Fachbereich 3
Mathematik - Informatik
Universität Bremen

Diploma thesis

Monadic Dynamic Logic: Application and Implementation

Dennis Walter

July 20, 2005

Supervised by
Lutz Schröder and Till Mossakowski

I hereby confirm that I independently worked on and wrote this thesis and that I only used the references and auxiliary means indicated herein.

Bremen, July 20, 2005

Dennis Walter

‘There must be some way out of here’
said the joker to the thief.
‘There’s too much confusion,
I can’t get no relief’

Bob Dylan

Contents

1 Introduction

1.1 Motivation and Classification

The study of formal methods, i. e. the mathematically rigorous specification, design and analysis of software systems, has a long tradition in the – by itself relatively short – history of computer science. It has, however, not gained as much attention for being an effective and efficient means of software design as for example object oriented software design or UML [7] modelling have. Quite the contrary, it is often considered a very complex and demanding way of creating software, requiring specialised skills in mathematics of all developers involved and taking a long time to finish. Its application is therefore often rejected and regarded as too expensive. A similar situation can be found in the field of software verification and validation, where the predominant method of operation is to perform clever and extensive testing.

Despite these facts, we consider formal methods to be a very valuable arrow in a computer scientist's quiver; the use of formal methods is the only known way to actually prove the absence of errors in a system, whereas other methods, e. g. testing, can only show their presence. Experience has shown (see [17, 36]) that there are applications in which formal methods is a means of not only writing better software, but writing it in proper time. Examples include the verification of AMD's floating point processing unit of the K7 CPU, which Intel also did for their Pentium Pro CPU, the verification of several cryptographic protocols, and the employment of various model-checkers in hardware design. We consider essential two features when using formal methods: firstly, it must be reasonably easy to understand and use and, secondly, there has to be a software tool that assists the user and relieves him of the duty of performing trivial but highly detailed proof steps.

Within the subject of formal methods, there are three major branches that are concerned with giving meaning to programs and programming languages and in particular with proving equivalences of programs; these are

- *Operational semantics*, in which the execution of programs is described by a transition (or evaluation) relation between program fragments, an overall state and the value in which an expression is supposed to result. Among the various incarnations of operational semantics, an approach popularised by Plotkin is used very commonly. This method employs rules that are structurally similar to those found in deduction systems to determine the evaluation of a program in a syntax directed way (see [29]).

  Other known examples of operational semantics, which are quite close to actual implementations of the respective language include Warren's abstract machine for interpreting Prolog programs or the SECD machine for evaluation of lambda terms.

- *Denotational semantics*, in which so-called semantic functions are defined, which map language elements into their intended interpretation in a mathematical model of the programming language at hand. In simple cases this is quite similar to giving a model for a language of first-order logic, but in common applications (e. g. when giving a semantics for a functional language featuring some kind of recursion) rather sophisticated mathematics (in concrete terms: the field of *domain theory* with its notions of least upper bounds, continuous functions and fixed points) become involved, cf. [30, 40]. A cornerstone of this kind of semantics is the compositionality of its semantic functions, i. e. semantic functions for composite terms can be explained through the meaning of their component parts alone.

- *Program logics* (often called axiomatic semantics), which differ from the above methods as they do not directly assign meaning to programs, but rather embed the programming language into a logical framework that allows for making statements about a program's behaviour and, hence, its correctness. Hoare's article [9] is the classic introductory paper about program logics, a special kind of which therefore are termed *Hoare logics*.

In this thesis we describe, apply and implement a program logic named *(propositional) monadic dynamic logic* [34] which allows one to prove properties of monadic programs. The logic allows to reason about partial correctness of programs, but also to prove termination and thus total correctness in one and the same framework.

Monads constitute an elegant technique for consistently abstracting and analysing several kinds of language features, e. g. side effects, nondeterminism, exceptions, input and output as well as combinations of these. The use of a logic of monadic programs is twofold: it can be used to rather directly reason about programming languages that support the notion of a monad (such as Haskell), but it can also be used to reason about programs written in imperative first-order languages, if one creates a monadic model of the key features of such a language and translates programs into this model. For Java this has been done recently (see [11]) and the calculus described in this thesis has been extended to deal with Java-like abnormal termination. This extension does not solely cover actual exceptions but also termination of a method through a `return` statement, or the interruption of execution of a while-loop through a `break` or `continue` statement.

An important feature of the logic is the fact that it is monad independent, which means that the general logical framework is applicable to every monad that allows the interpretation of dynamic logic, which is the case for nearly all computationally relevant monads. A notable exception to this is the continuation monad. Instantiations of the logic for concrete monads are realised through additional axioms determining the monad-specific operations, like reference writing in the state monad, or nondeterministic choice in the nondeterminism monad. While bearing some resemblance to Pitt's evaluation logic [27], the calculus described here is equipped with a purely monadic semantics, whereas Pitts provides a semantics only through certain hyperdoctrines acting on top of the monad. An alternative, but merely global semantics for the modal operators was given by Moggi [19]. However, a critical property of the modal operators is their *local* character, which is retained in the calculus described here. On top of it, a Hoare calculus for total correctness can easily be formulated.

1.2 Problem Setting

The aim of this work is twofold: on the one hand, it constitutes the first extended application of the recently developed calculus of monadic dynamic logic and thus demonstrates how this calculus can be applied to serious verification tasks. To name two examples, the total correctness of a breadth-first search algorithm and of a pattern matching algorithm involving Java-like exception handling have been established.

On the other hand, driven by the insight that due to the complexity even of relatively small software systems it is not feasible to carry out formal proofs about these manually, the calculus had to be implemented in some proof assistant tool. Furthermore, the formalisation within such a tool provides further evidence of the correctness of one's inferences – provided one trusts in the correctness of the tool, of course. We chose the generic proof assistant (often termed 'theorem prover') Isabelle/HOL in which we could base our implementation on a stable and well developed formalisation of higher-order logic. Isabelle/HOL comes with tools for proving theorems outright (by means of a classical tableau reasoner) as well as a term rewriting system that allows for equational reasoning and functional programming. Tasks during this implementation included the definition of a syntax for monadic dynamic logic, proving the theorems needed as foundations for the logic, and working out theorems and setting up Isabelle's automatic proof facilities to make life easier when applying the logic. The embedding into higher-order logic is a deep one in the sense that we define monadic logical connectives $\wedge_D$, $\longrightarrow_D$, etc. as well as a predicate $\vdash$ asserting the validity of monadic formulae. HOL formulae may, however, appear in monadic formulae thanks to existence of an insertion function *Ret* mapping HOL formulae into those of dynamic logic.

1.3 Structure of the Thesis

This thesis is structured as follows:

Chapter 2 introduces the theoretical background needed for the further development, which is the lambda calculus in its typed and untyped form, and the categorical concept of a monad as it is used in computer science.

Chapter 3 contains some preliminary work which eventually leads to the formulation of the calculus of monadic dynamic logic. This calculus is then extended to deal with the peculiarities of the exception monad such that a pattern match algorithm can be specified and proved correct.

Chapter 4 provides basic theorems characteristic of dynamic logics and it contains an extended application of the calculus to several monads. For example, the correctness of a tree search algorithm is established.

Chapter 5 gives an overview of the proof assistant Isabelle, its basic concepts, the higher-order logic HOL and the Isar proof language.

Chapter 6 describes the implementation of the calculus in Isabelle/HOL. This includes background work on properties of monadic programs as well as the setup of the calculus itself and the presentation of example specifications and proofs. Also, some differences between the calculus as laid out in [34] and its implementation are depicted.

Chapter 7 concludes by summarising the achievements and pointing out future work.

The appendix contains a Haskell implementation of the exception monad programs described in Section 3.4, a list of rules frequently used in Isabelle/HOL, and finally a typeset edition of the theory files which make up the calculus of monadic dynamic logic as implemented in Isabelle.

2 Theoretical Basis

We now provide the foundations needed to understand the further development of monadic dynamic logic and its implementation in Isabelle/HOL. A complete survey of all concepts involved would certainly go beyond the scope of this thesis, so that we assume basic familiarity with functional programming languages, especially Haskell, which is taught at the university of Bremen during the undergraduate studies, as well as basic knowledge of first-order logic. Instead we initially concentrate on two topics that are of fundamental importance in the following. First, we introduce the lambda calculus, in its pure and untyped as well as its typed form with added constants. A higher-order logic based on the lambda calculus will be described in Chapter 5 along with other foundations of Isabelle. Second, we devote a section to the description of monads and their applications in computer science. Although monads are a concept of category theory, we do not provide an introduction into the latter since we will merely use its basic terminology.

A good introduction to functional programming in Haskell with a focus on monadic programming is given in [10], whereas [1] introduces first-order and higher-order logic in a mathematically rigorous manner with an eye on historical developments. A book on category theory aimed at students of computer science is [26]; [18] delves even deeper into the topic, but with its focus geared towards readily educated mathematicians.

2.1 The Lambda Calculus

The lambda calculus is a formalism for describing and analysing functions. It has been developed by Alonzo Church in the 1930's and has influenced many programming languages since then. In particular, functional languages such as Haskell or ML have been directly influenced by the ideas underlying the lambda calculus, in particular its syntax. One of the key ideas of the lambda calculus is to make a function that takes its argument (say, x) to a certain expression containing that argument (e. g. $x+y$) an expression itself (in the example, this function would be denoted by $\lambda x. x+y$). Thus, lambda expressions (or: lambda terms) denote anonymous functions, which can be used as values themselves, for example as input into another function, like in $(\lambda x.x)(\lambda x.x+y)$, which furthermore indicates that the notation for function application is simply juxtaposition.

We will now explain some basic concepts on the basis of the *untyped lambda calculus*, in which all expressions are considered to have one universal type, since in this calculus the concepts are easier to explain. Later on typed calculi will be more important, as they are the basis of higher-order logic and modern functional programming languages. Nonetheless, the concepts introduced below provide a good starting point and apply to advanced calculi in similar form.

2.1.1 Syntax and Terminology

The untyped lambda calculus is conceptually very simple, but encompasses the whole expressive power of what is known as the computable functions or the Turing machine, i. e. to say every computable function can be formalised in the lambda calculus. Given a countably infinite set of variables *var* (e. g. the variable set $\{x_i \mid i \in \mathbb{N}\}$), the abstract syntax of lambda expressions can be given as

$$exp ::= var \mid \lambda var.\, exp \mid exp\ exp \tag{2.1}$$

where an expression of the form $\lambda x.\, e$ is called an *abstraction*, which is intuitively to be understood as a function mapping its argument x to the value denoted by the expression e. Expressions that have the form of the third alternative are called *applications* since they stand for applications of functions to arguments.

In a lambda expression $\lambda x.\, e$ the occurrence of the variable x directly succeeding the λ is called a *binding occurrence*, λ itself is called a *(variable) binder* and x is considered to be *bound* in the subexpression e, which is the *scope* of the binder. All variables in a lambda expression that are not bound are *free*. An expression that has no free variables is called a *closed* expression. To avoid unnecessary use of brackets when writing down concrete lambda expressions, we will stick to the common convention that the scope of a λ extends to the right as far as possible without breaking the existing bracketing hierarchy and that function application associates to the left.

Example 2.1. The expression $\lambda x.\, xx$ is to be read as $\lambda x.\, (xx)$, whereas $(\lambda x.\, x)(\lambda y.\, y)\lambda x.\, xx$ denotes $((\lambda x.\, x)(\lambda y.\, y))(\lambda x.\, (xx))$.

It is often useful to work with the set of all free variables of an expression, which leads to the following definition.

Definition 2.2. The set FV of *free variables* of a lambda expression is defined by induction on the structure of the expression. Thus, one has

$$\begin{aligned} FV(x) &= \{x\} \\ FV(ee') &= FV(e) \cup FV(e') \\ FV(\lambda x.\, e) &= FV(e) - \{x\} \end{aligned} \tag{2.2}$$

One further elementary concept is needed to formalise the idea of function evaluation in the untyped lambda calculus: the *substitution* of a lambda expression for a free variable.

Definition 2.3. The substitution of an expression e' for the variable x in e, written $e[e'/x]$ can be defined as follows

$$x[e'/x] = e' \tag{2.3}$$

$$y[e'/x] = y \qquad \text{provided } x \neq y \tag{2.4}$$

$$(\lambda x.\, e_0)[e'/x] = \lambda x.\, e_0 \tag{2.5}$$

$$(\lambda y.\, e_0)[e'/x] = \lambda y'.\, e_0[y'/y][e'/x] \qquad \text{provided } x \neq y \text{ and } y' \notin FV(e') \cup \{x\} \tag{2.6}$$

$$(e_0 e_1)[e'/x] = e_0[e'/x]e_1[e'/x] \tag{2.7}$$

where (2.5) and (2.6) make sure that the phenomenon of bound variable capture is avoided, i. e. after substitution all variables free in e' will be free in $e[e'/x]$. As a shortcut, one should let $y' = y$ in (2.6) whenever possible, i. e. when $y \notin FV(e')$.

The concepts of binding and bound variables are quite similar to those in first-order logic, where $\forall$ and $\exists$ are commonly used as binders. Since in both cases bound variables merely provide a local name with a local meaning that might differ from the meaning outside the scope of the binder, the lambda calculus also features the concept of bound variable renaming. Changing an expression e into an expression e' by renaming some of its bound variables in subexpressions is called *α-conversion*. It is intuitively clear that this process does not change the meaning of an expression, and in fact this can be shown. Hence, it makes sense to say that two expressions are equivalent up to renaming of bound variables (notation $e \equiv_\alpha e'$) if they can be converted into each other purely by applying α-conversion. It is often convenient to mentally identify expressions that are equivalent up to α-conversion, rather than making this identification a part of the formal system; in fact, it is possible to formalise the lambda calculus in such a way that all α-equivalent expressions are syntactically equal.

Example 2.4. The simplest case of α-conversion is to change the name of the bound variable in the identity function: we have $\lambda x.\, x \equiv_\alpha \lambda y.\, y$. There are, however, cases where more attention has to be paid: in renaming $\lambda x.\, \lambda y.\, xy$ into the obviously equivalent expression $\lambda y.\, \lambda x.\, yx$, the first step involves renaming the inner abstraction with the help of an intermediate variable: $\lambda x.\, \lambda y.\, xy \equiv_\alpha \lambda y.\, \lambda x'.\, yx' \equiv_\alpha \lambda y.\, \lambda x.\, yx$. Otherwise, a bound variable capture would occur, resulting in the entirely different expression on the right hand: $\lambda x.\, \lambda y.\, xy \not\equiv_\alpha \lambda y.\, \lambda y.\, yy$

2.1.2 Function Evaluation by Reduction

The concept of function evaluation is formalised in the lambda calculus through the concept of reduction. An application expression of the form $(\lambda x.\, e)e'$ is called a *redex*, which is short for reducible expression. A reduction then is the transformation of $(\lambda x.\, e)e'$ into $e[e'/x]$. The latter expression appears to be somewhat simpler, but this idea can be misleading, since it is possible for it to be larger than the former or in fact even equal to it. In any case, it coincides with the intuition behind function application: the function's argument (or formal parameter, in computer science parlance) is substituted by the value (or actual parameter) applied to it. Reducing an expression or one of its subexpressions in this way is called β-reduction. If an expression contains no redices it is said to be in *normal form*.

Another way of converting an expression is by the so called η-contraction, which allows to convert an expression $\lambda x.\, ex$, where e does not contain x as a free variable, into the simpler expression e. The idea is that one may see $\lambda x.\, ex$ as a function that takes its argument x simply to apply it to the function e and thus one may identify it with e.

Remark 2.5. The syntax of the lambda calculus suggests that there can only be functions that take exactly one argument; but this does not impose any restrictions concerning the expressibility of multi-argument functions, since a function taking n arguments may be expressed as $\lambda x_1.\lambda x_2.\ldots\lambda x_n.\, e$ (frequently abbreviated to $\lambda x_1 \ldots x_n.\, e$). The following reduction sequence may suggest how this works: $(\lambda f.\, \lambda x.\, fx)gy \rightsquigarrow (\lambda x.\, gx)y \rightsquigarrow gy$. The transformation of a function taking a single argument in form of a tuple into an equivalent one taking 'each argument at a time' as shown above has been proposed by Schönfinkel and Curry. Therefore, it is often called *currying*.

One might ask how the simple untyped lambda calculus can be used to express common functions like addition and multiplication on the natural numbers and, to that effect, how natural numbers themselves can be represented. Obviously, as there is nothing else available, they will have to be functions. To provide a short insight into this problem, we will now show

how to represent even simpler values and functions, namely the booleans and the conjunction function.

Lemma 2.6. *Let* True, False *and* $(\_ \wedge \_)$ *denote the lambda expressions defined below.*

$$\mathsf{True} := \lambda x.\, \lambda y.\, x \qquad \mathsf{False} := \lambda x.\, \lambda y.\, y \qquad e \wedge e' := (e\ e')\ \mathsf{False}$$

Then the following holds:

$$\begin{aligned} \mathsf{True} \wedge \mathsf{True} &\longrightarrow \ldots \longrightarrow \mathsf{True} & \mathsf{True} \wedge \mathsf{False} &\longrightarrow \ldots \longrightarrow \mathsf{False} \\ \mathsf{False} \wedge \mathsf{True} &\longrightarrow \ldots \longrightarrow \mathsf{False} & \mathsf{False} \wedge \mathsf{False} &\longrightarrow \ldots \longrightarrow \mathsf{False} \end{aligned}$$

Proof. By a direct calculation:

$$\begin{aligned} \mathsf{True} \wedge \mathsf{True} &\longrightarrow ((\lambda x.\, \lambda y.\, x)(\lambda x.\, \lambda y.\, x))\ \mathsf{False} \\ &\longrightarrow (\lambda y.\, (\lambda x.\, \lambda y.\, x))\ \mathsf{False} \\ &\longrightarrow \lambda x.\, \lambda y.\, x \longrightarrow \mathsf{True} \end{aligned}$$

$$\begin{aligned} \mathsf{True} \wedge \mathsf{False} &\longrightarrow ((\lambda x.\, \lambda y.\, x)(\lambda x.\, \lambda y.\, y))\ \mathsf{False} \\ &\longrightarrow (\lambda y.\, (\lambda x.\, \lambda y'.\, y'))\ \mathsf{False} \\ &\longrightarrow \lambda x.\, \lambda y'.\, y' \longrightarrow \mathsf{False} \end{aligned}$$

The remaining cases are analogous. □

Upon leaving the untyped calculus and turning our eyes to typed calculi possibly with additional constants, we state one central theorem that ensures that in what way an expression might ever be converted, it is always possible to 'cross the ways' of other strategies.

Proposition 2.7 (Church-Rosser Theorem). *If an expression e can be evaluated to e_0 in arbitrary steps according to the rules given above, and it can also be evaluated to e_1, then there is an expression $\bar{e}$ such that e_0 and e_1 can be converted to $\bar{e}$.*

2.1.3 Adding Types and Constants

Even if one accepts that the untyped lambda calculus is powerful enough to express every computable function, and that reduction to normal form is a kind of evaluation of these functions, it is obviously not very natural to directly work in this calculus. In fact, it took several decades until a denotational semantics for it was found by Dana Scott, which does not raise problems similar to those encountered in naive (untyped) set theory like Russell's paradox. Modern functional programming languages nowadays rely on type systems, where every expression is assigned a unique type. This idea goes back to Russell and Whitehead, who demonstrated the usefulness of types in higher-order logic within their influential work *Principia Mathematica* (1913). Church and Curry are the names commonly associated with typed lambda calculi (see [2] for a detailed comparison).

We will equip a variant of the lambda calculus with types and constants, thereby introducing some recurrent concepts of formal systems. First of all, the abstract syntax of lambda terms has to be extended slightly:

$$\begin{aligned} exp ::=\, & var \mid exp + exp \\ & \mid exp\ exp \mid \lambda var.\, exp \\ & \mid \langle exp, exp \rangle \mid exp.\mathsf{fst} \mid exp.\mathsf{snd} \end{aligned} \tag{2.8}$$

where $\langle e_1, e_2\rangle$ and $n_1 + n_2$ should respectively be interpreted as a pair of expressions e_1, e_2 and a sum of natural numbers n_1, n_2. Selection of the first and second components of a tuple are expressed by attaching $.\mathsf{fst}$ or $.\mathsf{snd}$ to it. Of course, this syntax alone does not prevent ill-typed expressions like $e_1 + e_2$ where, for example, e_1 is a function.

Types can be introduced in the following way. Usually one starts with a given set *btyp* of *base types* (which in common programming languages will include the type *int* of integers, the type *float* of floating point numbers and the type *char* of characters). Complex types can then be formed by applying *type constructors* to the base types and the already created compound types. We will take two type constructors $(\_ \to \_)$ (called the function type constructor) and $(\_ \times \_)$ (called the product type constructor) into the lambda calculus. They are introduced in a purely syntactic manner, but their intuitive interpretation ought to be that of a function and product type, respectively. In summary, the abstract syntax of types is the following:

$$typ ::= btyp \mid typ \to typ \mid typ \times typ \tag{2.9}$$

Next comes the concept of a *context*, which is a finite sequence $\Gamma = [x_1 : \sigma_1, \ldots, x_n : \sigma_n]$ of variable/type pairs, subject to the condition that $x_i \neq x_j$ for $i \neq j$. A context is used in *typing judgements* $\Gamma \vdash e : \sigma$ which should be read as "if the variables occurring in the context Γ have the assigned types, then expression e has type σ". Only typing judgements where each $x \in FV(e)$ occurs in the context Γ are allowed. Contexts may be compound, like in $\Gamma, x : \sigma \vdash e : \tau$, where it is implicitly assumed that x does not occur in Γ, or $\Gamma, \Gamma' \vdash e : \tau$, in which the sets of variables of Γ and Γ' have to be disjoint.

We would like to equip the calculus with a type *nat* representing the natural numbers and constants $\mathsf{0} : nat$ and $Suc : nat \to nat$ for zero and the successor function. This can be done in the following way: add *nat* to the set of base types (or pick an existing base type when appropriate), and let $\Gamma_0 = [\mathsf{0} : nat, Suc : nat \to nat]$ be the *base context*, where $\mathsf{0}$ and *Suc* are arbitrary variables which are given mnemonic names here. This context will be used implicitly in all typing judgements, such that $\Gamma \vdash e : \sigma$ actually means $\Gamma_0, \Gamma \vdash e : \sigma$, thus excluding their use as variables of different types due to the convention that Γ_0 and Γ must have disjoint variables. This will of course not introduce the properties of the natural numbers, e. g. with respect to addition (commutativity, associativity, zero as a unit element), but will merely make them available on the type-theoretical level.

Figure 2.1 lists the rules of a (decidable) deduction system, which will serve the purpose of determining whether given typing judgements are valid. These rules are to be read in the standard way: if the premisses above the horizontal bar are derivable in the calculus, one may also derive the conclusion below the bar. Now one defines a typing judgement to be valid if and only if there is a proof (see Definition 2.8 below) of the judgement from the given rules. The presentation of a proof will slightly deviate from the standard structure of a proof in natural deduction as it will be linearised to make presentation easier.

Definition 2.8. A *proof from rules* of a statement S is a sequence of statements $S_1, \ldots, S_n$ where $S_n = S$ and for each of the S_i one of the following holds:

- S_i is an axiom, i. e. a rule without premisses.
- S_i is the conclusion of a rule whose premisses $P_1, \ldots, P_k$ have been proved, i. e. for all P_j $(1 \leq j \leq k)$ there is an $S_{j'}$ $(1 \leq j' < i)$ such that $P_j = S_{j'}$.

$$\text{(var)}\quad \frac{}{\Gamma, x:\sigma, \Gamma' \vdash x:\sigma} \qquad \text{(wk)}\quad \frac{\Gamma \vdash e:\sigma}{\Gamma, \Gamma' \vdash e:\sigma}$$

$$\text{(abs)}\quad \frac{\Gamma, x:\sigma \vdash e:\tau}{\Gamma \vdash \lambda x.\, e:\sigma \to \tau} \qquad \text{(app)}\quad \frac{\Gamma \vdash e:\sigma \to \tau \quad \Gamma \vdash e':\sigma}{\Gamma \vdash e\, e':\tau}$$

$$\text{(prodI)}\quad \frac{\Gamma \vdash e_1:\sigma_1 \quad \Gamma \vdash e_2:\sigma_2}{\Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times \sigma_2} \qquad \text{(add)}\quad \frac{\Gamma \vdash e_1: \mathit{nat} \quad \Gamma \vdash e_2: \mathit{nat}}{\Gamma \vdash e_1 + e_2 : \mathit{nat}}$$

$$\text{(fst)}\quad \frac{\Gamma \vdash e:\sigma_1 \times \sigma_2}{\Gamma \vdash e.\mathsf{fst}:\sigma_1} \qquad \text{(snd)}\quad \frac{\Gamma \vdash e:\sigma_1 \times \sigma_2}{\Gamma \vdash e.\mathsf{snd}:\sigma_2}$$

Figure 2.1: Type inference rules for the simply typed lambda calculus

Example 2.9. Here is a proof of the typing judgement of a function that sums its arguments and adds one to it; recall that the base context is not shown. The right column indicates which rule has been used with which previous lines as premisses to obtain the respective statement.

(.1)	$\vdash 0 : \mathit{nat}$	(var)
(.2)	$\vdash \mathit{Suc} : \mathit{nat} \to \mathit{nat}$	(var)
(.3)	$x:\mathit{nat}, y:\mathit{nat} \vdash \mathit{Suc}\ 0 : \mathit{nat}$	(app: .2,.1; wk)
(.4)	$x:\mathit{nat}, y:\mathit{nat} \vdash x : \mathit{nat}$	(var)
(.5)	$x:\mathit{nat}, y:\mathit{nat} \vdash y : \mathit{nat}$	(var)
(.6)	$x:\mathit{nat}, y:\mathit{nat} \vdash x+y : \mathit{nat}$	(add: .4, .5)
(.7)	$x:\mathit{nat}, y:\mathit{nat} \vdash (x+y)+\mathit{Suc}\ 0 : \mathit{nat}$	(add: .6, .3)
(.8)	$x:\mathit{nat} \vdash \lambda y.\, (x+y)+\mathit{Suc}\ 0 : \mathit{nat} \to \mathit{nat}$	(abs: .7)
(.9)	$\vdash \lambda x.\, \lambda y.\, (x+y)+\mathit{Suc}\ 0 : \mathit{nat} \to \mathit{nat} \to \mathit{nat}$	(abs: .8)

The following example shows that certain functions like the identity function are polymorphic in the sense that there are proofs of different types of the syntactically identical function $(\lambda x.\, x)$:

(.1)	$x:\sigma \vdash x:\sigma$	(var)
(.2)	$x:\tau \vdash x:\tau$	(var)
(.3)	$\vdash \lambda x.\, x:\sigma \to \sigma$	(abs: .1)
(.4)	$\vdash \lambda x.\, x:\tau \to \tau$	(abs: .2)

An important point concerning the introduced 'simple' types, however, which partly explains why they are called that, is the fact that the simply typed lambda calculus lacks a genuine notion of polymorphism. This means that every function whose type is provable is assigned a fixed type σ, such that there is an identity function on the type *nat* of natural numbers $\lambda x.\, x : \mathit{nat} \to \mathit{nat}$, and an identity function $\lambda x.\, x : \mathit{nat} \times \mathit{nat} \to \mathit{nat} \times \mathit{nat}$ on pairs of *nat*'s, but these functions are not identical. This leads to the typical problems (or at least inconveniences) found in programming languages that also lack the concept of polymorphism, like the necessity to give different names to functions that essentially perform the same action (although in the small calculus described here there is no way to give functions a name, it could be easily extended to allow this, e. g. by the definition of let-terms). The lack of polymorphism also motivated the introduction of the projection operations fst and snd on the syntactical level rather than making them constants like *Suc*: we could only have expressed the type of $\mathsf{fst} : \sigma_1 \times \sigma_2 \to \sigma_1$ for fixed types σ_1 and σ_2, but they are intended to be used on any kind of tuple.

One might wonder if contexts in typing judgements are really necessary, since the initial goal was to assign types to expressions, but then one must recall that expressions may contain free variables, as opposed to (functional) programs, which may be identified with the closed lambda expressions. Typing judgements, then, essentially tell what the types of the free variables of an expression are. Furthermore, it is often convenient (and sometimes necessary to make typing decidable) to extend the syntax of typed calculi in such a way that types become a part of it. A common place where types are often explicitly annotated is at the binding occurrences of variables, like in $\lambda x : \sigma . e$. We will leave the types of bound variables implicit whenever possible, i. e. when they are determined by the context.

2.2 Monads in Computer Science

Originally arisen in category theory, monads have been introduced into computer science by Moggi [20] as an elegant device for dealing with manifold kinds of side effects. Initially, their value for enabling an abstract treatment of the semantics of several programming language constructs was appreciated, but it was soon realised that these benefits could also directly be exploited in purely functional programming languages. Wadler and Peyton Jones [37, 15] advocated the monadic style of functional programming for Haskell and it was finally included in the Haskell 98 language standard as the definitive way of communicating with the real world, i. e. for dealing with input and output.

In the field of denotational semantics, monads come into play when equipping a programming language with a categorical semantics – as opposed to a set-theoretic one – such that one reasons about *objects* instead of sets and *morphisms* instead of functions (see [13] for a categorical semantics of Java). Monads arise in this setting as a very natural and convenient concept for interpreting many kinds of side effects like exceptions or state changes in a uniform way.

We will first give some examples of concrete monads from the realm of functional programming, then we will introduce the abstract categorical concept of monads, and finally we will discuss Moggi's meta-language, which essentially is an equational logic that can be identified, in a sense yet to be specified, with categories equipped with strong monads.

2.2.1 Monads in Haskell

One of the most well-known applications of a monad is to simulate a global store of assignable variables in a way that does not conflict with referential transparency. The simplest idea to simulate a global store in the absence of assignable variables is to make the store explicit in every function by letting each function have one further argument that acts as the global store, e. g. a tuple containing all values involved, and furthermore extending its return value to be a pair of the actual return value and the possibly modified store. This way of proceeding is, however, extremely impractical and by no means modular: if the structure of the store has to be modified, this adaptation will have to be done in every single function.

The monadic approach to side effects does not suffer from such deficiencies and is thus much more elegant. The first step in turning the language feature of a global store into a monad (which is commonly called the *state monad*) is to define a datatype TA that represents the *computations* over values of type A. In this case, computations will simply be functions that take the global store[1] as input and return a value of type A together with the modified store. The expressions of type TA as given below are often called *state transformers* (note that a is a type variable, so one has types TA for each concrete type A).

```
type T a = (S -> (S, a))
```

The next step is to define the two basic polymorphic operations on computations, that on the one hand enable sequencing of computations, and on the other hand let us turn values into computations that do nothing except return the inserted value. The Haskell-style signatures of these functions are

```
(>>=) :: T a -> (a -> T b) -> T b
ret   :: a -> T a
```

where the infix operation $(\gg=)$ is called *binding*, in which the second parameter is a function that will be fed the resulting value of the computation which is the first parameter. The overall result of a computation $p \gg= f$ will be the result of f. To make these ideas clearer, we will provide the definitions of these operations for the state monad.

```
p >>= f   = \ s-> let (s', a) = p s in f a s'
ret x     = \ s-> (s, a)
```

where the backslash is Haskell syntax for a lambda abstraction. Recalling that p actually is a function from the state to a pair of state and return value, one sees that binding really implements a kind of sequencing: first, p is given the current state to evaluate to a new state and a value, which are then given as inputs to f, whose return value is also the return value of the overall computation.

What is called a monad in this context is the triple $(T, \gg=, ret)$, i. e. the type constructor T together with the two basic polymorphic operations. For the state monad to be useful, one naturally has to introduce further operations for reading the state and for updating it. Other operations can then be defined in terms of these. A possible signature for the former two operations is

```
get    :: T S
update :: S -> T ()

get       = \ s-> (s, s)
update s1 = \ s0-> (s1, ())
```

Finally, we present some computationally relevant monads, together with the possible definitions of T, $\gg=$ and ret, respectively. These definitions will be given in a set-theoretic

[1]The store is treated abstractly as a type S here, but may be imagined as a finite map of variable-name/value pairs

manner, but the translations to Haskell datatypes and functions should not constitute a problem. This is done so to motivate the more abstract definition of monads in the next section and because monads can not merely be used as a feature of a concrete programming language, but also to study programming languages themselves in an abstract way.

- The *state monad* has been described above. The appropriate definitions are
  $TA = (S \to S \times A)$ for some fixed set S representing the state, where $\times$ denotes the Cartesian product of sets and $X \to Y = \{f \mid f : X \to Y\}$ denotes the function space of all functions from X to Y,
  $(p \gg= f) = \lambda s.\, \mathsf{let}\ \langle s', a\rangle = p\ s\ \mathsf{in}\ f\ a\ s'$ and
  $ret\ x = \lambda s.\, \langle s, x\rangle$, where $\langle\rangle$ denotes pairing.

- The *exception monad* is used to model abnormal termination. One has
  $TA = (A + E)$, i. e. the disjoint union (corresponding to a sum datatype) of the result set A with some global set E of exceptions. In the simplest case, $E = \{\perp\}$, such that an exception indicates nontermination or failure,
  $(p \gg= f) = \mathsf{case}\ p\ \mathsf{of}\ (inl\ a) \to f\ a \mid (inr\ e) \to inr\ e$; this definition models the usual effect of an exception, in that the right-hand computation is evaluated only if the left-hand one did not raise an exception. The definition of the case-construct is standard. *inl* and *inr* stand for the left and right injections (corresponding to constructors of the same datatype), and
  $ret\ x = inl\ x$, which once more makes clear that *ret* actually is just an embedding of values into computations.

- The *nondeterminism monad* captures the effects of multiple possible outputs of a function by letting
  $TA = \mathscr{P}_{\mathrm{fin}}(A)$, i. e. T maps a set A to all its finite subsets,
  $(p \gg= f) = \bigcup\{f\ x \mid x \in p\}$; p is a subset of A, and f is applied to all elements of p, the result of which will be a set of sets, which is therefore flattened by taking the union of all these sets, and
  $ret\ x = \{x\}$, i. e. the singleton set containing only x.

- A combination of the *list monad* and a particular state monad is used in [12] to elegantly implement a library of monadic parser combinators. In it, one has
  $TA = (List\ I \to List\ (List\ I \times A))$, where I is a fixed, finite set of input tokens, and *List* maps a set A to the set of all finite lists of elements over A, and
  $p \gg= f = \lambda s.\, concat\ (map\ (\lambda \langle x, s'\rangle.\, f\ x\ s')\ (p\ s))$. Here, *concat* and *map* behave exactly like the well-known total functions of the same name as defined in the Haskell prelude. What happens is that p is applied to the the current state (a list of input tokens), returning a list of result pairs. To each result pair, the function f is applied, resulting in a list of lists of result pairs. These have to be flattened by *concat*, very much like in the nondeterminism monad. Finally, *ret* is once more just an embedding:
  $ret\ x = \lambda s.\, [\langle s, x\rangle]$, where $[e]$ denotes a list containing exactly one element e.

- The *continuation monad*, in which $TA = (A \to R) \to R$ for some fixed result type R, will not be described further in this thesis, since the continuation monad does not admit dynamic logic (see [34]).

2.2.2 Monads – the Abstract Way

We will now give a formal definition of what a monad is originally defined to be. Furthermore, we will give an alternative definition which is more suitable for our purposes and which comes closer to the intuitive introduction given in Section 2.2.1. Although we are not so much interested in applications of monads in category theory itself, we feel that it is reasonable to provide the original definition of a monad, as the term even appears in the title of this thesis. The following Definition 2.10 is taken from [18, Chapter VI, p. 137].

Definition 2.10. A *monad* $\mathbb{T} = (T, \eta, \mu)$ in a category $\mathbf{C}$ consists of an endofunctor $T : \mathbf{C} \to \mathbf{C}$ and two natural transformations η (called the unit) and μ (called multiplication), i. e. morphisms $\eta_A : A \to TA$ and $\mu_A : T^2A \to TA$ for each object A in $\mathbf{C}$, which make the following diagrams commute for every morphism $f : A \to B$ in $\mathbf{C}$

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[d, "f"'] & TA \arrow[d, "Tf"] \\ B \arrow[r, "\eta_B"'] & TB \end{tikzcd} \qquad \begin{tikzcd} T^2A \arrow[r, "\mu_A"] \arrow[d, "T^2f"'] & TA \arrow[d, "Tf"] \\ T^2B \arrow[r, "\mu_B"'] & TB \end{tikzcd}$$

$$\begin{tikzcd} TA \arrow[r, "\eta_{TA}"] \arrow[dr, "id_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "id_{TA}"] \\ & TA & \end{tikzcd} \qquad \begin{tikzcd} T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\ T^2A \arrow[r, "\mu_A"'] & TA \end{tikzcd}$$

where the upper two diagrams simply express the naturalness of η and μ, whereas the lower two diagrams express the required interplay of these.

