in $\mathcal{R}$ performs one of the relevant updates, with continuation of the loop again depending on the value of Z. Thus *any* execution of any program from a modality in ϕ can be emulated by the $\mathcal{R}$ program with the right choice of Z. For this reason, using the single formula $\mathcal{R}$ instead of the infinite family $\Box^* Q$ as a premise during the proof of Th. 4 is sufficient for obtaining a proof in ODL with the required createdness assertions.

What remains to be shown to complete the proof of Th. 4 without relative compactness assumption is the derivability of $\mathcal{R}$ in ODL$^+$. R45 performs an induction with the invariant

$$p \ := \ \bigwedge_{t \in T} \mathit{created}(t)$$

IA As a composition of createdness formulas, $c \doteq Z \ \vdash p$ can be derived by a sequence of R59 applications.

IE $p, \neg e \ \vdash p$ is immediate.

IS With α being the program in $\mathcal{R}$, the induction $p, e \ \vdash [\alpha]p$ is a true conjecture according to the restriction on α_i in Def. 5.1.3. Since a single execution of the loop body only performs one state transition (apart from c, d), the knowledge contained in R59 about createdness of any intermediate states is irrelevant to the proof. Hence, by relative completeness according to Th. 2 it can be derived within ODL plus R56 without any usage of R59.

Essentially, the induction proves that the programs contained in ϕ comply with the createdness preserving restriction of Def. 5.1.3 by examining the effect of each update in isolation. ■

5.3 Verification Examples

Example 5.3.1 The `new` *created remark* (R61) rule bears some subtleties, which involve updates to and simplification of access to $\mathtt{next}_\mathtt{C}$. The introduction of $\mathtt{next}_\mathtt{C}$ into the sequent by R61 leads to a formula depending on the particular execution context, especially the current accumulated update value of $\mathtt{next}_\mathtt{C}$. Notwithstanding, after all program statements have been processed and all updates have been promoted, this context reduces to a first-order situation in which information about the progress of $\mathtt{next_C}$ will be lost. Thus, the result of an application of R61 will give constraints of varying precision. However, new symbols for objects that already persist since prior to the execution of a program even satisfy the $< \mathtt{next_C}$ constraint in the initial situation. This constraint holds even more so in any state reached after some program statement execution in which $\mathtt{next_C}$ will never have decreased. Not initially using the R61 information about b leads to information loss but can keep branching low. On the other hand, new symbols for objects resulting from object creation already have an explicit $b \lhd \mathtt{obj_C}(n)$ form, which will be promoted to each corresponding occurrence of b by the update mechanism. Either way, soundness is assured.

Consider the case of symbols for newly constructed objects.

$$\langle x \lhd \mathtt{new\,C}(); x.v \lhd 1\rangle x.v \doteq 1$$

This example expands to the following formula, in which any application of R61 happens in a context where the $\mathtt{next_C}$ update information will have been kept.[15]

$$\langle x \lhd \mathtt{obj_C}(\mathtt{next_C}), \mathtt{next_C} \lhd \mathtt{next_C} + 1; x.v \lhd 1\rangle x.v \doteq 1$$

After promotion of the update, $\mathtt{next_C}$ update information will vanish, but then x will already have been replaced by $\mathtt{obj_C}(\mathtt{next_C})$.

Now consider the case of symbols denoting objects that have already been created prior to the execution of the piece of program at hand.

$$p.v \doteq 1 \rightarrow \langle x \lhd \mathtt{new\,C}()\rangle p.v \doteq 1$$

After the usual transformation according to §3.4.2 this looks as follows.

$$p.v \doteq 1 \rightarrow \langle x \lhd \mathtt{obj_C}(\mathtt{next_C}), \mathtt{next_C} \lhd \mathtt{next_C} + 1\rangle p.v \doteq 1$$

Now an application of the R61 rule is possible in the antecedent and within the update with both applications leading to $\mathtt{next_C}$ constraints of different precision. An application on the inner occurrence will lead to an additional premise

$$\begin{aligned} & \langle \mathtt{next_C} \lhd \mathtt{next_C} + 1\rangle (p \doteq \mathtt{obj_C}(X) \land X < \mathtt{next_C}) \\ \equiv \;\; & p \doteq \mathtt{obj_C}(X) \land X < \mathtt{next_C} + 1 \end{aligned}$$

[15] Compare for the update simplification restriction that $\mathtt{next_C}$ will never be removed from an update by *update occurrence* (R48).

Application on the outer occurrence will lead to the more precise premise

$$p \doteq \mathtt{obj_C}(X) \wedge X < \mathtt{next_C}$$

Still both constraints are correct since p already existed prior to the execution of the program snippet. Using less precise constraints may still be more appropriate for practical theorem proving because of the high branching potential in case of weak[16] static type information about p. Then, applying R61 can be delayed during the proof and can be focused on those remaining branches that truly need generatedness information, instead of introducing a reason for premature high forking of the proof. □

Example 5.3.2 (Anonymous Object Creation) Consider again the conjecture from Ex. 5.1.1 about a program involving object creation. This time, assume a translation from JAVA.

$$\begin{array}{l} b \neq \mathtt{null} \rightarrow \\ \langle b.x \triangleleft b.x + 1; \\ \;c \triangleleft \mathtt{new\,C}(); \\ \;c.x \triangleleft b.x + 2 \rangle c.x \doteq b.x + 2 \end{array}$$

During the proof, there is tremendous need to infer that b and c are distinct non-aliased objects by nature of their different sources of creation. In contrast to Ex. 4.3.4 establishing this information needs additional knowledge about the nature of programs, though.

Reusing the abbreviations from Ex. 4.3.4, the proof looks as follows.

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{b \neq \mathtt{null} \;\vdash b.x + 3 \doteq b.x + 1 + 2}}{b \neq \mathtt{null} \;\vdash \langle c \triangleleft o(n), b.x \triangleleft b.x + 1, o(n).x \triangleleft b.x + 1 + 2 \rangle \phi}}{b \neq \mathtt{null} \;\vdash \langle n \triangleleft n + 1, c \triangleleft o(n), b.x \triangleleft b.x + 1 \rangle \langle c.x \triangleleft \ldots \rangle \phi}}{b \neq \mathtt{null} \;\vdash \langle b.x \triangleleft b.x + 1 \rangle \langle c \triangleleft \ldots \rangle \phi}}{b \neq \mathtt{null} \;\vdash \langle b.x \triangleleft \ldots \rangle c.x \doteq b.x + 2}$$

Again, the last inference involves two auxiliary rewrite computations for update application. Abbreviate

$$\langle \mathcal{U} \rangle \quad := \quad \langle c \triangleleft o(n), b.x \triangleleft b.x + 1, o(n).x \triangleleft b.x + 3 \rangle$$

It is utterly important to discover that the update to $o(n).x$ does not affect the value of $b.x$ because b and $o(n)$ are no aliases. Yet, this is not deducible

[16] i.e. the static type information about p allows a large amount of most special dynamic types $\mathtt{D} \leq \mathtt{C}$.

with an application of R41 alone. Information about the origin of the object b is missing. Not yet knowing when b has been created, the prover only knows *that* it has been created at some time. *A priori*, it could have been instantiated with the construction of c just as well, provided that there has been an aliasing update like $b \lhd c$. When in doubt, this calls for the case distinction made by the update application rewrite rules.

$$\begin{array}{ll} & \langle\mathcal{U}\rangle(b.x) \\ \rightsquigarrow & \mathit{if}\, o(n) \doteq \langle\mathcal{U}\rangle b \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & \mathit{if}\, o(n) \doteq b \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & \mathit{if}\, \mathit{false} \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & b.x + 1 \end{array}$$

Within this rewrite inferences the reduction of $o(n) \doteq b$ to *false* is possible by the following argument. Applying R60 for the antecedent $b \neq \mathtt{null}$ gives an additional $(b \doteq o(X) \wedge X < n) \vee b \doteq \mathtt{null}$ with antecedental placement. The case $b \doteq \mathtt{null}$ immediately contradicts the antecedent $b \doteq \mathtt{null}$ with the remaining case being $b \doteq o(X) \wedge X < n$. From this, one is to conclude with simple arithmetic reasoning $X \neq n$ and by R41 $o(n) \neq o(X)$, allowing to continue the rewriting process after an application of R16.

$$\begin{array}{ll} \rightsquigarrow & \mathit{if}\, o(n) \doteq b \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & \mathit{if}\, o(n) \doteq o(X) \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & \mathit{if}\, \mathit{false} \;\mathit{then}\, b.x + 3 \;\mathit{else}\, b.x + 1 \;\mathit{fi} \\ \rightsquigarrow & b.x + 1 \end{array}$$

Table 5.1: Measurements for the Anonymous Object Creation Ex. 5.3.2, cf. Chapt. C

	Calculus	Inferences	Branches	Duration
E	*i*ODL	38	3	0s
E	*i*ODL +mo	46	4	0.1s
E	JavaCardDL nomo	> 120	> 2 (1 open)	> 0.4s
E	JavaCardDL mo	> 165	> 3 (1 open)	> 0.6s
C	*i*ODL	59	6	0.1s
C	*i*ODL +mo	68	7	0.2s
C	JavaCardDL nomo	> 399	> 14 (1 open)	> 1.1s
C	JavaCardDL mo	> 560	> 19 (1 open)	> 1.9s

The example has been run with two different magnitudes of subclasses. First, the case C of Fig. 5.2 having a subtype hierarchy, and second, the flat

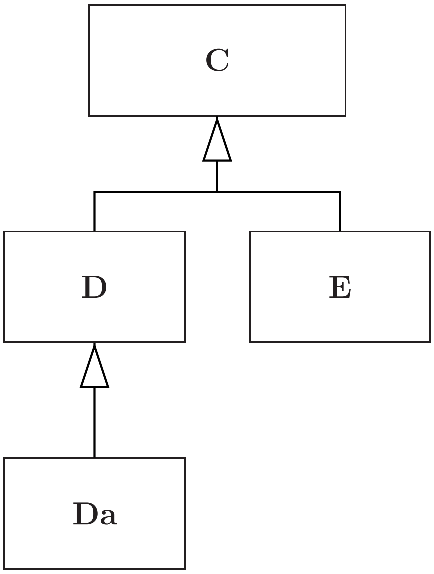

Figure 5.2: UML Class Diagram of Object Creation Type Hierarchy

hierarchy case E. In case C, the specification is unprovable in JavaCardDL with one remaining open goal that cannot be closed:[17]

$$b \doteq v,\ C.nextToCreate \doteq v\ \vdash null \doteq v$$

In class E case, the conjecture is still unprovable in JavaCardDL with one remaining open goal that cannot be closed and looks like this:

$$b \doteq E.nextToCreate\ \vdash$$

□

[17] *C.nextToCreate* is the JavaCardDL way of referring to the next object of type C. Contrary to ODL, JavaCardDL manages objects in a linked list advancing by the field *nextToCreate* of type C in class C.

Chapter 6

Implementation

The ODL calculus is implemented as a (semi-)automatic theorem prover. This implementation is based on an adaptation of the KeY system to ODL demands. The KeY System comes in two flavours:

1. KeY/JAVA for verification problems about programs written in the JAVA[1] programming language, and

2. KeY/ODL for ODL verification problems.

Just as the KeY/JAVA verification system is based on the JAVACARDDL calculus, the new KeY/ODL prover is based on the ODL calculus presented in §4.2. In order to overcome technical limitations of the KeY System and to attain a smooth and non-invasive integration, it has been necessary to attenuate the ODL calculus for matters of implementation and keep it a little bit closer to the JAVACARDDL implementation. Thus, what is implemented is a compromise between ODL and JAVACARDDL called *i*ODL . The *i*ODL implementation has been used for the measurements of the examples in §4.3.

The most apparent discrepancy between *i*ODL and ODL is the effect of the R23 and R25 inference rules. While the ODL rules only need a single inference to reach the final situation, *i*ODL approximates the same effect by a whole sequence of rule applications. Instead of moving the condition e from the program $\texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\}$ to the sequent $e\ \vdash ...$ all at the same time, the *i*ODL calculus only performs this conversion successively.

This behaviour is due to the strict distinction between program expressions of type boolean and formulas within KeY. In the case of JAVA there is indeed a difference. Unlike formulas, evaluation of program expressions of type boolean can produce side-effects or raise exceptions. For this reason,

[1]More precisely: JAVACARD.

the KeY System prohibits inference rules that directly move boolean expressions from programs to the level of formulas like R23 and R25 would do with the condition e. But ODL has no need to artificially distinguish between boolean program expressions and formulas. However, iODL has to respect the limitations of KeY, because of which the translation of a program-level e to a formula-level e passes several intermediate conversion stages. The conversion works inductively on the structure of the formula e. Especially the application of a single R23 ODL rule to a condition e, which is composed of n logical operators, corresponds to $n+1$ iODL inferences.

Example 6.1 Consider the following ODL inference.

$$\frac{a \land b \lor c \ \vdash \langle\alpha\rangle A \quad \neg(a \land b \lor c) \ \vdash \langle\gamma\rangle A}{\vdash \langle \mathtt{if}(a \land b \lor c)\,\{\alpha\}\mathtt{else}\{\gamma\}\rangle A}$$

This single ODL inference has to be emulated within iODL with the following sequence of inferences. For notational clarification denote, here, the program-level conjunction operator by &&, its translation as a function on boolean expression level as &, and the formula-level conjunction sign by $\land$. Further, denote by T the boolean literal `TRUE` of the KeY System, and `FALSE` by F.

$$\frac{\dfrac{\dfrac{a \doteq T \land b \doteq T \lor c \doteq T \ \vdash \langle\alpha\rangle A}{(a\&b) \doteq T \lor c \doteq T \ \vdash \langle\alpha\rangle A}}{(a\&b|c) \doteq T \ \vdash \langle\alpha\rangle A} \quad \dfrac{\dfrac{\dfrac{\neg(a \doteq T \land b \doteq T \lor c \doteq T) \ \vdash \langle\gamma\rangle A}{\neg((a\&b) \doteq T \lor c \doteq T) \ \vdash \langle\gamma\rangle A}}{\neg(a\&b|c) \doteq T \ \vdash \langle\gamma\rangle A}}{(a\&b|c) \doteq F \ \vdash \langle\gamma\rangle A}}{\vdash \langle \mathtt{if}(a\&\&b||c)\,\{\alpha\}\mathtt{else}\{\gamma\}\rangle A}$$

□

Contrary to the ODL principle outlined in §1.3, iODL adds on-the-fly transformations of definable operators for convenience. For example, instead of relying on a preprocessing transformation according to §3.4.2 to reduce object creation statements to ordinary updates, iODL permits `new C()` statements. Those object creation statements will be transformed according to Ex. 4.5.5 on demand rather than prior to starting the prover. This is an exemplary relaxation of the more strict ODL principle.

Apart from representing the ODL calculus in the KeY System and arranging the modifications to iODL, a major implementation effort has been the need for various extensions to the KeY built-in mechanisms. It has been necessary to introduce a new variety of schema variables for matching more general expressions, and slightly extend the parser to make this possible. Primarily, a set of new assignable generic symbols for $\mathtt{obj}_{\mathtt{C}}$ and $\mathtt{next}_{\mathtt{C}}$ needed to be introduced, along with a bunch of meta operators to overcome the remaining limitations of the KeY taclet support in dealing with generically sorted symbols and expressions. To improve the matching precision, generic sort variable conditions are added to the taclet mechanism. Finally it has been necessary to modify the update simplifier to properly treat the newly introduced generic symbols.