How this definition can be related to the one of Section 2.2.1 can be seen after the following definition and lemma:

Definition 2.11. A *Kleisli triple* on a category $\mathbf{C}$ is a triple $(T, \eta, \_^*)$ where $T : Ob\,\mathbf{C} \to Ob\,\mathbf{C}$ is a function, η is a family of morphisms η_A for each object A in $\mathbf{C}$ and $(\_^*)$ maps each morphism $f : A \to TB$ to a morphism $f^* : TA \to TB$. The following equations are required to hold – leaving the composition operation $(\cdot)$ implicit:

$$\eta_A^* = id_{TA} \qquad f^*\eta_A = f \qquad g^* f^* = (g^* f)^* \tag{2.10}$$

The meaning of Equations (2.10) can be understood best with the help of a derived operation called *Kleisli composition* $(\circ)$ that takes morphisms $f : A \to TB$ and $g : B \to TC$ to $g \circ f := g^* f$. Formulated with this operation, Equations (2.10) state that each η_A is a left and right unit, and that composition is associative:

$$\eta_A \circ f = f = f \circ \eta_A \qquad (f \circ g) \circ h = f \circ (g \circ h) \tag{2.11}$$

Another noteworthy point is that the binding operation $(p \gg= f)$ used above can be expressed as $f^*(p)$. The polymorphic operation *ret* can obviously be identified with η of the Kleisli triple.

The following Lemma shows that one may equally well use a Kleisli triple as the defining entity for a monad. Actually, one may even prove a stronger lemma establishing a one-one correspondence between Kleisli triples and monads.

Lemma 2.12. *Every Kleisli triple $(T', \eta, \_^*)$ determines a monad $\mathbb{T} = (T, \eta, \mu)$ by taking T to be the function T' extended to an endofunctor, defining $T\,f \equiv_{\text{def}} (\eta_B\, f)^*$ for each morphism $f : A \to B$, and by setting $\mu_A := (id_{TA})^*$.*

Proof. First of all, we must validate that the proposed extension T actually constitutes a functor, i. e. we must check compatibility with identities and composition; for $f : A \to B$ and $g : B \to C$ one has

$$T\,id_A = (\eta_A\, id_A)^* = \eta_A^* = id_{TA} \tag{2.12}$$

$$T\,(g\,f) = (\eta_C\, g\, f)^* = ((\eta_C\, g)^*\, \eta_B\, f)^* = (\eta_C\, g)^*\, (\eta_B\, f)^* = T\,f\,T g \tag{2.13}$$

where in (2.13) we used the definition of T applied to morphisms, the property of η being right-cancellable, and the special kind of associativity given to $\_^*$.

The fact that η and μ actually are natural transformations can be easily calculated, so we only show that they satisfy the equalities induced by the lower two diagrams. First comes the left-hand diagram:

$$\mu_A\, \eta_{TA} = (id_{TA})^*\, \eta_{TA} = id_{TA} \tag{2.14}$$

$$\begin{aligned} \mu_A\, T\, \eta_A &= (id_{TA})^*\, (\eta_{TA}\, \eta_A)^* \\ &= ((id_{TA})^*\, \eta_{TA}\, \eta_A)^* = \eta_A^* = id_{TA} \end{aligned} \tag{2.15}$$

which proves the required equality $\mu_A\, \eta_{TA} = id_{TA} = \mu_A\, T\, \eta_A$. Finally we have to show that $\mu_A\, T\, \mu_A = \mu_A\, \mu_{TA}$, which is expressed through the right-hand diagram.

$$\begin{aligned} \mu_A\, T\, \mu_A &= \mu_A\, (\eta_{TA}\, \mu_A)^* \\ &= (id_{TA})^*\, (\eta_{TA}\, (id_{TA})^*)^* = ((id_{TA})^*\, \eta_{TA}\, (id_{TA})^*)^* \\ &= ((id_{TA})^*)^* = ((id_{TA})^*\, id_{T^2A})^* = (id_{TA})^*\, (id_{T^2A})^* \\ &= \mu_A\, \mu_{TA} \end{aligned} \tag{2.16}$$

□

2.2.3 The Meta-language for Strong Monads

The so called 'do-notation' is known from its use in Haskell, where it is deployed to make the idea of sequential evaluation of monadic programs syntactically evident. This idea is not so apparent when monadic programs are expressed through $\gg=$ and *ret*. Nonetheless, the do-notation is only syntactical sugar for conventional monadic expressions, and the former is actually reducible to the latter (see [14] for details on how this is done).

Example 2.13. The expression $\mathsf{do}\,\{x \leftarrow p; q\}$ (where q is to be regarded as a syntactical variable for a monadic program and thus may contain x as a free variable) is translated into $p \gg= \lambda x.\, q$. Another example is the expression $\mathsf{do}\,\{p; q\}$, where the return value of p is ignored; a possible translation is $p \gg= \lambda u.\, q$, where u is a fresh variable, i. e. u does not occur in q.

In the domain of categorical semantics one may look at the do-notation as being a concise language to express morphisms – i. e. the denotations of concrete *programs* – in the categories used to interpret the programming language at hand. Taken this way, the do-notation provides a formal system to reason about monads, i. e. a basically *logical* view on the semantics, as opposed to the equational or diagrammatic view of category theory. This approach has been proposed by Moggi in [20], where a formal system called *meta-language* is developed which allows the formation of terms quite similar to do-terms (Moggi used a variant of let-terms instead, but one easily translates between the two formulations).

This meta-language is defined through term formation rules in much the same way as the typed lambda calculus has been defined in Section 2.1.3, so that terms are formed in a context and rules guide the way in which terms may be built. Additionally, inference rules for establishing equalities between terms are given, such that the equivalence of programs that are described by these do-terms can be established within the formal system. The key to make this formal system an internal language for (strong) monads is to interpret it in categories equipped with a strong monad in such a way that there is a one-one correspondence between the formal system and the category[2]. The meta-language can furthermore be extended to describe categories with additional structure, e. g. one might include product terms and appropriate rules in the language to describe categories that additionally have finite products.

Remark 2.14. The term *internal language* has its origins in the domain of categorical logic. An internal language is a means to reason about a category in a way that often makes proofs easier to follow than is possible through the typical 'diagram chasing'. In essence, an internal language is to be construed as a formal system giving names to relevant entities of the category at hand. This system is then given an interpretation in the category in such a way that theorems of the internal language translate into interesting statements about the category. For a detailed overview, see [28].

The formal system for the meta-language can on the one hand be used to define morphisms in the underlying category, and on the other hand to prove equivalences between these morphisms. Thanks to a soundness and completeness theorem provided in [20], one may abandon reasoning in categories with Kleisli triples and work in an adequate extension of the meta-language instead, which adds up to reasoning about do-terms in the following way:

1. Terms are formed in a context (which we shall often omit, as long as the types of all variables are obvious or do not matter), i. e. they have the structure $\Gamma \vdash e : \tau$. It should be noted that interpretations of terms depend on the context: if $\Gamma = [x_1 : \sigma_1, \ldots, x_n : \sigma_n]$, the interpretations of types σ_i are objects c_i in the underlying category, and τ is interpreted as object c, then $\Gamma \vdash e : \tau$ will denote a morphism $c_1 \times \cdots \times c_n \to c$.

2. We are given a type constructor T that takes values of type A into computations of type TA (the interpretation of T is exactly the function T of the Kleisli triple described in Definition 2.11).

3. The polymorphic operation *ret* embeds values into computations; it is polymorphic in the sense that it exists for each producible type.

[2] a *strong monad* is one that is additionally equipped with a natural transformation $t_{A,B} : A \times TB \to T(A \times B)$, called *tensorial strength* that must obey certain conditions given in [20]

4. do-terms of the form $\mathsf{do}\,\{x \leftarrow p; q\}$ allow to simultaneously express binding and sequencing, where x is a variable that gets bound to the resulting value of the computation p, and q is a computation which may contain x.

5. The notion of associativity of binding is reflected by the following equality between do-terms: for every program r not containing x, one has

$$(\mathsf{do}\,\{y \leftarrow \mathsf{do}\,\{x \leftarrow p; q\}; r\}) = (\mathsf{do}\,\{x \leftarrow p; \mathsf{do}\,\{y \leftarrow q\}; r\})$$

For notational clarity, repeated do-terms are abbreviated: Write $\mathsf{do}\,\{x_1 \leftarrow p_1; x_2 \leftarrow p_2; \ldots\}$ for $\mathsf{do}\,\{x_1 \leftarrow p_1; \mathsf{do}\,\{x_2 \leftarrow p_2; \ldots\}\}$

6. Corresponding to the properties of η, one has unit laws for *ret* (which actually is interpreted as η) in the following way

$$\begin{aligned}\mathsf{do}\,\{x \leftarrow p; \mathit{ret}\ x\} &= p\\ \mathsf{do}\,\{x \leftarrow \mathit{ret}\ a; p\} &= p[a/x]\end{aligned}$$

7. There are rules about equality, namely reflexivity, symmetry and transitivity, as well as a rule for substitution, stating that if an equation between terms $e_1 = e_2$ containing a variable x can be derived, then so can the equality $e_1[e/x] = e_2[e/x]$ for each well-formed term e not containing free variables that do not occur in e_1 or e_2.

As a final word on the meta-language, it should be pointed out that it is an equational theory. It therefore presents an instrument to prove equivalences between programs, i. e. an equality of the morphisms they denote in the interpretation. The logic that will be developed in the sequel goes far beyond the ability of proving equivalences. In monadic dynamic logic, it is possible to make much more specific statements about programs, e. g. one can specify under what conditions a program will terminate or one can prove that a given program in the state monad will modify the state in a certain way.

3 Monadic Dynamic Logic

In this chapter, the proof calculus of monadic dynamic logic is presented. First, properties of monadic programs are introduced that will be needed later on in order to develop the calculus; these include notions such as discardability and side effect freeness of programs. After that, the modal operators of dynamic logic are introduced in an axiomatic way and their meaning in the example monads of Section 2.2 is explained. All prerequisites gathered together, the monad-independent proof calculus for dynamic logic is described in Section 3.3. Finally, an extension of the calculus that is tailored towards the exception monad is developed.

In what follows the type of truth values will be denoted by Ω, and the entire formalisation is suited for an intuitionistic as well as a classical framework. T will denote the type constructor mapping a type of *values* into the type of *computations* or *programs* over these values. Formulae of dynamic logic will be taken to be terms of type $D\Omega$, where, for each A, DA is the subtype of TA of all deterministically side effect free programs, a notion depicted below. As a primary feature of the calculus, there will be modal operators $[x \leftarrow p]\,\_$ and $\langle x \leftarrow p\rangle\,\_$ for each program p that take a formula of dynamic logic ϕ (possibly containing x as a free variable) to another formula which may state properties of x being the result of executing p. The modal operators thus act as new variable binders; because we also allow program sequences to occur inside the operators – as in $[x \leftarrow p; y \leftarrow q]$ – they may bind several variables at once. An initial intuitive understanding of the box and diamond operators can be most easily given in the nondeterminism monad, where the formula $[x \leftarrow p]\,(x = 1)$ should be interpreted as "after executing p and binding the result to x, $(x = 1)$ will hold *for all* possible outcomes of p". On the other hand, $\langle x \leftarrow p\rangle\,(x = 1)$ states that there will be *some* result x of p such that $(x = 1)$ is true.

3.1 Preliminaries

The possibility for program sequences to occur inside the box and diamond operators instead of single bindings should be regarded as a mere notational convenience. That this does not add to the expressiveness of the operators can be seen by translating multiple bindings into bindings of tuples, e. g. the bound variables x and y in $[x \leftarrow p; y \leftarrow q]$ can be packaged into the single variable $z = (x, y)$ in $[z \leftarrow \mathsf{do}\,\{x \leftarrow p; y \leftarrow q; ret\ (x, y)\}]$. Horizontal bars above variables will indicate that actually a non-empty program sequence is under consideration rather than a single binding. Let $\bar{x} = [x_1, \dots, x_n]$ and $\bar{p} = [p_1, \dots, p_n]$; then $\bar{x} \leftarrow \bar{p}$ will denote the program $\mathsf{do}\,\{x_1 \leftarrow p_1; \cdots; x_n \leftarrow p_n; ret\ (x_1, \dots, x_n)\}$ or, if it appears inside a do-statement or a box or diamond operator, just the binding sequence $x_1 \leftarrow p_1; \cdots; x_n \leftarrow p_n$.

3.1.1 Properties of Monadic Programs

The property of a monadic program being deterministically side effect free (abbreviated to *dsef* in the following) relies on some simpler properties that will now be defined. The main idea behind the introduction of a subtype DA of dsef programs is that these programs have properties allowing them to be rearranged quite freely within a monadic program sequence. For example, if p and q are both dsef programs, the programs $\mathsf{do}\,\{x \leftarrow p; y \leftarrow q; r\}$ and $\mathsf{do}\,\{y \leftarrow q; x \leftarrow p; r\}$ will be equal for every program r (possibly containing x and y, in contrast to p and q, which may not mention them). This is an important fact when introducing connectives for formulae of dynamic logic: intuitively, $\phi : D\Omega$ and $\phi \wedge \phi : D\Omega$ should be regarded as equivalent formulae, but if ϕ has side effects or is nondeterministic, this equivalence might break down. Taking only terms of type $D\Omega$ as formulae makes sure such equivalences are retained in the calculus.

A more elaborate account of the information provided in this section can be found in [34], where virtually all lemmas and propositions stated here were proved. To avoid overly repeating facts already stated elsewhere, only the most important lemmas and some of their proofs are given here. Independently established proofs can be found in Section 3.4 covering extensions specific to the exception monad as well as in the chapters on application (Chapter 4) and implementation (Chapter 6) of the calculus.

Definition 3.1. Let 1 be the unit type and $*$ the single element of this type. A program $p : TA$ is called *discardable* if

$$\mathsf{do}\,\{x \leftarrow p; \mathit{ret}\ *\} = \mathit{ret}\ *$$

- In the state monad, p is discardable if it terminates and does not alter the state; since its result is not used in the remainder of the program on the left-hand side (i. e. in $\mathit{ret}\ *$), it might just as well be omitted altogether.

- In the nondeterminism monad, p is discardable if it yields at least one result; in that case, both sides of the equation yield $\{*\}$.

- The concept of discardability reveals differences between the list monad and the nondeterminism monad: in the list monad, p is discardable if it yields exactly one result, in which case both sides of the equation equal the singleton list $[*]$ – which may well be distinguished from the list $[*,*]$ containing the same element twice.

Definition 3.2. Let $p : TA$ be a program. p is called *stateless* if it is of the form $p = \mathit{ret}\ a$ for some $a : A$. Obviously, all stateless programs are discardable which follows immediately from the basic monad laws.

The following lemma confirms the appropriateness of Definition 3.1 for indicating when a program may be discarded at the head of an arbitrary program sequence:

Lemma 3.3. *Let $p : TA$ be discardable and $q : TB$ be an arbitrary program. Then*

$$\mathsf{do}\,\{p; q\} = q$$

Most proofs of the propositions in this section are by equational reasoning; here is an example proof of the above lemma.

Proof.

$$\begin{aligned}
\mathsf{do}\,\{p; q\} &= \mathsf{do}\,\{p; \mathit{ret}\ *; q\} && \text{(since } \mathsf{do}\,\{\mathit{ret}\ *; q\} = q\text{)}\\
&= \mathsf{do}\,\{\mathit{ret}\ *; q\} && \text{(} p \text{ discardable)}\\
&= q
\end{aligned}$$

$\square$

While discardability allows one to omit certain programs altogether whose return value is not used in the remainder, the following concept admits statements about the behaviour of certain programs when they are executed repeatedly:

Definition 3.4. Let $p : TA$ be a program. p is called *copyable* if the following equation holds:

$$\mathsf{do}\,\{x \leftarrow p; y \leftarrow p; ret\ (x,y)\} = \mathsf{do}\,\{x \leftarrow p; ret\ (x,x)\}$$

As with discardability, copyability entails a stronger form of program equality which expresses the fact that copyable programs may be doubled (or cancelled, taken the opposite way) without effect more directly:

Proposition 3.5. *Let $p : TA$ be a copyable and $r : TB$ be an arbitrary program possibly containing y as a free variable. Then one has*

$$\mathsf{do}\,\{x \leftarrow p; y \leftarrow p; r\} = \mathsf{do}\,\{x \leftarrow p; r[x/y]\}$$

For various monads, the deterministically side effect free programs comprise the copyable and discardable programs. That this type is not empty can easily be seen by considering stateless programs of the form *ret a*, which are discardable and copyable at any rate. These programs are also deterministically side effect free in the general sense, which depends on one further concept.

Definition 3.6. Let p and q be copyable and discardable programs with $x \notin FV(q)$ and $y \notin FV(p)$. Then *p commutes with q* if the following three equivalent conditions hold:

$$\mathsf{do}\,\{x \leftarrow p; y \leftarrow q; ret\ (x,y)\} \text{ is a copyable program} \tag{3.1}$$
$$\mathsf{do}\,\{x \leftarrow p; y \leftarrow q; ret\ (x,y)\} = \mathsf{do}\,\{y \leftarrow q; x \leftarrow p; ret\ (x,y)\} \tag{3.2}$$
$$\mathsf{do}\,\{x \leftarrow p; y \leftarrow q; r\} = \mathsf{do}\,\{y \leftarrow q; x \leftarrow p; r\} \tag{3.3}$$

Definition 3.7. A copyable and discardable program p that commutes with *all* copyable and discardable programs is called *deterministically side effect free* (dsef).

Proposition 3.8. *Dsef programs are stable under sequential composition, i. e. for every sequence $\bar{x} \leftarrow \bar{p}$ of dsef programs and every dsef program q, the program $\mathsf{do}\,\{\bar{x} \leftarrow \bar{p}; q\}$ is also dsef.*

Remark 3.9. As a rather technical aside, Isabelle imposes the restriction of quantifying not over all programs of all possible types, but merely over all programs of a fixed type. Fortunately, a program already commutes with all discardable and copyable programs if it commutes with all such programs of type $T\Omega$. Therefore, the property of a program p being dsef can also be expressed with more stringent type constraints in Isabelle.

3.1.2 Global Dynamic Judgements

Before introducing logical operators for dsef terms (viewed as formulae), we clarify when such a formula is to be regarded as valid. Contending the global validity of a term of type $T\Omega$ (notation $\boxdot\phi$)[1] amounts to saying that

$$\phi = \mathsf{do}\,\{a \leftarrow \phi; ret\ \top\}$$

[1]note that the type $T\Omega$ indicates that global validity is also defined for 'formulae' with side effects

i. e. basically ϕ evaluates to truth ($\top$) if it yields any results at all. In the state monad (resp. the exception monad), this equation also holds if ϕ is undefined (resp. throws an exception), while in the nondeterminism monad it also holds if ϕ does not produce any results at all. For the important special case when ϕ is discardable, $\boxed{G}\phi$ reduces to $\phi = \mathit{ret}\ \top$.

Although global validity is a sufficiently strong concept to express when a term of type $T\Omega$ is to be considered valid, there are monads (albeit rather exotic ones such as the free Abelian group monad, see [34, Section 3]) for which it is too weak to give a semantics to the box and diamond operators. For this to be possible in a most general manner, the similar but more powerful notion of a *global dynamic judgement* is necessary: let $[\overline{x} \leftarrow \overline{p}]_{\mathbf{G}}\,\phi$ abbreviate

$$\mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; \mathit{ret}\ (\overline{x}, \phi)\} = \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; \mathit{ret}\ (\overline{x}, \top)\}$$

(note that $\phi : \Omega$ in $[x \leftarrow p]_{\mathbf{G}}\,\phi$, i. e. ϕ is an actual formula, whereas $\psi : T\Omega$ is a monadic term in $\boxed{G}\psi$).

Remark 3.10. The monads that serve as examples in this thesis have been called *simple* in [34]; in simple monads the equivalence of the two statements $\boxed{G}(\mathsf{do}\,\{x \leftarrow p; \mathit{ret}\ \phi\})$ and $[x \leftarrow p]_{\mathbf{G}}\,\phi$ holds, such that one of these concepts would actually be sufficient.

The following lemma once more shows that a more general statement (in this case about global dynamic judgements) drops out of an apparently primitive definition.

Lemma 3.11. *If $[\overline{x} \leftarrow \overline{p}]_{\mathbf{G}}\,\phi$ holds,*

$$\mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; q[\phi/y]\} = \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; q[\top/y]\}$$

for each program q containing $y : \Omega$ as a free variable.

Proof. Again, by a direct calculation: let π_i denote the i-th projection function and let θ be the substitution $\{(\pi_1 \cdot \pi_1)z/x_1, \ldots, (\pi_n \cdot \pi_1)z/x_n, \pi_2 z/y\}$ which replaces x_i and y by their respective selection from the tuple $z = ((x_1, \ldots, x_n), y)$. Then

$$\begin{aligned}
\mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; q[\phi/y]\} &= \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; y \leftarrow \mathit{ret}\ \phi; q\} \\
&= \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; y \leftarrow \mathit{ret}\ \phi; z \leftarrow \mathit{ret}\ (\overline{x}, y); (q)\theta\} \\
&= \mathsf{do}\,\{z \leftarrow \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; y \leftarrow \mathit{ret}\ \phi; \mathit{ret}\ (\overline{x}, y)\}; (q)\theta\} \\
&= \mathsf{do}\,\{z \leftarrow \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; \mathit{ret}\ (\overline{x}, \phi)\}; (q)\theta\} \\
&= \mathsf{do}\,\{z \leftarrow \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; \mathit{ret}\ (\overline{x}, \top)\}; (q)\theta\} && (\star) \\
&= \mathsf{do}\,\{z \leftarrow \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; y \leftarrow \mathit{ret}\ \top; \mathit{ret}\ (\overline{x}, y)\}; (q)\theta\} \\
&= \ldots = \mathsf{do}\,\{\overline{x} \leftarrow \overline{p}; q[\top/y]\}
\end{aligned}$$

where to arrive at $(\star)$ the assumption $[\overline{x} \leftarrow \overline{p}]_{\mathbf{G}}\,\phi$ has been used. □

Corollary 3.12. *One has $[a \leftarrow \phi]_{\mathbf{G}}\,a$ if and only if $\boxed{G}\phi$. The implication from left to right is a direct consequence of Lemma 3.11, recalling that $\phi = \mathsf{do}\,\{a \leftarrow \phi; \mathit{ret}\ a\}$, whereas the implication from right to left is, again, a manipulation with the help of unit and associativity laws of monads.*

We will not devote ourselves to developing an entire calculus of global dynamic judgements – which indeed already is expressive enough to formulate a Hoare calculus for partial correctness with it, as has been done in [32] – but rather make use of it to define the modal operators of monadic dynamic logic. Global dynamic judgements are also useful to formalise what it means for a program to terminate:

Definition 3.13. A program p *terminates* if

$$[\bar{x} \leftarrow \bar{q}; p]_{\mathbf{G}}\, \phi \quad \text{implies} \quad [\bar{x} \leftarrow \bar{q}]_{\mathbf{G}}\, \phi$$

for each program sequence $\bar{x} \leftarrow \bar{q}$ and each formula $\phi : \Omega$.

Example 3.14. Obviously, $\phi : \Omega$ in $[\bar{x} \leftarrow \bar{q}; p]_{\mathbf{G}}\, \phi$ cannot mention the result of p since this result is not bound. To see how the above definition accords with the intuitive understanding of termination, consider the simplest possible exception monad where $TA = A + \{\bot\}$. In this setting, it is reasonable to talk of nontermination of a program p if it throws an exception (i. e. $p = \bot$). In this case $[\bar{x} \leftarrow \bar{q}; p]_{\mathbf{G}} \bot$ will be true for every program sequence $\bar{x} \leftarrow \bar{q}$ since $\mathsf{do}\,\{\bar{x} \leftarrow \bar{q}; p; \mathit{ret}\ (\bar{x}, \bot)\} = \bot = \mathsf{do}\,\{x \leftarrow q; p; \mathit{ret}\ (\bar{x}, \top)\}$ (recall the definition of binding in the exception monad). $[\bar{x} \leftarrow \bar{q}]_{\mathbf{G}} \bot$ will however be false for every program sequence $\bar{x} \leftarrow \bar{q}$ not throwing any exceptions.

Remark 3.15. Reasoning about termination in the state monad (recall that here TA takes the form $S \to S \times A$, i. e. a function space of *total* functions) only makes sense if either *partial* functions are considered or the theory of *complete partial orders* (cpos) with *continuous functions* between them is employed. In a setting where programs are partial functions $f : S \rightharpoonup S \times A$, one finds that the above definition of termination precisely identifies the terminating programs with the total functions – given an adapted definition of binding that takes the possibility of undefinedness of programs into account. Since in Isabelle/HOL every function is implicitly total, we also stick to this principle in the overall development, explicitly indicating when more structure is necessary, e. g. in the definition of arbitrarily recursive definitions like that of a while-loop.

The greater freedom in treatment that dsef programs are characterised by also shows up when they appear in global dynamic judgements. Several properties such as the equivalence of $[\bar{w} \leftarrow \bar{q}; x \leftarrow p; y \leftarrow p; \bar{z} \leftarrow \bar{r}]_{\mathbf{G}}\, \phi$ and $[\bar{w} \leftarrow \bar{q}; x \leftarrow p; \bar{z} \leftarrow \bar{r}[x/y]]_{\mathbf{G}}\, \phi[x/y]$ can be proved for a dsef program p. This leads to three notational conventions that allow one to use dsef programs in places where actual values are expected, and vice versa. Put concretely, we allow

1. a dsef program $p : DA$ to occur in places where a variable $x : A$ is expected; the program $q[p/x]$ decodes into $\mathsf{do}\,\{x \leftarrow p; q\}$.

2. a formula $\psi : D\Omega$ to occur in places where a genuine formula $a : \Omega$ can appear in global dynamic judgements; the judgement $[\bar{x} \leftarrow \bar{p}]_{\mathbf{G}}\, \phi[\psi/a]$ decodes into $[\bar{x} \leftarrow \bar{p}; a \leftarrow \psi]_{\mathbf{G}}\, \phi$. Note that here the evaluation of ψ takes place *after* having evaluated $\bar{x} \leftarrow \bar{p}$, whereas in (1.) p is evaluated *before* q.

3. formulae of type Ω to be inserted in places where actually a formula of type $D\Omega$ is expected, since the former type can easily be cast to the latter through *ret*. This is convenient if several stateless formulae are involved which, for instance, make statements about the data on which a program is supposed to work.

The specification of the tree-search algorithm in Section 4.4 is an example where this convention is employed. Compare also with Remark 6.2 on how this convention is handled in Isabelle.

3.2 Logical Operators

3.2.1 Primitive Connectives

The logical operators are defined in terms of already existing logical operators for the type of truth values Ω. So we assume that the background formalism at least allows the formulation of the standard propositional connectives; this is certainly the case for Isabelle/HOL which even allows the formulation of higher-order functions and predicates. We will use the same symbols for both actual formulae of type Ω as well as *formulae of dynamic logic* of type $D\Omega$; it will be clear from the context which of them is meant. Let *op* stand for conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$ or equivalence $\iff$ of two formulae of dynamic logic $\phi, \psi : D\Omega$ respectively. Then these connectives are defined as

$$\phi \; op \; \psi \quad \equiv_{\mathrm{def}} \quad \mathsf{do}\,\{a \leftarrow \phi; b \leftarrow \psi; ret\ (a\ op\ b)\} \tag{3.4}$$

Negation is of course similarly defined as

$$\neg\phi \quad \equiv_{\mathrm{def}} \quad \mathsf{do}\,\{a \leftarrow \phi; ret\ (\neg a)\}$$

First-order operators like a universal quantifier are not available for formulae of dynamic logic; they may however appear in *stateless formulae*, e. g. of the form $ret\ (\forall x.\, P(x))$, if the underlying formalism allows their formulation for formulae of type Ω. An example thereof can be found in Chapter 4 within the specification of a breadth-first search algorithm.

Asserting the validity of a formula of dynamic logic can be done in two equivalent ways, due to the existence of two different notations and their relation to each other. The 'global box' $\boxed{G}$ basically serves the purpose of asserting validity of a formula: $\boxed{G}(\phi \wedge \psi)$ decodes into $\boxed{G}(\mathsf{do}\,\{a \leftarrow \phi; b \leftarrow \psi; ret\ (a \wedge b)\})$ according to the definition of conjunction and is to be read as "it is globally true that $\phi \wedge \psi$ holds". By Corollary 3.12, an equivalent formulation is to say that $[a \leftarrow \phi; b \leftarrow \psi]_{\mathbf{G}}\,(a \wedge b)$ holds. It is important to note that all propositional tautologies carry over into the calculus of monadic dynamic logic: $\phi \Rightarrow (\psi \Rightarrow \phi)$ is globally valid, since global validity amounts to $[a \leftarrow \phi; b \leftarrow \psi; c \leftarrow \phi]_{\mathbf{G}}\,(a \Rightarrow (b \Rightarrow c))$ being valid. The latter judgement is valid because by Lemma 3.5 it is equivalent to $[a \leftarrow \phi; b \leftarrow \psi]_{\mathbf{G}}\,(a \Rightarrow (b \Rightarrow a))$ in which $a \Rightarrow (b \Rightarrow a)$ is a tautology (in Ω), thus equal to $\top$.

3.2.2 Boxes and Diamonds

The key feature of monadic dynamic logic is the existence of modal operators that allow building *formulae* (i. e. terms of type $D\Omega$) stating that after execution of a program some condition will necessarily or possibly hold. This is in contrast to the global box $\boxed{G}$ and the global dynamic judgements which, as the name suggests, merely allow the formulation of *global* statements about program sequences and properties of their bound variables. The semantics of the diamond and box operators $[x \leftarrow p]\,\phi$ and $\langle x \leftarrow p\rangle\,\phi$ is *local* in the sense that the state in which ϕ is evaluated may be modified by p, but the entire formula does not modify the state in which itself is evaluated. Hence, it may appear as a sub-formula without affecting the semantics of surrounding sub-formulae.

Example 3.16. The axiomatic introduction of the box and diamond operators given below does not quite point to an idea of what they intuitively express. We therefore give their intended interpretation for the monads described in Section 2.2 as a motivation for their usefulness.

- In the state monad of total functions $[x \leftarrow p]\,\phi$ and $\langle x \leftarrow p\rangle\,\phi$ depend on the state. They denote the same formula which is true in a state s if after execution of p the result x will satisfy ϕ. If partial functions are involved $[x \leftarrow p]\,\phi$ is actually weaker than $\langle x \leftarrow p\rangle\,\phi$ in that the former is also true if p is undefined.
- In the exception monad $[x \leftarrow p]\,\phi$ holds if p throws an exception or yields a value satisfying ϕ, whereas for $\langle x \leftarrow p\rangle\,\phi$ to hold it is additionally required that p does not throw an exception.
- In the nondeterminism monad, where $p : TA$ is a set of elements of A, $[x \leftarrow p]\,\phi$ holds if all elements in p satisfy ϕ (which also includes the case where $p = \emptyset$) and $\langle x \leftarrow p\rangle\,\phi$ is true if and only if p contains some value satisfying ϕ.
- Finally, in the combination of the list monad and the state monad the modal operators depend on the state as well. Validity of $[x \leftarrow p]\,\phi$ (or $\langle x \leftarrow p\rangle\,\phi$) in a state s means that all outcomes of p satisfy ϕ (or at least one outcome satisfies ϕ).

The following definition formalises the essential requirement that a monad must satisfy in order to allow the interpretation of monadic dynamic logic. The somehow dual operators $[x \leftarrow p]\,\_$ and $\langle x \leftarrow p\rangle\,\_$ are introduced independently of each other in order to make their particular interpretation possible in intuitionistic logics as well. In a classical setting, one might define $\langle x \leftarrow p\rangle\,\phi$ as $\neg[x \leftarrow p]\,\neg\phi$, and in fact this equivalence is shown to hold in Isabelle later on.

Definition 3.17. A monad *admits dynamic logic* if there exist formulae $[\overline{y} \leftarrow \overline{q}]\,\phi$ and $\langle \overline{y} \leftarrow \overline{q}\rangle\,\phi$ for each program sequence $\overline{y} \leftarrow \overline{q}$ and each formula $\phi : D\Omega$ such that for each program sequence $\overline{x} \leftarrow \overline{p} = x_1 \leftarrow p_1; \ldots; x_n \leftarrow p_n$ containing $x_i : \Omega$ $(1 \leq i \leq n)$ the following equivalences hold:

$$[\overline{x} \leftarrow \overline{p}]_{\mathbf{G}}\,(x_i \Rightarrow [\overline{y} \leftarrow \overline{q}]\,\phi) \iff [\overline{x} \leftarrow \overline{p}; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(x_i \Rightarrow \phi)$$
$$[\overline{x} \leftarrow \overline{p}]_{\mathbf{G}}\,(\langle \overline{y} \leftarrow \overline{q}\rangle\,\phi \Rightarrow x_i) \iff [\overline{x} \leftarrow \overline{p}; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(\phi \Rightarrow x_i)$$

The purpose of using the variable x_i is generality: one can express every formula ψ in context of the other x_j through it: simply put $x_i = x_n$ and $p_n = \mathit{ret}\ \psi$. Note also that the above equivalences make use of the notational convention of letting formulae of monadic logic appear where a formula of type Ω is expected: in decoded form the first equivalence reads as

$$[\overline{x} \leftarrow \overline{p}; a \leftarrow [\overline{y} \leftarrow \overline{q}]\,\phi]_{\mathbf{G}}\,(x_i \Rightarrow a) \iff [\overline{x} \leftarrow \overline{p}; \overline{y} \leftarrow \overline{q}; b \leftarrow \phi]_{\mathbf{G}}\,(x_i \Rightarrow b)$$

and similar for the second one.

We state some basic properties that accompany the box and diamond operators.

Proposition 3.18 (Unique determination). *One can turn the type of dsef programs $D\Omega$ into a partial order by setting $\phi \leq \chi$ if and only if $\phi \Rightarrow \chi$. Then $[\overline{y} \leftarrow \overline{q}]\,\phi$ is the greatest formula ψ such that $[a \leftarrow \psi; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(a \Rightarrow \phi)$ and $\langle \overline{y} \leftarrow \overline{q}\rangle\,\phi$ is the smallest formula ψ such that $[a \leftarrow \psi; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(\phi \Rightarrow a)$.*

A proof of this proposition involves two steps: first, it has to be shown that for each formula ψ satisfying $[a \leftarrow \psi; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(a \Rightarrow \phi)$ (or $[a \leftarrow \psi; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,(\phi \Rightarrow a)$) one has $\psi \Rightarrow [\overline{y} \leftarrow \overline{q}]\,\phi$ (or $\langle \overline{y} \leftarrow \overline{q}\rangle\,\phi \Rightarrow \psi$). Second, it must be shown that $[\overline{y} \leftarrow \overline{q}]\,\phi$ ($\langle \overline{y} \leftarrow \overline{q}\rangle\,\phi$) in fact satisfy the judgements. Both parts of the proof follow more or less immediately from the definition of the box and diamond operators.

Proposition 3.19 (Global validity of box formulas). *Let $\bar{y} \leftarrow \bar{q}$ be an arbitrary program sequence and $\phi : D\Omega$ a formula. Then $\boxed{G}([\bar{y} \leftarrow \bar{q}]\,\phi)$ is equivalent to $[\bar{y} \leftarrow \bar{q}]_{\mathbf{G}}\,\phi$.*

The following equivalence allows us to reason about termination within the calculus of monadic dynamic logic without having to fall back to reasoning about global dynamic judgements.

Proposition 3.20 (Termination). *A program p terminates in the sense of Definition 3.13 if and only if $\langle p \rangle \top$ holds.*

Defining the Modal Operators

In monads with additional structure that besides imposing some minor logical well-behavedness basically allows one to 'read the current state' – a property which virtually all of the running example monads possess – it is possible to directly define the box operator[2]. This definition is shown now as it enlightens the particular locality of the box operator's semantics.

The general idea is that dsef programs can essentially be regarded as programs that may read the 'state', but not alter it, i. e. there is an isomorphism between the type DA of dsef programs over A and the function space $F \to A$, where F is the type of states (see below). With the help of this isomorphism, one may describe the box operator $[x \leftarrow p]\,\phi$ as a function that maps the current state to a global dynamic judgement (hence, a formula of type Ω) asserting that after setting this state and executing p, the formula ϕ will be true. We need some definitions to make these ideas precise. The notion of state has to be abstracted from the set of concrete state values S in the state monad to a concept that also makes sense in other monads.

Definition 3.21. A *state* is a terminating program $s : T1$ such that for each dsef program $p : DA$ there exists an element $a : A$ such that

$$\mathsf{do}\,\{s; p\} = \mathsf{do}\,\{s; ret\ a\}$$

If for each terminating program q one furthermore has

$$s = \mathsf{do}\,\{q; s\}$$

then s is called a *forcible state*. The subtype of $T1$ of all forcible states is denoted by F.

In the state monad, a state as just defined would rather be thought of as an update operation: the function $update\ s' = \lambda s : S.\,(s', *)$ of Section 2.2.1 yields a state when it is applied to an element of the set of concrete states S. In the exception and nondeterminism monads there is only the trivial state $ret\ *$, which in both cases is forcible; the special kind of list monad we have described does not have forcible states: its states take the form $s_c = \lambda i : List\ I.\,[(c, *)]$ for $c : List\ I$, but for the program $q = \lambda i : List\ I.\,[(a, *), (b, *)]$ one has

$$\mathsf{do}\,\{q; s_c\} = \lambda i : List\ I.\,[(j, *), (j, *)] \neq s_c$$

The basic problem is that an element can occur multiple times in lists, in contrast to sets so that forcibility is only available when the latter are used, e. g. in the nondeterminism monad.

[2] even in the intuitionistic case the diamond operator can then be defined in terms of the box operator, albeit in a rather contrived way that we will not present here

For the definition of the box operator we need a further operation that allows one to extract the state. It is determined by the property that accessing the state with the help of it and then immediately executing this state has no effect (since the state will be the same afterwards as beforehand):

Definition 3.22. A program $d : DF$ is called a *state discloser* if the term $\mathsf{do}\,\{x \leftarrow d; x\}$ is discardable.

It is now possible to establish that $DA \cong (F \to A)$ by defining two isomorphisms $\kappa_A : DA \to (F \to A)$ and its inverse $\kappa_A^{-1} : (F \to A) \to DA$ for each type A (the index A will be omitted in the following). While to be able to define κ one needs a Hilbert description operator, its inverse κ_A^{-1} can be defined purely by means already available. Let $d : DF$ be a state discloser, then for each function $f : F \to A$ the program $\kappa^{-1}(f)$ accesses the current state and applies f to it, i. e. one has

$$\kappa^{-1}(f) \quad \equiv_{\text{def}} \quad \mathsf{do}\,\{s \leftarrow d; ret\ (f\ s)\}$$

which is a dsef program of type DA, recalling that both d and ret are dsef programs. This mapping allows us to describe the box operator as a function in $F \to \Omega$, i. e. as a state dependent truth value, and then subsequently inject it into DA: $[\overline{y} \leftarrow \overline{q}]\,\phi$ can be interpreted in $F \to \Omega$ as a function that returns the global validity of ϕ after executing the state s followed by $\overline{y} \leftarrow \overline{q}$. This is formalised by the following definition of the box operator:

$$[\overline{y} \leftarrow \overline{q}]\,\phi \quad \equiv_{\text{def}} \quad \kappa^{-1}(\lambda s : F.\, [s; \overline{y} \leftarrow \overline{q}]_{\mathbf{G}}\,\phi)$$

3.3 The Monad-independent Proof Calculus

The entire proof calculus for monadic dynamic logic can be formalised by adding the rules and axioms of Figure 3.1 to the set of propositional tautologies in $D\Omega$. Certainly the inclusion of *all* tautologies is overkill which might be prevented by only including an independent and complete set of axioms for propositional logic[3], but here we are mainly concerned with rules and axioms for the modal operators. The soundness of the calculus has been established in [34], whereas its completeness is still an open issue.

The side condition '$\overline{x}$ not free in assumptions' in the necessitation rule is a typical side condition analogous to the one for the universal quantifier in first-order logic; the term *assumptions* is to be understood as it is used in natural deduction and does not refer to the *premiss* of the rule, ϕ. The axioms K3$\Box$ and K3$\Diamond$ refer to stateless formulae that are mere injections of formulae $\phi : \Omega$. The first one expresses the fact that stateless formulae continue to hold after execution of programs (whereas the inverse is not true due to possible nontermination of the program p), and the second one expresses the fact that stateless formulae that hold after terminating executions of p also hold unconditionally. The sequencing axioms seq$\Box$ and seq$\Diamond$ allow one to freely split and join boxes and diamonds.

Essentially the K axioms are the intuitionistic counterpart to the usual K axiom of classical modal logic, which is called $K1$ here (see [35]). Further K axioms are however necessary to be able to prove intuitionistically valid formulae. This is mainly due to the fact that the box and diamond operators are defined independently of each other. It will be seen in Chapter 6 that the implementation of the calculus in Isabelle behaves classically, so that in it the classical equivalence of $\langle x \leftarrow p \rangle\, P$ and $\neg [x \leftarrow p]\, \neg P$ can be shown.

[3] complete in the sense that every tautology can be proved from these axioms together with modus ponens

Rules:

$$(\textbf{nec})\ \frac{\phi}{[\bar{x}\leftarrow\bar{p}]\,\phi}\ \ \begin{array}{c}\bar{x}\text{ not free}\\ \text{in assumptions}\end{array} \qquad (\textbf{mp})\ \frac{\phi\Rightarrow\psi;\quad\phi}{\psi}$$

Axioms:

$$\begin{array}{lll}
(\mathrm{K1}) & [\bar{x}\leftarrow\bar{p}]\,(\phi\Rightarrow\psi)\Rightarrow[\bar{x}\leftarrow\bar{p}]\,\phi\Rightarrow[\bar{x}\leftarrow\bar{p}]\,\psi & \\
(\mathrm{K2}) & [\bar{x}\leftarrow\bar{p}]\,(\phi\Rightarrow\psi)\Rightarrow\langle\bar{x}\leftarrow\bar{p}\rangle\,\phi\Rightarrow\langle\bar{x}\leftarrow\bar{p}\rangle\,\psi & \\
(\mathrm{K3}\Box) & \mathit{ret}\ \phi\Rightarrow[p]\,\mathit{ret}\ \phi & \\
(\mathrm{K3}\Diamond) & \langle p\rangle\,\mathit{ret}\ \phi\Rightarrow\mathit{ret}\ \phi & \\
(\mathrm{K4}) & \langle\bar{x}\leftarrow\bar{p}\rangle\,(\phi\vee\psi)\Rightarrow(\langle\bar{x}\leftarrow\bar{p}\rangle\,\phi\vee\langle\bar{x}\leftarrow\bar{p}\rangle\,\psi) & \\
(\mathrm{K5}) & (\langle\bar{x}\leftarrow\bar{p}\rangle\,\phi\Rightarrow[\bar{x}\leftarrow\bar{p}]\,\psi)\Rightarrow[\bar{x}\leftarrow\bar{p}]\,(\phi\Rightarrow\psi) & \\
(\mathrm{seq}\Box) & [\bar{x}\leftarrow\bar{p};y\leftarrow q]\,\phi\iff[\bar{x}\leftarrow\bar{p}]\,[y\leftarrow q]\,\phi & \\
(\mathrm{seq}\Diamond) & \langle\bar{x}\leftarrow\bar{p};y\leftarrow q\rangle\,\phi\iff\langle\bar{x}\leftarrow\bar{p}\rangle\,\langle y\leftarrow q\rangle\,\phi & \\
(\mathrm{ctr}\Box) & [x\leftarrow p;y\leftarrow q]\,\phi\Rightarrow[y\leftarrow\mathsf{do}\,\{x\leftarrow p;q\}]\,\phi & (x\notin FV(\phi)) \\
(\mathrm{ctr}\Diamond) & \langle x\leftarrow p;y\leftarrow q\rangle\,\phi\Leftarrow\langle y\leftarrow\mathsf{do}\,\{x\leftarrow p;q\}\rangle\,\phi & (x\notin FV(\phi)) \\
(\mathrm{ret}\Box) & [x\leftarrow\mathit{ret}\ a]\,\phi\iff\phi[a/x] & \\
(\mathrm{ret}\Diamond) & \langle x\leftarrow\mathit{ret}\ a\rangle\,\phi\iff\phi[a/x] & \\
(\mathrm{dsef}\Box) & [x\leftarrow p]\,P\iff P[p/x] & (p\text{ is }\mathit{dsef}) \\
(\mathrm{dsef}\Diamond) & \langle x\leftarrow p\rangle\,P\iff P[p/x] & (p\text{ is }\mathit{dsef})
\end{array}$$

Figure 3.1: The generic proof calculus of monadic dynamic logic

Two further axioms that are needed in Chapter 6 can only be proved in so called *logically regular monads* (cf. [34, Def. 5.14]). Essentially, logical regularity means that arbitrary formulae $c:\Omega$ implying some global dynamic judgement can be moved into the scope of that judgement, as follows

$$c\Rightarrow[x\leftarrow p]_{\mathbf{G}}\,\phi \quad \mathit{implies} \quad [x\leftarrow p]_{\mathbf{G}}\,(c\Rightarrow\phi)$$

This restriction is only necessary in the intuitionistic case; if the underlying logic is classical one can show that all monads are logically regular. Even in the intuitionistic case all monads that are under consideration here are logically regular. The axioms allow one to substitute equals for equals inside boxes and diamonds:

Axioms:

$$\begin{array}{ll}
(\mathrm{eq}\Box) & p=q\Rightarrow[x\leftarrow p]\,\phi\Rightarrow[x\leftarrow q]\,\phi \\
(\mathrm{eq}\Diamond) & p=q\Rightarrow\langle x\leftarrow p\rangle\,\phi\Rightarrow\langle x\leftarrow q\rangle\,\phi
\end{array}$$

3.3.1 Hoare Calculi

The calculus of monadic dynamic logic can be applied in order to define a Hoare logic for partial as well as one for total correctness of monadic programs. In Hoare logics for partial correctness of imperative programs one has assertions of the form $\{\phi\}\,p\,\{\psi\}$, which are to be understood as "if the precondition ϕ holds before execution of p, then the postcondition ψ will hold afterwards if p terminates". This idea also makes sense for monadic programs, but in fact it is already incorporated in dynamic logic by formulae of the form $\phi\Rightarrow[p]\,\psi$. Likewise, one can give meaning to Hoare assertions for total correctness by adding the requirement that a program terminates. This leads to the following definition.

Definition 3.23. A *Hoare assertion for partial correctness* of monadic programs is a formula

$$\phi \Rightarrow [\bar{x} \leftarrow \bar{p}]\, \psi \qquad \text{(written as} \quad \{\phi\}\, p\, \{\psi\})$$

A *Hoare assertion for total correctness* also requires the termination of the program under consideration and hence takes the following form:

$$\phi \Rightarrow ([\bar{x} \leftarrow \bar{p}]\, \psi \wedge \langle \bar{x} \leftarrow \bar{p} \rangle \top) \qquad \text{(written as} \quad [\phi]\, p\, [\psi])$$

Classical Hoare rules like a sequencing rule or a context weakening rule

$$\frac{[\phi]\, \bar{x} \leftarrow \bar{p}\, [\psi] \qquad [\psi]\, \bar{y} \leftarrow \bar{q}\, [\chi]}{[\phi]\, \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}\, [\chi]} \qquad \frac{[\phi]\, \bar{x} \leftarrow \bar{p}\, [\psi] \qquad \phi' \Rightarrow \phi \qquad \forall \bar{x}.\, \psi \Rightarrow \psi'}{[\phi']\, \bar{x} \leftarrow \bar{p}\, [\psi']}$$

(which of course also exist for partial correctness assertions) are easily derived in the proof calculus of dynamic logic. In the next section we will make use of a Hoare logic definable in this way for specifying and proving correct a pattern match algorithm. While Hoare logic represents a convenient way of reasoning about programs in the state monad (which naturally comes quite close to reasoning about simple imperative programming languages), e. g. the queue monad used in the next chapter does not lend itself to an axiomatisation simply by means of Hoare assertions about the basic queue operations. Hence, proofs about the queue monad will be conducted in the calculus of dynamic logic.

3.4 Specific Extensions for the Exception Monad

We have mentioned in Example 3.14 that non-termination in the (simple) exception monad means that an exception has been thrown. So, given an operation $raise : E \to TA$ which raises an exception from the set E of exceptions, one has $[raise\ e]\perp$ so that "anything can be proved in the presence of an exception". This might be acceptable as long as exceptions simply indicate some kind of failure and it does not matter much which error eventually occurred. In this case, partial correctness explicitly does *not* say anything about whether the program actually terminated and total correctness excludes all situations in which an exception occurred. But as soon as exceptions are employed to deliberately manipulate the control flow and if they may carry values (e. g. in the monad for Java of [13, 11]) this turns out to be a serious lack of expressiveness. An extension of the basic Hoare calculus described above has been given in [33] which makes it possible to reason about so called *abnormal postconditions* required to hold if an exception has been thrown (as opposed to the *normal postcondition* which must be satisfied in case of regular termination). This extension relies on the presence of an operation to turn an exceptional state back into a normal one, which is, of course, the well-known $catch : TA \to T(A+E)$ operation. As indicated by this signature, *catch* simply makes an exception visible rather than additionally requiring a handler to cope with the exceptional situation, as in Haskell's $catch : TA \to (E \to TA) \to TA$. The latter is easily definable in terms of the former. In [33] a categorical definition of exception monads is given[4], from which however one can derive all equations that may intuitively be expected

[4] in which *catch* is taken to be a natural transformation between T and $T(\_+E)$ such that it equalises the strong monad morphisms $catch_{\_+E}$ and $T\,inl$

to hold, e. g.

$$\begin{aligned} &catch\ \mathsf{do}\,\{x \leftarrow p; q\ x\} = \\ &\quad \mathsf{do}\,\{y \leftarrow catch\ p; \mathsf{case}\ y\ \mathsf{of}\ inl\ a \rightarrow catch\ (q\ a) \mid inr\ e \rightarrow ret\ (inr\ e)\} \\ &catch\ (ret\ x) = ret\ (inl\ x) \\ &catch\ (raise\ e) = ret\ (inr\ e) \end{aligned} \tag{3.5}$$

where the first equation states how *catch* behaves under sequential composition of programs (in particular the second program q is only executed if p did not throw any exceptions), the second one states that *ret* does not throw any exceptions and the third one expresses how *catch* interacts with *raise*, namely that it precisely returns the exception thrown by this operation.

With these defining equations for *catch* available, one may reason in the regular Hoare calculus by wrapping up all programs with a *catch* and doing a case distinction about the return value of *catch* in the postcondition:

$$\{\phi\}\, y \leftarrow (catch\ x \leftarrow p)\, \{\mathsf{case}\ y\ \mathsf{of}\ inl\ x \rightarrow \psi \mid inr\ e \rightarrow S\ e\}$$

The abnormal postcondition $S : E \rightarrow D\Omega$ is a stateful predicate on exception values and may not mention the normal return value x, whereas ψ of the normal postcondition may contain x freely. This scheme can be given a more convenient notation by explicitly distinguishing between normal and abnormal postcondition and leaving the ubiquitous *catch* unmentioned:

$$\{\phi\}\, x \leftarrow p\, \{\psi \parallel S\}$$

It is now possible to derive a Hoare calculus to reason about exception monads, including rules for sequential composition of programs, a rule for *raise* as well as one for *catch*, etc. Figure 3.3 lists all rules that apply to arbitrary exception monads; in particular note rule (raise) which shows how the problem of giving a reasonable postcondition for *raise* has been resolved.

3.4.1 Parameterised Exceptions

As a concrete example, we will now describe how to translate the exception handling mechanism of the Java programming language into the calculus described here. It will then appear that one further extension has to be made, since in Java even `return` statements terminate abnormally, resulting in exceptions carrying values of an arbitrary type. The stipulation that `return` (and `break` and `continue`) statements terminate abnormally is not specific to the model of Java given here, but rather settled in the Java language specification [16]. To deal with this situation a conversion function *mbody* is required that mediates between slightly different monads. This is due to the fact that every concrete monad may only carry exception values of a fixed type, as will be seen, whereas *return exceptions* of different methods may have entirely unrelated types – which is naturally so, since methods may have different return types.

The fact that certain statements terminate abnormally suggests the following data type be used as the type of exceptions – ignoring for the time being the class-hierarchy of exceptions rooting in class `Exception`, i. e. all run-time errors like `ArrayOutOfBoundsException` or `IOException`. The main point to be made here is how to model the hidden exceptions that do not show up as such within a real Java program. So let

$$E\ a = MRet\ a \mid FallenOff \mid Break \mid Cont \mid Error$$

where *MRet a* represents a return exception carrying the value which was the argument of the `return` statement that raised the exception. *FallenOff* will be raised by the yet to be defined *mbody* operation to indicate that its argument illegally terminated normally, *Break* and *Cont* are exceptions raised by `break` and `continue` statements respectively, and *Error* is an exception that slightly over-simplifyingly models all other cases.

The monad in which the semantics of sequential Java is modelled best is the state monad extended by exceptions and nontermination (where the latter is treated similar to an exception by the binding operation)

$$T\ a\ b = S \to S \times (b + E\ a) + 1$$

such that $T\ a$ is an exception state monad for each each type a in which binding respects exceptions, i. e. in do $\{x \leftarrow p; q\}$ the program q is only evaluated if p did not raise an exception. In this monad one has $\mathit{catch} : T\ a\ b \to T\ a\ (b + E\ a)$ and $\mathit{raise} : E \to T\ a\ b$ for all types a and b. The type of *catch* already points out that it is not possible to switch between monads of different exception types; this precludes the applicability of this model in situations where e. g. one method of Java-return type `int` is called within another method of return type `boolean`. The following example demonstrates the problem.

Example 3.24. Let *mret x* abbreviate $\mathit{raise}\ (\mathit{MRet}\ x)$, then the Java methods

```
public static int f(int x) {
  if (g(x) < 0)
    return x + 1;
  else
    return x - 1;
}

public static boolean g(int x) {
  return x*x < 100;
}
```

might naïvely be translated into the monadic model to obtain

$$\begin{aligned}
&f : \mathit{Int} \to T\ \mathit{Int}\ a\\
&f\ x = \mathsf{do}\ \{\ r \leftarrow \mathit{catch}\ (g\ x);\\
&\qquad\qquad \mathsf{case}\ r\ \mathsf{of}\\
&\qquad\qquad\quad \mathit{inl}\ (\mathit{MRet}\ b) \to \mathsf{if}\ b\ \mathsf{then}\ \mathit{mret}\ (x+1)\ \mathsf{else}\ \mathit{mret}\ (x-1)\\
&\qquad\qquad\quad _ \quad \to \mathit{raise}\ \mathit{Error}\ \}
\end{aligned}$$

$$\begin{aligned}
&g : \mathit{Int} \to T\ \Omega\ a\\
&g\ x = \mathit{mret}\ (x \cdot x < 100)
\end{aligned}$$

But this results in a type error, since the program $\mathit{catch}\ (g\ x)$ has type $T\ \Omega\ (a + \mathit{Int})$ in f, which itself is a monadic computation in $T\ \mathit{Int}$. Thus, the two monadic computations are incompatible. Intuitively, it should be possible to resolve this incompatibility, as the type of exceptions g may throw is not of importance to the exception type of f (all calls to methods are enclosed by *catch* and hence cannot propagate into f). In fact, this can be achieved in a way that simultaneously avoids having to enclose every method call by a *catch*. The key to this solution is the observation that every exception monad T can be obtained by applying the *exception monad transformer* (well known as `ErrorT` from the Haskell libraries) to some existing monad R such that T is isomorphic to $R(\_+E)$. Basically, this says that for every exception monad there is some underlying monad such that they share the same structure, but the exception monad only lives on result types enriched by some set E of exceptions. In the case at hand, R simply is the state monad with non-termination, and binding in $R\ (\_+E) = S \to S \times (\_+E) + 1$ means binding as defined for the state monad and not for the exception monad. The practical consequence of this relationship is that one can also write programs in $R(\_+E)$ and convert them to T via `ErrorT`, which is precisely what is done for *mbody*. We refer to Appendix A, p. 98, for a Haskell implementation of *mbody* and the exception monad transformer. The pivotal property of *mbody* from the viewpoint of the exception monad T is that it converts the exceptional state of a computation back into a normal one if a return exception has been raised, but lets all other exceptions pass – thus making it *polymorphic in its own exception type*. Additionally, in case of normal termination of its argument, *mbody* will raise a *FallenOff* exception. Its type therefore is

$$\mathit{mbody} : T\ a\ b \to T\ c\ a$$

When translating Java methods into the monadic setting, one will thus enclose the translation m of every method body `m` of function `f` by *mbody* to obtain the translated function f. Conducted in this manner, the translation of the above Java methods then is

$$\begin{array}{l} f : \mathit{Int} \to T\ a\ \mathit{Int} \\ f\ x = \mathit{mbody}\ (\\ \qquad\qquad \mathsf{do}\ \{\ b \leftarrow g\ x; \\ \qquad\qquad\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathit{mret}\ (x+1)\ \mathsf{else}\ \mathit{mret}\ (x-1) \\ \qquad\qquad \}\ ) \end{array}$$

$$\begin{array}{l} g : \mathit{Int} \to T\ b\ \Omega \\ g\ x = \mathit{mbody}\ (\ \mathit{mret}\ (x \cdot x < 100)\ ) \end{array}$$

Since every program p obtained from a translation of a Java method into the monadic setting will now contain an occurrence of *mbody*, it is necessary at this point to specify and prove a Hoare rule for this construct which captures its decisive properties (see also [38]). Fortunately, a single rule suffices for this purpose in the case of partial as well as total correctness assertions (and both rules look alike so that only one of them is shown), noting that one will only want to prove properties of programs that terminate abruptly with a return exception.

$$(\textbf{mbody}) \quad \frac{\{\phi\}\, x \leftarrow p\, \{\bot \parallel \lambda e.\, \mathsf{case}\ e\ \mathsf{of}\ \mathit{MRet}\ y \to \psi \mid e \to S\ e\}}{\{\phi\}\, y \leftarrow \mathit{mbody}\ p\, \{\psi \parallel S\}}$$

Correctness of a pattern match algorithm

As an example of how to apply the extended calculus to realistic programs, we will specify and prove the correctness of a pattern match algorithm which searches for a given sub-pattern in a given base pattern. The algorithm is implemented in an exception monad with dynamic references and a while loop; the existence of the latter implicitly presupposes additional structure on the monad, see [34, Section 7] for details and Appendix A for an implementation. One therefore has to axiomatise additional operations on the monad (apart from *ret* and $\gg=$);

the corresponding specification is shown in Figure 3.2. A condensed version of this proof already appeared in [38], while here we provide the full picture.

$$
\begin{array}{l}
\mathit{pmatch} : \mathit{List}\ a \to \mathit{List}\ a \to T\ e\ \mathit{Nat} \\
\mathit{pmatch}\ \mathit{base}\ \mathit{pat} = \mathit{mbody}\ (\ \mathsf{do}\ \{ \\
\qquad r \leftarrow \mathit{new}\ 0; \\
\qquad s \leftarrow \mathit{new}\ 0; \\
\qquad \mathit{while}\ (\mathit{ret}\ \top) \\
\qquad\qquad (\mathsf{do}\ \{\ u \leftarrow *r; \\
\qquad\qquad\qquad v \leftarrow *s; \\
\qquad\qquad\qquad \mathsf{if}\ u = \mathit{len}\ \mathit{pat} \\
\qquad\qquad\qquad\quad \mathsf{then}\ \mathit{mret}\ v \\
\qquad\qquad\qquad\quad \mathsf{else\ if}\ v + u = \mathit{len}\ \mathit{base} \\
\qquad\qquad\qquad\qquad \mathsf{then}\ \mathit{raise}\ \mathit{Error} \\
\qquad\qquad\qquad\qquad \mathsf{else\ if}\ \mathit{base}!!(v+u) = \mathit{pat}!!u \\
\qquad\qquad\qquad\qquad\quad \mathsf{then}\ r := u + 1 \\
\qquad\qquad\qquad\qquad\quad \mathsf{else\ do}\ \{\ s := v + 1;\ r := 0\} \\
\qquad\qquad \}) \\
\qquad \})
\end{array}
$$

This definition of *pmatch* is almost identical to the Haskell implementation to be found in Appendix A, with slight modifications to retain the notation used so far. It introduces a type constructor *List* mapping each type a to the type of lists over a, a length function $\mathit{len} : \mathit{List}\ a \to \mathit{Nat}$ and an indexing function $!! : \mathit{List}\ a \to \mathit{Nat} \to a$ operating on these lists in the usual way – where the latter is undefined if the index exceeds the bounds of the list. Further it requires a natural numbers type *Nat* and makes use of *existential equality* when comparing elements of lists. This means that a comparison $v!!i = w!!j$ yields true if and only if both $v!!i$ and $w!!j$ are defined and equal. An informal specification of this algorithm is as follows.

- *pmatch* returns the first – i. e. least – index x such that the pattern *pat* occurs in *base* starting at index x.
- If no such index exists, *pmatch* will fail with an exception *Error*.

The specification in Figure 3.2 extends the axiomatisation of the dynamic reference monad given in [32] by abnormal postconditions, which in most cases are $\bot$, asserting that the corresponding operations do not raise exceptions. An exception is the rule (new-distinct), which states that the subsequent creation of references, with an arbitrary program p (which may raise exceptions) executed in between, produces distinct references. We prove total correctness of the algorithm generically, i. e. without further assumptions on the underlying monad other than the axioms of Figure 3.2 and the interpretability of a *while* construct. Figure 3.3 displays the generic Hoare calculus for total exception correctness. The calculus for partial correctness is essentially identical (where the square brackets are of course replaced by curly brackets) except for rule (stateless) in which there is no need for a premiss.

For the actual method body p, i. e. the argument of *mbody* in function *pmatch*, we claim that it terminates abnormally, raising either a return exception carrying as its value an index x that is the starting position of the first occurrence of the pattern in the base string, or a

Operations

$read : Ref\ a \to T\ b\ a$ $\quad (read\ r \equiv_{\mathrm{def}} *r)$

$write : Ref\ a \to a \to T\ b\ 1$ $\quad (write\ r\ x \equiv_{\mathrm{def}} r := x)$

$new : a \to T\ b\ (Ref\ a)$

Axioms

$\mathrm{dsef}(read)$ (dsef-read)

$[]\ r := x\,[x = *r \parallel \bot]$ (read-write)

$[x = *r \wedge \neg r = s]\ s := y\,[x = *r \parallel \bot]$ (read-write-other)

$[]\ r \leftarrow new\ x\,[x = *r \parallel \bot]$ (read-new)

$[x = *r \wedge \neg r = s]\ s \leftarrow new\ y\,[x = *r \parallel \bot]$ (read-new-other)

$[\phi]\ r \leftarrow new\ x; p\,[\top \parallel \top] \Rightarrow$

$\quad [\phi]\ r \leftarrow new\ x; p; s \leftarrow new\ y\,[\neg r = s \parallel \top]$ (new-distinct)

Figure 3.2: Specification of the exception reference monad

failure exception *Error* indicating that there is no occurrence of the pattern in the base string. Compare this to the specification (3.20) for *pmatch* itself, which actually *returns* the index x if found:

$$[]\ p\ [\bot \parallel \lambda e.\ \mathsf{case}\quad e\ \mathsf{of} \quad \begin{array}{l} \ \ MRet\ i \to MPOS\ i \wedge \forall j.\ MPOS\ j \Rightarrow i \leq j \\ \mid Error \to \neg\exists i.\ MPOS\ i \\ \mid \_ \to \bot] \end{array} \tag{3.6}$$

The abnormal postcondition above will be denoted by *POST* below. Here, *MPOS i* states that the pattern is matched at position i in the base string:

$$MPOS\ i \equiv \forall j.\ 0 \leq j < len\ pat \Rightarrow base!!(i+j) = pat!!j.$$

In order to apply the total exception while rule (while) of Figure 3.3, we need to provide a loop invariant *INV* and a termination measure t. Putting

$$\begin{array}{rl} INV \equiv & (\forall i.\ 0 \leq i < *r \Rightarrow base!!(*s+i) = pat!!i)\ \wedge \\ & \forall i.\ MPOS\ i \Rightarrow *s \leq i \end{array}$$

(which implies $0 \leq *r \leq len\ pat$ and $0 \leq *s + *r \leq len\ base$) guarantees that the dsef term $t = (len\ base - *s, len\ pat - *r)$ always yields results of type $Nat \times Nat$, on which we have the lexicographic ordering as a well-founded relation.

Establishing the invariant upon entrance into the loop is easy, since from the axioms given above,

$$[]\ r \leftarrow new\ 0; s \leftarrow new\ 0\,[*s = *r = 0 \wedge \neg(r = s) \parallel \bot] \tag{3.7}$$

can be derived by the rules (seq), (conj), (read-new-other) and (new-distinct). Inside the loop, there are essentially four branches, arising from three applications of the rule (if), so

$$(\textbf{seq})\quad \frac{\begin{array}{c}[\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi\,\|\,S]\\ [\psi]\,\overline{y}\leftarrow\overline{q}\,[\chi\,\|\,S]\end{array}}{[\phi]\,\overline{x}\leftarrow\overline{p};\overline{y}\leftarrow\overline{q}\,[\chi\,\|\,S]}\qquad (\textbf{ctr})\quad \frac{\begin{array}{c}[\phi]\ \ldots;x\leftarrow p;y\leftarrow q;\overline{z}\leftarrow\overline{r}\,[\psi\,\|\,S]\\ x\notin FV(\overline{r})\cup FV(\psi)\end{array}}{[\phi]\ \ldots;y\leftarrow(\mathsf{do}\,\{x\leftarrow p;q\});\overline{z}\leftarrow\overline{r}\,[\psi\,\|\,S]}$$

$$(\textbf{conj})\quad \frac{\begin{array}{c}[\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi_1\,\|\,S_1]\\ [\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi_2\,\|\,S_2]\end{array}}{[\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi_1\wedge\psi_2\,\|\,S_1\wedge S_2]}\qquad (\textbf{disj})\quad \frac{\begin{array}{c}[\phi_1]\,\overline{x}\leftarrow\overline{p}\,[\psi\,\|\,S]\\ [\phi_2]\,\overline{x}\leftarrow\overline{p}\,[\psi\,\|\,S]\end{array}}{[\phi_1\vee\phi_2]\,\overline{x}\leftarrow\overline{p}\,[\psi\,\|\,S]}$$

$$(\textbf{wk})\quad \frac{\begin{array}{c}[\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi\,\|\,S]\\ \phi'\Rightarrow\phi\quad \psi\Rightarrow\psi'\\ \forall e.\,S\,e\Rightarrow S'\,e\end{array}}{[\phi']\,\overline{x}\leftarrow\overline{p}\,[\psi'\,\|\,S']}\qquad (\textbf{stateless})\quad \frac{[ret\ \phi]\,q\,[\top\,\|\,\top]}{[ret\ \phi]\,q\,[ret\ \phi\,\|\,\lambda e.\,ret\ \phi]}$$

$$(\textbf{dsef})\quad \frac{p\ \text{dsef}}{[\phi]\,x\leftarrow p\,[\phi\wedge x=p\,\|\,\bot]}\qquad (\textbf{if})\quad \frac{\begin{array}{c}[\phi\wedge b]\,x\leftarrow p\,[\psi\,\|\,S]\\ [\phi\wedge\neg b]\,x\leftarrow q\,[\psi\,\|\,S]\end{array}}{[\phi]\,x\leftarrow\mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q\,[\psi\,\|\,S]}$$

$$(\textbf{catch})\quad \frac{[\phi]\,\overline{x}\leftarrow\overline{p}\,[\psi[inl\ \overline{x}/y]\,\|\,\lambda e.\,\psi[inr\ e/y]]}{[\phi]\,y\leftarrow(catch\ \overline{x}\leftarrow\overline{p})\,[\psi\,\|\,\bot]}$$

$$(\textbf{raise})\quad \frac{}{[\phi]\,raise\ e_0\,[\bot\,\|\,\lambda e.\,(\phi\wedge e=e_0)]}$$

$$(\textbf{while})\quad \frac{\begin{array}{c}[\phi\wedge b]\,p\,[\top\,\|\,\top]\\ \{\phi\wedge b\wedge t=z\}\,p\,\{\phi\wedge b\wedge t<z\,\|\,\top\}\\ \{\phi\wedge b\}\,p\,\{\top\,\|\,S\}\end{array}}{[\phi]\,while\ b\ p\,[\phi\wedge\neg b\,\|\,S]}$$

Figure 3.3: The generic Hoare calculus for total exception correctness

that the three premisses of the total exception while rule are split into twelve proof goals (the two read operations $u \leftarrow *r$ and $v \leftarrow *s$ are dealt with by rules (dsef) and (seq)). The total exception while rule proof obligations, stated informally, are first to prove termination of the program at hand, then to prove that the invariant is maintained as well as that the termination measure decreases strictly, and finally to prove that the abnormal postcondition can be established given the loop invariant as a precondition. We now prove the goals for each branch, with some of those having obvious proofs omitted. Furthermore, we will leave the pre- and postcondition $\neg(r = s)$ implicit, since it obviously prevails in the whole proof thanks to rule (stateless).

(i) Returning an Index.

$$[\mathit{INV} \wedge *r = u \wedge *s = v \wedge u = \mathit{len\ pat}]\ \mathit{mret}\ v\ [\top \,\|\, \top] \tag{3.8}$$

$$\begin{gathered}\{\mathit{INV} \wedge *r = u \wedge *s = v \wedge u = \mathit{len\ pat} \wedge (\mathit{len\ base} - v, \mathit{len\ pat} - u) = z\} \\ \mathit{mret}\ v \\ \{\mathit{INV} \wedge (\mathit{len\ base} - *s, \mathit{len\ pat} - *r) < z \ \|\ \top\}\end{gathered} \tag{3.9}$$

$$\{\mathit{INV} \wedge *r = u \wedge *s = v \wedge u = \mathit{len\ pat}\}\ \mathit{mret}\ v\ \{\top \,\|\, \mathit{POST}\} \tag{3.10}$$

By the total and partial variants of rules (raise) and (wk), recalling that $\mathit{mret}\ v$ is just an abbreviation for $\mathit{raise}\ (\mathit{MRet}\ v)$ one easily obtains (3.8) and (3.9). It remains to show (3.10); now from its precondition we infer $\mathit{MPOS}\ v \wedge \forall i.\, \mathit{MPOS}\ i \Rightarrow v \le i$, a stateless formula. Further we may derive $\{\}\ \mathit{mret}\ v\ \{\bot \,\|\, \lambda e.\ e = (\mathit{MRet}\ v)\}$ by (raise). Hence, by (stateless), (conj), and (wk), we obtain

$$\begin{aligned}&\{\mathit{INV} \wedge *r = u \wedge *s = v \wedge *r = \mathit{len\ pat}\}\ \mathit{mret}\ v \\ &\qquad\qquad \{\bot \,\|\, \lambda e.\, e = \mathit{MRet}\ v \wedge \mathit{MPOS}\ v \wedge \forall i.\, \mathit{MPOS}\ i \Rightarrow v \le i\}\end{aligned}$$

The formula in the abnormal postcondition implies POST, since the kind of exception is identified as MRet and thus the formula can be extended to the case construct of POST. This means we are finished by another application of (wk).

(ii) Failing to Find an Index.

$$[\mathit{INV} \wedge *r = u \wedge *s = v \wedge \neg(u = \mathit{len\ pat}) \wedge u + v = \mathit{len\ base}]\ \mathit{raise\ Error}\ [\top \,\|\, \top] \tag{3.11}$$

$$\{\mathit{INV} \wedge (\mathit{len\ base} - *s, \mathit{len\ pat} - *r) = z \wedge \ldots\}\ \mathit{raise\ Error}\ \{\mathit{INV} \wedge \ldots \,\|\, \top\} \tag{3.12}$$

$$\begin{gathered}\{\mathit{INV} \wedge *r = u \wedge *s = v \wedge \neg(u = \mathit{len\ pat}) \wedge u + v = \mathit{len\ base}\} \\ \mathit{raise\ Error} \\ \{\top \,\|\, \mathit{POST}\}\end{gathered} \tag{3.13}$$

Again, (3.11) and (3.12) – which has not even been written out in full; the pattern is as in (i) – are immediate by rules (raise) and (wk). To show (3.13) we note that by (raise) one has

$$\{\}\ \mathit{raise\ Error}\ \{\bot \,\|\, \lambda e.\ e = \mathit{Error}\} \tag{3.14}$$

and letting $P \equiv_{\text{def}} INV[u/{*r}, v/{*s}] \wedge u + v = len\ base \wedge \neg(u = len\ pat)$ by (stateless) one obtains

$$\{P\}\ raise\ Error\ \{P \,\|\, \lambda e.\, P\} \tag{3.15}$$

After strengthening the precondition of (3.14) by (wk) one may combine it with (3.15) by rule (conj). But the formula thus obtained in the abnormal postcondition implies *POST*: informally this can be seen because the exception type is *Error*; the substituted invariant guarantees that no pattern has been found up to v and none can appear later on as the end of the pattern has been reached and u is less than *len pat*; this means that no occurrence of *pat* in *base* exists.

(iii) Proceeding With a Partial Match. In this branch the first and third goals are trivially proved, since assignment terminates and does not raise exceptions. The second goal is

$$\begin{gathered}\{INV \wedge {*r} = u \wedge {*s} = v \wedge \neg(u = len\ pat) \wedge \neg(u + v = len\ base)\\ \wedge (len\ base - v, z = len\ pat - u) \wedge base!!(u+v) = pat!!u\}\\ r := u + 1\\ \{INV \wedge (len\ base - {*s}, len\ pat - {*r}) < z \ \|\ \top\}\end{gathered} \tag{3.16}$$

There are two parts to be shown here: on the one hand it has to be established that the invariant holds in the postcondition, and on the other hand one must show that the termination measure t decreases. Both proofs hinge on rules (read-write) and (read-write-other) from which one infers

$$\{\neg(r = s) \wedge {*s} = v\}\ r := u + 1\ \{{*s} = v \wedge {*r} = u + 1 \,\|\, \bot\} \tag{3.17}$$

so that in (3.16) the value of $*s$ carries over from the pre- to the postcondition, while the value of $*r$ is increased exactly by one. Taken together, this forces the measure t to decrease strictly. Regarding the invariant a similar point can be made: analogous to (ii) the invariant with u and v replacing $*r$ and $*s$ respectively carries over from the precondition to the postcondition since it is stateless. Moreover, the facts that the partial match may be extended, i. e. one has $base!!(u+v) = pat!!u$, and (3.17) establish the invariant proper.