In order to provide a good automatic theorem prover ODL has a new dedicated prover strategy that pursues the right balance of additional inference rules. R22 receives lower priority than most of the other inference rules but still higher priority than that of R26. This leads to the effect that intermediate sequential composition will only be attempted when the alternative would consist of obstinate unfolding of loops. Intermediate operator conversion according to Ex. 6.1, on the other hand, has a very high priority such that it almost always performs first, such that the intermediate operators & vanish again as quickly as possible. To avoid unnecessary cluttering of the proof in *i*ODL, the attention span of the prover is focused so that – in JAVA mode – he almost only applies R61 when there are no other possibilities of closing the current goal.

Equation ordering is another important aspect for the automatic prover capabilities. KeY follows a – possibly incomplete – canonicaliser approach to term rewriting by equations. In automatic theorem proving, it is, of course, vital to prevent applying R16 and R20 to the same equation over and over again in a circular way, which would lead to much inferences with no progress at all. The KeY solution to this antagonism is to assess left hand side and right hand side and elect one side as the "canonical"[2] representation. Then all equations are only used to substitute occurrences of the greater side with the "canonical" side. This election is based on a comparison of the terms involved according to a combination of lexicographic and maximum term-depth ordering. In the ODL setting, $\mathtt{obj}_{\mathtt{C}}(n)$ is the preferable form for object expressions, regardless of its depth. This preference is hard-coded into the depth-based term comparison algorithm, with the effect that the theorem prover automatically replaces any occurrence of an object expression with the "canonical" $\mathtt{obj}_{\mathtt{C}}(n)$ whenever possible to maximise the available knowledge about all symbols in each context.

[2]Classical computer algebra terminology would prefer the term "normal", reserving canonicity to situations where the projection ensures unique canonical representatives.

Chapter 7

Summary

This thesis defines an object-oriented dynamic logic, ODL, reduced to the essentials of object-orientation and presents a sound and relatively complete calculus for ODL. The conceptual design of the language ODL has been guided by the ambition to find an adequate non-technical logic in between the comprehensive, object-oriented but complicated JavaCardDL and the frugal but only imperative dynamic logic for the While programming language. Usually, language semantics, inference rules and meta property proofs equally experience a far higher technical complexity in logics with an overwhelming burden of features. Although a full 100% treatment of real Java is, of course, vitally important for practical implementations of real-world verification systems, it is not very well-suited for all types of theoretical investigations. The tremendous amount of special cases and side-conditions imposed by the inclusion of language features like exceptional evaluation, reasons for abrupt completion, or side-effecting expression evaluation strongly supports this ambition – not to mention the sheer amount of syntactical variations of loops and assignment statements to consider. Within feature-rich logics like JavaCardDL, experience shows that several inference rules are surprisingly complicated for the general case of arbitrarily "vicious" programs, but would simplify tremendously for the typical program involving less breaks, less labels, less intermediate side-effects and less "chaotic" exception raising. An investigation of those programs of a simpler and more structural organisation thus seems worthwhile.

Unlike JavaCardDL, ODL has – due to its simplistic nature – very useful properties. ODL possesses standard sequential composition rules because there is no need to consider an abnormal program completion context like surrounding `try-catch` blocks for exception handling. It guarantees regular evaluation of all expressions because of the exclusion of partiality as would otherwise be caused by undefined subterms evaluating to `null`. The exclusion of side-effects supports the liberation of the calculus from forcibly respecting the mandatory evaluation order during the proof. Furthermore, the rigorously structural and local control-flow permits simpler and more straightforward inference rules.

During our investigation we have discovered that only very few features of object-orientation are truly essential characteristics. From a logical point of view, most features reduce to mere syntactic sugar for coding convenience but do not contribute to the logical qualities of object-orientation. Which is the reason why they have been excluded from ODL in comparison to full JavaCardDL. Those more contingent features of object-orientation have been shown to possess a simple translation to ODL without the need to add any surplus language capabilities. Additionally, this translation is effective and natural, which means that it works in a uniform and structural way without coding. Even on a schematological level, ODL and more comprehensive object-oriented languages like JavaCardDL possess the same expressiveness.

In comparison to While the object-oriented programming language underlying ODL provides dynamic type checks and updates, i.e. operations to change the interpretation of function symbols. ODL updates work locally, i.e. pointwise, and can bundle changes to multiple locations of multiple function symbols to one simultaneous update.

ODL handles object creation in an intrinsic way by providing object enumerator symbols that allow access to new objects in a bijective way. Apart from the addition of derived inference rules for optimisation purposes, the axiomatisation of object creation is evident. Practical experiments show that the bijective object enumerators are by far superior to list-based approaches to object creation; not only in terms of execution speed but also by the amount of programs that are provable at all. A large class of conjectures about practically relevant Java programs are provable in automatic mode with the ODL calculus, although they cannot be proven in JavaCardDL at all. Hence, object enumerators constitute a viable choice for handling object creation both in theoretical investigations and practical theorem proving.

ODL has been equipped with a calculus that has been proven both sound and relatively complete. It is based on a classical sequent calculus for the While programming language (Harel, 1979; Harel, 1984; Harel *et al.*, 2000). In order to deal with function update operations, rewrite rules have been introduced that promote the effect of an update throughout the affected formula. To resolve potential aliasing situations, the update promotion process may introduce conditional terms. Both, updates and conditional terms help to defer branching of the proof until necessary for the progress of the proof. Therewith, common parts of the proof can be kept on the same proof branch as long as possible, for efficiency.

The ODL completeness proof is even relative to arithmetic and reveals a flaw in the classical proofs for While (Harel, 1979; Schlager, 2000). This gap in the characterisation treatment of assignments in programs with more than one variable has been closed in our work. Further, a particularly astounding fact is that the completeness and expressibility proofs presented in this thesis even extend to the case of uncomputable functions.

To sum up, the feasibility of presenting an insightful essentials-only verification calculus for general object-oriented programming, which is sound and complete relative to classical first-order arithmetic, has been proven in this thesis.

Appendix A

Properties

A.1 Semantical Relationships

The next result establishes the formal relationship between branching on program level and conditional terms on formula level. Though not needed during the course of this thesis it may clarify the connexion between both concepts in order to show that they are closely connected but still slightly different.

Remark A.1.1

$$\langle \texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\}\rangle\phi \quad\equiv\quad \mathit{if}\, e\ \mathit{then}\, \langle\alpha\rangle\phi\ \mathit{else}\, \langle\gamma\rangle\phi\ \mathit{fi}$$

Proof: Let $w \in W = \ell(\ell)$ be any state of any interpretation ℓ.

$$\begin{array}{ll}
 & \mathit{val}_\ell(w, \langle \texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\}\rangle\phi) = \mathit{true} \\
\Longleftrightarrow & \text{there is } t \in W\ \ s\rho_\ell(\texttt{if}(e)\,\{\alpha\}\texttt{else}\{\gamma\})t\,,\ \mathit{val}_\ell(t,\phi) = \mathit{true} \\
\Longleftrightarrow & \text{there is } t \in W \text{ with } \mathit{val}_\ell(w,e) = \mathit{true} \text{ and } s\rho_\ell(\alpha)t, \\
 & \mathit{or}\, \mathit{val}_\ell(w,e) = \mathit{false} \text{ and } s\rho_\ell(\gamma)t \\
 & \text{and – in either case – } \mathit{val}_\ell(t,\phi) = \mathit{true} \\
\Longleftrightarrow & \text{if } \mathit{val}_\ell(w,e) = \mathit{true}, \text{ there is } t \in W\ s\rho_\ell(\alpha)t \text{ and } \mathit{val}_\ell(t,\phi) = \mathit{true} \\
 & \text{resp.} \\
 & \text{if } \mathit{val}_\ell(w,e) = \mathit{false}, \text{ there is } t \in W\ s\rho_\ell(\gamma)t \text{ and } \mathit{val}_\ell(t,\phi) = \mathit{true} \\
\Longleftrightarrow & \text{if } \mathit{val}_\ell(w,e) = \mathit{true} \text{ then } \mathit{val}_\ell(w,\langle\alpha\rangle\phi) = \mathit{true} \\
 & \text{resp. if } \mathit{val}_\ell(w,e) = \mathit{false} \text{ then } \mathit{val}_\ell(w,\langle\gamma\rangle\phi) = \mathit{true} \\
\Longleftrightarrow & \mathit{val}_\ell(w, \mathit{if}\, e\ \mathit{then}\, \langle\alpha\rangle\phi\ \mathit{else}\, \langle\gamma\rangle\phi\ \mathit{fi}) = \mathit{true}
\end{array}$$

∎

A.2 Meta-Proving

Proposition A.2.1 *The following statements interrelate derivations in the* ODL *calculus.*

1. *If* $\phi \vdash \psi$ *is derivable, then so is* $\vdash \phi \to \psi$.
2. *If* $\vdash \phi \to \psi$ *is derivable, then so is* $\phi \vdash \psi$.
3. *If* $\vdash \phi$ *and* $\phi, \Gamma \vdash \Delta$ *are derivable, then so is* $\Gamma \vdash \Delta$.
4. *If some derivation of* $\phi \vdash \psi$ *can be continued to a derivation of* $\phi' \vdash \psi'$, *then* $\phi \to \psi \vdash \phi' \to \psi'$ *is derivable as well:*

$$\begin{array}{cccc} & (\phi \vdash \psi & \Rightarrow & \phi' \vdash \psi') \\ \Rightarrow & \phi \to \psi & \vdash & \phi' \to \psi' \end{array}$$

Proof:

1. $$\frac{\phi \ \vdash \psi}{\vdash \phi \to \psi}$$

   can extend a derivation by the $\to$ *right* (R10) inference rule.

2. By *modus ponens* (R55), $\phi, \phi \to \psi \vdash \psi$ is directly derivable. In combination with the derivation of $\vdash \phi \to \psi$, (3) allows to conclude $\phi \vdash \psi$. This reasoning leads to the inverse of the $\to$ *right* (R10) inference rule.

3. $$\frac{\overline{\phi, \Gamma \ \vdash \Delta} \quad \overline{\Gamma \ \vdash \phi, \Delta}^{\overline{\ \vdash \phi\ }}}{\Gamma \ \vdash \Delta}$$

   extends the assumed derivations of $\vdash \phi$ and $\phi, \Gamma \vdash \Delta$ by R6 resp. R12 weakening and combines them with a *cut* (R5).

4. By premise there is a "partial"[1] derivation

$$\frac{\dfrac{\phi \ \vdash \psi}{\vdots}}{\phi' \ \vdash \psi'}$$

[1] i.e. a derivation following the ODL inference rules without starting with axioms.

Extending all the inferences with an additional antecedent $\phi \to \psi$ leads to the following proof, which has been continued by $\to$ *right* (R10) and closed to a complete proof by R55.

$$\dfrac{\dfrac{\dfrac{}{\phi \to \psi, \phi \ \vdash \psi}}{\dfrac{\vdots}{\phi \to \psi, \phi' \ \vdash \psi'}}}{\phi \to \psi \ \vdash \phi' \to \psi'}$$

■

Proposition A.2.2 *When A is derivable in* ODL*, adding A to the antecedent of any sequent is always a sound inference:*

$$\frac{\Gamma, A \ \vdash \Delta}{\Gamma \ \vdash \Delta}$$

Proof: Using a *cut* (R5) to introduce the axiom into the antecedent the derivation looks as follows, with the right branch closing according to the derivation of $\vdash A$.

$$\dfrac{\dfrac{\dots}{\Gamma, A \ \vdash \Delta} \quad \dfrac{}{\Gamma \ \vdash A, \Delta}}{\Gamma \ \vdash \Delta}$$

■

A.3 Deduction

This section contains a proof of the global deduction theorem Lem. 2.3.14, which has been used in Th. 4.

Unlike the other parts of this thesis, the proof of Lem. 2.3.14, profits from a shift in notation, which will be performed to better highlight the essential aspects. As those aspects are of importance during the proof, instead of interpretations ℓ with a unified notation, this section considers structures (W, ρ, β) that are explicitly split into a set W of states, an[2] accessibility relation ρ and a variable assignment β on V. In order to avoid cluttering of notation during the other chapters of this thesis, those more detailed structures are only used during this section.

[2]See below for the reason why it is sufficient to consider only one accessibility relation in this section.

Lem. 2.3.14 considers for arbitrary programs all possible transitions. The same effect is achieved when considering all transitions possible by program execution at all.

Definition A.3.1 (Blurred Structures) *For an interpretation ℓ with a set W of states and accessibility relations $\rho_\ell(\alpha)$, define the* blurred *structure (W, ρ, β) with*

$$\begin{aligned} \rho &:= \bigcup_{\alpha \in \mathrm{Prg}(\Sigma \cup V)} \rho_\ell(\alpha) \\ \beta &:= (x \mapsto val_\ell(w, x)) \end{aligned}$$

Thereby note that for variables $x \in V$, $val_\ell(w, x)$ does not depend on the state w according to Def. 2.3.1. As modalities belonging to ρ use $\Box$ and $\Diamond$ instead of $[\alpha]$ and $\langle\alpha\rangle$, with a semantics that directly transfers the notions from Def. 2.3.8 to ρ instead of the parametric $\rho_\ell(\alpha)$. The notation $(W, \rho, \beta), s \vDash F$ and $(W, \rho, \beta) \vDash F$ also generalise from Def. 2.3.10. The set of states W along with the accessibility relation ρ is called a Kripke frame.

Remark A.3.2 *$(W, \rho, \beta), t \vDash \Box F$ holds if and only if $(W, \rho, \beta), t \vDash [\alpha]F$ holds for all $\alpha \in \mathrm{Prg}(\Sigma \cup V)$. Similarly, $(W, \rho, \beta), t \vDash \Diamond F$ holds if and only if $(W, \rho, \beta), t \vDash \langle\alpha\rangle F$ holds for some $\alpha \in \mathrm{Prg}(\Sigma \cup V)$.*

With those concepts, Lem. 2.3.14 simplifies to Lem. A.3.3 below, since $\{\Box^n F\}$ now is a singleton set and receives its standard meaning by[3]

$$\Box^n F := \underbrace{\Box \ldots \Box}_{n \text{ times}} F$$

Lemma A.3.3 (Global Deduction Theorem)

$$\Phi \cup \{F\} \vDash \Psi \rhd \chi \iff \Phi \vDash \Psi \cup \bigcup_{n \in \mathbf{N}} \{\Box^n F\} \rhd \chi$$

Proof:

"$\Leftarrow$" simple

"$\Rightarrow$" Let (G, ρ, β) be an arbitrary structure, with $(G, \rho, \beta) \vDash \Phi$, i.e. for each $\phi \in \Phi (G, \rho, \beta) \vDash \phi$. Furthermore, provided that $s \in G$ is a state with $s \vDash \Psi, s \vDash F, s \vDash \Box F, s \vDash \Box\Box F, s \vDash \Box\Box\Box F, \ldots$, then assume that it was $s \nvDash \chi$, which we have to lead to a contradiction.

[3]Hence the odd abbreviation in Lem. 2.3.14, which is now justified in retrospect by equivalence.