(iv) Starting a New Match. Again, the first and third goals are not shown, because the situation is essentially the same as for (iii), the only difference being that two assignments are executed instead of one.

$$\begin{gathered}\{INV \wedge {*r} = u \wedge {*s} = v \wedge \neg(u = len\ pat) \wedge \neg(v + u = len\ base)\\ \wedge z = (len\ base - v, len\ pat - u) \wedge \neg(base!!(u+v) = pat!!u)\}\\ \mathsf{do}\,\{s := v + 1; r := 0\}\\ \{INV \wedge (len\ base - {*s}, len\ pat - {*r}) < z\}\end{gathered} \tag{3.18}$$

Once more the crucial fact can be obtained from (read-write) and (read-write-other):

$$\{\neg(r = s)\}\ s := v + 1; r := 0\ \{{*r} = 0 \wedge {*s} = v + 1 \,\|\, \bot\} \tag{3.19}$$

This forces the termination measure to decrease strictly, and enables one to retain the invariant in the postcondition. Informally this is valid due to the fact that $\neg(base!!(u+v) = pat!!u)$,

i. e. the current partial match cannot be completed. It is then legal to increase s by one to search for another match further on, validating the second conjunct of the invariant. By setting r to zero the first conjunct of the invariant becomes vacuously true.

Altogether, having arrived at proving Formula (3.6) by composing (3.7) with the conclusion of the total exception while rule, we can then apply rule (mbody) to obtain the total correctness of the whole algorithm:

$$[]\ i \leftarrow mbody\ p\ [MPOS\ i \wedge \forall j.\, MPOS\ j \Rightarrow i \leq j \parallel \lambda e.\,\mathsf{case}\ e\ \mathsf{of}\ \begin{array}{l} Error \rightarrow \neg\exists i.\, MPOS\ i \\ \mid \_ \rightarrow \bot]. \end{array} \tag{3.20}$$

4 Verification with Dynamic Logic

We will now apply the general calculus as well as monad-specific extensions of it to prove properties of monadic programs. These proofs will be fairly detailed, which is so because of their being formal proofs. On the one hand this provides rigorous evidence of their correctness, but on the other hand it definitely prompts for the employment of a (semi-) automatic proof assistant to dispose of the necessity of doing the most trivial proof steps by hand. We begin with some standard lemmas which are typical of dynamic logic.

4.1 Basic Lemmas of Dynamic Logic

An important and quite natural fact is that one may prove formulae of the form $[x \leftarrow p]\,(\bigwedge \phi_i)$ by proving each $[x \leftarrow p]\,\phi_i$ separately:

Lemma 4.1. $[x \leftarrow p]\,(\phi \wedge \psi)$ *if and only if* $[x \leftarrow p]\,\phi \wedge [x \leftarrow p]\,\psi$

Proof. "$\Rightarrow$" We have $\phi \wedge \psi \Rightarrow \phi$, which is a tautology, so by (nec) and one-time application of modus ponens to (K1) we obtain $[x \leftarrow p]\,(\phi \wedge \psi) \Rightarrow [x \leftarrow p]\,\phi$. Dually, we arrive at $[x \leftarrow p]\,(\phi \wedge \psi) \Rightarrow [x \leftarrow p]\,\psi$ when starting from $\phi \wedge \psi \Rightarrow \psi$. Taken together, the proposition is proved.

"$\Leftarrow$" Beginning with the tautology $\phi \Rightarrow (\psi \Rightarrow \phi \wedge \psi)$, by (nec) and two-time application of modus ponens to (K1) we arrive at $[x \leftarrow p]\,\phi \Rightarrow ([x \leftarrow p]\,\psi \Rightarrow [x \leftarrow p]\,(\phi \wedge \psi))$ which is tautologically equivalent to $[x \leftarrow p]\,\phi \wedge [x \leftarrow p]\,\psi \Rightarrow [x \leftarrow p]\,(\phi \wedge \psi)$.

□

Lemma 4.2 (Regularity). *The following are valid rules of inference.*

$$(\mathbf{reg}\Box)\quad \frac{\forall x.\ \phi \Rightarrow \psi}{[x \leftarrow p]\,\phi \Rightarrow [x \leftarrow p]\,\psi} \qquad (\mathbf{reg}\Diamond)\quad \frac{\forall x.\ \phi \Rightarrow \psi}{\langle x \leftarrow p\rangle\,\phi \Rightarrow \langle x \leftarrow p\rangle\,\psi}$$

Proof. For $(reg\Box)$, assume $\forall x.\ \phi \Rightarrow \psi$, apply necessitation to obtain $[x \leftarrow p]\,\phi \Rightarrow \psi$, from which the conclusion can be derived by modus ponens with (K1). The proof for rule $(reg\Diamond)$ is identical, except that (K2) has to be used in the final step. □

Lemma 4.3. *The following two rules that resemble modus ponens, only 'inside' boxes and diamonds, are valid derived rules of inference.*

$$(\mathbf{wk}\Box)\quad \frac{[x \leftarrow p]\phi \quad \forall x.\, \phi \Rightarrow \psi}{[x \leftarrow p]\psi} \qquad (\mathbf{wk}\Diamond)\quad \frac{\langle x \leftarrow p\rangle\phi \quad \forall x.\, \phi \Rightarrow \psi}{\langle x \leftarrow p\rangle\psi}$$

Proof. Concerning rule (wk$\Box$) we have to deduce the conclusion $[x \leftarrow p]\psi$ under the assumptions $[x \leftarrow p]\phi$ and $\forall x.\ \phi \Rightarrow \psi$. By regularity, we immediately obtain $[x \leftarrow p]\,\phi \Rightarrow [x \leftarrow p]\,\psi$, which provides the conclusion through an application of modus ponens with the assumption $[x \leftarrow p]\,\phi$. Once again, the proof of (wk$\Diamond$) is dual. □

The following lemmas, which can also be found in [8], show some distributivity properties of the modal operators. It should be pointed out that the implications in the other directions are not valid (except for the first lemma, where the reverse implication is axiom (K4)).

Lemma 4.4. $\langle x \leftarrow p \rangle\, \phi \vee \langle x \leftarrow p \rangle\, \psi \Rightarrow \langle x \leftarrow p \rangle\, \phi \vee \psi$

Proof. This proof is rather typical and the same scheme will be applied to the following ones. First, we start with the tautology $\forall x.\ \phi \Rightarrow \phi \vee \psi$; strictly speaking this is not a tautology due to the universal quantifier, but this formula can easily be obtained from the tautologous $\phi \Rightarrow \phi \vee \psi$ by universal generalisation, so we will talk of a formula as being a tautology even if it is the universal closure of one.

By regularity we derive $\langle x \leftarrow p \rangle\, \phi \Rightarrow \langle x \leftarrow p \rangle\, \phi \vee \psi$, and we can also gain $\langle x \leftarrow p \rangle\, \psi \Rightarrow \langle x \leftarrow p \rangle\, \phi \vee \psi$ in a similar fashion. From these, twofold application of (mp) to the tautology scheme $(\Phi \Rightarrow \Theta) \Rightarrow (\Psi \Rightarrow \Theta) \Rightarrow (\Phi \vee \Psi \Rightarrow \Theta)$, where $\Phi = \langle x \leftarrow p \rangle\, \phi, \Psi = \langle x \leftarrow p \rangle\, \psi$ and $\Theta = \langle x \leftarrow p \rangle\, \phi \vee \psi$, gives the desired result. □

Lemma 4.5. $\langle x \leftarrow p \rangle\, \phi \wedge \psi \Rightarrow \langle x \leftarrow p \rangle\, \phi \wedge \langle x \leftarrow p \rangle\, \psi$

Proof. Here the tautology scheme is $(\Theta \Rightarrow \Phi) \Rightarrow (\Theta \Rightarrow \Psi) \Rightarrow \Theta \Rightarrow \Phi \wedge \Psi$ allowing us to separately prove $\langle x \leftarrow p \rangle\, \phi \wedge \psi \Rightarrow \langle x \leftarrow p \rangle\, \phi$ and $\langle x \leftarrow p \rangle\, \phi \wedge \psi \Rightarrow \langle x \leftarrow p \rangle\, \psi$ and then applying modus ponens twice. But these two formulae are directly provable from the obvious tautologies and application of rule (reg$\Diamond$). □

Lemma 4.6. *The following implications are valid in the calculus. Proofs thereof are very similar to the previous ones and are omitted here. Instead we refer to Section C.5 in the Appendix where the formulae have been verified with Isabelle.*

$$\langle x \leftarrow p \rangle\, \phi \wedge [x \leftarrow p]\, \psi \Rightarrow \langle x \leftarrow p \rangle\, \phi \wedge \psi$$
$$[x \leftarrow p]\, \phi \vee [x \leftarrow p]\, \psi \Rightarrow [x \leftarrow p]\, \phi \vee \psi$$

4.2 Axiomatising the Queue-Monad

Following the axiomatic approach to reasoning about a particular monad, the first step is to characterise the monad by the signature of its basic operations and a set of additional axioms. This is in contrast to the definitional approach of [23], where one preferably defines the operations of the monad and derives its properties as lemmas in the calculus on hand. The following is a possible specification of a *queue monad*, which comes with operations to insert an element into the queue, to remove an element from the queue and simultaneously return it as well as an operation for testing whether the queue is empty. The signature of the operations is

Operations

$enq : A \to Q\ 1$

$deq : Q\ A$

$empty : Q\ \Omega$

where A is a fixed type of queue elements, i. e. *enq* and *deq* are not polymorphic. A possible implementation of this monad is as a specific state monad that maintains a list of elements of type A as its state value.

Axioms

$\mathrm{dsef}(empty)$ (dsef-empty)

$\langle enq\rangle\,\top$ (enq-term)

$\neg empty \Rightarrow \langle deq\rangle\,\top$ (deq-term)

$empty \Rightarrow [deq]\,\bot$ (empty-deq)

$[enq\ z]\,\neg empty$ (non-empty)

$empty \Rightarrow [enq\ z; x \leftarrow deq]\,(x = z \wedge empty)$ (enq-deq)

$\neg empty \wedge [enq\ z; x \leftarrow deq]\,\phi \iff \neg empty \wedge [x \leftarrow deq; enq\ z]\,\phi$ (swap)

With these axioms we are able to establish some simple proofs about the queue monad. For example, given an empty queue we can insert two items, fetch and bind two items thereafter, and make a statement about the equality of items inserted and fetched:

Proposition 4.7. $empty \Rightarrow [enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq](x = a \wedge y = b)$

Proof. We proceed in two steps, first asserting (a) $empty \Rightarrow [enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq](x = a)$, then (b) $empty \Rightarrow [enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq](y = b)$ and conclude by combining these two results by Lemma 4.1

(a)

$$empty \Rightarrow [enq\ a; x \leftarrow deq](x = a) \wedge empty \qquad \text{by (enq-deq)}$$

Noting that $x = a$ is stateless and thus applying (K3) and (seq$\Box$) we obtain

$$empty \Rightarrow [enq\ a; x \leftarrow deq; enq\ b](x = a) \tag{4.1}$$

By (swap) we have

$$\neg empty \Rightarrow [x \leftarrow deq; enq\ b]\phi \Rightarrow [enq\ b; x \leftarrow deq]\phi$$

to which we apply (nec) and subsequently (K1) twice:

$$\begin{aligned} [enq\ a]\neg empty &\Rightarrow [enq\ a][x \leftarrow deq; enq\ b]\phi \\ &\Rightarrow [enq\ a][enq\ b; x \leftarrow deq]\phi \end{aligned}$$

This can be simplified by (non-empty) and (seq$\Box$):

$$[enq\ a; x \leftarrow deq; enq\ b]\phi \Rightarrow [enq\ a; enq\ b; x \leftarrow deq]\phi \tag{4.2}$$

'Connecting' (4.1) and (4.2) by rule (wk$\Box$) provides

$$empty \Rightarrow [enq\ a; enq\ b; x \leftarrow deq](x = a)$$

from which, finally, the proposition (a) can be derived by application of (K3) and (seq$\Box$).

(b) We have to show $empty \Rightarrow [enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq](y = b)$ proceeding as follows and leaving applications of (seq□) implicit. By (enq-deq) we respectively have

$$empty \Rightarrow [enq\ a; x \leftarrow deq]empty$$
$$empty \Rightarrow [enq\ b; y \leftarrow deq](y = b)$$

These can be connected (with the help of rule wk□) to form

$$empty \Rightarrow [enq\ a; x \leftarrow deq; enq\ b; y \leftarrow deq](y = b) \tag{4.3}$$

Also, by (swap) we have

$$\neg empty \wedge [x \leftarrow deq; enq\ b]\phi \Rightarrow [enq\ b; x \leftarrow deq]\phi$$

We once more apply (nec) and (K1) which brings us close to our goal:

$$[enq\ a]\neg empty \Rightarrow [enq\ a; x \leftarrow deq; enq\ b]\phi \Rightarrow$$
$$[enq\ a; enq\ b; x \leftarrow deq]\phi$$

The premiss can be disposed of by axiom (non-empty) so that we now instantiate ϕ with $[y \leftarrow deq](y = b)$ arriving at

$$\begin{aligned} &[enq\ a; x \leftarrow deq; enq\ b; y \leftarrow deq](y = b) \Rightarrow \\ &\quad [enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq](y = b) \end{aligned} \tag{4.4}$$

Now connect (4.3) and (4.4) and we are done.

□

We would now like to maintain an assertion concerning the termination of the program sequence given above. This amounts to stating

Proposition 4.8. $\langle enq\ a; enq\ b; deq; deq\rangle \top$

Intuitively, one would say that any program sequence containing only *enq*'s and *deq*'s in which every execution of *deq* is preceded by more *enq*'s than *deq*'s should terminate unconditionally. Moreover, any program sequence with the stated property and in which the total number of enq's exceeds the number of deq's should enforce the queue not to be empty. This idea leads to the following definition and theorem, from which the above proposition can be proved with ease.

Definition 4.9. In chance analogy to [5], we say that a program sequence p in the queue monad is *well-formed* iff it is a non-empty sequence of programs $enq\ z_i$ or $x_i \leftarrow deq$ in which every initial subsequence has the property of containing at least as many programs of the former type as of the latter type and in which $x_i \neq x_j$ for $i \neq j$.

Example 4.10. $(enq\ a; enq\ b; x \leftarrow deq; y \leftarrow deq)$ is a well-formed program sequence, whereas $(enq\ a; x \leftarrow deq; y \leftarrow deq)$ is not.

Theorem 4.11. *For every well-formed program sequence p containing more enq's than deq's one has* $[p]\neg empty$.

Proof. By induction on the number of occurrences of *enq*.

In the base case $n = 1$ there is only one possible program sequence, namely *enq* z for some z. Then, axiom (non-empty) gives us $[enq\ z]\neg empty$.

In the inductive step, w.l.o.g. let p consist of programs $enq\ z_i$, $1 \le i \le n+1$ and $x_j \leftarrow deq$, $1 \le j \le m$ where necessarily $m \le n$. We take a look at the final occurrence of an *enq* and distinguish two possible cases:

(i) $p = (\ldots; enq\ z_{n+1})$, i. e. the final *enq* appears at the end of the program sequence. In this easier case, by repeated application of rule (nec) to $[enq\ z_{n+1}]\neg empty$, which is an instance of axiom (non-empty), one obtains $[p]\neg empty$.

(ii) $p = (\ldots; enq\ z_{n+1}; x_{m-j} \leftarrow deq; \ldots; x_m \leftarrow deq)$. This can be proved by induction on j. In the base case, where $j = 0$, we have $p = (\ldots; enq\ z_{n+1}; x_m \leftarrow deq)$ which means we can apply the 'outer' induction hypothesis to the $(\ldots)$ part providing $[\ldots]\neg empty$. By (swap) we have

$$\neg empty \Rightarrow [x_m \leftarrow deq; enq\ z_{n+1}]\neg empty \Rightarrow [enq\ z_{n+1}; x_m \leftarrow deq]\neg empty$$

and it thus suffices to show $[x_m \leftarrow deq; enq\ z_{n+1}]\neg empty$ which can be done by applying rule (nec) to $[enq\ z_{n+1}]\neg empty$, an instance of axiom (non-empty).

In the inductive step $(j-1 \to j,\ j > 0)$, $[\ldots; enq\ z_{n+1}; x_{m-j} \leftarrow deq; \ldots; x_m \leftarrow deq]\neg empty$ has to be asserted. By the outer inductive hypothesis we again have $[\ldots]\neg empty$, so by (swap) it suffices to show

$$[\ldots; x_{m-j} \leftarrow deq; enq\ z_{n+1}; x_{m-j+1} \leftarrow deq; \ldots; x_m \leftarrow deq]\neg empty$$

This is true due to the 'inner' inductive hypothesis. □

Now we return to the deferred task of proving the termination of the above program sequence, i. e. we will prove $\langle enq\ a; enq\ b; deq; deq\rangle\top$.

Proof. By Lemma 4.11 we have

$$[enq\ a; enq\ b]\neg empty \quad \text{and} \quad [enq\ a; enq\ b; x \leftarrow deq]\neg empty \tag{4.5}$$

Now, $\langle enq\ a\rangle\top$ and $\langle enq\ b\rangle\top$ which is equivalent to $\top \Rightarrow \langle enq\ b\rangle\top$. Thus, by rule (wk$\diamond$) and (seq$\diamond$):

$$\langle enq\ a; enq\ b\rangle\top \tag{4.6}$$

We prove $\langle enq\ a; enq\ b\rangle\neg empty$ by application of (K2) to (4.5) and (4.6) once again noting that $\phi \iff (\top \Rightarrow \phi)$.

In order to proceed to $\langle enq\ a; enq\ b; x \leftarrow deq\rangle\top$ we apply rule (wk$\diamond$) with the help of (deq-term). With the right-hand statement of (4.5) we can, in a very similar manner to the one just pointed out, assert $\langle enq\ a; enq\ b; x \leftarrow deq\rangle\neg empty$ in which we only need one further application of rule (wk$\diamond$) and axiom (deq-term) to finish the proof. □

4.3 Specification of a Reference Monad

The algorithm of Section 4.4 will make use of a single reference to store a result value in. Therefore we briefly review the axioms of a monad in which such references are available.

Further details can be found in [32]. The reference monad R is equipped with operations for reading a reference $r : \mathit{Ref}\, A$, i. e. a reference containing a value of type A, and writing to it:

Operations

$* \_ : \mathit{Ref}\, A \to R\, A$

$(\_ := \_) : \mathit{Ref}\, A \to A \to R\, 1$

These operations should behave as expected, so that reading a value should be *dsef*, after writing to a reference, it should hold this value, and writing to a reference should not interfere with existing values of distinct references.

Axioms

$\mathrm{dsef}(*r)$ (dsef-read)

$[r := x]\, x = *r$ (read-write)

$\langle r := x \rangle\, \top$ (write-term)

$(x = *r) \Rightarrow [s := y]\, (x = *r \vee r = s)$ (read-write-other)

4.4 Correctness of a Breadth-First Search Algorithm

Breadth-first search is a commonly used, if memory intensive, technique for finding an element in a tree satisfying a certain condition ([31]). Basically, this algorithm will be defined in the previously axiomatised *queue monad* Q, which is extended so as to include a single *reference of type* A which will be used to store elements of a tree over A. Although in finite trees a proper search algorithm will always terminate, its canonical definition requires the existence of an iteration construct that resembles the while-loop of imperative languages. This iteration construct is practically by definition not interpretable by a total function – as is known it is the basic source of nontermination in simple imperative languages. Therefore we will assume for this section that the underlying monad allows the interpretation of arbitrary recursive definitions, e. g. via fixed point recursion on cpos. Although quite a far-reaching condition, there exist monads that allow the interpretation of all operations used in this section. Moreover we assume existence of a classical type of truth values *Bool* that is needed to interpret the *if-then-else* construct in the usual manner – this requirement is of course not necessary if the underlying logic is classical, so that *Bool* is the type of truth values anyway. In this vein we can recursively define a while-loop in the following way:

$\mathit{while} : Q\, \mathit{Bool} \to Q1 \to Q1$

$\mathit{while}\ b\ p = \mathsf{do}\,\{x \leftarrow b; \mathsf{if}\ x\ \mathsf{then}\ \mathsf{do}\,\{p; \mathit{while}\ b\ p\}\ \mathsf{else}\ \mathit{ret}\ *\}$

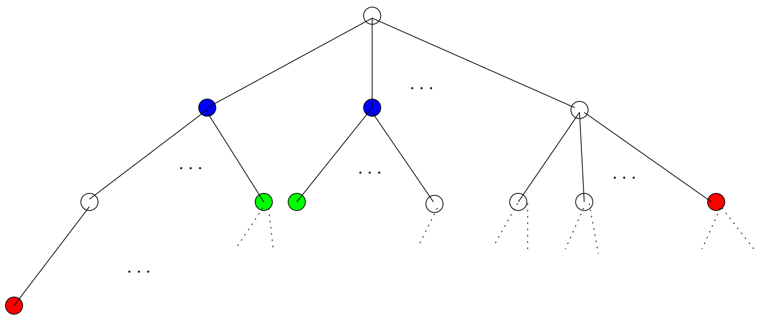

Figure 4.1: Graphical representation of a finitely branching tree; identically coloured nodes represent direct neighbours in the sense of $\prec_1$

The algorithm whose correctness will be verified is then defined as

$$
\begin{array}{l}
\mathit{bfs} : (A \to \mathit{Bool}) \to \mathit{Tree}\ A \to Q\ 1 \\
\mathit{bfs}\ p\ r = \mathsf{do}\ \{ \\
\qquad x := \mathit{inl}\ *; \\
\qquad \mathit{enq}\ r; \\
\qquad \mathit{while}\ (*x = \mathit{inl}\ * \wedge \neg \mathit{empty}) \\
\qquad\qquad \mathsf{do}\ \{\ t \leftarrow \mathit{deq}; \\
\qquad\qquad\quad \mathsf{if}\ (p\ t)\ \mathsf{then}\ x := \mathit{inr}\ t \\
\qquad\qquad\qquad \mathsf{else}\ \mathit{enqAll}\ (\mathit{chld}\ t) \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

where *enqAll* is a primitive recursive function that simply inserts all given elements into the queue:

$$
\begin{array}{ll}
\mathit{enqAll}\ [] & = \mathit{ret}\ * \\
\mathit{enqAll}\ (x : \mathit{xs}) & = \mathsf{do}\ \{\ \mathit{enq}\ x; \mathit{enqAll}\ \mathit{xs}\ \}
\end{array}
$$

To keep the discussion independent of a concrete implementation of a tree of elements of type A, we simply assume its existence as well as some kind of *access function chld* returning a list of a tree's child nodes. *inl* and *inr* are the usual left and right injections for sum datatypes, while $*$ is the single inhabitant of the unit datatype 1, so that x is a reference over values of $1 + \mathit{Tree}\ A$. In what follows we will talk about a fixed, yet arbitrary *finite tree* r.

The typical property of breadth-first search is that it 'finds' the shallowest node in the tree r satisfying the property p, i. e. in our case it assigns this node to the reference x. Therefore, we impose an order $\prec_1$ on the elements of the tree by defining a subtree t_1 to *directly precede* a subtree t_2 (written $t_1 \prec_1 t_2$) iff t_1 lies on the same level as t_2 does, i. e. has the same depth, and the former is its direct left-hand neighbour, or t_1 is the rightmost element in some level n and t_2 is the leftmost element in level $n+1$ (with respect to a graphical representation depicted in Figure 4.1). By taking the transitive closure $\prec$ of $\prec_1$

$$t \prec t' \quad \equiv_{\mathrm{def}} \quad \exists t_1 \ldots t_n.\ t \prec_1 t_1 \wedge t_1 \prec_1 t_2 \wedge \cdots \wedge t_n \prec_1 t'$$

we obtain a means to say that a subtree t *precedes* some other subtree t'. From these definitions, it is clear that

$$t_1 \prec t_2 \wedge \neg \exists t \in r.\ t_1 \prec t \prec t_2 \quad \text{iff} \quad t_1 \prec_1 t_2$$

To put it formally, our goal will then be to prove

$$(\exists t \in r.\ p\ t) \Rightarrow [bfs\ p\ r](*x = inr\ t_0 \wedge (p\ t_0) \wedge \forall t \in r.\ p\ t \Rightarrow t = t_0 \vee t_0 \prec t)$$

where, in the following we will be a bit sloppy about the value of $*x$ and use $*x$ in place of the tree t if $*x = inr\ t$, and say $*x = *$ if actually $*x = inl\ *$. This will not lead to ambiguities, since no tree is of the form $*$.

Remark 4.12. One has $t_1 \prec t_2 \Rightarrow \forall c_1 \in chld\ t_1, c_2 \in chld\ t_2.\ c_1 \prec c_2$, which is immediate from the definition of relation $\prec$. Also, for each tree t where $chld\ t = [c_1, \ldots, c_n]$ it is clear that $c_i \prec_1 c_{i+1}$ for $1 \leq i < n$.

In order to reason about the contents of the current queue, we need two additional monadic predicates $relq : (A \to A \to \Omega) \to Q\ \Omega$ and $inq : A \to Q\ \Omega$ which intuitively state that a given relation holds for adjacent elements in the queue, respectively that an element is contained in the queue. One could define these predicates by means of the iteration construct *iter* for which an inference rule exists (see [34]). In this case, however, the definitions as well as the proofs involving them become quite unwieldy. We therefore take another approach and axiomatise one further deterministically side-effect free operation *get*, which lets us *look inside* the queue by returning a list of all elements in the queue. We will use notation $(x : xs)$ for a list with head x and tail list xs as well as $(xs \uparrow x)$ for a list with endmost element x and initial part xs.

Axioms

$\mathrm{dsef}(get)$	(dsef-get)
$(get = xs) \Rightarrow [enq\ x]\,(get = (xs \uparrow x))$	(enq-app)
$(get = (x : xs)) \Rightarrow [y \leftarrow deq]\,(x = y \wedge get = xs)$	(deq-tl)
$empty \iff (get = [])$	(empty-nil)

An essential operation on queues we will need in our correctness proof is *last*. With *get* available, this is just an abbreviation, assuming there is a function *lst* on lists that returns the last element in the list:

$$(x = last) \quad \equiv_{\mathrm{def}} \quad x = (lst\ get)$$

Obviously, one has $(get = xs \uparrow z) \Rightarrow (last = z)$.

Definition 4.13 (*relq* and *inq*).

$$relq\ R \quad \equiv_{\mathrm{def}} \quad get = q \Rightarrow (\forall i.\ 0 \leq i < len\ q - 1 \Rightarrow q^i R q^{i+1}) \tag{4.7}$$

$$inq\ x \quad \equiv_{\mathrm{def}} \quad get = q \Rightarrow (\exists i.\ 0 \leq i < len\ q \wedge q^i = x) \tag{4.8}$$

Where q^i denotes the i-th element of the list q, with the count starting at zero.

The main problem, as often encountered in proofs involving a while-loop, is to establish a loop invariant, i. e. a condition that holds before the loop and is re-established at each iteration of the loop. Figure 4.2 shows the invariant for the while loop of the *bfs* algorithm. The first thing to remain invariant is the in-queue relation $relq\ \prec_1$, as we will see. This makes sure that all items in the tree are searched ‘in order’. Furthermore, if x has not been assigned a

$$
\begin{array}{llll}
 & relq \prec_1 & & \\
\wedge & *x = * & \Rightarrow & \neg empty \wedge \quad [t \leftarrow deq](NF(t) \wedge CIN(t)) \\
 & & & \vee (empty \wedge \neg \exists t \in r.\ p\ t) \\
\wedge & \neg(*x = *) & \Rightarrow & p *x \wedge \forall t \in r.\ p\ t \Rightarrow *x = t \vee *x \prec t
\end{array}
$$

Figure 4.2: Loop invariant *INV* for the proposed breadth-first search

value, there are two cases: either the queue is empty, in which case there is no element in the tree satisfying p (which would contradict the assumptions), or the queue is not empty and two conditions hold, abbreviated as follows:

$$NF(t) \equiv_{\mathrm{def}} \forall t' \in r.\ t' \prec t \Rightarrow \neg p t' \tag{4.9}$$

$$CIN(t) \equiv_{\mathrm{def}} \forall c \in r.\ inq\ c \iff \exists t' \in r.\ c \in chld\ t' \wedge t' \prec t \prec c \tag{4.10}$$

$[t \leftarrow deq]NF(t)$ states that for all elements preceding t property p does not hold, and $[t \leftarrow deq]CIN(t)$ states that the elements in the queue are exactly the children of elements t' preceding t, whose children are preceded by t. Finally the case $\neg(*x = *)$ must be considered, where it is said that $p *x$ holds and all elements before $*x$ do not have property p.

4.4.1 Basic Facts

Before providing the proof, we note some basic facts we will use later on.

Lemma 4.14. *In a non-empty queue, enqAll and deq may be swapped:*

$$
\begin{array}{ll}
 & \neg empty \wedge [enqAll\ xs][t \leftarrow deq]\varphi \\
\iff & \neg empty \wedge [t \leftarrow deq][enqAll\ xs]\varphi
\end{array}
$$

Proof. By induction on the structure of *xs*. In the base case, $xs = []$, by (dsef□) we have $[ret\ *]\varphi \iff \varphi$ and thus $[enqAll\ xs]\varphi \iff \varphi$ by the definition of *enqAll*. So the base case is trivially true.

In the inductive step, let $xs = (y : ys)$, so we need to show

$$
\begin{array}{ll}
 & \neg empty \wedge [enq\ y; enqAll\ ys][t \leftarrow deq]\varphi \\
\iff & \neg empty \wedge [t \leftarrow deq][enq\ y; enqAll\ ys]\varphi
\end{array}
$$

By the inductive hypothesis, the left-hand part of the formula can be equivalently reformulated as $\neg empty \wedge [enq\ y][t \leftarrow deq][enqAll\ ys]\varphi$ and then, by axiom (swap) this is equivalent to the right-hand side of the formula □

Lemma 4.15. *Under the stated conditions, we can add an element into the queue without losing property relq R:*

$$
\begin{array}{ll}
(i) & \neg empty \wedge last\ R\ x \wedge relq\ R \Rightarrow [enq\ x]relq\ R \\
(ii) & empty \Rightarrow [enq\ x]relq\ R
\end{array}
$$

Proof. For (i), we reformulate $\neg empty$ as $get = xs \uparrow y$ (which indeed is an existential statement: there are some xs and y with this property), from which it follows that $last\ R\ x$ is yRx and $relq\ R$ simplifies to $\forall i.\ 0 \leq i < len\ xs \Rightarrow (xs \uparrow y)^i R (xs \uparrow y)^{i+1}$. The latter two formulae are stateless, such that together with axiom (get-app) one has

$$\begin{aligned} get = (xs \uparrow y) \wedge yRx \wedge \forall i.\ 0 \leq i < len\ xs \Rightarrow (xs \uparrow y)^i R (xs \uparrow y)^{i+1} \Rightarrow \\ [enq\ x]\, get = (xs \uparrow y \uparrow x) \wedge yRx \wedge \forall i.\ 0 \leq i < len\ xs \Rightarrow (xs \uparrow y)^i R (xs \uparrow y)^{i+1} \end{aligned}$$

where the formula in the scope of the box operator implies $relq\ R$, which finishes the proof by an application of rule (wk□).

Concerning (ii), the conclusion is obvious from the premiss and the definition of *get* and *relq*. □

Remark 4.16. One can generalise Lemma 4.15 in the sense that it is also possible to insert lists of items $[x_1, \ldots, x_n]$ for all $n \in \mathbb{N}$ if $x_i R x_{i+1}$ for $i \in \{1, \ldots, n-1\}$ and x_1 may be enqueued without breaking the relation $relq\ R$. The proof thereof proceeds by structural induction on the to-be-inserted list.

Lemma 4.17. *If the relation R holds in the queue, i. e. relq R, then after removing one element, R still holds:* $relq\ R \Rightarrow [x \leftarrow deq] relq\ R$.

Proof. For $get = []$, the formula holds trivially, so assume $get = (y : ys)$. From the definition of *relq*, we can deduce $\forall i.\ 0 \leq i < len (y : ys) - 1 \Rightarrow (y : ys)^i R (y : ys)^{(i+1)}$, so in particular R holds for all adjacent elements in ys. By (deq-tl) we obtain the desired result. □

Lemma 4.18. *After inserting some elements xs into the queue, for each* $x \in xs$ *we have inq x. Put formally:*

$$[enqAll\ xs](\forall x \in xs.\ inq\ x) \quad \textit{for all lists } xs$$

Proof. Since *get* is dsef and thus always defined, we always have $get = ys$ for some list ys. Now as usual we proceed by induction on the structure of xs and leave out the base case, where *enqAll* does nothing and there are no elements to make a statement about. So let $xs = (x' : xs')$. It then follows by (get-app) that $[enq\ x'](get = (ys \uparrow x'))$ and so $[enq\ x'](inq\ x')$. By the induction hypothesis we have

$$[enqAll\ xs'](\forall x \in xs'.\ inq\ x)$$

and by application of (nec) we obtain

$$[enq\ x'][enqAll\ xs'](\forall x \in xs'.\ inq\ x)$$

The missing ingredient for finishing the proof is

$$inq\ x \Rightarrow [enqAll\ xs] inq\ x \quad \text{for all } x \text{ and } xs$$

But this fact is again provable by induction on the mentioned xs and follows quite directly. Altogether we arrive at

$$[enq\ x'][enqAll\ xs'](\forall x \in xs'.\ inq\ x \wedge inq\ x')$$

which actually is what we claimed, recalling that $(x' : xs') = xs$ □

Lemma 4.19. *If the relation $\prec$ (or in fact any other strict partial order) holds in the queue, then after removing an element x from it, there is no element y in the queue with $x = y$*

$$relq \ \prec \quad \Rightarrow \quad [x \leftarrow deq](\neg inq\ x)$$

Proof. We only need to consider the case where $get = (y : ys)$. Assuming $relq\ \prec$ amounts to saying that

$$\forall i.\ 0 \leq i < len\ (y : ys) - 1 \Rightarrow (y : ys)^i \prec (y : ys)^{i+1} \tag{4.11}$$

holds. By (deq-tl), after dequeuing only the ys remain in the queue:

$$get = (y : ys) \Rightarrow [x \leftarrow deq](get = ys) \tag{4.12}$$

Noting that $\prec$ is a transitive and irreflexive relation (i. e. $\forall xyz.\ x \prec y \wedge y \prec z \Rightarrow x \prec z$ and $\forall x.\ x \not\prec x$) we may by (4.11) infer that there is no y' in ys such that $y' = y$. But then, by (4.12), we are already done: after dequeuing x, the ys remain, in which there is no element equal to x. □

Lemma 4.20. *Dequeuing an element does not affect existence of other elements inside the queue:*

$$inq\ x \Rightarrow [y \leftarrow deq](x = y \vee inq\ x)$$

Proof. For $get = []$, $inq\ x$ is obviously false for every x. For $get = [x_1, \ldots, x_n]$, assuming $inq\ x$ amounts to saying that there is an $x_i = x$ for some i, $1 \leq i \leq n$. By (deq-tl) have $[y \leftarrow deq](y = x_1 \wedge get = [x_2, \ldots, x_n]$ and thus for $x = x_1$ have $[y \leftarrow deq](x = y)$ whereas for $x \neq x_1$ – i. e. $x = x_i$ for $1 < i \leq n$ – have $[y \leftarrow deq](inq\ x)$, so altogether $[y \leftarrow deq](x = y \vee inq\ x)$ (cf. also Lemma 4.6). □

4.4.2 Auxiliary Rules

In merging the specifications of the queue monad and the reference monad, a typical frame-problem arises: The question *'what remains the same in a changing world?'* can be instantiated here as *'what happens to references if we modify the queue?'* The answer will certainly be *'nothing'*, which we formalise as follows.

$$(x = *r) \Rightarrow [qop](x = *r) \quad \text{for } qop \in \{deq, enq, empty\} \tag{4.13}$$

The simplest way to answer the converse question *'what happens to the queue if we modify a reference?'* is by relating get to reference writing:

$$(get = xs) \Rightarrow [r := x](get = xs) \tag{4.14}$$

Reference to one of these axioms will be indicated by (frame).

In [34] a Hoare calculus for total correctness has been developed, in which Hoare rules such as

$$\textbf{(seq)} \quad \frac{[\varphi]\bar{x} \leftarrow \bar{p}[\psi] \quad [\psi]\bar{y} \leftarrow \bar{q}[\chi]}{[\varphi]\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}[\chi]}$$

appear. It has been said in Section 3.3.1 that a Hoare rule $[\varphi]\bar{x} \leftarrow \bar{p}[\psi]$ is meant to be interpreted as $\varphi \Rightarrow (\langle \bar{x} \leftarrow \bar{p} \rangle \top \wedge [\bar{x} \leftarrow \bar{p}]\psi)$. In this way, partial correctness as well as termination of a program sequence $\bar{p}$ and thus total correctness are concisely captured.

Because we are working with formulae of dynamic logic and do not want to switch into the Hoare calculus, yet we would like to use the results of the latter, we simply translate some Hoare rules of [34] back into rules for dynamic logic.

$$(\textbf{dsef1})\quad \frac{p \text{ dsef}}{\varphi \Rightarrow [p]\varphi} \qquad (\textbf{if})\quad \frac{\begin{array}{c} b \text{ dsef} \\ \varphi \wedge b \Rightarrow [x \leftarrow p]\psi \\ \varphi \wedge \neg b \Rightarrow [x \leftarrow q]\psi \end{array}}{\varphi \Rightarrow [x \leftarrow \mathsf{if}\ b\ \mathsf{then}\ p\ \mathsf{else}\ q]\psi}$$

$$(\textbf{while})\quad \frac{\begin{array}{c} t : DB \\ \_ < \_ : B \times B \to \Omega \text{ is well-founded} \\ \varphi \wedge b \Rightarrow \langle p \rangle \top \\ (\varphi \wedge b \wedge t =_B z) \Rightarrow [p](\varphi \wedge t < z) \end{array}}{\varphi \Rightarrow [\mathit{while}\ b\ p]\,(\varphi \wedge \neg b) \wedge \langle \mathit{while}\ b\ p \rangle \top}$$

In rule (while) termination is ensured by letting the term t decrease strictly in every iteration. Since $<$ is well-founded, it is impossible for the final premiss to be true infinitely often. The so called *ghost variable* $z : B$ does not appear within the program and simply serves the purpose of relating the value of t before and after execution of p. In particular, t is not equal to z as a computation, but rather its value equals z.

Now we are equipped with all we need to prove total correctness of the program *bfs*, in particular – as can be seen from the rule for while – termination of the while-loop.

4.4.3 Proof of Total Correctness

In what follows, we try not to be too formalistic and therefore make reference to common laws such as transitivity of equivalence or other obvious validities without proving them for each separate instance. We further assume that the underlying formalism is classical, i. e. we allow reasoning by case distinction over some formula $\phi \vee \neg\phi$. In a Hilbert-style calculus with essentially only modus ponens available as an inference rule, methods such as proof by contradiction are to be conceived as first proving $\neg P \Rightarrow \mathsf{False}$ and then applying (mp) to the tautologous $(\neg P \Rightarrow \mathsf{False}) \Rightarrow P$. Likewise, substitutivity of equivalence makes use of the tautology scheme $(P \iff Q) \Rightarrow R[P/x] \Rightarrow R[Q/x]$.

It will now first be established that INV, the loop invariant, holds before the while loop, i. e. with $PRE \equiv_{\text{def}} \exists t \in r.\ p\ t$ (a stateless formula) we show

$$PRE \wedge empty \Rightarrow [x := *; enq\ r](INV) \tag{4.15}$$

By (read-write) and (frame)

$$[x := *; enq\ r](*x = *) \tag{4.16a}$$

From the definition of *relq* we can infer

$$empty \Rightarrow [enq\ r](relq\ \prec) \tag{4.16b}$$

which by (frame) can be extended to

$$empty \Rightarrow [x := *; enq\ r](relq\ \prec) \tag{4.16c}$$

Again with (frame), (enq-deq) gives us

$$empty \Rightarrow [x := *; enq\ r][t \leftarrow deq](r = t \wedge empty) \tag{4.16d}$$

Now from $r = t$ we can deduce $NF(t)$, because there simply is no element $t' \prec r$ in r. Similarly, we infer $CIN(t)$ because $inq\ c$ is false for every element in r and again there is no element $t' \prec r$, so the equivalence in CIN holds. Combining (4.16a), (4.16c) and (4.16d) we obtain the desired result.

The while Rule The next step is to gather the premisses of the (while) rule as stated above to draw the conclusion of selfsame. The premiss $INV \wedge *x = * \wedge \neg empty \Rightarrow \langle body \rangle \top$ asserting termination of the loop body *body* is quite obvious, since the only source of non-termination is the *deq*-operation, which will however only be executed if the queue is not empty. The formalisation of this argument can be conducted along the lines of the following proof of the most integral part:

$$\begin{aligned} &INV \wedge *x = * \wedge \neg empty \wedge vol = z \Rightarrow \\ &\quad [t \leftarrow deq; \mathsf{if}\ p\ t\ \mathsf{then}\ x := inl\ t\ \mathsf{else}\ enqAll\ chld\ t](INV \wedge vol < z) \end{aligned} \tag{4.17}$$

where we introduce the termination measure *vol* which computes the total number of elements reachable from any subtree contained in the queue. Employing the list functions $sum : [Nat] \to Nat$ and $map : (A \to B) \to [A] \to [B]$ – whose definitions are straightforward and can be found, e. g., in the Haskell Prelude – it might be defined like this:

$$\begin{aligned} &vol : Q\ Nat \\ &vol = \mathsf{do}\ \{\ q \leftarrow get; \\ &\qquad\qquad ret\ sum\ (map\ volume\ q) \\ &\quad \mathsf{where}\ volume : Tree\ A \to Nat \\ &\qquad\qquad volume\ t = 1 + sum\ (map\ volume\ (chld\ t)) \end{aligned}$$

The intuition behind this approach is that the overall volume of the queue must strictly decrease after dequeuing some subtree t and enqueuing its children, because the volume of t is defined to be by 1 larger than the sum of volumes of its children. *vol* is a dsef operation since it is composed solely of dsef operations (it has been shown in Isabelle that dsef programs are stable under composition).

We note the following equivalence which we shall use for simplification purposes and whose right-hand part we will denote by SI.

$$\begin{aligned} &INV \wedge *x = * \wedge \neg empty \\ \Longleftrightarrow\quad &relq\ \prec_1 \wedge \neg empty \wedge [t \leftarrow deq](NF(t) \wedge CIN(t)) \wedge *x = * \end{aligned} \tag{4.18}$$

By Lemma 4.17 we have

$$SI \Rightarrow [t \leftarrow deq](relq\ \prec_1)$$

so by (frame) *relq* still holds after assignment to x:

$$SI \Rightarrow [t \leftarrow deq][x := t](relq\ \prec_1) \tag{4.19}$$

Then-branch Working our way through the then-branch of the loop body, we also need the next statement. This is obtained from (read-write) and the fact that NF and $p\,t$ are stateless.

$$NF(t) \wedge p\,t \Rightarrow [x := t](*x = t \wedge p\,*x \wedge NF(t)) \tag{4.20}$$

Now, $NF(t) \wedge p\,*x \wedge *x = t$, i. e. that all elements in the tree smaller than t do not have property p, but t and therefore $*x$ does, can be reformulated as $p\,*x \wedge \forall t \in r.\ p\,t \Rightarrow *x = t \vee *x \prec t$.

In combining (4.19) and (4.20) we obtain the following, where the formula in the scope of the $[x := t]$ box is in fact stronger than INV

$$\begin{aligned} & relq \prec_1 \wedge NF(t) \wedge p\,t \\ \Rightarrow\ & [x := t](*x = t \wedge p\,*x \wedge relq \prec_1 \\ & \qquad \wedge (\forall t' \in r.\ p\,t \Rightarrow *x = t' \vee *x \prec t') \end{aligned} \tag{4.21}$$

Else-branch Because all ingredients needed for the then-part are now assembled, we turn our eyes to the else-part, which actually is the harder one. ‘Inside’ the $[t \leftarrow deq]$ box of (4.17) we have $CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge *x = *$. We will, in accordance with the if-rule, furthermore assume $\neg p\,t$ and prove the following, in which again the formula inside the $[enqAll\ (chld\ t)]$ box implies INV

$$\begin{aligned} & CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge \neg p\,t \wedge *x = * \\ \Rightarrow\ & [enqAll\ (chld\ t)](relq \prec_1 \wedge *x = * \\ & \quad \wedge (\neg empty \wedge [t' \leftarrow deq](NF(t') \wedge CIN(t')) \\ & \qquad \vee (empty \wedge \neg\exists t'' \in r.\ p\,t''))) \end{aligned} \tag{4.22}$$

This can by Lemma 4.1 be done in three steps, each asserting the truth of the above formula reduced to one of the three conjunct clauses in the scope of the enqAll box.

Part i

$$*x = * \quad \Rightarrow \quad [enqAll\ (chld\ t)]\ (*x = *)$$

Now this is an obvious generalisation of one of the (frame) axioms.

Part ii

$$\begin{aligned} & CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge \neg p\,t \wedge *x = * \\ \Rightarrow\ & [enqAll\ (chld\ t)]\ (relq \prec_1) \end{aligned}$$

This formula asserts that we may enqueue t’s children without destroying the relation $relq \prec_1$ inside the queue. For $chld\ t = []$ we must then prove

$$\ldots \wedge relq \prec_1 \wedge \ldots \Rightarrow [ret\ *]\, relq \prec_1$$

which essentially is given by (ret□). So let $chld\ t = (x : xs)$. Then by Remark 4.16 all children may be inserted through $enqAll$ without invalidating $relq \prec_1$ if x may be enqueued through enq. For $empty$ this is clearly true, so consequently we’ll add the premiss $\neg empty$. Then $CIN(t)$ tells us $inq\ c$ holds for exactly all the child elements c of predecessors of t. Thus $last \prec x$ certainly holds (cf. Remark 4.12). Because $\neg\exists a \in r.\ last \prec a \prec x$, even $last \prec_1 x$ is true, providing all the premisses of Lemma 4.15 and letting us draw the desired conclusion. $\neg\exists a \in r.\ last \prec a \prec x$ can be shown by contradiction: assume $\exists a \in r.\ last \prec a \prec x$; Then it directly follows that there is t'' such that $a \in chld\ t''$ and $t'' \prec t$ ($t'' = t$ cannot be the case since $a \prec x$, and for the same reason $t \prec t''$ neither). But then, because of $CIN(t)$, $inq\ a$ holds, which together with $last \prec a$ violates the given premiss $relq \prec_1$. We conclude that part ii is true.

Part iii

$$\begin{aligned} & CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge \neg p\ t \wedge *x = * \\ \Rightarrow\ & [enqAll\ (chld\ t)]\,(\neg empty \wedge [t' \leftarrow deq](NF(t') \wedge CIN(t')) \\ & \quad \vee (empty \wedge \neg\exists t'' \in r.\ p\ t'')) \end{aligned}$$

This part makes sure that after inserting t's child elements we either have seen each element in the tree and none satisfies p, or there are elements left and after dequeuing another element t' all its predecessors don't have property p and the elements remaining in the queue are exactly the children of predecessors of t', which themselves are succeeding t'.

We proceed by case distinction over $empty \vee \neg empty$. We have

$$empty \Rightarrow [enqAll\ (chld\ t)]\ empty \quad \text{iff} \quad chld\ t = []$$

But in this case, i. e. when $empty$ holds in the box, t must be the final element in the tree r since all children of predecessors would otherwise be in the queue (by CIN). Extend $NF(t)$ and $\neg p\ t$ to $\neg\exists t'' \in r.\ p\ t''$ and obtain $[enqAll\ (chld\ t)](empty \wedge \neg\exists t'' \in r.\ p\ t'')$ making the conclusion of part (iii) true. For $chld\ t = (x : xs)$ one has $empty \Rightarrow [enqAll\ (chld\ t)](\neg empty)$. Here, $t \prec_1 x$ must hold, i. e. t's first child element is its direct successor, because no element before t has child elements that are in the queue by $CIN(t) \wedge empty$. Now

$$[enq\ x; enqAll\ xs][t' \leftarrow deq](NF(t') \wedge CIN(t')))$$

is by Lemma 4.14 equivalent to

$$[enq\ x; t' \leftarrow deq][enqAll\ xs](NF(t') \wedge CIN(t'))$$

and because of (enq-deq) one has:

$$empty \Rightarrow [enq\ x; t' \leftarrow deq; enqAll\ xs](x = t')$$

So it suffices to prove the implication

$$\ldots \Rightarrow [enq\ x; t' \leftarrow deq][enqAll\ xs](CIN(t') \wedge NF(t'))$$

where $\ldots$ denotes the premisses $\neg pt$, $t \prec_1 x$, $CIN(t)$, $NF(t)$ and $empty$.

The NF part is fairly easy to see: one certainly has $NF(t) \wedge t \prec_1 t' \wedge \neg pt$ inside the box, which implies $NF(t')$, where t' replaced x due to their being equal. $CIN(t')$, which decodes into $CIN(t') \equiv_{\text{def}} \forall c \in r.\ inq\ c \iff \exists t'' \in r.\ c \in chld\ t'' \wedge t'' \prec t' \prec c$, is true due to the fact that exactly the xs are in the queue, and for each $x' \in xs$ we have $x' \prec x$. That finishes the case where $empty$ is true.

Now for the case where $\neg empty$ is taken as a premiss and – to restate the other ones – $CIN(t)$, $NF(t)$, $relq\ \prec_1$ and $\neg pt$. Obviously one then has $\ldots \Rightarrow [enqAll\ (chld\ t)](\neg empty)$, so it remains to be proved that

$$\ldots \Rightarrow [enqAll\ (chld\ t)][t' \leftarrow deq](NF(t') \wedge CIN(t'))$$

or, equivalently and quite similar to the case above we can show

$$\ldots \Rightarrow [t' \leftarrow deq][enqAll\ (chld\ t)](NF(t') \wedge CIN(t'))$$

For $NF(t')$ alone, this can be done if $\ldots \Rightarrow [t' \leftarrow deq](t \prec_1 t')$ can be shown, because unlike $CIN(t')$, $NF(t')$ is indeed a stateless formula about a property of the tree r and not about the monadic queue. Hence $NF(t') \Rightarrow [enqAll\ (chld\ t)](NF(t'))$ by (K3$\Box$). For the same reason, however, $NF(t)$ holds after execution of *deq*: $NF(t) \Rightarrow [t' \leftarrow deq](NF(t))$ so that at least for $NF(t')$ the proof goes through: we have

$$\ldots \Rightarrow [t' \leftarrow deq](NF(t) \wedge t \prec_1 t') \tag{4.23}$$

because the direct successor of t must be in the queue, asserted by $CIN(t)$ together with $\neg empty$, and it must be 'the next one to drop out of it', given by $relq\ \prec_1$. From this and $\neg pt$ we infer

$$[t' \leftarrow deq](NF(t'))$$

And then by the argument given above

$$\ldots \Rightarrow [t' \leftarrow deq][enqAll\ (chld\ t)](NF(t'))$$

Continuing with the premisses $\neg empty$ and $CIN(t) \wedge NF(t)$, $relq\ \prec_1$ and $\neg pt$ we will now show the final piece of the puzzle, viz. that these imply

$$[t' \leftarrow deq][enqAll\ (chld\ t)](CIN(t')) \tag{4.24}$$

We proceed as follows; let $get = [x_1, \ldots, x_n]$, $n \geq 1$. By Lemma 4.19 and fact (4.23) we have

$$\ldots \Rightarrow [t' \leftarrow deq](\neg inq\ t' \wedge get = [x_2, \ldots, x_n] \wedge t \prec_1 t' \wedge t' = x_1)$$

$CIN(t)$ tells us that the x_i ($1 \leq i \leq n$) are exactly those elements for which $x_i \in chld\ t_i \wedge t_i \prec t \prec x_i$ is true for appropriate t_i. With $t \prec_1 t'$ it is clear that all elements c satisfying $c \in chld\ t_i \wedge t_i \prec t' \prec c$ for appropriate t_i are $x_2, \ldots, x_n$ (a possibly empty sequence) plus the child elements of t (pointing out that t' cannot be a child of t because $t' = x_1$ and therefore is a child of some predecessor of t by $CIN(t)$). With $chld\ t = [c_1, \ldots, c_k]$ one has by structural induction

$$get = [x_1, \ldots, x_n] \Rightarrow [t' \leftarrow deq][enqAll\ (chld\ t)](get = ((\ldots([x_2, \ldots, x_n] \uparrow c_1) \uparrow \ldots) \uparrow c_k))$$

or slightly more readable

$$[t' \leftarrow deq][enqAll\ (chld\ t)](get = [x_2, \ldots, x_n, c_1, \ldots, c_k])$$

from which we conclude by the foregoing argument that for the given premisses we can show

$$\ldots \Rightarrow [t' \leftarrow deq][enqAll\ (chld\ t)](CIN(t'))$$

Assembling the Results

We may finally apply rule (if) to formulae (4.21) and (4.22) repeating that in both ones, the sub-formulae inside the boxes imply *INV*

$$\begin{aligned} & CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge *x = * \\ \Rightarrow\ & [\mathsf{if}\ p\ t\ \mathsf{then}\ x := t\ \mathsf{else}\ enqAll\ (chld\ t)]\,(INV) \end{aligned} \tag{4.25}$$

Referring to (4.18), we can say

$$SI \Rightarrow [t \leftarrow deq](CIN(t) \wedge NF(t) \wedge relq \prec_1 \wedge *x = *) \tag{4.26}$$

Regarding the decrease in volume, which has silently been passed over until now, one has $\neg empty \iff get = [x_1,\dots,x_n]$ for some elements x_i and some n and thus by (deq-tl) and the definition of *vol* resp. *volume*

$$\begin{aligned} & volume\ x > 0 \\ & SI \wedge get = [x_1,\dots,x_n] \wedge vol = z \\ \Rightarrow\ & [t \leftarrow deq](get = [x_2,\dots,x_n] \wedge vol = (z - volume\ x_1)) \\ & \text{so by (frame)} \\ & SI \wedge get = [x_1,\dots,x_n] \wedge vol = z \\ \Rightarrow\ & [t \leftarrow deq; x := t](vol < z) \end{aligned} \tag{4.27}$$

Now in addition let $chld\ t = [c_1,\dots,c_k]$ such that after enqueuing these one still has a smaller volume than before dequeuing t, since t's volume is defined to be by one larger than the sum of volumes of its child elements:

$$\begin{aligned} & volume\ t = 1 + \textstyle\sum_{i=1}^{k}(volume\ c_i) \\ & SI \wedge get = [x_1,\dots,x_n] \wedge vol = z \\ \Rightarrow\ & [t \leftarrow deq; enqAll\ (chld\ t)](get = [x_2,\dots,x_n,c_1,\dots,c_k] \wedge vol < z) \end{aligned} \tag{4.28}$$

Having ascertained the termination of the loop by (4.27), (4.28), we apply rule (wk□) to (4.25), (4.26) to finally verify the premisses of rule (while) (cf. (4.17)) and thus conclude

$$INV \Rightarrow [while\ cond\ prog](INV \wedge (x \neq * \vee empty))$$

$$\begin{aligned} \text{where}\quad cond &= x = * \wedge \neg empty \\ prog &= t \leftarrow deq; \mathsf{if}\ p\ t\ \mathsf{then}\ x := t\ \mathsf{else}\ enqAll\ (chld\ t) \end{aligned}$$

The definitely last step is now to derive the postcondition

$$(p *x \wedge \forall t \in r. p\ t \Rightarrow *x = t \vee *x \prec t)$$

from what the while loop left us with:

$$(INV \wedge (x \neq * \vee empty))$$

but this can be done easily, recalling that the stateless formula warranting existence of an element satisfying p still holds after execution of *bfs*

$$(\exists t \in r.\ p\ t) \Rightarrow [bfs\ p\ r](\exists t \in r.\ p\ t)$$

□

5 The Theorem Prover Isabelle

Isabelle is an interactive theorem proving environment, i. e. an assistant for performing formal proofs. The fact that Isabelle is generic in the sense that it allows one to define and reason within several kinds of logics distinguishes it from most other proof assistants. Examples of logics that have been defined within Isabelle's framework are classical first-order logic (FOL), constructive type theory (CTT), or higher-order logic (HOL) which constitutes the base logic in our development of monadic dynamic logic.

We will now introduce the foundations of Isabelle which are the so called meta-logic, its syntax and inference rules. We then introduce higher-order logic as formalised in Isabelle. Finally, we provide insight into basic proof methods whose knowledge is necessary to comprehend or at least read printed Isabelle proofs. A full account of all facilities that were applied cannot be given in this thesis; very readable introductions to Isabelle and Isabelle/Isar can be found in [22, 23]

But first, a note about terminology and the development of Isabelle is in order: Initially, communicating with Isabelle meant sequentially applying ML functions, since Isabelle is written in this functional language. This user interface has recently been discharged in favour of an independent proof and theory language called *Isar*, making proofs substantially more readable (and maintainable). The combination of Isabelle with Isar is named Isabelle/Isar, which becomes Isabelle/Isar/HOL when referring to the specific logic HOL, expressed in Isar. In the following, we will often use the term Isabelle for all these phrases, stating once and for all that the formal proofs in this thesis are presented in Isabelle/Isar with HOL as the underlying logic.

5.1 The Meta-logic

Isabelle lets the user define his own logics, so that he does not have to work within a fixed logic that might not suit his needs. In doing so, one needs some means to express the syntax of one's newly defined logic, to express inference rules, and to impose side conditions on these rules. Take the following natural deduction rule governing the introduction of the $\forall$-quantifier as an example:

$$\frac{P(x)}{\forall x.\, P(x)} \quad (x \text{ not free in assumptions}) \tag{5.1}$$

The annotation 'x not free in ...' is a very typical side condition, while the horizontal bar expresses a possible logical inference from the premisses (displayed above the bar) to the conclusion (below the bar).

Besides determining the basic syntax of all definable logics, it is the task of the *meta-logic* to enable the formulation of such 'meta-logical' constructs, i. e. to formalise properties of concrete object-logics. Put shortly, the meta-logic is an intuitionistic higher-order logic with polymorphic functions in the style of ML or Haskell that possesses a universal quantifier, implication and equality as its constants.

5.1.1 Basic Syntax and Terminology

The meta-logic is syntactically based on the simply typed lambda calculus as described in Section 2.1.3 (although without product types). The additional possibility to define polymorphic functions means that function types may contain *type variables*, e. g. the identity function $id : \alpha \to \alpha$ exists for *every* type α. *Type declarations* allow the introduction of new base types, whereas *type classes* may be seen as collections of types that share some structure (a well-known example is the class *ord*, which the types with a notion of order among their elements belong to). The latter concept comes close to Haskell's type classes, but is not powerful enough to embrace Haskell's constructor classes as well. In particular, the notion of a type constructor being an instance of a monad cannot be specified in Isabelle. A remark about how this problem has been resolved in the implementation can be found in Section 6.2.

Some peculiarities of Isabelle's syntax should be noted before proceeding:

- The base type of truth values is named *prop*.
- Type annotations are denoted by two successive colons instead of one.
- Function types may be built from existing types by means of the function type constructor $\Rightarrow$, such that $f :: \sigma \Rightarrow \tau$ is Isabelle's notation for $f : \sigma \to \tau$. The type constructor $\Rightarrow$ associates to the right.
- The types of curried functions taking n arguments, $f :: \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \sigma$ may be written in a list-like notation $f :: [\sigma_1, \ldots, \sigma_n] \Rightarrow \sigma$.
- Type variables are written as Latin letters prefixed with an apostrophe ($'$), e. g. $'a$, $'b$, $'b_1$ are type variables. Inside normal text we will however not use this style.

The constants of the meta-logic are a universal quantifier, (denoted by the symbol $\bigwedge$), implication ($\Longrightarrow$[1]) and equality ($\equiv$). An interesting property of higher-order logics that spring from the lambda calculus is the fact that no variable binders other than λ are needed: predicates are simply interpreted as functions into truth values (e. g. a predicate on the type *nat* of natural numbers might be expressed as a function $P : nat \to prop$), and quantifiers are interpreted as higher-order functions from predicates to truth values. Thus, the type of the universal quantifier is

$$\bigwedge_\alpha :: (\alpha \Rightarrow prop) \Rightarrow prop \tag{5.2}$$

for each type α; the polymorphism of Isabelle is restricted in the same way as in ML or Haskell in that it does not allow higher-order functions to take polymorphic functions as arguments. This is made explicit here by indexing the quantifier with the appropriate type under consideration.

5.1.2 Defining Logics

Users are not expected to work within the meta-logic itself, but rather to formalise their own logics by extending the meta-logic through the introduction of new types and constants and through axioms capturing the properties of these constants. An example is given in Section

[1] note the difference between this symbol and the shorter one for the function type constructor $\Rightarrow$; both however associate to the right and there also is a list-like notation for repeated implication of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$

5.2, where the formalisation of HOL within the meta-logic is described. The outline of such a formalisation is as follows:

1. Introduce a new type for truth values, thereby distinguishing it from the type of truth values of the meta-logic. Furthermore introduce a predicate *Trueprop* converting from object-level truth to meta-level truth; it has proved sensible to keep these two kinds of truth values apart. Other useful types may be added as well, of course.

2. Name and assign types to the constants that will serve as basic functions of the logic to be defined; examples include propositional connectives $\wedge$, $\longrightarrow$, etc., or even modal operators. It is possible to decorate constants with concrete syntax (by so called *mixfix annotations*, cf. [25]) that makes operations more readable than is possible with the minimalistic syntax of the lambda calculus. One way or the other, functions of the respective object-logic conventionally have higher precedence than those of the meta-logic.

3. Extend the meta-logic by further axioms that capture the properties of these constants and types. The basic idea is that axioms of the meta-logic are to be interpreted as rules in the object logic. For example, the typical rules for conjunction introduction and universal generalisation in first-order logic

$$\frac{P \quad Q}{P \wedge Q} \qquad \frac{P\,x}{\forall x.\,P\,x}\ (x \text{ not free in assumptions})$$

might be formalised as

$$[\![P; Q]\!] \Longrightarrow P \wedge Q \quad \text{and} \quad (\bigwedge x.\,P\,x) \Longrightarrow \forall x.\,P\,x$$

Proofs from rules within the object-logic are then basically proofs from corresponding axioms within the meta-logic.

5.1.3 Meta-logic Rules

To perform such proofs inside the meta-logic, a collection of meta-rules is necessary. These rules are hard-wired into Isabelle, which means they are implemented as ML functions operating on meta-logic terms rather than being terms of the meta-logic itself. A complete exposition of these rules can be found in [24, Section 2.4], which we do not repeat here, since the meta-rules are virtually never applied in proofs inside object-logics. Instead, we merely summarise the rules, giving an idea of the relative compactness of the meta-logic.

The meta-rules can roughly be put into three categories:

1. Introduction and elimination rules for the constants $\bigwedge$, $\Longrightarrow$ and $\equiv$;

2. Rules concerning lambda terms; put concretely, there is a rule for α-conversion, a rule for β-reduction admitting the conclusion $a[b/x]$ from the premiss $(\lambda x.\,a)\ b$, and a rule of extensionality;

3. Finally, there are basic rules for equality.

Constant	*Term*	*written as*
$Not :: bool \Rightarrow bool$	$Not\ P$	$\neg P$
$True :: bool$		
$False :: bool$		
$If :: [bool,'a,'a] \Rightarrow' a$	$If\ b\ p\ q$	$\mathsf{if} b \mathsf{then} p \mathsf{else} q$
$The :: ('a \Rightarrow bool) \Rightarrow' a$	$The\ P$	$THE\ x.\ P\ x$
$All :: ('a \Rightarrow bool) \Rightarrow bool$	$All\ P$	$\forall x.\ P\ x$
$Ex :: ('a \Rightarrow bool) \Rightarrow bool$	$Ex\ P$	$\exists x.\ P\ x$
$Let :: ['a,'a \Rightarrow' b] \Rightarrow' b$	$Let\ t\ \lambda x.\ e$	$\mathsf{let} x = t \mathsf{in} e$
$= :: ['a,'a] \Rightarrow bool$	$a = b$	
$\wedge, \vee, \longrightarrow :: [bool, bool] \Rightarrow bool$	$P \wedge Q$, etc.	

Table 5.1: Constants extending the meta-logic to HOL

5.2 Higher-order Logic (HOL)

In this section we introduce the formalisation of the simply typed higher-order logic HOL. The outstanding feature of higher-order logics is their capability of expressing higher-order functions (in a sense similar to that of functional programming languages), but also of expressing predicates and quantification on arbitrarily typed terms. For example, one may state the property of a set S being infinite by expressing that there is an injective function from S into a proper subset $S' \subset S$:

$$S\ infinite \quad \text{iff} \quad \exists S'.\ S' \subset S \wedge \exists f :: S \Rightarrow S'.\ f\ injective$$

Because of the quantification on the function f this statement is inherently higher-order; it cannot even be expressed equivalently in first-order languages. In HOL all functions are required to be total; an extension incorporating concepts from domain theory that allows the formulation of arbitrary computable functions is HOLCF [21]. For in-depth descriptions of higher-order logic and its implementation in Isabelle, see [1, 23].

5.2.1 Constants

HOL as implemented in Isabelle extends the meta-logic by a number of constants that are to be interpreted as the usual logical connectives, like conjunction, universal quantification, or boolean case distinction (the familiar *if-then-else* construct). Differing from the notation used so far, implication is denoted by a simple long arrow $\longrightarrow$. Some of the operations come in two flavours, namely their functional form (as actual constants in the lambda calculus of the meta-logic) and with some syntactical sugaring; Table 5.1 lists the most important ones. The function *The* is a *definite description operator*; $THE\ x.\ P\ x$ is meant to be interpreted as "the x, such that $P\ x$ holds" and will yield an arbitrary value of the appropriate type if no such x exists. The interpretation of the remaining functions and values is standard, but one should note that quantification exists for arbitrary types, just as equality, *if-then-else* and *let* do.

HOL inherits the ability to express functions as lambda terms from the meta-logic by identifying HOL types and functions with the types and functions of the meta-logic[2]. This way, HOL also exploits Isabelle's built-in type checker, which is a great help in immediately refuting ill-typed expressions. Nonetheless it has its own type of truth values, classically named *bool*. In fact, HOL is a classical logic (as opposed to a constructive or intuitionistic logic) featuring the law of excluded middle (cf. rule *True-or-False* in Table 5.2).

There is an interesting difference between variables in HOL and the more syntactical variables encountered in the definition of logics 'on paper', where a rule of substitutivity of equality might be defined as follows

$$\frac{a = b \quad \phi}{\phi[b/a]} \tag{5.3}$$

In this rule, ϕ is a syntactical variable in the sense that it stands for an arbitrary formula (i. e. a term of type *bool* in HOL), probably containing a as a free variable – otherwise substituting b for a would be pointless. To the contrary, in HOL there is no need for an explicit notion of substitution, and the rule under consideration is expressed as

$$\frac{a = b \quad \phi\ a}{\phi\ b} \tag{5.4}$$

making $\phi :: \sigma \Rightarrow \mathit{bool}$ a function variable provided that $a, b : \sigma$. Here is a simple example to visualise the difference.

Example 5.1. Assuming some proof has reached a state such that $a = b$ and $f\,a = g\,x$ have been proved. In this case, ϕ of (5.3) can be instantiated to $f\,a = g\,x$, whereas ϕ of (5.4) is $\lambda y.\ f\,y = g\,x$. Applying rule (5.4) yields $(\lambda y.\ f\,y = g\,x)\ b$ which can be converted to $f\,b = g\,x$ by the β-rule of the meta-logic.

5.2.2 Definitions

To avoid unnecessary redundancy, logics – including HOL – often only axiomatise the properties of a minimal set of constants, with everything else being defined in the form of abbreviations (the definition of implication through negation and disjunction is a case in point, although in HOL implication is the basic connective). It is here, where the constants of the meta-logic come into play: we may use meta-equality to describe definitions, meta-implication to express rules and the use of meta-quantification is a convenient way to capture many common side conditions. Table 5.2 shows the axiomatisation of HOL as an extension of the meta-logic, where the usual connectives are still missing; their definitions are presented in Table 5.3. Within the latter, the left column shows the logical constants with their types, while their definition is presented in the right column.

Remark 5.2. To ensure that this representation of higher-order logic is actually sensible, one would now go on and prove a kind of equivalence between a higher-order logic defined in the usual way (by axioms and rules with side conditions) and this extension of the meta-logic, showing that for every proof in the one system, there is always a corresponding proof in the other system. This meta-proof cannot be expressed within Isabelle, though.

[2]this might seem an obvious choice, but some logics follow a different approach to make type systems possible that do not fit into the one provided by the meta-logic, cf. e. g. the formulations of Zermelo-Fraenkel set theory or CTT

eq-reflection	$(x = y) \Longrightarrow (x \equiv y)$
refl	$(x = x)$
subst	$[\![s = t; P\,s]\!] \Longrightarrow P\,t$
ext	$(\bigwedge x. f\,x = g\,x) \Longrightarrow \lambda x.\, f\,x = \lambda x.\, g\,x$
the-eq-trivial	$(\varepsilon x. x = a) = a$
impI	$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
mp	$[\![P \longrightarrow Q; P]\!] \Longrightarrow Q$
iff	$(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$
True-or-False	$P = True \vee P = False$

Table 5.2: Axiomatisation of HOL in Isabelle

Constant	*Definition*		
$True :: bool$	$True$	$\equiv$	$(\lambda x :: bool.\, x) = \lambda x.\, x$
$All :: ('a \Rightarrow bool) \Rightarrow bool$	$\forall x.\, P\,x$	$\equiv$	$P = \lambda x.\, True$
$Ex :: ('a \Rightarrow bool) \Rightarrow bool$	$\exists x.\, P\,x$	$\equiv$	$\forall b.\, (\forall x.\, P\,x \longrightarrow b) \longrightarrow b$
$False :: bool$	$False$	$\equiv$	$\forall b.\ b$
$Not :: bool \Rightarrow bool$	$\neg P$	$\equiv$	$P \longrightarrow False$
$\wedge :: [bool, bool] \Rightarrow bool$	$P \wedge Q$	$\equiv$	$\forall R.\, (P \longrightarrow Q \longrightarrow R) \longrightarrow R$
$\vee :: [bool, bool] \Rightarrow bool$	$P \vee Q$	$\equiv$	$\forall R.\, (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

Table 5.3: Definitions of some common logical constants in HOL

Example 5.3. To make the definitions of Table 5.3 a little bit more convincing, we take a closer look at two of them:

- The most basic notion of HOL is equality, so it is tempting to define truth in terms of equality: $\mathit{True} \equiv (\lambda x :: \mathit{bool}.\, x) = \lambda x.\, x$. This term is entirely closed, i. e. it neither contains free term variables nor free type variables, which is why this definition is used instead of the seemingly simpler $x = x$.

- Universal quantification is a predicate on predicates: if $\mathit{All}\, P$ or equivalently $\forall x.\, P\, x$ is true, this says that P is a predicate that constantly yields true, no matter what argument it is applied to (of course, all arguments must have the appropriate type). So, one can define $(\forall x.\, P\, x) \equiv (P = \lambda x.\, \mathit{True})$.

5.3 Proof Methods

Performing proofs from rules in an object-logic – in examples this will always be HOL – means proving theorems in the meta-logic. Such proofs would be incredibly tedious if only the meta-rules described in Section 5.1.3 had to be used. Fortunately, there is a powerful proof method whose correctness is assured by the axiomatic properties of $\bigwedge$ and $\Longrightarrow$: *higher-order resolution*. As with first-order resolution, known from logic programming in Prolog, this concept involves the *unification* of terms. As usual, if θ is a unifier of terms t_1 and t_2, i. e. an assignment of terms to variables, the simultaneous substitution of all variables mentioned in θ by the according terms is written as $(t_1)\theta$ and $(t_2)\theta$, respectively. Due to the fact that Isabelle employs the lambda calculus as its formal basis, it sometimes has to unify lambda abstractions that do not have a *most general unifier* (mgu), which is in contrast to first-order unification, where two terms either are not unifiable or have exactly one mgu (up to equivalence). The effect of this problem mainly is that sometimes the user must assist Isabelle in finding a unifier by supplying instantiations of variables.

Remark 5.4. Isabelle distinguishes two kinds of variables that logically have the same meaning. On the one hand there are the usual variables with standard lexical syntax (x, y, x_1, P are variables of this kind). On the other hand there are *schematic variables* which may be used as variables for substitution during unification. These are prefixed with a question mark to emphasise their role as placeholders (e. g. $?x$, $?P$). The usual way of proceeding is that theorems are stated solely with normal variables. After they have been proved, Isabelle internally converts all free variables of the theorem into schematic variables. This is in accordance with intuition: in proving a theorem T, one would certainly not want T's variables to be replaced by some concrete term; but one should be able to replace the free variables of already proved theorems, as they eventually represent arbitrary terms.

5.3.1 Higher-order Resolution

In what follows we will talk of the left-hand side of a meta-implication as the premiss (or premisses, if the $[\![\ldots]\!]$ notation is used) and of the right-hand side as the conclusion, to emphasise the role of meta-implication for object-logics. Given two theorems $[\![P_1, \ldots, P_n]\!] \Longrightarrow P$ and $[\![Q_1, \ldots, Q_m]\!] \Longrightarrow Q$ in the meta-logic, such that $(P_i \equiv Q)\theta$ holds for some $i \in \{1, \ldots, n\}$ and some unifier θ, resolution allows us to prove a new theorem that has P as its conclusion and all the P_j and Q_j except P_i as premisses, but with θ applied to the whole term

$$\frac{[\![P_1,\ldots,P_n]\!] \Longrightarrow P \quad [\![Q_1,\ldots,Q_m]\!] \Longrightarrow Q}{([\![P_1,\ldots,P_{i-1},Q_1,\ldots,Q_m,P_{i+1},\ldots,P_n]\!] \Longrightarrow P)\theta} \tag{5.5}$$

Apart from the substitution θ, this rule is intuitively clear: if the Q_j imply Q and $Q \equiv P_i$, then the Q_j are a suitable surrogate for P_i as premisses for the conclusion P. The involvement of substitution makes this idea even more general by admitting terms that are only equal under a given substitution θ.