Consider the variant $M' := (\overline{\{s\}}, \rho|_{\overline{\{s\}}}, \beta)$ from A.3.5. According to A.3.4, there it still is $M', s \vDash \Psi, M', s \vDash F, M', s \vDash \Box F, M', s \vDash \Box\Box F, \dots$ (thus $M' \vDash F$ everywhere on $\overline{\{s\}}$), and still $M', s \nvDash \chi$. But when we can further show that it still is $M' \vDash \Phi$, we have constructed a model M' satisfying $M', s \vDash \chi$ by premise, which is a contradiction.

So it only remains to show that $M' \vDash \Phi$. for each $t \in \overline{\{s\}}$ closing the mini-frame $\{t\}$ by A.3.5 and localising $\overline{\{t\}}$ in the frame $\overline{\{s\}}$ by A.3.4 leads to $M', t \vDash \Phi \overset{A.3.4}{\iff} (\{s\} \circ \rho^*) \cap (\{t\} \circ \rho^*), t \vDash \Phi$ which, (because $t \in \overline{\{s\}} = \{s\} \circ \rho^*$ implies $\{t\} \circ \rho^* \subseteq \{s\} \circ \rho^*$)[4] is equivalent to $\{t\} \circ \rho^*, t \vDash \Phi \overset{A.3.4}{\iff} G, t \vDash \Phi$ which is true by premise.

■

Lemma A.3.4 (Localisation Lemma[5]) *If H is a* closed *subframe[6] of the Kripke frame G, i.e. $H \circ \rho \subseteq H$, then*

$$\textit{for each } g \in H \ \textit{ for each } A \in \mathrm{Fml}(\Sigma) \ \textit{ for each assignment } \beta$$
$$(G, \rho, \beta), g \vDash A \iff (H, \rho|_H, \beta), g \vDash A$$

Proof: by definition, because of the unidirectional navigation within ρ during the interpretation of $\Box$ and $\Diamond$. ■

Lemma A.3.5 (Modal closure) *For a subframe H of the Kripke frame G,*

$$\bar{H} := \bigcap_{H \subseteq F \subseteq G \ \textit{closed}} F = H \circ \rho^* \ \textit{ is closed}$$

Proof:

"s" $\bar{H}$ is closed

"$\subseteq$" $H \circ \rho^*$ takes part in the intersection as an F because of

- $H \subseteq H \circ \rho^*$.
- $H \circ \rho^* \subseteq G$ since G is closed in itself.
- $\rho|_{H \circ \rho^*} = \rho$ on $H \circ \rho^*$ by definition of restriction.

[4]This argument shows that the result $(\{s\} \circ \rho^*) \cap (\{t\} \circ \rho^*)$ of localising $\overline{\{t\}}$ in $\overline{\{s\}} = (\{s\} \circ \rho^*)$ is the same as the result of immediately localising $\overline{\{t\}}$ in M.

[5]Also called forward property in the case $H := \overline{\{g\}}$.

[6]H is a subframe of G if it has less states, but the same accessibility on $H \subseteq G$.

– $$(H \circ \rho^*) \circ \rho = H \circ (\rho^* \circ \rho) = H \circ \bigcup_{0<n\in\mathbf{N}} \rho^n \subseteq H \circ \rho^*$$

since $\circ$ is associative and distributive over $\bigcup$.

"$\supseteq$" for each $x \in G \setminus (H \circ \rho^*)$ we have to show that $x \notin \bar{H}$. Let

$$\tilde{F} := \overline{H \cup \{x\}} \supseteq \bar{F} \text{ when } F := \tilde{F} \setminus (\rho^* \circ \{x\}) \subseteq G$$

– $F \supseteq H$, since it is for each $h \in H$ not $(h\rho^* x)$, by premise on x.

– F is closed due to the fact that for each $f \in F$ for each $y \in G$ $f\rho y \overset{\tilde{F}\text{ closed}}{\Rightarrow} y \in \tilde{F}$. And if we assume that $y \in \tilde{F} \setminus F = \rho^* \circ \{x\}$ then $f\rho y\,,\, y\rho^* x \Rightarrow f\rho^* x \Rightarrow f \in \rho^* \circ \{x\} = \tilde{F} \setminus F$ Contradiction!

Therefore the set F takes part in the intersection of the left hand side, which proves that $\bar{H} \subseteq F \not\ni x$.

$\blacksquare$

Appendix B

Examples

B.1 Update Examples

Example B.1.1 An example which saves a lot of case distinctions.

$$
\begin{aligned}
&\langle f(s) \lhd s\rangle g(f(f(s)))\\
\rightsquigarrow\ & g\big(\langle f(s) \lhd s\rangle f(f(s))\big)\\
\rightsquigarrow\ & g\Big(\mathit{if}\, s \doteq {\scriptstyle \langle f(s) \lhd s\rangle f(s)}\ \mathit{then}\, s\ \mathit{else}\, f\big(\langle f(s) \lhd s\rangle f(s)\big)\ \mathit{fi}\Big)\\
\rightsquigarrow\ & g\Big(\mathit{if}\, s \doteq {\scriptstyle \langle f(s) \lhd s\rangle f(s)}\ \mathit{then}\\
&\qquad s\\
&\mathit{else}\\
&\qquad f\big(\mathit{if}\, s \doteq \langle f(s) \lhd s\rangle s\ \mathit{then}\, s\ \mathit{else}\, f(\langle f(s) \lhd s\rangle s)\ \mathit{fi}\big)\\
&\mathit{fi}\Big)\\
\rightsquigarrow\ & g\Big(\mathit{if}\, s \doteq \big({\scriptstyle \mathit{if}\, s \doteq \langle f(s) \lhd s\rangle s\ \mathit{then}\, s\ \mathit{else}\, f(\langle f(s) \lhd s\rangle s)\ \mathit{fi}}\big)\ \mathit{then}\\
&\qquad s\\
&\mathit{else}\\
&\qquad f\big(\mathit{if}\, s \doteq \langle f(s) \lhd s\rangle s\ \mathit{then}\, s\ \mathit{else}\, f(\langle f(s) \lhd s\rangle s)\ \mathit{fi}\big)\\
&\mathit{fi}\Big)\\
\rightsquigarrow\ & g\Big(\mathit{if}\, s \doteq \big({\scriptstyle \mathit{if}\, s \doteq s\ \mathit{then}\, s\ \mathit{else}\, f(\langle f(s) \lhd s\rangle s)\ \mathit{fi}}\big)\ \mathit{then}\\
&\qquad s\\
&\mathit{else}\\
&\qquad f\big(\mathit{if}\, s \doteq s\ \mathit{then}\, s\ \mathit{else}\, f(\langle f(s) \lhd s\rangle s)\ \mathit{fi}\big)
\end{aligned}
$$

$$
\begin{array}{ll}
 & \quad \mathit{fi}\Big) \\
\rightsquigarrow & g\Big(\mathit{if}\, s \doteq \big({\scriptstyle \mathit{if}\, s \doteq s\, \mathit{then}\, s\, \mathit{else}\, f(s)\, \mathit{fi}}\big)\; \mathit{then} \\
 & \qquad s \\
 & \quad \mathit{else} \\
 & \qquad f\big(\mathit{if}\, s \doteq s\; \mathit{then}\, s\; \mathit{else}\, f(s)\; \mathit{fi}\big) \\
 & \quad \mathit{fi}\Big) \\
\text{“}\rightsquigarrow\text{”} & \Big(s \doteq s \to g\big(\mathit{if}\, s \doteq s\; \mathit{then}\, s\; \mathit{else}\, f(s)\; \mathit{fi}\big)\Big) \wedge \\
 & \Big(s \neq s \to g\big(\mathit{if}\, s \doteq f(s)\; \mathit{then}\, s\; \mathit{else}\, f(f(s))\; \mathit{fi}\big)\Big) \\
\text{“}\rightsquigarrow\text{”} & \big(s \doteq s \wedge s \doteq s \to g(s)\big) \wedge \\
 & \big(s \doteq s \wedge s \neq s \to g(f(s))\big) \wedge \\
 & \big(s \neq s \wedge s \doteq f(s) \to g(s)\big) \wedge \\
 & \big(s \neq s \wedge s \neq f(s) \to g(f(f(s)))\big) \\
\text{“}\rightsquigarrow\text{”} & g(s)
\end{array}
$$