A complication concerning the applicability of resolution arises when the premisses of a meta-theorem contain a meta-implication or meta-quantification themselves, as in the derived HOL rule (impI): $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$. The single premiss of this meta-theorem will only be unifiable with the conclusion of another meta-theorem if the latter consists of a variable or is of the form $X \Longrightarrow Y$, but both forms seldom appear in theorems. To circumvent this problem, Isabelle is able to *lift* a rule into a context, which can be formalised by the rule

$$\frac{[\![P_1,\ldots,P_n]\!] \Longrightarrow P}{[\![Q \Longrightarrow P_1,\ldots,Q \Longrightarrow P_n]\!] \Longrightarrow (Q \Longrightarrow P)} \tag{5.6}$$

This transformation is done automatically during resolution if necessary.

Although forward proof is also possible in Isabelle– mainly to derive new theorems from existing ones in a rather direct manner – theorems are usually proved in a backward style: By applying rules backwards, a theorem is reduced into simpler parts until the remaining propositions are trivially true (in particular by reducing propositions to axioms, of course). The ideas presented so far can best be understood with the help of an example.

Example 5.5. The backward proof a theorem T within the object-logic always starts with the trivial meta-theorem $T \Longrightarrow T$. This theorem is then transformed by the meta-rules and resolution until T has been derived. The following are HOL rules, derivable from the axioms given in Table 5.2.

$(?A \Longrightarrow ?B) \Longrightarrow ?A \longrightarrow ?B$	(impI)
$[\![?A; ?B]\!] \Longrightarrow ?A \wedge ?B$	(conjI)
$[\![?A \wedge ?B]\!] \Longrightarrow ?A$	(conjunct1)
$[\![?A \wedge ?B]\!] \Longrightarrow ?B$	(conjunct2)

Here is a proof of $A \wedge B \longrightarrow B \wedge A$ from these rules:

(.1)	$(A \wedge B \longrightarrow B \wedge A) \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	
(.2)	$[\![A \wedge B \Longrightarrow B \wedge A]\!] \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	(impI)
(.3)	$[\![A \wedge B \Longrightarrow B; A \wedge B \Longrightarrow A]\!] \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	(conjI, lifted)
(.4)	$[\![A \wedge B \Longrightarrow ?A \wedge B; A \wedge B \Longrightarrow A]\!] \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	(conjunct2, lifted)
(.5)	$(A \wedge B \Longrightarrow A) \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	(assumption)
(.6)	$(A \wedge B \Longrightarrow A \wedge ?B) \Longrightarrow (A \wedge B \longrightarrow B \wedge A)$	(conjunct1, lifted)
(.7)	$(A \wedge B \longrightarrow B \wedge A)$	(assumption)

To derive (.2), the premiss of (.1) has been resolved with the conclusion of rule (impI), where $?A$ has been unified with $(A \wedge B)$ and $?B$ has been instantiated to $(B \wedge A)$. To arrive at

(.3) lifting is necessary, because there is no rule that would otherwise match the premiss of (.2). Lifting rule (conjI) (to become $[\![?C \Longrightarrow ?A; ?C \Longrightarrow ?B]\!] \Longrightarrow (?C \Longrightarrow ?A \wedge ?B)$) makes it possible to resolve it with the premiss of (.2). The step from (.3) to (.4) is justified by lifting rule (conjunct2) and then resolving with the first premiss of (.3). Note that at this point a new schematic variable $?A$ is introduced which is entirely independent from A. This introduction is due to the fact that (conjunct2) contains $?A$ in its premiss, but not in the conclusion. We arrive at (.5) by dismissing an assumption which is trivially true after unification of $?A$ with A. This type of proof step is called *proof by assumption*. The remaining steps are analogous.

5.3.2 A Different Perspective

Another way to look at a proof of theorem T that is a bit more natural is to start with $T \Longrightarrow T$, but ignore the conclusion T and simply look at the premisses, regarding them as *goals*, i. e. statements that are yet to be proved in order to finish the proof of T. Thus, the initial goal is the theorem itself. Resolution of the theorem at hand with other theorems as described above can then be imagined as the application of rules to the current goal. For example, if the current goal is to show $A \longrightarrow B$ for some formulae A and B in the proof of T (i. e. internally the theorem $A \longrightarrow B \Longrightarrow T$ has been derived), we may 'apply the rule (impI)' to turn this goal into $A \Longrightarrow B$. Making one further step of abstraction, this term can be taken as the goal B, to be proved from the *assumption* A. Lifting of rules into a context suddenly takes the form of preservation of assumptions: In the above proof of $A \wedge B \longrightarrow B \wedge A$ the step from (.2) to (.3) preserves the assumption $A \wedge B$ for the two new *subgoals* $A \wedge B \Longrightarrow B$ and $A \wedge B \Longrightarrow A$.

One speaks of applying a *rule* in Isabelle parlance if it is applied in this standard way. There are other ways of applying a rule that do not enlarge the set of provable theorems, but that come in quite handy sometimes. Assume the current subgoal is $[\![P_1; \ldots; P_n]\!] \Longrightarrow P$ and we try to apply the rule $[\![T_1; \ldots; T_k]\!] \Longrightarrow T$, which is an already proved theorem.

- The standard rule application unifies P with T giving a unifier θ. It then replaces the subgoal by k new subgoals $([\![[\![P_1; \ldots; P_n]\!] \Longrightarrow T_1; \ldots; [\![P_1; \ldots; P_n]\!] \Longrightarrow T_k]\!])\theta$.

- Applying a *drule* (for destruction rule) is useful to modify a subgoal's assumptions. It unifies T_1 with some assumption – which for simplicity we assume to be P_1 – and yields the subgoals

$$([\![[\![P_2; \ldots; P_n]\!] \Longrightarrow T_2; \ldots; [\![P_2; \ldots; P_n]\!] \Longrightarrow T_k; [\![P_2; \ldots; P_n; T]\!] \Longrightarrow P]\!])\theta$$

  The idea is that T_1 is among the current assumptions (it is unifiable with P_1 here) and can thus be proved trivially. It then remains to prove T_2 to T_k, but if this can be done, it is reasonable to take T as an assumption in proving P, since all of T's premisses can be proved from the current assumptions.

- The application of an *erule* (for elimination rule) lets P be unified with T and simultaneously unifies T_1 (called the *major premiss* in this context) with one of the current assumptions (let it be P_1). It replaces the current subgoal with the new ones

$$([\![[\![P_2; \ldots; P_n]\!] \Longrightarrow T_2; \ldots; [\![P_2; \ldots; P_n]\!] \Longrightarrow T_k]\!])\theta$$

  This rule application is obviously quite similar to the standard way, but it deletes the assumption P_1 and it proves one subgoal immediately.

5.3.3 Advanced Proof Methods

For a proof assistant to be helpful in serious verification tasks, one may expect it to come with more powerful proof methods than just the application of axiomatically established rules in a backward proof. We now shortly present some important principles supported by Isabelle and which are regularly encountered in proofs.

- *Derived rules.* Every theorem that has been proved in Isabelle can be given a name and subsequently be used as if it were a rule of the object-logic. The rules (conjI), (impI), etc. shown above are examples for derived rules: they represent valid modes of reasoning in HOL and extend the logic in a conservative way, i. e. they do not enlarge the set of provable statements in HOL. In practice the largest part of rules applied in a proof will be derived rules of inference. A list of customary rules can be found in Appendix B.

- *The simplifier.* Isabelle provides a powerful and extensible term rewriting (or simplification) tool. Term rewriting works by subsequently transforming terms with the help of *rewrite rules* in a bottom-up fashion. The set of applicable rewrite rules is comprised of definitions and theorems. Adding the definition of Pierce's arrow $P \downarrow Q \equiv \neg P \wedge \neg Q$ to the set of rewrite rules lets the simplifier replace occurrences of $\downarrow$ by the defining term; this can be useful if no theorems about $\downarrow$ are known yet, but for $\wedge$ and $\neg$ there are some. Certain theorems are also good candidates for term rewriting; given associativity and commutativity of addition, the simplifier is able to prove equations like $(a+b)+(c+d) = (a+(b+(d+c)))$ outright, relieving the user of several applications of these rules by hand.

  To avoid looping on so-called permutative rewrite rules in which the left-hand side of the equation is equal to the right-hand side up to a renaming of variables – e. g. the rule $a+b = b+a$ – the simplifier performs *ordered rewriting* so that terms are only rewritten by permutative rules if they become lexicographically smaller. Hence, $a+b$ may be rewritten to $b+a$, but not the other way round.

- *A classical tableau prover.* In contrast to the simplifier – which can be employed as an intermediate proof step leaving a goal that is simpler to prove by hand, and which is able to manipulate arbitrary terms – there also is a tool for proving logical formulae directly. This tool is known as the `blast` method and it is capable of proving theorems like $(\exists y.\, \forall x.\, P\; x\; y) \longrightarrow (\forall x.\, \exists y.\, P\; x\; y)$ without intervention from the user (this theorem could not even be altered by the simplifier in any way). It cannot modify theorems however, e. g. to make the structure of the problem more apparent: if it fails to finish the proof, it fails completely.

5.3.4 An Example Proof

Concluding the presentation of Isabelle, we provide a short example proof, thereby explaining basic syntactic elements.

lemma *imp-uncurry*: $P \longrightarrow (Q \longrightarrow R) \Longrightarrow (P \wedge Q) \longrightarrow R$
apply (*rule impI*)
apply (*erule conjE*)
apply (*drule mp*)

apply *assumption*
by (*drule mp*)

Read as a rule of the object-logic HOL, *imp-uncurry* says that given the implication $P \longrightarrow (Q \longrightarrow R)$, one may conclude $(P \wedge Q) \longrightarrow R$. These formulae are well known to be equivalent, so we might even have proposed $(P \longrightarrow Q \longrightarrow R) = (P \wedge Q \longrightarrow R)$ (omitting all unnecessary parentheses) which we have not done to keep the example short. Let's walk through this proof step by step: As has been said, the initial goal is the theorem (or lemma) itself. Applying rule (impI) turns the goal into

$$[\![P \longrightarrow Q \longrightarrow R; P \wedge Q]\!] \Longrightarrow R$$

i. e. it assumes $P \wedge Q$ and imposes the proof of R. The next step uses the elimination rule (conjE) which is

$$[\![?P \wedge ?Q; [\![?P; ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R \qquad \text{(conjE)}$$

This results in the subgoal

$$[\![P \longrightarrow Q \longrightarrow R; P; Q]\!] \Longrightarrow R \tag{5.7}$$

What happens is that $?P \wedge ?Q$ is matched against $P \wedge Q$ and $?R$ is matched against R. The only remaining subgoal is then to prove $[\![P; Q]\!] \Longrightarrow R$ from the assumption $P \longrightarrow Q \longrightarrow R$ for which (5.7) is just a different notation. As a final step of detailed analysis we show what subgoals are yielded by applying rule (mp) destructively:

$$[\![[\![P; Q]\!] \Longrightarrow P; [\![P; Q; Q \longrightarrow R]\!] \Longrightarrow R]\!]$$

The rest of the proof consists of proof by assumption and another application of drule (mp). The **by** statement concludes a proof, possibly undertaking further steps of proof by assumption if necessary.

5.4 The Isar Proof Language

The proof style displayed in Section 5.3.4 above – occasionally termed the *apply style* due to its excessive use of the **apply** method – has two major drawbacks. The first one is that proof scripts comprising a long sequence of **apply**s are hard to read, because there is no information about intermediate proof states shown. The second one, which becomes evident in the presence of large numbers of theories, is maintainability: if, for example, the simplifier by changing its configuration becomes more powerful, an application of the *simp* method which previously resulted in a certain proof state might now result in quite a different one. This often means that subsequent rules of the original proof script are no longer applicable, so that the script has to be adjusted.

Another issue is that pure backward-oriented proofs are sometimes quite unnatural to perform. This is especially true for proofs involving applications of modus ponens. If at some point in a proof the goal A remains, which one wants to prove from the globally given facts B and $B \longrightarrow A$, then an application of rule (mp) results in the two new subgoals $?P \longrightarrow A$ and $?P$, thus introducing a new unification variable $?P$. In this simple case the structure of the goals containing the unification variable is very similar to the structure of the given facts, but in practice their relation can be hard to guess, since $?P$ may stand for any formula. This problem is of course closely related to the reason why cut-freeness and the sub-formula property are desired properties of logical calculi (see [1]).

5.4.1 Introducing Isar by Example

The Isar proof language has been conceived as a formalism for writing proof scripts that are both machine- and human-readable. Strictly speaking, one already works within Isar when employing the apply style, since **apply** is an Isar command rather than one of basic Isabelle. However, this mode of usage closely resembles the original Isabelle style in which ML functions were called directly. Full Isar comes with several advanced features which are best introduced with the help of a simple example. This is how a proof of the above lemma *imp-uncurry* looks like in Isar:

lemma *imp-uncurry2*: $P \longrightarrow (Q \longrightarrow R) \Longrightarrow (P \wedge Q) \longrightarrow R$
proof
  assume *a1*: $P \longrightarrow Q \longrightarrow R$
  assume *a2*: $P \wedge Q$
  show R
  proof $-$
    from *a2* **have** P **by** (*rule conjunct1*)
    with *a1* **have** *qr*: $Q \longrightarrow R$ **by** (*rule mp*)
    from *a2* **have** Q **..**
    with *qr* **show** *?thesis* **..**
  qed
qed

Compound Isar proofs are commenced by the keyword **proof**. In its pure form this statement tries to find a rule that can be applied to the goal – in the example, the implication introduction rule (impI) is selected. This kind of implicit rule application, which is much the same as **apply**ing a rule in a backward-oriented proof, can be avoided by appending a hyphen '$-$' or the rule selection can be made explicit by providing a concrete rule. Applying (impI) here results in the Isabelle proof state

$$[\![P \longrightarrow Q \longrightarrow R; P \wedge Q]\!] \Longrightarrow R$$

which is exactly mirrored by the following two **assume** commands introducing the valid assumptions (which may be given a name for future reference) in the proof script. Moreover, the succeeding **show** command precisely depicts the statement that remains to be shown. In every compound proof there occurs exactly one **show**. To prove R another compound proof has to be initiated, this time without applying a backward rule. From the given assumptions *a1* and *a2* it is very natural to prove R by forward reasoning: basically, two applications of modus ponens to assumption *a1* should yield the desired result. This is exactly what we find in the proof script: first, we derive P from $P \wedge Q$ by rule (conjunct1) as an intermediate fact, then we may apply modus ponens to $a1$ to obtain fact *qr*, i. e. $Q \longrightarrow R$. The same procedure can be executed once more (this time on *qr*) to finally show the thesis.

Several concepts of Isar have been used to achieve this result. The **by** command represents *basic proofs* which are finished immediately through an application of the rule handed to it (e. g. rule (conjunct1) or (mp)) and possibly further steps of proof by assumption. But how can a rule having itself some premisses be used to prove a pending subgoal? For this purpose the **from** command is needed, which feeds facts into a proof so that these are unified with the premisses of the applied rule. In the concrete example, the fact $P \wedge Q$ is fed into the proof by rule (conjunct1) to obtain P. A handy abbreviation is **with**, which behaves like **from**, but additionally feeds the most recent fact into the subsequent proof. For example, to obtain

$Q \longrightarrow R$ by (mp), one must feed the two premisses $P \longrightarrow Q \longrightarrow R$ and P into the proof, where P is the most recently established result. Hence, **with** *a1* yields all that is required to finish the proof by modus ponens. Finally, **qed** concludes a compound proof and two dots '..' are shorthand for **by** *standard rules*, i. e. a basic proof established through the standard rule set which includes (mp), (impI), (conjunct1) and many more. See Appendix B for frequently used rules in HOL and refer to [22, 39] for further details about the Isar proof language. Some more specialised features will also be explained in Chapter 6 as required.

6 Implementation in Isabelle

In this chapter we describe how the calculus of propositional dynamic logic has been implemented in Isabelle. The implementation can roughly be divided into three parts, which are first prerequisites like introducing the basic operations of a monad and setting up a convenient syntax – namely the do-notation – for compound monadic programs, second the definition or derivation of the logical operators as well as several proof rules accompanying these, and third two substantial example specifications from the realms of monadic parser combinators and a classical while-program performing Russian multiplication.

To keep the notation within the main text and the inserted Isabelle example specifications consistent, we will use the notation of Isabelle throughout this chapter. One major change caused thereby is that we will write $'a \Rightarrow 'b\ T$ for the type of a polymorphic function which would otherwise be denoted by $a \to T\ b$ (cf. Section 5.1.1). Because the commonly used symbols for the propositional connectives like $\wedge$ or $\longrightarrow$ are reserved for HOL, monadic connectives will be indexed by a D, as in $\wedge_D$ or $\longrightarrow_D$. Note also that implication is denoted by a simple arrow $\longrightarrow$ and not by a double arrow $\Rightarrow$.

6.1 Theory Files

The following listing of the theory files that have been created provides a more detailed explanation of the overall structure of the implementation. Besides that, Figure 6.1 shows the dependency graph of these theories. In this diagram, a link between two theories indicates that the theory below imports all theorems and definitions of the one above. In this way a simple acyclic theory hierarchy can be created in Isabelle. The figure moreover visualises the fact that the calculus directly builds on HOL, Isabelle's formulation of higher-order logic. Theory *Pure* is Isabelle's meta-logic, hence the base theory for every other logic.

Monads first of all defines a type constructor T that takes values of type $'a$ to monadic programs (or computations) of type $'a\ T$. Further it defines the monadic primitive operations $\gg=$, $\gg$ and *ret* for binding, sequencing and creating monadic programs. Finally, a do-notation quite similar to the one found in Haskell is defined through Isabelle's syntax facility.

MonProp formalises the notions of discardability, copyability and deterministic side-effect freeness of monadic programs and the properties that these programs possess. The subtype $'a\ D$ of dsef programs in $'a\ T$ is introduced and operations $liftM, liftM2$, etc., are defined allowing to lift HOL functions into the monadic setting. These will be used to define the propositional connectives.

MonLogic constitutes the setup of the propositional part of monadic dynamic logic. It defines the propositional connectives in terms of the ones of HOL, enables the simplifier to solve propositional tautologies in the new logic automatically and proves 'lifted' analogues of standard HOL rules like *conjI*, *disjE* or *excluded-middle*.

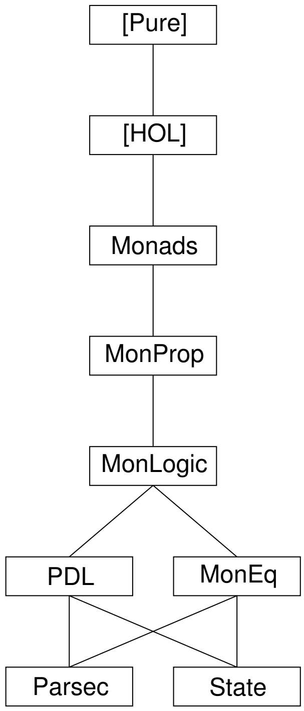

Figure 6.1: Dependency graph of the Isabelle theories

PDL completes the setup of the basic calculus by declaring the box and diamond operators, providing a convenient syntax for these, and formalising the proof calculus for dynamic logic of Section 3.3. Additionally, it is shown how the classical relationship between the box and diamond operator is automatically established by basing the logic on HOL, which itself is classical. The theory file ends with several proof rules that are derived from the basic calculus.

MonEq is a rather short theory file adding equality to the set of lifted operations. Rules representing transitivity, reflexivity and symmetry of monadic equality are also given.

Parsec contains the axiomatisation of the basic operations of a monad for parser combinators in the style of [12]. Subsequently, the specification and verification of a parser for natural numbers which is defined in terms of the basic parsers is presented.

State specifies a monad with readable and writable references as well as a while loop. In this monad, the algorithm for Russian multiplication is specified and proved correct.

6.2 Monads in Isabelle

While in Haskell the common ground of all (computable) monads can at least be captured at the level of operation types[1], Isabelle's concept of *axiomatic type classes* is not strong enough to suit this purpose. Axiomatic type classes are like Haskell's type classes, with the supplementary possibility of specifying what properties the operations over a certain type class must satisfy. For example, the type class *parord* of partial orders requires its instances to provide the operations $<$ and $\leq$, but additionally demands that the latter satisfies the usual properties of transitivity, reflexivity and antisymmetry. For the specification of monads however one does not require a class of types but rather a class of type constructors, namely the class of all those type constructors mapping a given base type into the type of specific computations over this type.

Due to the lack of this concept our implementation simply declares a polymorphic abstract type $'a\ T$, where T is supposed to stand for the monad in question. This way of proceeding precludes the exact definition of concrete monads and their primitive operations, since the structure of the monad is not visible. From the viewpoint of Isabelle's *definitional approach* – where HOL is supposed to be supplemented only by further definitions and theorems rather than axioms – this may be considered an imperfection, because additional operations acting on the structure of the monad have to be described axiomatically. For instance, there will be no way to define what precisely the operations of writing to or reading a reference in the state monad do, but these can only be described via their logical effects. Nonetheless, the way chosen here adheres to the one suggested in [34] and, in any case, the alternative would have been to have distinct base theories for all concrete monads, which is hard to maintain and tedious to implement.

typedecl $'a\ T$

consts
$bind :: {'a}\ T \Rightarrow ({'a} \Rightarrow {'b}\ T) \Rightarrow {'b}\ T \quad (\textbf{infixl} \gg\!= 20)$
$ret :: {'a} \Rightarrow {'a}\ T$

constdefs
$seq :: {'a}\ T \Rightarrow {'b}\ T \Rightarrow {'b}\ T \qquad (\textbf{infixl} \gg 20)$
$p \gg q \equiv (p \gg\!= (\lambda x.\ q))$

This is the concrete Isabelle notation for the introduction of the type $'a\ T$ of monadic programs and the basic operations *bind*, *ret* and *seq*, where the latter is defined in terms of the binding. The so-called *mixfix annotations* on the right margin declare infix notation, $\gg\!=$ for *bind* and $\gg$ for *seq*, which in their simple form given here resemble the syntax annotations for infix operators in Haskell. As stated above *bind* and *ret* can only be declared as abstract constants through a **consts** declaration, while *seq* can be given a declaration as well as a concrete definition (albeit in terms of the abstractly defined operation *bind*, of course) through the **constdefs** statement. The latter combines the effects of the statements **consts** and **defs**, where the **defs** statement serves the purpose of providing a definition for a previously introduced constant.

The following is a specification of the monad laws of Equation (2.10) in Isabelle. The

[1] which is done by making the respective type constructors like `[]` (being syntactical sugar for `List`), `Maybe`, etc., instances of the constructor class `Monad`

[simp] instruction makes Isabelle hand a theorem or axiom to the simplifier as a rewrite rule automatically. We have included a specification that *ret* is injective. From these axioms we can prove the associativity of $\gg$ immediately.

axioms
bind-assoc [*simp*]: $(p \gg= (\lambda x.\, f\, x \gg= g)) = (p \gg= f \gg= g)$
ret-lunit [*simp*]: $(ret\ x \gg= f) = f\, x$
ret-runit [*simp*]: $(p \gg= ret) = p$
ret-inject: $ret\ x = ret\ z \Longrightarrow x = z$

lemma *seq-assoc* [*simp*]: $(p \gg (q \gg r)) = (p \gg q \gg r)$
by (*simp add*: *seq-def*)

6.2.1 The do-Notation

Next comes the setup of the do-notation by means of Isabelle's syntax translation facility. This basically is a term-rewriting mechanism on abstract syntax trees which can be configured by adding rewrite rules for either the transformation of concrete input into a valid Isabelle term or vice versa. We will not go into the details of this mechanism, which is laid out in the Isabelle reference manual [25]. The implementation can be found in Appendix C, p. 101.

The syntax translations make it possible to write monadic programs in a much more convenient way that mirrors the sequentiality inherent in these programs. In the implementation we make use of this notation exclusively. As an example, one may write the following

$$\mathsf{do}\,\{x \leftarrow p; q\ x\} \quad \mathsf{do}\,\{x \leftarrow p; y \leftarrow q; r\ x\ y\} \qquad \mathsf{do}\,\{x \leftarrow p; y \leftarrow q; z \leftarrow r; ret\ (x,y,z)\}$$

instead of

$$p \gg= \lambda x.\, q\ x \qquad p \gg= (\lambda x.\, q \gg= \lambda y.\, r\ x\ y) \quad \mathsf{do}\,\{x \leftarrow p; \mathsf{do}\,\{y \leftarrow q; \mathsf{do}\,\{z \leftarrow r; ret\ (x,y,z)\}\}\}$$

where the third column indicates that multiple bindings may be input as a sequence rather than in a nested fashion.

Remark 6.1. The fact that do-terms are simply syntactical sugar also means that we do not formalise the inference rules of the meta-language for monads described in Section 2.2.3, but rather work with monadic programs and their properties directly and just display them in the more convenient do-notation. That such a translation can be achieved purely by syntax transformations indicates how closely the meta-language is related to actual monadic programs.

6.2.2 Properties of Monadic Programs

Our main goal for now is to obtain a subtype *'a D* of deterministically side effect free (*dsef*) programs over *'a T* so that programs of type *bool D* can be used as formulae of our logic. The kind of subtyping supported by Isabelle proceeds by defining a new type in terms of a subset of elements of an existing type. Isabelle then generates a bijection between this subset of the existing type and the new type which consists of an *abstraction function* from the existing type into the new one – which is only sensibly defined for elements that really have a corresponding element in the new type – and a *representation function* mapping elements of the new type back to their representatives in the existing type.

It is straightforward to formalise the concepts of discardability and copyability, the concepts on which the property *dsef* builds. The latter is itself defined in terms of the former ones as follows.

constdefs
$dis :: {'a}\ T \Rightarrow bool$
$dis(p) \equiv (do\ \{x \leftarrow p;\ ret()\}) = ret\ ()$

$cp\ :: {'a}\ T \Rightarrow bool$
$cp(p) \equiv (do\ \{x \leftarrow p;\ y \leftarrow p;\ ret(x,y)\}) = (do\ \{x \leftarrow p;\ ret(x,x)\})$

$dsef :: {'a}\ T \Rightarrow bool$
$dsef(p) \equiv cp(p) \wedge dis(p) \wedge (\forall q{::}bool\ T.\ cp(q) \wedge dis(q) \longrightarrow$
$cp(do\ \{x \leftarrow p;\ y \leftarrow q;\ ret(x,y)\}))$

The definition of *dsef* deserves explanation for two reasons. First, it should be repeated that there are three equivalent formulations of what it means for a program to commute with some other program (cf. Def. 3.6), from which we have chosen (3.1). Second, this formulation restricts the types of programs that the given program p has to commute with to those of type *bool* (see also Definition 3.7 and Remark 3.9). This is required because Isabelle[2] does not allow for a quantification over type variables in a definition. But this is exactly what would be done, if implicitly, in the case that the right-hand side of the definition mentioned an arbitrary program $q :: {'a}\ T$. As $'a$ would be arbitrary, any type might serve as an instantiation. An explicit lemma *commute-bool-arb* is needed to derive the commutativity of a certain program p with copyable and discardable programs of *any* type from the commutativity of p among copyable and discardable programs of type *bool*. Because the implementation of global dynamic judgements was the subject of a different diploma thesis, this 'lemma' is in fact provided as an axiom in this thesis; given a more elaborate infrastructure, it would however be provable.

Several properties of copyable and discardable programs discussed in Section 3.1 have been formalised, the most frequently employed of which are Lemmas 3.3 and 3.5

lemma *cp-arb*: $cp\ p \Longrightarrow do\ \{x \leftarrow p;\ y \leftarrow p;\ r\ x\ y\} = do\ \{x \leftarrow p;\ r\ x\ x\}$
lemma *dis-left*: $dis(p) \Longrightarrow do\ \{p;\ q\} = q$

Notice how the substitution of x for y in r of lemma *cp-arb* is achieved by making r a function of x and y. With the above definitions and lemmas at our disposal the type $'a\ D$ can be defined.

typedef $(Dsef)\ ('a)\ D = \{p{::}'a\ T.\ dsef\ p\}$
apply$(rule\ exI[of\ \text{-}\ ret\ x])$
apply$(blast\ intro{:}\ dsef\text{-}ret)$
done

The proof obligation in the type definition arises due to the restriction that types must not be empty. We use the program *ret x* as a witness, since stateless programs are always dsef.

[2] to be precise, this statement is only true for logics like HOL which inherit their type mechanism from Isabelle's meta-logic

This fact has of course been proved as lemma *dsef-ret* in Isabelle beforehand. The **typedef** statement declares the new type $'a\ D$ to be in bijective correspondence to the set *Dsef* of dsef programs in $'a\ T$. The definition of this set is subsequently available under the name *Dsef-def*. What's more, two functions $\mathit{Abs\text{-}Dsef} :: {'a}\ T \Rightarrow {'a}\ D$ and $\mathit{Rep\text{-}Dsef} :: {'a}\ D \Rightarrow {'a}\ T$ are generated that mediate between these two types. As the functions may appear quite often in certain formulae, two abbreviations are introduced: $\Uparrow p$ stands for $\mathit{Abs\text{-}Dsef}\ p$ and $\Downarrow P$ stands for $\mathit{Rep\text{-}Dsef}\ P$. This is quite suggestive, in particular in those cases where terms of the form $\Uparrow\Downarrow P$ or $\Downarrow\Uparrow p$ appear since one is visually reminded that these operations cancel each other out.

Remark 6.2. The reason why terms of the form $\Uparrow p$ will appear is that one may only write monadic programs in T, while the formulae of our logic live in D. This means that a compound truth-valued program $p = \mathsf{do}\{x_1 \leftarrow p_1; \cdots; x_n \leftarrow p_n; r\, x_1 \cdots x_n\}$ that is dsef will nonetheless have type *bool T*. This program has to be shifted to *bool D* to form the monadic formula $\Uparrow p$. Furthermore, there are several atomic programs – with *ret x* being the predominant one – which are dsef and hence may appear in formulae when shifted. We initiate the convention of defining a formula $\mathit{Prog} \equiv \Uparrow \mathit{prog}$ for each atomic dsef program *prog*. Hence the shifted version of *ret* is $\mathit{Ret} :: {'a} \Rightarrow {'a}\ D$.

Theory MonProp also contains proofs of characteristic properties of dsef programs which are not shared by discardable or copyable programs. The two most important facts are that neighbouring dsef programs may be swapped (Theorem *commute-dsef*, p. 108) and that dsef programs are stable under sequential composition (Theorem *dsef-seq*, p. 108). While the first one is quite immediate from the definitions, the second one asks for a bit more work.

theorem *dsef-seq*: $[\![\mathit{dsef}\ p;\ \forall x.\ \mathit{dsef}\ (q\ x)]\!] \Longrightarrow \mathit{dsef}\ (\mathit{do}\ \{x{\leftarrow}p;\ q\ x\})$

According to the definition of *dsef* proving that $\mathsf{do}\{x \leftarrow p; q\ x\}$ (call it r in the following) is dsef amounts to showing three facts. The first one is that r is discardable. This follows from the fact that p and $q\ x$ are discardable for all x. The second one, namely that r is copyable, follows from the fact that p and $q\ x$ commute with each other, so that the defining equality of copyability holds for r by the copyability of p and $q\ x$. It must be noted here that while we used condition (3.1)[3] as part of the defining property of dsef programs, condition (3.3)[4] can easily be inferred from (3.1), a point that has been shown in lemma *commute-1-3*. The final fact to be shown is that r commutes with all copyable and discardable *bool*-valued programs. This follows similarly to the second fact, noting that p and $q\ x$ alone commute with all discardable and copyable *bool*-valued programs.

6.2.3 Equational Reasoning in Isar

We will now shortly explain how Isar supports equational reasoning. As it is used in this thesis, equational reasoning means reasoning by chains of equations, where each separate step is justified mainly by substituting equals for equals. Take the following lemma, representing the formalisation of how to infer (3.2) from (3.1), as an example.

lemma *commute-1-2*: $[\![\mathit{cp}\ q;\ \mathit{cp}\ p;\ \mathit{dis}\ q;\ \mathit{dis}\ p]\!] \Longrightarrow \mathit{cp}\ (\mathit{do}\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ \mathit{ret}(x,y)\})$
$\Longrightarrow \mathit{do}\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ \mathit{ret}(x,y)\} = \mathit{do}\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ \mathit{ret}(x,y)\}$

[3] stating that the composition of two discardable and copyable programs is again copyable
[4] which states the property of commutativity more instructively by actually swapping two programs

proof −
 assume *a*: *cp q cp p dis q dis p*
 assume *c*: *cp* (*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)})
 let *?s* = *do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)}
 have *?s* = *do* {*z*←*?s*; *ret* (*fst z*, *snd z*)} **by** *simp*
 also from *c* **have** ... = *do* {*w*←*?s*; *z*←*?s*; *ret* (*fst z*, *snd w*)} **by** (*simp add*: *cp-arb*)
 also from *a* **have** ... = *do* {*v*←*q*; *x*←*p*; *ret*(*x*,*v*)} **by** (*simp add*: *mon-ctr dis-left2*)
 finally show *?thesis* **.**
qed

After stating the valid assumptions and setting ?*s* as an abbreviation for the left-hand side of the equation that is to be shown, a chain of equations starts beginning with ?*s* and ending with the right-hand side of the main goal. This kind of successive equational reasoning is realised in Isar through a sequence of **have**... **also have**... statements and a concluding **finally** statement. In its simplest form, the **also** statement combines two facts of the form $a = b$ and $b = c$ to yield the fact $a = c$, thus simply exploiting transitivity of equality. The **finally** statement reiterates the transitive chain build up so far and feeds it into the concluding proof – which in the example is precisely the goal thesis. As a convenience, three dots '...' within a term refer to the right-hand side of the most recently established equality. The main workhorse for performing the intermediate proof steps is the simplifier, since it is ideally suited for handling equalities and substitution. [3] contains a detailed description of extended features of this mechanism, showing how it can also be applied to inequalities.

6.2.4 Lifting HOL Constants

The definition of the propositional connectives in Section 3.2.1 suggests the introduction of *lifting operators* that allow one to embed HOL operators into the monadic setting. These lifting operators are well known from Haskell and their definition in Isabelle does not look that different. The basic idea is that to apply an n-ary operator $f :: [a_1, \ldots, a_n] \Rightarrow b$ to n monadic programs $p_1 :: a_1\,T, \ldots, p_n :: a_n\,T$, one simply evaluates these programs in turn and applies the operator to the results. In principle all HOL operators like equality, comparisons, addition, etc. could be lifted this way, but for simplicity we will only lift the propositional connectives and equality in the sequel.

constdefs
 liftM :: [′*a* ⇒ ′*b*, ′*a T*] ⇒ ′*b T*
 liftM f p ≡ *do* {*x* ← *p*; *ret* (*f x*)}
 liftM2 :: [′*a* ⇒ ′*b* ⇒ ′*c*, ′*a T*, ′*b T*] ⇒ ′*c T*
 liftM2 f p q ≡ *do* {*x* ← *p*; *y* ← *q*; *ret* (*f x y*)}

Thanks to lemma *dsef-seq* it is very easy to prove that applying a lifted operation to dsef programs yields a dsef program:

lemma *dsef-liftM2*: ⟦*dsef p*; *dsef q*⟧ ⟹ *dsef* (*liftM2 f p q*)

This fact is essential when introducing the propositional connectives in this manner, since e. g. the conjunction of two formulae is of course required to be a formula, hence dsef.

6.3 Setting up the Logic

Apart from a slight visual clutter induced by the occurrences of the shifting functions $\Uparrow$ and $\Downarrow$ the definition of global validity (which we denote here by a turnstile $\vdash$ instead of the global box $\boxed{G}$) and of the propositional connectives is now straightforward. We take conjunction, disjunction and implication as primitives: the constant for falsity does not have do be defined, since it is available via the injection of *False* into the monad, i. e. via *Ret False*.

consts
$Valid \quad :: bool\ D \Rightarrow bool \qquad ((\vdash\text{ -})\ 15)$
$\wedge_D \quad :: [bool\ D, bool\ D] \Rightarrow bool\ D \quad (\textbf{infixr}\ 35)$
$\vee_D \quad :: [bool\ D, bool\ D] \Rightarrow bool\ D \quad (\textbf{infixr}\ 30)$
$\longrightarrow_D \quad :: [bool\ D, bool\ D] \Rightarrow bool\ D \quad (\textbf{infixr}\ 25)$
defs
$Valid\text{-}def: \vdash P \equiv \Downarrow P = do\ \{x \leftarrow (\Downarrow P); ret\ True\}$
$conjD\text{-}def: P \wedge_D Q \equiv \Uparrow (liftM2\ (op\ \wedge)\ (\Downarrow P)\ (\Downarrow Q))$
$disjD\text{-}def: P \vee_D Q \equiv \Uparrow (liftM2\ (op\ \vee)\ (\Downarrow P)\ (\Downarrow Q))$
$impD\text{-}def: P \longrightarrow_D Q \equiv \Uparrow (liftM2\ (op\ \longrightarrow)\ (\Downarrow P)\ (\Downarrow Q))$

Other operators like equivalence $\longleftrightarrow$ and negation $\neg$ are defined as abbreviations in the usual way:

constdefs
$iffD \quad :: [bool\ D, bool\ D] \Rightarrow bool\ D \quad (\textbf{infixr} \longleftrightarrow_D 20)$
$P \longleftrightarrow_D Q \equiv (P \longrightarrow_D Q) \wedge_D (Q \longrightarrow_D P)$
$NotD \quad :: bool\ D \Rightarrow bool\ D \qquad (\neg_D\text{ - }[40]\ 40)$
$\neg_D P \equiv P \longrightarrow_D Ret\ False$

The notion of global validity can be simplified, since dsef programs are discardable. This fact can be stated either as an equality in T or as an equality in D:

lemma *Valid-simp*: $(\vdash p) = (\Downarrow p = ret\ True)$
lemma *Valid-simpD*: $(\vdash P) = (P = Ret\ True)$

Remark 6.3. While the formalisation of the proof calculus as given in [34] is tailored towards an intuitionistic framework, an immediate consequence of the definitions presented thus far is that *the implementation of the calculus in Isabelle is classical*. This follows from the fact that the logical operators are defined in terms of the HOL operators and that *bool* is classical, i. e. contains only two values *True* and *False*. A representative theorem confirming that a logic is classical is the law of excluded middle. The formulation in the monadic setting reads as

theorem *pdl-excluded-middle*: $\vdash P \vee_D (\neg_D P)$

The outline of the proof of this theorem is as follows: first, decode $P \vee_D (\neg_D P)$ into the program $\Uparrow \mathrm{do}\{a \leftarrow \Downarrow P; b \leftarrow \Downarrow P; ret\ (a \vee \neg b)\}$. By copyability of $\Downarrow P$ – noting that all programs of the form $\Downarrow \_$ are dsef, therefore copyable – this program is equal to $\Uparrow \mathrm{do}\{a \leftarrow \Downarrow P; ret\ (a \vee \neg a)\}$. At this point, reasoning in HOL reduces $a \vee \neg a$ to *True*, so that by discardability of $\Downarrow P$ the whole program is equal to *Ret True*, hence globally valid.