□

Example B.1.2 For higher arities, it is even more important to detect unnecessary duplicate case distinctions in order to avoid branching as the following complicated derivation demonstrates. For notation, use $\langle \mathcal{U} \rangle$ as an abbreviation for $\langle f(s_1, s_2) \lhd r \rangle$.

$$
\begin{array}{ll}
 & \langle f(s_1, s_2) \lhd r \rangle f(f(o_1, o_2), f(p_1, p_2)) \\
\rightsquigarrow & \mathit{if}\, s_1 \doteq {\scriptstyle \langle f(s_1, s_2) \lhd r \rangle f(o_1, o_2)} \wedge s_2 \doteq {\scriptstyle \langle f(s_1, s_2) \lhd r \rangle f(p_1, p_2)}\; \mathit{then} \\
 & \qquad r \\
 & \mathit{else} \\
 & \qquad f\big(\langle f(s_1, s_2) \lhd r \rangle f(o_1, o_2), \langle f(s_1, s_2) \lhd r \rangle f(p_1, p_2)\big) \\
 & \mathit{fi} \\
\rightsquigarrow & \mathit{if}\, s_1 \doteq {\scriptstyle \langle f(s_1, s_2) \lhd r \rangle f(o_1, o_2)} \wedge s_2 \doteq {\scriptstyle \langle f(s_1, s_2) \lhd r \rangle f(p_1, p_2)}\; \mathit{then} \\
 & \qquad r \\
 & \mathit{else} \\
 & \qquad f\big(\mathit{if}\, s_1 \doteq \langle \mathcal{U} \rangle o_1 \wedge s_2 \doteq \langle \mathcal{U} \rangle o_2\; \mathit{then}\, r\; \mathit{else}\, f(o_1, o_2)\; \mathit{fi}, \\
 & \qquad\quad \mathit{if}\, s_1 \doteq \langle \mathcal{U} \rangle p_1 \wedge s_2 \doteq \langle \mathcal{U} \rangle p_2\; \mathit{then}\, r\; \mathit{else}\, f(p_1, p_2)\; \mathit{fi}\big) \\
 & \mathit{fi}
\end{array}
$$

$$
\begin{aligned}
\rightsquigarrow\ & \mathit{if}\, s_1 \doteq \left(\scriptstyle \mathit{if}\, s_1 \doteq o_1 \wedge s_2 \doteq o_2\ \mathit{then}\, r\ \mathit{else}\, f(o_1, o_2)\ \mathit{fi}\right) \\
&\qquad\quad \wedge s_2 \doteq \left(\scriptstyle \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}\right)\ \mathit{then} \\
&\qquad r \\
& \mathit{else} \\
&\qquad f\big(\mathit{if}\, s_1 \doteq o_1 \wedge s_2 \doteq o_2\ \mathit{then}\, r\ \mathit{else}\, f(o_1, o_2)\ \mathit{fi}, \\
&\qquad\quad \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}\big) \\
& \mathit{fi} \\
\text{“}\rightsquigarrow\text{”}\ & \Big(s_1 \doteq o_1 \wedge s_2 \doteq o_2 \rightarrow \mathit{if}\, s_1 \doteq r \\
&\qquad\quad \wedge s_2 \doteq \left(\scriptstyle \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}\right)\ \mathit{then} \\
&\qquad r \\
& \mathit{else} \\
&\qquad f(r, \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}) \\
& \mathit{fi}\Big) \wedge \\
& \Big(s_1 \neq o \vee s_2 \neq o_2 \rightarrow \mathit{if}\, s_1 \doteq f(o_1, o_2) \\
&\qquad\quad \wedge s_2 \doteq \left(\scriptstyle \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}\right)\ \mathit{then} \\
&\qquad r \\
& \mathit{else} \\
&\qquad f(f(o_1, o_2), \mathit{if}\, s_1 \doteq p_1 \wedge s_2 \doteq p_2\ \mathit{then}\, r\ \mathit{else}\, f(p_1, p_2)\ \mathit{fi}) \\
& \mathit{fi}\Big) \\
\text{“}\rightsquigarrow\text{”}\ & \big((s_1 \doteq o_1 \wedge s_2 \doteq o_2) \wedge (s_1 \doteq p_1 \wedge s_2 \doteq p_2) \rightarrow \\
&\qquad \mathit{if}\, s_1 \doteq r \wedge s_2 \doteq r\ \mathit{then}\, r\ \mathit{else}\, f(r, r)\ \mathit{fi}\big) \wedge \\
& \big((s_1 \doteq o_1 \wedge s_2 \doteq o_2) \wedge (s_1 \neq p_1 \vee s_2 \neq p_2) \rightarrow \\
&\qquad \mathit{if}\, s_1 \doteq r \wedge s_2 \doteq f(p_1, p_2)\ \mathit{then}\, r\ \mathit{else}\, f(r, f(p_1, p_2))\ \mathit{fi}\big) \wedge \\
& \big((s_1 \neq o \vee s_2 \neq o_2) \wedge (s_1 \doteq p_1 \wedge s_2 \doteq p_2) \rightarrow \\
&\qquad \mathit{if}\, s_1 \doteq f(o_1, o_2) \wedge s_2 \doteq r\ \mathit{then}\, r\ \mathit{else}\, f(f(o_1, o_2), r)\ \mathit{fi}\big) \\
& \big((s_1 \neq o \vee s_2 \neq o_2) \wedge (s_1 \neq p_1 \vee s_2 \neq p_2) \rightarrow \\
&\qquad \mathit{if}\, s_1 \doteq f(o_1, o_2) \wedge s_2 \doteq f(p_1, p_2)\ \mathit{then}\, r\ \mathit{else}\, f(f(o_1, o_2), f(p_1, p_2))\ \mathit{fi}\big) \\
\text{“}\rightsquigarrow\text{”}\ & \big((s_1 \doteq o_1 \wedge s_2 \doteq o_2) \wedge (s_1 \doteq p_1 \wedge s_2 \doteq p_2) \wedge (s_1 \doteq r \wedge s_2 \doteq r)) \rightarrow \\
&\qquad r\big) \wedge \\
& \big((s_1 \doteq o_1 \wedge s_2 \doteq o_2) \wedge (s_1 \doteq p_1 \wedge s_2 \doteq p_2) \wedge (s_1 \neq r \vee s_2 \neq r)) \rightarrow \\
&\qquad f(r, r)\big) \wedge
\end{aligned}
$$

$$\big((s_1 \doteq o_1 \wedge s_2 \doteq o_2) \wedge (s_1 \neq p_1 \vee s_2 \neq p_2) \rightarrow$$
$$\quad \mathit{if}\, s_1 \doteq r \wedge s_2 \doteq f(p_1, p_2)\ \mathit{then}\, r\ \mathit{else}\, f(r, f(p_1, p_2))\ \mathit{fi}\big) \wedge$$
$$\big((s_1 \neq o \vee s_2 \neq o_2) \wedge (s_1 \doteq p_1 \wedge s_2 \doteq p_2) \rightarrow$$
$$\quad \mathit{if}\, s_1 \doteq f(o_1, o_2) \wedge s_2 \doteq r\ \mathit{then}\, r\ \mathit{else}\, f(f(o_1, o_2), r)\ \mathit{fi}\big)$$
$$\big((s_1 \neq o \vee s_2 \neq o_2) \wedge (s_1 \neq p_1 \vee s_2 \neq p_2) \rightarrow$$
$$\quad \mathit{if}\, s_1 \doteq f(o_1, o_2) \wedge s_2 \doteq f(p_1, p_2)\ \mathit{then}\, r\ \mathit{else}\, f(f(o_1, o_2), f(p_1, p_2))\ \mathit{fi}\big)$$