An interesting connection between the *Ret* function and every operator *op* that has been lifted by the *liftM∗* functions to form an operator op_D is that *Ret* is a homomorphism between *op*-terms in HOL and op_D-terms in *D*. This is reflected by the following equations, which all hinge on the fact that the operators have simply been lifted.

lemma *conjD-Ret-hom*: $Ret\ (a \wedge b) = ((Ret\ a) \wedge_D (Ret\ b))$
lemma *impD-Ret-hom*: $Ret\ (a \longrightarrow b) = ((Ret\ a) \longrightarrow_D (Ret\ b))$
lemma *NotD-Ret-hom*: $Ret\ (\neg\ P) = (\neg_D\ (Ret\ P))$

Dual statements hold for disjunction, equivalence and the like.

6.3.1 Basic Proof Rules

Besides theorem *pdl-excluded-middle* there are several other analogues of proof rules of HOL given in Section C.3.3. These include modus ponens, introduction and elimination rules for conjunction and disjunction, some rules concerning negation and so forth. It would thus be tempting to try and formulate a natural deduction calculus for the propositional part of the logic. However, this fails at one critical point: the introduction rule for implication, which might be formulated as

$$\textit{pdl-impI} \quad (\vdash P \Longrightarrow \vdash Q) \Longrightarrow \vdash P \longrightarrow_D Q$$

is not provable, and what's worse, not even valid. This is quite obvious, since one may not expect any relationship between the *global* validity of P and the global validity of the formula $P \longrightarrow_D Q$. Hence it does not make sense to assume the global validity of P, prove $\vdash Q$ and then conclude that $P \longrightarrow_D Q$ must be globally valid. It is a common phenomenon that natural deduction systems – and the proof calculus for HOL basically is formulated as such – have to be modified if they are to be used for modal logics. For simple logics involving unparameterised modal operators this can be done rather easily (see [6]), but it is as yet unclear how it might be accomplished for the logic discussed here, which includes modal operators for every possible program sequence.

The lack of this single rule has quite profound consequences, since the simplest theorems like $\vdash P \longrightarrow_D Q \longrightarrow_D P$ cannot be proved 'logically', i. e. with the natural deduction rules. Like every classical tautology this theorem however has a semantic proof which proceeds in analogy to the proof of *pdl-excluded-middle* discussed above by unfolding the definition of global validity and then manipulating the resulting do-terms. Having to step back to the semantic definition of the connectives when proving valid formulae is not desirable since this does not lend itself easily to automation and it makes proofs very unstructured in comparison to those conducted in a proof calculus. To obtain a purely Hilbert-style calculus for the propositional part of the logic it would theoretically suffice to prove an appropriate set of axiom schemes semantically and then conduct proofs from these axioms by modus ponens. This way of proceeding would lead to rather cumbersome proofs and substantially blow up the amount of work required to verify programs of realistic size, so an alternative solution had to be found.

6.3.2 Proving Tautologies Automatically

The solution that has been adopted in the implementation is to use the simplifier, i. e. to employ the technique of term rewriting, and enhance it in such a way that it can prove *classical* propositional tautologies automatically. The first step to this solution is to regard the propositional part of the logic as a Boolean algebra. It is a standard exercise [4, Chapter 5] to verify that $(bool\ D, \wedge_D, \vee_D, Ret\ False, Ret\ True)$ is such an algebra which further gives rise to a boolean ring, i. e. a commutative ring in which all elements are idempotent, i. e. $X^2 = X$ for all X. Taking $\wedge_D$ as the multiplication and exclusive disjunction[5] $\oplus_D$ as the addition of the Boolean ring this equation certainly holds, since $X \wedge_D X = X$ is valid. All other requirements of a Boolean ring like distributivity of multiplication over addition, associativity of these operations etc. are also satisfied. The major insight then is that a complete set of rewrite rules for ordered rewriting can be given for Boolean rings. A complete set of rewrite rules is one that is terminating and confluent, such that every term can be rewritten into a unique normal form and it does not matter which path of possible reductions one follows (cf. the Church-Rosser property of the untyped lambda calculus in Prop. 2.7 and the description of the simplifier in Section 5.3.3). But this is exactly what is needed to prove a classical tautology T automatically, since it can then be rewritten to its normal form *Ret True*, so that proving $\vdash T$ amounts to proving the trivial statement $\vdash$ *Ret True*. This final proof step can of course be done automatically, too.

Section C.3.2 presents all rules the simplifier has to be equipped with to prove tautologies automatically. For shortage of time the rules were given as axioms, and only some of them were proved as examples on how such proofs can be carried out. The rules include associativity and commutativity of $\wedge_D$ as well as $\oplus_D$, unit laws for $\wedge_D$ with respect to *Ret True* and absorption laws for $\wedge_D$. Furthermore, the behaviour of $\wedge_D$ and $\oplus_D$ with regard to falsity and the distribution of $\wedge_D$ over $\oplus_D$ are laid out. All these laws – together with translation rules that let all connectives be expressed through $\wedge_D$ and $\oplus_D$ plus falsity – are collected in the rule set *pdl-taut*. Tautologies are now proved in one fell swoop:

lemma $\vdash (P \longrightarrow_D Q) \wedge_D (\neg_D P \longrightarrow_D R) \longleftrightarrow_D (P \wedge_D Q \vee_D \neg_D P \wedge_D R)$
 by (*simp only*: *pdl-taut Valid-Ret*)

6.3.3 Modal Operators and the Proof Calculus

We will now make up for the definition of the box and diamond operators which have been overlooked up to this point. Due to the fact that an elaborate formalisation of global dynamic judgements has been worked out in a different diploma thesis, the box and diamond operators are in fact not *defined* through their unique defining property as given in Proposition 3.18, but rather treated as abstract constants.

consts
 $Box :: {}'a\ T \Rightarrow ({}'a \Rightarrow bool\ D) \Rightarrow bool\ D \quad$ ([# -]- [0, 100] 100)
 $Dmd :: {}'a\ T \Rightarrow ({}'a \Rightarrow bool\ D) \Rightarrow bool\ D \quad$ ($\langle$-$\rangle$- [0, 100] 100)

Each operator maps a program and a formula depending on the return value of the program into a monadic formula. The syntax annotations make it possible to write, e. g. $[\#\ \mathsf{do}\,\{x \leftarrow p; q\}]\, Q$[6] or $\langle \mathsf{do}\,\{x \leftarrow p; q\}\rangle\, (\lambda y.\, P\, y)$ – note that both Q and P are function predicates depending on the return value of the entire do-term inside the box or diamond respectively, and *not*

[5] recall the definition of exclusive disjunction: $A \oplus B \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$
[6] the sharp sign '#' is needed to disambiguate box formulae from lists, for which the square bracket notation is already in use

on the variable x bound in these do-terms. This means that the notion of variable binding that is performed by these operators differs from that which has been proposed in [34], where the multiple bindings that may occur inside a modal operator all constitute variable binders for the formula in the scope of the operator.

With the help of some intricate syntax translation instructions it becomes possible to mimic this kind of multiple variable binding in Isabelle. The idea is to write a sequence of bindings inside the modal operator and use these bound variables freely in the formula in scope of the operator, like so:

$$[\# \, x_1 \leftarrow p_1; \cdots; x_n \leftarrow p_n] \, P \, x_1 \cdots x_n$$

The binding sequence is then transformed into an actual do-term by collecting all bound variables to form a tuple and appending a *ret* expression that takes this tuple as its argument. The free occurrences of these variables in the formula in the scope of the modal operator become bound by turning the formula into a lambda abstraction that expects the tuple of variables as an argument. So the result of translating the above binding sequence is

$$[\# \, \mathsf{do} \, \{x_1 \leftarrow p_1; \cdots; x_n \leftarrow p_n; \mathit{ret} \; (x_1, \ldots, x_n)\}] \, \lambda \, (x_1, \ldots, x_n). \, P \, x_1 \cdots x_n$$

The notation thus set up is in particular nice for sequences of length one, because $\langle x \leftarrow p \rangle \, (P \; x \wedge_D Q \; x)$ and $\langle p \rangle \, (\lambda x. \, P \; x \wedge_D Q \; x)$ denote the same formula, with the former one emphasising the connection between the return value x of p and its use in the subsequent conjunction.

With the modal operators readily defined, the proof calculus for propositional dynamic logic can be implemented, resulting in the following specification.

axioms
pdl-nec: $(\forall x. \vdash P \; x) \Longrightarrow \vdash [\# \, x \leftarrow p](P \; x)$
pdl-mp-: $[\![\vdash (P \longrightarrow_D Q); \vdash P]\!] \Longrightarrow \vdash Q$

pdl-k1: $\vdash [\# \, x \leftarrow p](P \; x \longrightarrow_D Q \; x) \longrightarrow_D [\# \, x \leftarrow p](P \; x) \longrightarrow_D [\# \, x \leftarrow p](Q \; x)$
pdl-k2: $\vdash [\# \, x \leftarrow p](P \; x \longrightarrow_D Q \; x) \longrightarrow_D \langle x \leftarrow p \rangle(P \; x) \longrightarrow_D \langle x \leftarrow p \rangle(Q \; x)$
pdl-k3B: $\vdash \mathit{Ret} \; P \longrightarrow_D [\# \, x \leftarrow p](\mathit{Ret} \; P)$
pdl-k3D: $\vdash \langle x \leftarrow p \rangle(\mathit{Ret} \; P) \longrightarrow_D \mathit{Ret} \; P$
pdl-k4: $\vdash \langle x \leftarrow p \rangle(P \; x \vee_D Q \; x) \longrightarrow_D (\langle x \leftarrow p \rangle(P \; x) \vee_D \langle x \leftarrow p \rangle(Q \; x))$
pdl-k5: $\vdash (\langle x \leftarrow p \rangle(P \; x) \longrightarrow_D [\# \, x \leftarrow p](Q \; x)) \longrightarrow_D [\# \, x \leftarrow p](P \; x \longrightarrow_D Q \; x)$
pdl-seqB: $\vdash [\# \, x \leftarrow p; \, y \leftarrow q \; x](P \; x \; y) \longleftrightarrow_D [\# \, x \leftarrow p][\# \, y \leftarrow q \; x](P \; x \; y)$
pdl-seqD: $\vdash \langle x \leftarrow p; \, y \leftarrow q \; x \rangle(P \; x \; y) \longleftrightarrow_D \langle x \leftarrow p \rangle \langle y \leftarrow q \; x \rangle(P \; x \; y)$
pdl-ctrB: $\vdash [\# \, x \leftarrow p; \, y \leftarrow q \; x](P \; y) \longrightarrow_D [\# \, y \leftarrow \mathit{do} \; \{x \leftarrow p; \, q \; x\}](P \; y)$
pdl-ctrD: $\vdash \langle y \leftarrow \mathit{do} \; \{x \leftarrow p; \, q \; x\} \rangle(P \; y) \longrightarrow_D \langle x \leftarrow p; \, y \leftarrow q \; x \rangle(P \; y)$
pdl-retB: $\vdash [\# \, x \leftarrow \mathit{ret} \; a](P \; x) \longleftrightarrow_D P \; a$
pdl-retD: $\vdash \langle x \leftarrow \mathit{ret} \; a \rangle(P \; x) \longleftrightarrow_D P \; a$
pdl-dsefB: $\mathit{dsef} \; p \Longrightarrow \vdash \Uparrow (\mathit{do} \; \{a \leftarrow p; \Downarrow (P \; a)\}) \longleftrightarrow_D [\# \, a \leftarrow p](P \; a)$
pdl-dsefD: $\mathit{dsef} \; p \Longrightarrow \vdash \Uparrow (\mathit{do} \; \{a \leftarrow p; \Downarrow (P \; a)\}) \longleftrightarrow_D \langle a \leftarrow p \rangle(P \; a)$

This specification does not look all that different from the original one presented in Figure 3.1. The side-condition in the necessitation rule that variable x must not occur free in the assumptions can be formalised by requiring that $P \; x$ holds for all x, since this precludes any assumptions to be made on x. The K axioms really are almost identical to the original specification. The axioms for dsef programs, *pdl-dsefB* and *pdl-dsefD*, cannot be stated in the convenient notation in which dsef programs of type $'a \; T$ are simply substituted for actual values of type $'a$, since this results in a type error. So in the specification one has to resort to the decoded forms, where the dsef program is evaluated first, and the resulting value a is used in the formula $P\ a$.

For the structural rules there also do not appear to be notable differences, but this is not quite true: the syntax translations transform, e. g., the depicted formula *pdl-seqB*

$$[\#\, x \leftarrow p; y \leftarrow q\ x]\, P\ x\ y \longleftrightarrow_D [\#\, x \leftarrow p]\, [\#\, y \leftarrow q\ x]\, P\ x\ y$$

into the genuine Isabelle term

$$[\#\ \mathsf{do}\ \{x \leftarrow p; y \leftarrow q\ x; ret\ (x,y)\}]\ \lambda(x,y).\, P\ x\ y \longleftrightarrow_D [\#\ p]\ \lambda x.\, [\#\ q\ x]\ \lambda y.\, P\ x\ y$$

which has a rather complicated structure. This complexity and the fact that the *ret* expression only appears on the left-hand side of the equivalence will make it hard to apply the axiom in actual proofs about compound programs, because these will hardly ever unify with the program structure imposed by the axiom.

It has been shown in [34] that simple monads (cf. Rem. 3.10) satisfy the converses of the contraction axioms *pdl-ctrB* and *pdl-ctrD*, i. e. the same formulae with the implication arrows reversed. These converse axioms make it possible to prove simplified versions of the sequencing axioms which have turned out to be more effective in practice. Whether there is a proof for these theorems in monads that are not simple, too, is currently unclear. For the box operator the corresponding theorem is

axioms
pdl-seqB-simp: $\vdash$ ($[\#\ x{\leftarrow}p][\#\ y{\leftarrow}q\ x](P\ y)$) $\longleftrightarrow_D$ ($[\#\ y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}](P\ y)$)

with the equivalence of $[\#\ x \leftarrow p; y \leftarrow q\ x]\, P\ y$ and $[\#\ y \leftarrow \mathsf{do}\ \{x \leftarrow p; q\ x\}]\, P\ y$ being the key fact required for the proof. This equivalence is precisely what is provided by the contraction rules and their converses.

Although the simpler rendition of the sequencing axiom does not allow for reasoning about intermediate results – as P only depends on the return value of $q\ x$, but not that of p – one has to remember that in rule *pdl-seqB* the intermediate results are only made available to P by packing all of them into a tuple $\overline{x}$ and making $ret\ \overline{x}$ the final expression of the do-term. The formulation as given in *pdl-seqB-simp* is more flexible, since $q\ x$ may or may not consist of or at least end with such a *ret* expression. In the case where one is in fact not interested in the value of x (e. g. when nothing is to be said about intermediate results), it turns out to be more convenient to dispose of the final *ret* expression. And the general contraction rules are too weak to make this possible when working with *pdl-seqB*. Given some further rules that will be described below it is easily possible to employ *pdl-seqB-simp* to prove theorems about the equivalence of multiply split boxes and their 'joint box':

lemma $\vdash [\#\ do\ \{x1{\leftarrow}p1;\ x2{\leftarrow}p2;\ x3{\leftarrow}p3;\ r\ x1\ x2\ x3\}]P \longleftrightarrow_D$
$[\#\ x1{\leftarrow}p1][\#\ x2{\leftarrow}p2][\#\ x3{\leftarrow}p3][\#\ r\ x1\ x2\ x3]P$

6.3.4 Theorems and Proof Rules Involving Modal Operators

All theorems of Section 4.1 – which include the distribution of the box operator over finite conjunctions (named *box-conj-imp-distrib* here), the regularity and weakening rules (*pdl-box-reg*, *pdl-dmd-reg*, *pdl-wkB*, *pdl-wkD*) as well as several other lemmas – have also been proved in Isabelle. The proofs thereof are heavily inspired by how they have been carried out 'on paper', so we just refer to Section C.5.2 in the Appendix for a formal Isabelle verification.

A more interesting proof which has not been presented before, because it relies on the underlying logic being classical, is the relationship between the box and diamond operator. It has already been stated that the propositional part of the logic behaves classical, but the following theorem confirms that this is also true for the relation between the modal operators.

theorem *dmd-box-rel*: $\vdash \langle x{\leftarrow}p\rangle(P\ x) \longleftrightarrow_D \neg_D\ [\#\ x{\leftarrow}p](\neg_D\ P\ x)$

The formula is proved in two steps, each one validating one direction of the equivalence. The first half, in which the definition of negation[7] has already been unfolded, looks as follows. The Isar keyword **is** introduces an abbreviation for the term preceding it. In the case at hand, ?*b* and ?*d* are matched against and bound to the box and diamond formulae $[\#\ x \leftarrow p]\ P\ x \longrightarrow_D \mathit{Ret\ False}$ and $\langle x \leftarrow p\rangle\ P\ x$ respectively.

lemma *dmd-box-rel1*: $\vdash ([\#\ x{\leftarrow}p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \langle x{\leftarrow}p\rangle(P\ x)$
  (**is** $\vdash (?b \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?d$)
proof –
  have $\vdash (?d \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?b$
  proof –
    have *f1*: $\vdash ((?d \longrightarrow_D [\#\ x{\leftarrow}p](\mathit{Ret\ False})) \longrightarrow_D ?b) \longrightarrow_D$
        $(?d \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?b$
      by (*simp add*: *pdl-taut*)
    have *f2*: $\vdash (?d \longrightarrow_D [\#\ x{\leftarrow}p](\mathit{Ret\ False})) \longrightarrow_D ?b$
      by (*rule pdl-k5*)
    from *f1 f2* **show** *?thesis* **by** (*rule pdl-mp*)
  qed
  thus *?thesis* **by** (*simp add*: *pdl-taut*)
qed

The proof proceeds by classical contraposition, i. e. instead of proving the main goal we initially show the following formula

$$(\langle x \leftarrow p\rangle\ (P\ x) \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D [\#\ x \leftarrow p]\ (P\ x \longrightarrow_D \mathit{Ret\ False})$$

(call it Φ) to hold and let the simplifier conclude the proof by equalising Φ and the main goal with the help of *pdl-taut*. Noticing that Φ already looks quite similar to an instance of axiom *pdl-k5* – with only the leftmost *Ret* term to be replaced by the formula $[\#\ x \leftarrow p]\ (\mathit{Ret\ False})$ – we recognise that the axiom really implies the goal, i. e. for an appropriate instance Ψ of axiom *pdl-k5* we have that $\Psi \longrightarrow_D \Phi$ is a tautology that can once again be proved by the simplifier. Hence, a final application of modus ponens finishes the proof.

The second half of the equivalence is easily proved, as it is tautologically implied by *pdl-k3D* and *pdl-k2*:

lemma *dmd-box-rel2*: $\vdash \langle x{\leftarrow}p\rangle(P\ x) \longrightarrow_D [\#\ x{\leftarrow}p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}$
proof –
  have $\vdash (\langle x{\leftarrow}p\rangle(\mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D$
    $([\#\ x{\leftarrow}p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \langle x{\leftarrow}p\rangle(P\ x) \longrightarrow_D \langle x{\leftarrow}p\rangle(\mathit{Ret\ False})) \longrightarrow_D$

[7] $\neg_D P \equiv P \longrightarrow_D \mathit{Ret\ False}$

$\langle x{\leftarrow}p\rangle(P\ x) \longrightarrow_D [\#\ x{\leftarrow}p](P\ x \longrightarrow_D Ret\ False) \longrightarrow_D Ret\ False$
by (*simp add*: *pdl-taut*)
from *this pdl-k3D pdl-k2* **show** *?thesis* **by** (*rule pdl-mp-2x*)
qed

6.4 A Specification of Parser Combinators

In this section it is shown how the monad-independent specification of the calculus of dynamic logic can be extended by axioms to describe a monad of basic parser combinators. This specification has been heavily influenced by the Haskell implementation presented in [12], but in contrast to that work we specified a *deterministic* parser monad with fall back alternatives. The basic operations of this parser are

consts
item :: *nat T*
fail :: $'a\ T$
alt :: $'a\ T \Rightarrow 'a\ T \Rightarrow 'a\ T$
getInput :: *nat list T*
setInput :: *nat list* $\Rightarrow$ *unit T*

where *item* parses exactly one natural number from the finite stream of input numbers, *fail* is a parser that always fails, thus representing a dead end, the combinator *alt* (syntactically sugared by two parallel bars '‖') takes two parsers p and q and yields a parser that runs the first argument of *alt* – let it be p – first, and only if it fails the second parser q is tried. Every producible parser thus always yields at most one result. Finally there are operations *getInput* and *setInput* to read and set the remaining input stream. As a typical implementation of this monad one might use a deterministic state monad with an added exception representing failure.

As an abbreviation we also introduced the operation *eot* (for 'end of text') which is defined through *getInput* in the obvious way. In accordance with the convention of Remark 6.2 the operations *Eot* and *GetInput* denote the operations in D corresponding to the dsef operations in T written in lower case.

6.4.1 Specification of the Basic Parsers

axioms
determ: $\vdash \langle x{\leftarrow}p\rangle(P\ x) \longleftrightarrow_D [\#\ x{\leftarrow}p](P\ x) \wedge_D \langle x{\leftarrow}p\rangle(Ret\ True)$
dsef-getInput: *dsef getInput*
fail-bot: $\vdash [\#\ fail](\lambda x.\ Ret\ False)$
eot-item: $\vdash Eot \longrightarrow_D [\#\ x{\leftarrow}item](Ret\ False)$
set-get: $\vdash \langle setInput\ x\rangle(\lambda u.\ GetInput =_D Ret\ x)$
get-item: $\vdash GetInput =_D Ret\ (y\#ys) \longrightarrow_D \langle x{\leftarrow}item\rangle(Ret\ (x = y) \wedge_D GetInput =_D Ret\ ys)$
altB-iff: $\vdash [\#\ x{\leftarrow}p\|q](P\ x) \longleftrightarrow_D (\ [\#\ x{\leftarrow}p](P\ x) \wedge_D \langle x{\leftarrow}p\rangle(Ret\ True)\ ) \vee_D$
$(\ [\#\ x{\leftarrow}q](P\ x) \wedge_D [\#\ x{\leftarrow}p](Ret\ False)\ )$
altD-iff: $\vdash \langle x{\leftarrow}p\|q\rangle(P\ x) \longleftrightarrow_D \langle x{\leftarrow}p\rangle(P\ x) \vee_D (\langle x{\leftarrow}q\rangle(P\ x) \wedge_D [\#\ x{\leftarrow}p](Ret\ False))$

An interesting axiom is *determ* which captures the fact that we are working in a deterministic monad. The characteristic feature of such a monad is that the box and diamond operators denote nearly the same formula, with the diamond being stronger in the sense that it additionally asserts termination. So when formalising the total correctness of parsers in the parser monad one can either keep partial correctness and termination separated, or one can jointly specify them by using the diamond operator. The latter has been done in the remainder of the specification

The operations *getInput*, *setInput* and *item* behave as one would expect, such that reading the remaining input is deterministically side effect free (*dsef-getInput*), trying to read further input when the end has been reached results in an error (*eot-item*), after setting the input to x this value can be read by *getInput* (*set-get*) and reading an item when input is available is a terminating operation that diminishes the remaining input by one item and yields this item as a result (*get-item*).

Remark 6.4. Attention has to be paid when specifying the equality of a dsef term with a stateless value. For example, to express that the remaining input equals some list of numbers l, one cannot write $\mathit{GetInput} = l$ as this is a type error. It is moreover also wrong to write $\mathit{GetInput} = \mathit{Ret}\ l$ instead, since this would generally make *GetInput* a stateless program always yielding the same result l. So what one actually wants to express is a *monadic equality* (denoted by $=_D$) to be defined as the lifted counterpart to standard equality – in the same way as for the propositional connectives:

$$a =_D b \quad \equiv \quad \Uparrow (\mathit{liftM2}(\mathit{op} =)(\Downarrow a)(\Downarrow b))$$

Axiom *altB-iff* characterises the more complex behaviour of this monad. It states in what cases a formula P holds for the outcome of the combined parser $p\|q$. The fall back behaviour of this parser with respect to q is captured by the assertion that $[\#\, x \leftarrow p\|q]\, P\ x$ holds if and only if p makes P true and p terminates, or q makes P true, but p does not terminate. In the case where both parsers fail $[\#\, x \leftarrow p\|q]\, P\ x$ will always be provable due to axiom *fail-bot*. Describing the total correctness behaviour of *alt*, i. e. the formula $\langle x \leftarrow p\|q\rangle\, P\ x$, axiom *altD-iff* looks quite similar, only that one assertion in the left part of the disjunction – namely that p must terminate – may be omitted, since it is implied by the formula $\langle x \leftarrow p\rangle\, P\ x$.

6.4.2 Defining Complex Parsers

One can now define complex parsers in terms of the basic ones. The following are a parser *sat* that accepts an item if it satisfies a given predicate and otherwise fails as well as a parser that accepts numbers between zero and nine, which we will treat as *digits* in the sequel.

constdefs

$\mathit{sat} \quad :: (\mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow \mathit{nat}\ T$
$\mathit{sat}\ p \ \equiv\ \mathit{do}\ \{x \leftarrow \mathit{item};\ \mathit{if}\ p\ x\ \mathit{then}\ \mathit{ret}\ x\ \mathit{else}\ \mathit{fail}\}$
$\mathit{digitp} \quad :: \mathit{nat}\ T$
$\mathit{digitp}\ \equiv\ \mathit{sat}\ (\lambda x.\ x < 10)$

A useful compound parser is one that repeatedly applies a given parser p, collecting the results of p until p fails. Sometimes it is useful to require that at least one run of p has to yield a result, leading to the definition of the combinators *many* and *many1*. Unfortunately, *many* has to be axiomatised rather than defined, because its definition would not result in a total function (cf. Rem. 6.5 in the next section for an exposition).

consts

$many\ \ ::\ 'a\ T \Rightarrow 'a\ list\ T$
$many1 :: 'a\ T \Rightarrow 'a\ list\ T$
axioms
$many\text{-}unfold$: $many\ p = ((do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\}) \parallel ret\ [])$
defs
$many1\text{-}def$: $many1\ p \equiv (do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\})$

The *many* combinator critically depends on the *alt* operation, as it tries to run p as many times as possible, but as soon as p fails, it will fall back on its alternative, which is to return an empty list. The axiom *many-unfold* can be used to formulate a rule for *many* that resembles its operational semantics, i. e. one can prove

lemma *many-step*: $[\![\vdash \langle (do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\}) \rangle P \vee_D$
$\langle ret\ [] \rangle P \wedge_D [\#\ x{\leftarrow}p](Ret\ False)\ ]\!] \Longrightarrow \vdash \langle many\ p \rangle P$

What one actually would like to have is some kind of introduction rule for *many*, i. e. one in which *many* occurs only in the conclusion. Ideally, this would then make proofs about *many* much like proofs involving while loops in the state monad, where an assertion about a loop can be reduced to an assertion involving only the loop body. However, as yet we do not see what such a rule might look like, respectively whether it can be formulated in the calculus at all.

As an example specification within the monad presented here we define a parser that extends *digitp* to obtain a parser for natural numbers, i. e. a parser that reads as many digits as possible and turns them into the corresponding number. For instance, given the input $[1,2,3,42]$ it is supposed to parse the digits up to and including 3 and yield 123 as a result. The remaining input is then expected to be $[42]$. This parser can easily be defined with the help of *many1*:

constdefs
$natp :: nat\ T$
$natp \equiv do\ \{ns \leftarrow many1\ digitp;\ ret\ (foldl\ (\lambda r\ n.\ 10 * r + n)\ 0\ ns)\}$

One can now go on prove the *total* correctness of this parser for concrete inputs. This can be done rather conveniently, due to the fact that we can cover both partial correctness and termination by expressing the assertion in terms of the diamond operator. The following simple example can now be proved in a straightforward manner (cf. Section C.6.3 for the complete proof).

theorem *natp-corr*: $\vdash \langle do\ \{uu{\leftarrow}setInput\ [1];\ natp\} \rangle (\lambda n.\ Ret\ (n = 1) \wedge_D Eot)$

6.5 A Specification of Russian Multiplication

This final part of the overview of how the calculus of monadic dynamic logic has been implemented in Isabelle describes the specification of a reference monad with while loops. The specification only allows for partial correctness proofs since we only provide axioms for the box operator. The extensions required to be able to perform total correctness proofs are mostly straightforward, with merely the rule for total correctness of while loops being an exception. To specify the latter one has to introduce a termination measure along the lines of the rule found in Section 4.4.2.

consts
newRef :: $'a \Rightarrow 'a\ ref\ T$
readRef :: $'a\ ref \Rightarrow 'a\ T$
writeRef :: $'a\ ref \Rightarrow 'a \Rightarrow unit\ T$ $((\text{-} := \text{-})\ [100, 10]\ 10)$
monWhile :: $bool\ D \Rightarrow unit\ T \Rightarrow unit\ T$ $(WHILE\ (4\text{-})\ /DO\ (4\text{-})\ /END)$

These are the basic operations of the monad, where $'a\ ref$ is the type of references containing values of type $'a$. The syntax annotations for the while-loop operator *monWhile* let one write *WHILE b DO p END* instead of *monWhile b p*. A further syntactical sugaring is provided by the term $*r$ which is short for $\Uparrow (readRef\ r)$, i. e. the formula representing the value of reference r; here we have chosen to stick to the convention of using the asterisk notation $*r$ instead of introducing an operation *ReadRef r*.

Remark 6.5. The while-loop operator is in fact not a truly basic operation of the monad. One would certainly prefer to define it recursively in the natural way:

$$monWhile\ b\ p \quad \equiv \quad \mathsf{do}\,\{a \leftarrow b; \mathsf{if}\ a\ \mathsf{then}\ \mathsf{do}\,\{p; monWhile\ b\ p\}\ \mathsf{else}\ ret\ *\}$$

but this is impossible in HOL since the above equation is not a real definition: it mentions *monWhile* on both sides. What one actually wants to state is that *monWhile* is the *least* function satisfying this equation (cf. [30] for a discussion of least fixed points). To make such a statement possible one would either have to add a substantial amount of infrastructure to HOL to enable it to cope with cpos and function definitions thereupon, or base the calculus of dynamic logic on HOLCF [21], the framework of computable functions on top of HOL. Fortunately, it is not so important to be able to define *monWhile*, because we are only interested in its logical characterisation, which can be given in HOL, too.

axioms
dsef-read: $dsef\ (readRef\ r)$
read-write: $\vdash [\#\, r := x](\lambda uu.\ *r =_D Ret\ x)$
read-write-other-gen: $\vdash\ \Uparrow (do\ \{u {\leftarrow} readRef\ r;\ ret\ (f\ u)\}) \longrightarrow_D$
$[\#\, s := y](\lambda uu.\ Ret\ (r {\neq} s) \longrightarrow_D\ \Uparrow (do\ \{u {\leftarrow} readRef\ r;\ ret\ (f\ u)\}))$
while-par: $\vdash P \wedge_D b \longrightarrow_D [\#\, p](\lambda u.\ P) \Longrightarrow \vdash P \longrightarrow_D [\#\ WHILE\ b\ DO\ p\ END](\lambda x.\ P \wedge_D \neg_D\ b)$

Rule *while-par* is really just a translation of the standard while rule for partial correctness into dynamic logic, such that the formula P can be thought of as some kind of loop invariant. A peculiarity of this specification is axiom *read-write-other-gen*, which constitutes a generalisation of axiom *read-write-other* of Section 4.3. It expresses the fact that any stateless assertion that holds for the value of a reference r – notice that this means that the asserting formula itself is *not* stateless, as it depends on $*r$ – continues to hold after a value is assigned to a different reference s. It is in fact not necessary to employ do-terms to specify this fact since one can prove the following equivalence and solely work with the *Ret* terms of the left-hand side.

lemma $\vdash *r =_D Ret\ b \wedge_D Ret\ (f\ b) \longleftrightarrow_D\ \Uparrow (do\ \{a {\leftarrow} readRef\ r;\ ret\ (f\ a \wedge a = b)\})$

Remark 6.6. We have opted to work with do-terms for the following reason. The invariant P present in the rule for while loops virtually always involves some non-trivial arithmetical

$$
\begin{aligned}
&\mathit{rumult}\ a\ b\ x\ y\ r \ \equiv \\
&\quad \mathsf{do}\ \{x := a;\ y := b;\ r := 0; \\
&\qquad \mathit{WHILE}\ (0 < *x) \\
&\qquad \mathit{DO}\ \mathsf{do}\ \{\mathsf{if}\ (\mathit{odd}\ *x)\ \mathsf{then}\ r := (*r + *y)\ \mathsf{else}\ \mathit{ret}\ *; \\
&\qquad\qquad x := *x\ \mathit{div}\ 2; \\
&\qquad\qquad y := *y * 2\ \} \\
&\qquad \mathit{END}; \\
&\qquad *r\ \}
\end{aligned}
$$

Figure 6.2: Simplified specification of the Russian multiplication algorithm

relation between the references occurring in the program. To state this relation in terms of monadic formulae one would have to lift several arithmetical operations like addition, multiplication, integer division, etc. to form monadic operators. But for these, there would be no automatic proof procedure available – and it would indeed require a quite an amount of work to change this fact. We found it preferable to go along with the slightly less readable do-notation and in turn be able to employ the arithmetical reasoner *arith* that is built into HOL. As an example, take the following two formulae which are equivalent and both valid, but where the second one requires a lifted multiplication $\cdot_D$ and where to prove the second formula one would initially have to decode it into the first formula manually.

$$\vdash \Uparrow \mathsf{do}\,\{a \leftarrow \mathit{readRef}\ r; b \leftarrow \mathit{readRef}\ s; \mathit{ret}\ (b = 0 \longrightarrow a \cdot b = 0)\}$$
$$\vdash *s =_D \mathit{Ret}\ 0 \longrightarrow_D *r \cdot_D *s =_D \mathit{Ret}\ 0$$

It is now possible to verify the partial correctness of several imperative programs. The specification in Figure 6.2 determines a program performing the so-called *Russian multiplication* that carries out the multiplication of two natural numbers loosely resembling the way how (unoptimised) multiplication is performed in hardware. The specification as well as the following proof outline is presented in a stylised form as it is done in Chapter 4; we refer to Section C.7 for the concrete Isabelle definition which lacks the notational conventions applied to dsef terms here. The function *rumult* expects references x, y and r and two values a and b. It will set the reference r to the value of $a \cdot b$, making auxiliary use of x and y.

6.5.1 Proof Sketch

The partial correctness specification for the Russian multiplication algorithm is straightforward: assuming there are three distinct variables x, y and r, execution of *rumult* will yield the value $a \cdot b$.

$$\vdash \mathit{Ret}\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ \mathit{rumult}\ a\ b\ x\ y\ r]\,(\lambda x. \mathit{Ret}\ (x = a \cdot b))$$

The major proof steps as conducted in Isabelle are documented in Section C.7.3, so we only convey the basic ideas here. The first part of the proof is to make one's way up to the while loop, i. e. one has to unfold the definition of *rumult* and employ the rules *read-write* and *read-write-other* and *pdl-k3B*. These cannot be applied directly however; several applications of structuring rules have to be interspersed that manipulate the unification variable representing the desired 'postcondition' so as to obtain the right form. To make this idea clearer, take the following as an example. Imagine one has arrived at the proof goal

$$A \longrightarrow_D [\# \, x := a] \, ?B \tag{6.1}$$

with $?B$ being the unification variable that must be instantiated, or the 'postcondition'[8] that has to be found. Given the rules

$$A \longrightarrow_D [\# \, x := a] \, C \quad \text{and} \quad A \longrightarrow_D [\# \, x := a] \, D \tag{6.2}$$

one has to invent, i. e. prove, a structuring rule like

$$\frac{\begin{array}{c} A \longrightarrow_D [\# \, x := a] \, C \\ A \longrightarrow_D [\# \, x := a] \, D \end{array}}{A \longrightarrow_D [\# \, x := a] \, (C \wedge_D D)}$$

so that an application of this rule to (6.1) unifies $?B$ with $C \wedge_D D$ and one can prove the resulting two new goals by the given facts of (6.2). Of course one *can* prove (6.1) with each of the two given facts, but this would make the instantiation of $?B$ too weak a formula to be useful in the sequel in many cases.