□

B.2 Substitutions versus Updates

Contrary to syntactical substitutions, updates also affect terms that bear no deeper syntactical similarity but only partial semantical identity (by aliasing). This effect can be seen by the distinct effect of the two following inferences. First consider the effect of the $\doteq$ *subst* (R16) rule.

$$\frac{s \doteq t, f(s) \doteq 1 \ \vdash 1 > 0 \wedge f(t) > 0}{s \doteq t, f(s) \doteq 1 \ \vdash f(s) > 0 \wedge f(t) > 0}$$

Then compare it to the effect of a comparable update inference, which affects the term $f(t)$ on the basis of the additional knowledge that $s \doteq t$.

$$\frac{s \doteq t \ \vdash 1 > 0 \wedge 1 > 0}{s \doteq t \ \vdash \langle f(s) \lhd 1 \rangle (f(s) > 0 \wedge f(t) > 0)}$$

The difference in effects is even more convincing in the case of unknown aliasing relationships, where updates introduce a case distinction to cover all possibilities of whether or not the affected terms are subject to aliasing.

$$\cfrac{\cfrac{s \doteq t \ \vdash 1 > 0 \wedge 1 > 0 \qquad s \neq t \ \vdash 1 > 0 \wedge t > 0}{\vdash 1 > 0 \wedge (\mathit{if}\, s \doteq t\ \mathit{then}\, 1\ \mathit{else}\, t\ \mathit{fi}) > 0}}{\vdash \langle f(s) \lhd 1 \rangle (f(s) > 0 \wedge f(t) > 0)}$$

Appendix C

Measurements

All the measurements in this thesis have been performed on an Athlon™ XP 2700+ from AMD with 512MB DDR-SDRAM RAM PC2700 running Suse Linux 9.0 with a 2.6.5-25 kernel. KeY has been run on a Java HotSpot™ Client Virtual Machine, build 1.4.2-b28, in mixed mode. All measurements are based on the ODL branch of the KeY system, which is an extended descendant of the KeY major revision 0.1164.

The measurements have been repeated five times with the result being the average of the last three runs. Disposing of the first two runs is important for meaningful time measurements. Due to Java dynamic class loading and caching effects initial runs tend to have a considerable variance. Those effects stabilise after the first few runs. Nevertheless, due to the large amount of nondeterministic factors involved, especially the timing measurements are not statistically significant. Rather the timing information is intended to give a feeling for magnitudal differences. Further, timing information is adequate to balance measurements with contradicting number of inference and branching information. Measurements that produce more inferences yet involve less case distinctions are not easy to compare, qualitatively, to measurements of less inferences scattered over a multitude of branches. Timing information helps to judge an appropriate balance of inferences in comparison to branches.

Moreover, due to the coarse time resolution and the small durations, timing information can result in zero seconds, which is an artifact. Still it stands for quick runtime and – due to rounding effects – cannot be considered significantly different from 0.1 seconds. With greater quantities the timing measurements get more precise, though.

While the Java Virtual Machine itself already contributes to nondeterministic effects that result from dynamic class loading, garbage collection and caching, the implementation of the KeY system itself is subject to nondeterminism as well. At least at the time of writing, one source of nondeterminism is term ordering. KeY considers equations as directed, and will only rewrite occurrences of the term that is greater according to the term ordering with the smaller term. The default term ordering however uses the location of the symbol object in memory amongst other deterministic criteria.

The measurements in this thesis compare the implementation variant *i*ODL of the ODL calculus to the original JavaCardDL calculus of the KeY System. Apart from the difference of the calculi themselves, unlike JavaCardDL, the ODL calculus does not employ an optimisation of update merging, which will be referred to as "mo" during the course of this work. The *merge optimisation* mo applied in JavaCardDL transforms an update $o.x \lhd t,\ p.x \lhd s$ to

$$o.x \lhd (\mathit{if}\, o \doteq p\, \mathit{then}\, o.x\, \mathit{else}\, t\, \mathit{fi}), \quad p.x \lhd s$$

In order to support an orthogonal comparison of the ODL versus JavaCardDL calculi, independent of the question of whether or not to use the mo optimisation, the measurements will be provided in several flavours. The measurements have been repeated for the *i*ODL calculus with the mo transformation (called "*i*ODL +mo" in the measurement tables) and without mo (called only "*i*ODL"). Further the measurements have been performed for the JavaCardDL calculus as is[1] (called "JavaCardDL mo") and for the JavaCardDL calculus with mo disabled (called "JavaCardDL nomo"). To compare the proper *i*ODL and JavaCardDL approaches, compare the "*i*ODL" to the "JavaCardDL mo" measurements. The other measurements are only provided to isolate the cause.

There is, of course, no need to emphasise that the *i*ODL inference measurements only place upper bounds to the ODL calculus. Apart from the changes in the calculus this is due to the nature of the automatic theorem prover, which does not search for the shortest proof or even cancel out void inferences, but only derives as much as possible in short time. Some automatic derivations still contain unnecessary intermediate steps.

[1] i.e. with mo transformation

References

AHRENDT, WOLFGANG, BAAR, THOMAS, BECKERT, BERNHARD, BUBEL, RICHARD, GIESE, MARTIN, HÄHNLE, MENZEL, WOLFRAM, MOSTOWSKI, WOJCIECH, ROTH, ANDREAS, SCHLAGER, STEFFEN, & SCHMITT, PETER H. 2004. The KeY Tool. *Software and System Modeling*. To appear in print.

APT, KRZYSZTOF R., & OLDEROG, ERNST-RÜDIGER. 1997. *Verification of Sequential and Concurrent Programs*. 2nd edn. Texts and Monographs in Computer Science. Springer-Verlag.

BAADER, FRANZ, & NIPKOW, TOBIAS. 1998. *Term rewriting and all that*. Cambridge Univ. Press. Baader.

BALCER, MARC J., & MELLOR, STEPHEN J. 2002. *Executable UML: A Foundation of Model Driven Architecture*. 1st edn. Addison-Wesley.

BALZERT, HELMUT. 2001. *Lehrbuch der Software-Technik - Software-Entwicklung*. Lehrbücher der Informatik. Spektrum Akademischer Verlag.

BARENDREGT, HENK P. 1984. *The Lambda Calculus*. revised edition edn. North Holland.

BARENDREGT, HENK P. 1992. *Lambda calculi with types*. Oxford University Press, Inc. Pages 117–309.

BECKERT, BERNHARD. 2000. A Dynamic Logic for Java Card. *Pages 111–119 of: Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*.

BECKERT, BERNHARD, & KLEBANOV, VLADIMIR. 2004. Proof Reuse for Deductive Program Verification. *In: Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press. To appear.

BECKERT, BERNHARD, & SASSE, BETTINA. 2001. Handling Java's Abrupt Termination in a Sequent Calculus for Dynamic Logic. *Pages 5–14 of:* BECKERT, B., FRANCE, R., HÄHNLE, R., & JACOBS, B. (eds), *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy.* Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena.

BECKERT, BERNHARD, & SCHLAGER, STEFFEN. 2004. Software Verification with Integrated Data Type Refinement for Integer Arithmetic. *In: Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK.* LNCS. Springer. To appear.

BÖRGER, EGON, & STÄRK, ROBERT. 2003. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer-Verlag.

CHURCH, ALONZO. 1940. A formulation of the simple theory of types. *J. Symbolic Logic*, 56–68.

COOK, STEPHEN A. 1978. Soundness and Completeness of an Axiom System for Program Verification. *j-SIAM-J-COMPUT*, **7**(1), 70–90.

CORMEN, THOMAS H., LEISERSON, CHARLES E., & RIVEST, RONALD L. 1990. *Introduction to Algorithms.* MIT Press/McGraw-Hill.

CROSSLEY, JOHN N. (ed). 1981. *Aspects of Effective Algebra: Proceedings of a Conference at Monash University 1-4, August 1979.* Steel Creek, Australia: Upside Down A Book Company.

DAVEY, B. A., & PRIESTLEY, H. A. 2002. *Introduction to Lattices and Order.* Cambridge University Press.

DE BOER, FRANK S. 1999. A WP-calculus for OO. *Pages 135–149 of: Proceedings of Foundations of Software Science and Computation Structurers (FOSSACS'99).* LNCS, vol. 1578. Springer.

DERSHOWITZ, NACHUM. 1982. Orderings for term-rewriting systems. *Theoretical Computer Science*, **17**, 279–301.

DERSHOWITZ, NACHUM, & JOUANNAUD, JEAN-PIERRE. 1990. Rewrite Systems. *Pages 243–320 of: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B).*

DERSHOWITZ, NACHUM, & PLAISTED, DAVID A. 2001. Rewriting. *Chap. 9, pages 535–610 of:* ROBINSON, A., & VORONKOV, A. (eds), *Handbook of Automated Reasoning*, vol. I. Elsevier Science.

DIJKSTRA, EDSGER WYBE. 1976. *A Discipline of Programming*. Prentice-Hall.

DOWD, TYSON, SCHACHTE, PETER, HENDERSON, FERGUS, & SOMOGYI, ZOLTAN. 2000 (April). *Using impurity to create declarative interfaces in Mercury*. Tech. rept. 2000/17. Department of Computer Science, University of Melbourne, Melbourne, Australia.

FITTING, MELVIN, & MENDELSOHN, RICHARD L. 1998. *First-Order Modal Logic*. 1st edn. Kluwer Academic Publishers.

GÖDEL, KURT. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, **38**, 173–198. english translation: On formally undecidable propositions of Principia Mathematica and Related Systems I, Oliver & Boyd, London, 1962.

GOSLING, JAMES, JOY, BILL, STEELE, GUY L., & BRACHA, GILAD. 1996. *The Java Language Specification*. The Java Series. Massachusetts: Addison-Wesley.

GUREVICH, YURI. 2000. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, **1**(July), 77–111.

HACK, SEBASTIAN, GEI§, RUBINO, & GLESNER, SABINE. 2004. *Applying SPO Graph Rewriting to Machine-Dependent Optimizations*. Institute of Program Structures and Data Organization, University of Karlsruhe, Germany.

HAREL, D., KOZEN, D., & TIURYN, J. 2000. *Dynamic logic*. MIT Press.

HAREL, DAVID. 1979. *First-Order Dynamic Logic*. Springer-Verlag New York, Inc.

HAREL, DAVID. 1984. *Dynamic Logic*. 1st edn. Handbook of Philosophical Logic, vol. II. Dordrecht: Reidel. Chap. 10, pages 497–604.

IGARASHI, ATSUSHI, PIERCE, BENJAMIN, & WADLER, PHILIP. 1999 (October). *Featherweight Java: A Minimal Core Calculus for Java and GJ*. Tech. rept. MS-CIS-99-25. University of Pennsylvania.

Igarashi, Atsushi, Pierce, Benjamin C., & Wadler, Philip. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, **23**(3), 396–450.

JavaCard. 2004. *Java Card 2.2 Platform Specification.* `http://java.sun.com/products/javacard/`.

Kruskal, Joseph B. 1960. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, **95**(may), 210–225.

Leavens, Gary T., Baker, Albert L., & Ruby, Clyde. 1998 (June). *Preliminary Design of JML: A Behavioural Interface Specification Language for Java.* Tech. rept. 98-06y. Department of Computer Science, Iowa State University. revised several times until June 2004.

Lindholm, Tim, & Yellin, Frank. 1999. *Java Virtual Machine Specification.* 2nd edn. Addison-Wesley Longman Publishing Co., Inc.

MDA. 2003. *OMG Model Driven Architecture.* `http://www.omg.org/mda/`.

Menzel, Wolfram, & Schmitt, Peter H. 2000. *Formale Systeme.* Vorlesungsskriptum Fakultät für Informatik , Universität Karlsruhe.

Nash-Williams, C. St. J. A. 1963. On well-quasi-ordering finite trees. *Proc. of the Cambridge Philosophical Society*, **59**(4), 833–835.

Nipkow, Tobias. 2003. Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language.

Nipkow, Tobias, & Paulson, Lawrence C. 2000. *Isabelle HOL - The Tutorial.*

Nipkow, Tobias, Paulson, Lawrence C., & Wenzel, Markus. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS, vol. 2283. Springer.

Paulson, Lawrence C. 1994. *Isabelle: A Generic Theorem Prover.* Springer. LNCS 828.

Platzer, André. 2004. *Using a Program Verification Calculus for Constructing Specifications from Implementations.* Minor Thesis, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems. `http://www.functologic.com/logic/Minoranthe.ps`.

Rumbaugh, James, Jacobson, Ivar, & Booch, Grady. 1998. *The Unified Modeling Language Reference Manual*. Addison-Wesley.

Rumbaugh, James, Jacobson, Ivar, & Booch, Grady. 1999. *The Unified Modeling Language User Guide*. Addison-Wesley.

Schlager, Steffen. 2000. *Erweiterung der Dynamischen Logik um temporallogische Operatoren*. M.Phil. thesis, Universität Karlsruhe, Fakultät für Informatik. `http://i12www.ira.uka.de/~schlager/publications/Studienarbeit.ps.gz`.

Schlager, Steffen. 2002. *Behandlung von Integer-Arithmetik bei der Verifikation von Java-Programmen*. M.Phil. thesis, Universität Karlsruhe, Fakultät für Informatik. `http://www.key-project.org/olderPublicat.html`.

Schmitt, Peter H. 2003 (Mai). *Nichtklassische Logiken*. Vorlesungsskriptum Fakultät für Informatik , Universität Karlsruhe.

Somogyi, Zoltan, Henderson, Fergus, Conway, Thomas, & O'Keefe, Richard. 1995. Logic programming in the real world. *In: Proceedings of the ILPS'95 Postconference Workshop on Visions of the Future of Logic Programming*. Porland, Oregon.

Stärk, Robert F., & Nanchen, Stanislas. 2001. A Logic for Abstract State Machines. *Pages 217–231 of:* Fribourg, L. (ed), *Computer Science Logic (CSL 2001)*. Springer-Verlag, Lecture Notes in Computer Science 2142.

Thai, Thuan L., & Lam, Hoang. 2001. *.NET Framework Essentials*. O'Reilly & Associates, Inc.

TogetherSoft. 2003. *TogetherSoft WWW Homepage*. `http://www.togethersoft.com/`.

von Oheimb, David. 2001. Hoare Logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, **13**(13). `http://isabelle.in.tum.de/Bali/papers/CPE01.html`.

von Oheimb, David, & Nipkow, Tobias. 2001. Hoare Logic for NanoJava: Auxiliary Variables, Side Effects and Virtual Methods revisited.

Wilhelm, Reinhard, & Maurer, Dieter. 1997. *Übersetzerbau: Theorie, Konstruktion, Generierung*. 2nd edn. Berlin, Heidelberg, New York: Springer-Verlag. 2., überarb. und erw.Aufl.

XviD. 2004. *The XviD project*. release 1.0.2. http://www.xvid.org/.

Index