The heart of the invariant of the while loop is the relation between the references x, y and r. We state explicitly

$$INV \quad \equiv_{\text{def}} \quad *x \cdot *y + *r = a \cdot b \wedge_D Ret \; (x \neq y \wedge y \neq r \wedge x \neq r)$$

i. e. all references remain distinct and the value of $*r$ added to the product of $*x$ and $*y$ is equal to the desired result $a \cdot b$. Since $*x = 0$ will hold after termination of the while loop, one will be able to infer that $*r = a \cdot b$ holds, so that the final read operation on r (cf. Fig. 6.2) makes *rumult* yield the specified result.

Having applied the while rule *while-par* in the proof, it remains to be shown that the loop invariant can be re-established after a single execution of the loop body. The main arithmetical facts exploited within the loop body relate to integer division by two. One has

$$\begin{aligned} (n \; div \; 2 + n \; div \; 2) &= n && \text{if } n \text{ is even} \\ (n \; div \; 2 + n \; div \; 2) + 1 &= n && \text{if } n \text{ is odd} \end{aligned}$$

so the following assertion can be shown

$$INV \longrightarrow_D [\# \, \mathsf{if} \; (odd \; *x) \; \mathsf{then} \; r := (*r + *y) \; \mathsf{else} \; ret \; *] \, ((*x \; div \; 2 + *x \; div \; 2) \cdot *y + *r = a \cdot b)$$

The remaining two assignments to x and y inside the loop body then transform the formula in the scope of the box back into the loop invariant. The stateless formula ensuring the distinctness of all three references prevails in the whole body by virtue of *pdl-k3B*.

To obtain a *total correctness verification* of this algorithm in an extended specification of the reference monad, one essentially needs to find a termination measure living in a type equipped with a well-founded relation. This measure then has to decrease strictly (with respect to the well-founded relation) in each run of the loop body. The obvious candidate for such a measure is $*x$, which will unconditionally be decreased in each run, since the assertion provided by the loop exit condition $(0 < *x)$ ensures that $*x$ is strictly greater than $*x \; div \; 2$.

[8] we speak of a postcondition here, since the structure of the formula is precisely that which may be used to interpret Hoare assertions in dynamic logic

6.5.2 Similarity to Hoare Logic Proofs

Proofs in the reference monad are basically just Hoare logic proofs retranslated into the syntax of dynamic logic. A reference monad containing a while loop as its sole algorithmically expressive construct is just the monadic model of a simple while-language. It is to such a language that Hoare logics have been applied successfully first, and they can indeed be regarded as a natural way of doing verification in such a language. Recalling that a Hoare triple $\{A\}\, x \leftarrow p\, \{B\}$ can be encoded by the formula $A \longrightarrow_D [\#\, x \leftarrow p]\,(B\, x)$, the sequencing rule of an appropriate Hoare calculus

$$\frac{\{A\}\, x \leftarrow p\, \{B\} \qquad \{B\}\, y \leftarrow q\, \{C\}}{\{A\}\, x \leftarrow p; y \leftarrow q\, \{C\}}$$

is basically just the weakening rule $(wk\Box)$ of Lemma 4.3 which has been implemented as rule *pdl-wkB* in Isabelle. This is to say that proofs about programs in the monad presented here proceed stepwise – i. e. by handling each atomic expression separately – by applications of the following rule. This rule combines the effects of the weakening and sequencing rules.

$$(\textit{pdl-plugB-lifted1}) \quad \frac{\vdash A \longrightarrow_D [\#\, x \leftarrow p]\,(B\, x) \qquad \forall x.\ \vdash B\, x \longrightarrow_D [\#\, y \leftarrow q\, x]\,(C\, y)}{A \longrightarrow_D [\#\, x \leftarrow p; y \leftarrow q\, x]\,(C\, y)}$$

In a backward proof this rule introduces two goals with an initially uninstantiated formula variable B. To look for an optimal initialisation of this variable with respect to the first premiss is the same as trying to find the *strongest postcondition* of the corresponding Hoare assertion $\{A\}\, x \leftarrow p\, \{?\}$, i. e. an instantiation P of B such that for every other formula Q making $A \longrightarrow_D [\#\, x \leftarrow p]\,(Q\, x)$ true, one has $\forall x.\, P\ x \longrightarrow_D Q\ x$. The notion of strongest postcondition might be well worth being formalised in our calculus, but in the example verification of the algorithm for Russian multiplication we have only established those ‘postconditions’ that suffice for the remaining proof to go through.

7 Conclusion and Outlook

In this thesis we have described a program logic for programs formulated in the do-notation of monads. After having recalled that monads are an elegant and effective means to model several kinds of computational effects like state, input and output, exceptions, or nondeterminism we have depicted the development of this *monadic dynamic logic*. The prominent features of the logic are that

1. Programs with certain well-behavedness properties making them deterministically side effect free are taken as formulae of the logic

2. Modal operators allow one to make statements of the form “after execution of the program p, the formula ϕ will hold”

3. The modal operators are entirely interpreted within the underlying monad (presupposing the monad satisfies certain additional conditions); no additional structure is required.

The calculus has been extended by further axioms, rules and the *mbody* operation to evolve into a suitable logic for reasoning about abrupt termination in Java. In this extension the correctness of a pattern match algorithm has been verified. Back in the basic calculus we have then specified and proved correct an implementation of a breadth-first search algorithm in the queue monad, which represents a rather complex example on how to apply the general calculus to realistic programs. Finally, the calculus has been implemented on top of higher-order logic in the proof assistant Isabelle. In this formalisation further monads like the reference monad and a monad for parser combinators have been specified. To help automatise simple proof obligations, Isabelle’s simplifier has been extended to become able to prove tautologies of dynamic logic automatically.

The implementation in Isabelle made it obvious that the formulation of the calculus in Hilbert-style, i. e. with several axioms and only the two proof rules necessitation and modus ponens, makes proofs of rather simple theorems quite expensive in terms of the required proof steps. The extension of the simplifier to solve tautologies automatically is already a great help, but of course tautologies do not constitute the most interesting part of the valid formulae of dynamic logic. It has been pointed out that the major problem why we cannot provide a natural deduction system for the calculus is the lack of an appropriate rule for implication introduction. This also obviates the employment of Isabelle’s classical reasoner; it is thus an interesting question whether a sequent or tableaux calculus can be found for the logic that allows for more automation than has been achieved in this thesis.

It has turned out that proofs in monads where a Hoare calculus for total correctness can be given – most notably this applies to the state monad – proofs as conducted in dynamic logic actually resembled the proof style for Hoare logics. This is to say that proofs mostly proceeded in a sequential fashion in which the modal operators were mainly indexed by the program fragment to be verified; thus the only necessary modal expression was to state what will hold after execution of the main program. It might therefore turn out to be useful to formulate a Hoare calculus on top of the formalisation of dynamic logic in Isabelle in which modal formulae do not appear in the precondition or postcondition. The formulation of such a calculus for total correctness would have the additional benefit of removing the duplicate proof obligations that arose in dynamic logic due to the fact that in the latter termination and correctness are expressed by two distinct formulae.

Finally, it would be nice to undermine the implementation provided in this thesis with further foundations to make several axioms unnecessary. In particular, the formalisation of global dynamic judgements would make it possible for several monads to actually define the modal operators. Since this formalisation is currently being worked out in a different diploma thesis this should not constitute a major problem. Given a definition of the modal operators, it would also be much more rewarding to actually define concrete monads instead of just axiomatising their characteristic properties, because then one could go on and actually establish these properties as theorems.

Acknowledgements

First and foremost, I want to thank my family and Jenny for their love and support especially in but of course not limited to the time during which I wrote this thesis. Thanks also to my fellow students Martin Kühl and Tina Kraußer for fruitful discussions and suggestions on how to improve this thesis. Last but not least thanks go to my supervisors Lutz Schröder and Till Mossakowski who always offered their expertise and advice when problems arose.

Appendix A

Haskell Implementation of mbody

We present here a complete Haskell implementation of the *mbody* construct described in Section 3.4.1. As an example application the pattern match algorithm that has been verified in Section 3.4.1 and in [38] is used.

```
module MBodyTrans
where

import Control.Monad.Error
import Control.Monad.State

data Exception a = Excpt String
                 | Ret a
                 | DropOff
                   deriving (Show)


-- Needed for class dependencies; actually only for fail
-- which is not used in our calculus.
instance Error (Exception a) where
    noMsg = Excpt ""
    strMsg = Excpt
```

Rather than defining the binding in an exception monad by ourselves, we make use of the *exception monad transformer* `ErrorT` from the Haskell libraries. The type of exceptions consists of three alternatives; exceptions either signal failure through `Excpt` with a message attached, or they carry a return value of some method, or they indicate that a method has illegally terminated normally (`DropOff`). For simplicity, `continue` and `break` have been left out, but could easily be added.

For every monad m we can construct a monad $(Ex\ m\ e)$ that behaves just like m in the absence of an exception, but also allows exceptions to be thrown and caught.

```
type Ex m e a = ErrorT (Exception e) m a
```

Recall the instance definition of `ErrorT` from the Haskell libraries:

```
instance (Monad m, Error e) => Monad (ErrorT e m) where
        return a = ErrorT $ return (Right a)
        m >>= k  = ErrorT $ do
                a <- runErrorT m
```

```
                case a of
                        Left  l -> return (Left l)
                        Right r -> runErrorT (k r)
        fail msg = ErrorT $ return (Left (strMsg msg))
```

which precisely captures the intended behaviour of the binding in the presence of an exception, namely that the right-hand argument is only evaluated if the left one terminated normally. The function `runErrorT` simply unpacks the inner monad, i. e. drops the constructor `ErrorT`.

The concrete state monad that will be used below needs a single reference of type `Int`, but the general variable mapping can be defined as follows. A variable map consists of an ID for the next reference and a function mapping references to their values:

```
type Ref = Int

type VMap a = (Int, Ref -> a)
```

Next comes the `mbody` construction that catches `Ret` exceptions and converts them into normal return values. All other exceptions are propagated unchanged. This implementation is polymorphic in the exception type of the result and thus allows for switching between monads. Whether the input computation should be polymorphic in its return type or whether $*$ should be enforced is a matter of taste. `mret` emulates the actual Java `return` statements – whereas `return` is the usual monadic *ret* function.

```
mret :: Monad m=> e -> Ex m e a
mret x = throwError (Ret x)

mbody :: Monad m=> Ex m e () -> Ex m e1 e
mbody p = ErrorT $ do
          a <- runErrorT p      -- binding in the "inner" monad
          case a of
            Right () -> return (Left DropOff)
            Left e   -> case e of
              Ret x    -> return (Right x)
              Excpt s  -> return (Left (Excpt s))
              DropOff  -> return (Left DropOff)
```

There are three state-related operations on exception state monads: reading, writing and creation of variables. A generic while loop for the exception state monad is also easily defined.

```
readVar :: Ref -> Ex (State (VMap a)) e a
readVar r = do (_, f) <- get
               return (f r)

wrtVar :: Ref -> a -> Ex (State (VMap a)) e ()
wrtVar r x = do (n, f) <- get
                put (n, \ k-> if k == r then x else f k)
```

```
newVar :: a -> Ex (State (VMap a)) e Ref
newVar v = do (n, f) <- get
              put (n+1, \ k-> if k == n then v else f k)
              return n

while :: Monad m=> Ex m e Bool -> Ex m e () -> Ex m e ()
while b p = do v <- b
               if v then do p; while b p
                    else return ()
```

The pattern match algorithm, as described in [38, 11]. For testing purposes, here's how to evaluate pmatch with an initial state with all references defaulting to 0:

```
evalState (runErrorT (pmatch base1 pat1)) (0, const 0)
```

which will evaluate (correctly) to `Right 9`

```
pmatch :: String -> String -> Ex (State (VMap Int)) e Int
pmatch base pat = mbody $ do
    r <- newVar 0
    s <- newVar 0
    while (return True)
          (do u <- readVar r
              v <- readVar s
              if u == length pat
                then mret v
                else if v + u == length base
                  then throwError (Excpt "Pattern not found")
                  else if base!!(v+u) == pat!!u
                    then wrtVar r (u+1)
                    else do wrtVar s (v+1); wrtVar r 0)

-- Some sample patterns
base1 :: String
base1 = "puff the magic dragon"

pat1 :: String
pat1 = "magic"

pat2 :: String
pat2 = "mary"
```

Appendix B

Table of Rules of Isabelle/HOL

Since the main purpose of the implementation in Isabelle was to set up a new logic, only few deep theorems of Isabelle/HOL itself, on which the logic is based, have been made use of. Further, many rules are applied implicitly when employing the simplifier or the classical reasoner. The following is a list of the rules that appear verbatim in the implementation.

allI	$(\bigwedge x. P\,x) \Longrightarrow \forall x. P\,x$
arg_cong	$x = y \Longrightarrow f\ x = f\ y$
cong	$[\![f = g;\ x = y]\!] \Longrightarrow f\ x = g\ y$
conjE	$[\![P \wedge Q;\ [\![P;\ Q]\!] \Longrightarrow R]\!] \Longrightarrow R$
conjI	$[\![P;\ Q]\!] \Longrightarrow P \wedge Q$
conjunct1	$[\![P \wedge Q]\!] \Longrightarrow P$
exE	$[\![\exists x. P\ x;\ \bigwedge x. P\ x \Longrightarrow Q]\!] \Longrightarrow Q$
FalseE	$\mathit{False} \Longrightarrow P$
iffD1	$[\![Q = P;\ Q]\!] \Longrightarrow P$
iffD2	$[\![P = Q;\ Q]\!] \Longrightarrow P$
iffI	$[\![P \Longrightarrow Q;\ Q \Longrightarrow P]\!] \Longrightarrow P = Q$
impI	$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
mp	$[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$
notE	$[\![\neg P;\ P]\!] \Longrightarrow R$
notI	$(P \Longrightarrow \mathit{False}) \Longrightarrow \neg P$
refl	$t = t$
spec	$\forall x.\ P\ x \Longrightarrow P\ y$
subst	$[\![s = t;\ P\ s]\!] \Longrightarrow P\ t$

Table B.1: Derived rules of inference for HOL

Appendix C

Isabelle Theories

The following sections present the concrete implementation of the calculus of dynamic logic in Isabelle. The typesetting has been automatically taken care of by the `isatool` mechanism of the Isabelle distribution, which directly extracts this information from the given theory files. The proofs of some rather technical statements which are only used as auxiliary lemmas in other proofs have been omitted. This chapter is intended for reference usage and not so much for being perused sequentially. Refer to Chapter 6 for a conceptual description of the implementation.

C.1 Basic Monad Definitions and Laws.

theory *Monads* = *Main*:

For the lack of constructor classes in Isabelle, we initially use functor T as a parameter standing for the monad in question.

typedecl $'a\ T$

arities $T :: (type)type$

Monadic operations, decorated with Haskell-style syntax.

consts
$bind :: {'a}\ T \Rightarrow ({'a} \Rightarrow {'b}\ T) \Rightarrow {'b}\ T \quad (\textbf{infixl} \gg= 20)$
$ret\ :: {'a} \Rightarrow {'a}\ T$

constdefs
$seq :: {'a}\ T \Rightarrow {'b}\ T \Rightarrow {'b}\ T \qquad (\textbf{infixl} \gg 20)$
$p \gg q \equiv (p \gg= (\lambda x.\ q))$

The usual monad laws for bind and ret (not the Kleisli triple ones) including injectivity of *ret* for convenience.

axioms
$bind\text{-}assoc\ [simp]: (p \gg= (\lambda x.\ f\ x \gg= g)) = (p \gg= f \gg= g)$
$ret\text{-}lunit\ [simp]: (ret\ x \gg= f) = f\ x$
$ret\text{-}runit\ [simp]: (p \gg= ret) = p$
$ret\text{-}inject: ret\ x = ret\ z \Longrightarrow x = z$

lemma $seq\text{-}assoc\ [simp]: (p \gg (q \gg r)) = (p \gg q \gg r)$
by $(simp\ add: seq\text{-}def)$

This sets up a Haskell-style '$do\ \{x{\leftarrow}p;\ q\}$' syntax with multiple bindings inside one *do* term.

nonterminals
monseq

syntax (*xsymbols*)
$\textit{-monseq} :: \textit{monseq} \Rightarrow {}'a\ T \qquad ((do\ \{(\text{-})\})\quad [5]\ 100)$
$\textit{-mongen} :: [\textit{pttrn}, {}'a\ T, \textit{monseq}] \Rightarrow \textit{monseq} \quad ((\text{-}{\leftarrow}(\text{-});/\ \text{-})\ \ [10, 6, 5]\ 5)$
$\textit{-monexp} :: [{}'a\ T, \textit{monseq}] \Rightarrow \textit{monseq} \qquad ((\text{-};/\ \text{-}) \qquad [6, 5]\ 5)$
$\textit{-monexp0} :: [{}'a\ T] \Rightarrow \textit{monseq} \qquad ((\text{-}) \qquad 5)$

translations
— input macros; replace do-notation by $op \gg\!= / op \gg$
$\textit{-monseq}(\textit{-mongen}\ x\ p\ q) \rightharpoonup p \gg\!= (\%x.\ (\textit{-monseq}\ q))$
$\textit{-monseq}(\textit{-monexp}\ p\ q) \rightharpoonup p \gg (\textit{-monseq}\ q)$
$\textit{-monseq}(\textit{-monexp0}\ q) \rightharpoonup q$
— Retranslation of into the do-notation
$\textit{-monseq}(\textit{-mongen}\ x\ p\ q) \leftharpoondown p \gg\!= (\%x.\ q)$
$\textit{-monseq}(\textit{-monexp}\ p\ q) \leftharpoondown p \gg q$
— Normalization macros 'flattening' do-terms
$\textit{-monseq}(\textit{-mongen}\ x\ p\ q) \leftharpoondown \textit{-monseq}\ (\textit{-mongen}\ x\ p\ (\textit{-monseq}\ q))$
$\textit{-monseq}(\textit{-monexp}\ p\ q) \leftharpoondown \textit{-monseq}\ (\textit{-monexp}\ p\ (\textit{-monseq}\ q))$

Actually, this rule does not contract, but rather expand monadic sequences, but for historical reasons. . .

lemma *mon-ctr*: $(do\ \{x \leftarrow (do\ \{y \leftarrow p;\ q\ y\});\ f\ x\}) = (do\ \{y \leftarrow p;\ x \leftarrow q\ y;\ f\ x\})$
by(*rule bind-assoc*[*symmetric*])

end

C.2 Basic Notions of Monadic Programs

theory *MonProp* = *Monads*:

C.2.1 Discardability and Copyability

Properties of monadic programs which are needed for the further development, e.g. for the definition of a subtype $'a\ D$ of deterministically side-effect free (*dsef*) programs.

constdefs
— Discardable programs
$dis :: {}'a\ T \Rightarrow bool$
$dis(p) \equiv (do\ \{x{\leftarrow}p;\ ret()\}) = ret\ ()$
— Copyable programs
$cp :: {}'a\ T \Rightarrow bool$
$cp(p) \equiv (do\ \{x{\leftarrow}p;\ y{\leftarrow}p;\ ret(x,y)\}) = (do\ \{x{\leftarrow}p;\ ret(x,x)\})$
— *dsef* programs are *cp* and *dis* and commute with all such programs
$dsef :: {}'a\ T \Rightarrow bool$
$dsef(p) \equiv cp(p) \wedge dis(p) \wedge (\forall q{::}bool\ T.\ cp(q) \wedge dis(q) \longrightarrow$
$\qquad cp(do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\}))$

lemma *dsef-cp*: *dsef p* $\Longrightarrow$ *cp p*
 apply(*unfold dsef-def*)
by *blast*

lemma *dsef-dis*: *dsef p* $\Longrightarrow$ *dis p*
 apply(*unfold dsef-def*)
by *blast*

This is Lemma 4.5 of [34] that allows us to actually discard discardable programs in front of arbitrary programs.

lemma *dis-left*: *dis*(*p*) $\Longrightarrow$ *do* {*p*; *q*} = *q*
proof –
 assume *d*: *dis*(*p*)
 have *do* {*p*; *q*} = *do* {*p*; *ret* (); *q*}
  by (*simp add*: *seq-def*)
 also from *d* **have** ... = *do* {*ret* (); *q*}
  by (*simp add*: *dis-def seq-def del*: *ret-lunit*)
 also have ... = *q* **by** (*simp add*: *seq-def*)
 finally show *?thesis* **.**
qed

Essentially the same as *dis-left*, but expressed with binding.

lemma *dis-left2*: *dis p* $\Longrightarrow$ *do* {*x*←*p*; *q*} = *q*
proof –
 assume *a*: *dis p*
 have *do* {*x*←*p*; *q*} = *do* {*p*; *q*} **by** (*simp only*: *seq-def*)
 also from *a* **have** ... = *q* **by** (*rule dis-left*)
 finally show *?thesis* **.**
qed

This is Lemma 4.22 of [34] which allows us to insert or remove copies of *cp* programs whose result values may be substituted for each other in the following program sequence *r*.

lemma *cp-arb*: *cp p* $\Longrightarrow$ *do* {*x*←*p*; *y*←*p*; *r x y*} = *do* {*x*←*p*; *r x x*}
proof (*unfold cp-def*)
 assume *c*: *do* {*x*←*p*; *y*←*p*; *ret* (*x*, *y*)} = *do* {*x*←*p*; *ret* (*x*, *x*)}
 have *do* {*x*←*p*; *y*←*p*; *r x y*} = *do* {*x*←*p*; *y*←*p*; *z*←*ret*(*x*,*y*); *r* (*fst z*) (*snd z*)}
  by (*simp*)
 also have ... = *do* {*z*←*do* {*x*←*p*; *y*←*p*; *ret*(*x*,*y*)}; *r* (*fst z*) (*snd z*)}
  by (*simp add*: *mon-ctr*)
 also from *c* **have** ... = *do* {*z*←*do* {*x*←*p*; *ret*(*x*,*x*)}; *r* (*fst z*) (*snd z*)}
  by *simp*
 also have ... = *do* {*x*←*p*; *z*←*ret* (*x*,*x*); *r* (*fst z*) (*snd z*)}
  by (*simp add*: *mon-ctr*)
 also have ... = *do* {*x*←*p*; *r x x*}
  by *simp*
 finally show *?thesis* **.**
qed

This is Lemma 4.23 of [34], asserting a weak composability of copyable programs. It is generally not the case that sequences of copyable programs constitute a copyable program.

lemma *weak-cp-seq*: *cp p* $\Longrightarrow$ *cp* (*do* {*x*←*p*; *ret* (*f x*)})
proof –

assume *c*: *cp p*
let *?q* = *do* {*x*←*p*; *ret* (*f x*)}
have *do* {*u*←*?q*; *v*←*?q*; *ret*(*u*,*v*)} = *do* {*x*←*p*; *u*←*ret* (*f x*); *y*←*p*; *v*←*ret* (*f y*); *ret*(*u*,*v*)}
by (*simp add*: *mon-ctr*)
also have ... = *do* {*x*←*p*; *y*←*p*; *ret* (*f x*, *f y*)}
by *simp*
also from *c* **have** ... = *do* {*x*←*p*; *ret* (*f x*, *f x*)}
by (*simp add*: *cp-arb*)
also have ... = *do* {*x*←*p*; *u*←*ret* (*f x*); *ret*(*u*,*u*)}
by *simp*
also have ... = *do* {*u*←*?q*; *ret*(*u*,*u*)}
by (*simp add*: *mon-ctr*)
finally show *?thesis* **by** (*simp add*: *cp-def*)
qed

One can reduce the copyability of a program of a certain form to a simpler form.

lemma *cp-seq-ret*: *cp* (*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)}) ⟹ *cp* (*do* {*x*←*p*; *y*←*q*; *ret* (*f x y*)})
proof −
assume *cp* (*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)})
hence *c*: *cp* (*do* {*u*←*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)}; *ret* (*f* (*fst u*) (*snd u*))})
by (*simp add*: *weak-cp-seq*)
have *do* {*u*←*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)}; *ret* (*f* (*fst u*) (*snd u*))}
= *do* {*x*←*p*; *y*←*q*; *ret* (*f x y*)}
by (*simp add*: *mon-ctr*)
with *c* **show** *?thesis* **by** *simp*
qed

We also have a weak notion of stability under sequencing for *dsef* programs.

lemma *weak-dis-seq*: *dis p* ⟹ *dis* (*do* {*x*←*p*; *ret* (*f x*)})
proof −
assume *d*: *dis p*
have *do* {*z*←*do* {*x*←*p*; *ret* (*f x*)}; *ret* ()} = *do* {*x*←*p*; *z*←*ret* (*f x*); *ret* ()}
by (*simp only*: *mon-ctr*)
also have ... = *do* {*x*←*p*; *ret*()}
by *simp*
also from *d* **have** ... = *ret* () **by** (*simp add*: *dis-def*)
finally show *?thesis* **by** (*simp add*: *dis-def*)
qed

The following lemmas *commute-X-Y* are proofs of the Propositions 4.24 of [34] where *X* is the respective premiss and *Y* is the conclusion.

lemma *commute-1-2*: ⟦*cp q*; *cp p*; *dis q*; *dis p*⟧ ⟹ *cp* (*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)})
⟹ *do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)} = *do* {*y*←*q*; *x*←*p*; *ret*(*x*,*y*)}
proof −
assume *a*: *cp q cp p dis q dis p*
assume *c*: *cp* (*do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)})
let *?s* = *do* {*x*←*p*; *y*←*q*; *ret*(*x*,*y*)}
have *?s* = *do* {*z*←*?s*; *ret* (*fst z*, *snd z*)} **by** *simp*
also from *c* **have** ... = *do* {*w*←*?s*; *z*←*?s*; *ret* (*fst z*, *snd w*)} **by** (*simp add*: *cp-arb*)
also from *a* **have** ... = *do* {*v*←*q*; *x*←*p*; *ret*(*x*,*v*)} **by** (*simp add*: *mon-ctr dis-left2*)
finally show *?thesis* **.**
qed

lemma *commute-2-3*: $[\![cp\ q;\ cp\ p;\ dis\ q;\ dis\ p]\!] \Longrightarrow$
$do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ ret(x,y)\} \Longrightarrow$
$\forall r.\ do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\}$
proof
fix *r*
assume *a*: *cp q cp p dis q dis p*
assume *b*: $do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ ret(x,y)\}$
have $do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ z{\leftarrow}ret(x,y);\ r\ (fst\ z)\ (snd\ z)\}$
by *simp*
also have $\ldots = do\ \{z{\leftarrow}do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\};\ r\ (fst\ z)\ (snd\ z)\}$
by (*simp only*: *mon-ctr*)
also from *b* **have** $\ldots = do\ \{z{\leftarrow}do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ ret(x,y)\};\ r\ (fst\ z)\ (snd\ z)\}$
by *simp*
also have $\ldots = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\}$ **by** (*simp add*: *mon-ctr*)
finally show $do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\}$ **.**
qed

In this case, type annotations are necessary, since we cannot quantify over types of programs. The type for *r* given here is precisely what is needed for the proof to go through.

lemma *commute-3-1*: $[\![cp\ q;\ cp\ p;\ dis\ q;\ dis\ p]\!] \Longrightarrow$
$\forall r{::}'a \Rightarrow {}'b \Rightarrow (('a{*}'b){*}('a{*}'b))\ T.$
$do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\} \Longrightarrow$
$cp\ (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y){::}('a * {}'b)\ T\})$
proof –
let $?s = do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\}$
assume *a*: *cp q cp p dis q dis p*
assume *c*: $\forall r{::}'a \Rightarrow {}'b \Rightarrow (('a{*}'b){*}('a{*}'b))\ T.$
$do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\}$
have $do\ \{w{\leftarrow}?s;\ z{\leftarrow}?s;\ ret\ (w,z)\} = do\ \{u{\leftarrow}p;\ v{\leftarrow}q;\ x{\leftarrow}p;\ y{\leftarrow}q;\ ret((u,v),(x,y))\}$
by (*simp add*: *mon-ctr*)
also from *c* **have** $\ldots = do\ \{u{\leftarrow}p;\ x{\leftarrow}p;\ v{\leftarrow}q;\ y{\leftarrow}q;\ ret((u,v),(x,y))\}$ **by** *simp*
also from *a* **have** $\ldots = do\ \{u{\leftarrow}p;\ v{\leftarrow}q;\ ret((u,v),(u,v))\}$ **by** (*simp only*: *cp-arb*)
also have $\ldots = do\ \{w{\leftarrow}?s;\ ret(w,w)\}$ **by** (*simp add*:*mon-ctr*)
finally show *?thesis* **by** (*simp add*: *cp-def*)
qed

lemma *commute-1-3*: $[\![cp\ q;\ cp\ p;\ dis\ q;\ dis\ p]\!] \Longrightarrow$
$cp\ (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x,y)\}) \Longrightarrow$
$\forall r.\ do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ r\ x\ y\} = do\ \{y{\leftarrow}q;\ x{\leftarrow}p;\ r\ x\ y\}$
— More or less just transitivity of implication
apply(*rule commute-2-3*)
apply(*simp-all*)
apply(*rule commute-1-2*)
apply(*simp-all*)
done

This weird axiom is needed to obtain the general commutativity of a discardable and copyable program from its commuting with all *bool*-valued programs.

axioms

$commute\text{-}bool\text{-}arb$: $[\![dis\ p;\ cp\ p;\ \forall q1{::}bool\ T.\ cp(q1) \wedge dis(q1) \longrightarrow$
$cp(do\ \{x{\leftarrow}p;\ y{\leftarrow}q1;\ ret(x{,}y)\})]\!] \Longrightarrow$
$(\forall q.\ cp(q) \wedge dis(q) \longrightarrow cp(do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ ret(x{,}y)\}))$

In order to introduce the subtype of *dsef* programs, we must prove it is not empty.

theorem *dsef-ret* [*simp*]: *dsef* (*ret x*)
proof (*unfold dsef-def*)
 have *cp* (*ret x*) **by** (*simp add*: *cp-def*)
 moreover have *dis* (*ret x*) **by** (*simp add*: *dis-def*)
 moreover have $(\forall q.\ cp\ q \wedge dis\ q \longrightarrow cp\ (do\ \{x{\leftarrow}ret\ x;\ y{\leftarrow}q;\ ret\ (x,\ y)\}))$
  by (*simp add*: *weak-cp-seq*)
 ultimately show $cp\ (ret\ x) \wedge dis\ (ret\ x) \wedge$
  $(\forall q.\ cp\ q \wedge dis\ q \longrightarrow cp\ (do\ \{x{\leftarrow}ret\ x;\ y{\leftarrow}q;\ ret\ (x,\ y)\}))$
  by *blast*
qed

C.2.2 Introducing the Subtype of *dsef* Programs

Introducing the subtype $'a\ D$ of $'a\ T$ comprising the *dsef* programs; since Isabelle lacks true subtyping, it is simply declared as a new type with coercion functions *Rep-Dsef* :: $'a\ D \Rightarrow\ 'a\ T$ and *Abs-Dsef* :: $'a\ T \Rightarrow\ 'a\ D$ where *Abs-Dsef p* is of course only sensibly defined if *dsef p* holds.

typedef (*Dsef*) ($'a$) $D = \{p{::}'a\ T.\ dsef\ p\}$
 apply(*rule exI*[*of - ret x*])
 apply(*blast intro*: *dsef-ret*)
done

Minimizing the clutter caused by *Abs-Dsef* and *Rep-Dsef*.

syntax
 -absdsef :: $'a\ T \Rightarrow\ 'a\ D$ ($\Uparrow$ - [*200*] *199*)
 -repdsef :: $'a\ D \Rightarrow\ 'a\ T$ ($\Downarrow$ - [*200*] *199*)
translations
 $\Uparrow p \rightleftharpoons Abs\text{-}Dsef\ p$
 $\Downarrow p \rightleftharpoons Rep\text{-}Dsef\ p$

All representatives of terms of type $'a\ D$ are dsef and thus in particular discardable and copyable.

lemma *dsef-Rep-Dsef* [*simp*]: *dsef* ($\Downarrow a$)
proof (*induct a rule*: *Abs-Dsef-induct*)
 fix *a*
 assume *a* : *Dsef*
 thus *dsef* ($\Downarrow$ ($\Uparrow a$))
  by (*simp add*: *Abs-Dsef-inverse Dsef-def*)
qed

lemma *dis-Rep-Dsef*: *dis* ($\Downarrow a$)
 apply(*insert dsef-Rep-Dsef*[*of a*])
 apply(*unfold dsef-def*)
 apply(*blast*)
done

lemma *cp-Rep-Dsef*: *cp* ($\Downarrow a$)

apply(*insert dsef-Rep-Dsef*[*of a*])
apply(*unfold dsef-def*)
apply(*blast*)
done

Convention: We will denote functions in D that are simply abstracted versions of appropriate functions in T by the same name with the first letter capitalised.

constdefs
$Ret :: {}'a \Rightarrow {}'a\ D$
$Ret\ x \equiv \Uparrow (ret\ x)$

lemma *Ret-ret*: $\Downarrow (Ret\ x) = ret\ x$
proof $-$
have $\Downarrow (Ret\ x) = \Downarrow (\Uparrow (ret\ x))$ **by** (*simp only*: *Ret-def*)
also have $\ldots = ret\ x$ **by** (*simp add*: *Dsef-def Abs-Dsef-inverse*)
finally show *?thesis* **.**
qed

Lifting operations will allow us to introduce monadic connectives $\wedge$, $\vee$, etc. by simply lifting the HOL ones. Theorem *dsef-ret* will assert these to be *dsef* (see below).

constdefs
$liftM :: [{}'a \Rightarrow {}'b, {}'a\ T] \Rightarrow {}'b\ T$
$liftM\ f\ p \equiv do\ \{x \leftarrow p; ret\ (f\ x)\}$
$liftM2 :: [{}'a \Rightarrow {}'b \Rightarrow {}'c, {}'a\ T, {}'b\ T] \Rightarrow {}'c\ T$
$liftM2\ f\ p\ q \equiv do\ \{x \leftarrow p; y \leftarrow q; ret\ (f\ x\ y)\}$
$liftM3 :: [{}'a \Rightarrow {}'b \Rightarrow {}'c \Rightarrow {}'d, {}'a\ T, {}'b\ T, {}'c\ T] \Rightarrow {}'d\ T$
$liftM3\ f\ p\ q\ r \equiv do\ \{x \leftarrow p; y \leftarrow q; z \leftarrow r; ret\ (f\ x\ y\ z)\}$
— The most general form of lifting; the above may be expressed by it
$ap :: [({}'a \Rightarrow {}'b)\ T, {}'a\ T] \Rightarrow {}'b\ T$ (**infixl** $$ *100*)
$ap\ m\ p \equiv do\ \{f \leftarrow m; x \leftarrow p; ret\ (f\ x)\}$

lemma *liftM-ap*: *liftM f x* = (*ret f* $$ *x*)
by (*simp add*: *ap-def liftM-def*)

lemma *liftM2-ap*: *liftM2 f x y* = (*ret f* $$ *x* $$ *y*)
by (*simp add*: *mon-ctr ap-def liftM2-def*)

lemma *liftM3-ap*: *liftM3 f x y z* = *ret f* $$ *x* $$ *y* $$ *z*
by(*simp add*: *mon-ctr ap-def liftM3-def*)

theorem *dsef-ret-ap*: *dsef p* $\Longrightarrow$ *dsef* (*ret f* $$ *p*)
apply(*simp add*: *ap-def dsef-def*)
apply(*clarify*)
apply(*rule conjI*)
apply(*erule weak-cp-seq*)
apply(*rule conjI*)
apply(*erule weak-dis-seq*)
apply(*clarify*)
apply(*drule-tac* $x = q$ **in** *spec*)
apply(*simp add*: *mon-ctr weak-cp-seq*)
apply(*simp* (*no-asm-simp*) *only*: *cp-seq-ret*)

done

dsef programs may be swapped.

lemma *commute-dsef*: ⟦*dsef p*; *dsef q*⟧ ⟹
  ∀*r*. *do* {*x*←*p*; *y*←*q*; *r x y*} = *do* {*y*←*q*; *x*←*p*; *r x y*}
 apply(*rule commute-1-3*)
 apply(*simp-all add*: *dsef-def*)
 apply(*clarify*)
 apply(*drule commute-bool-arb*)
 apply(*assumption*)+
 apply(*drule-tac x* = *q* **in** *spec*)
by(*blast*)

lemma *commute-bool*: ⟦*dsef p*; *cp* (*q*::*bool T*); *dis q*⟧ ⟹
  ∀*r*. *do* {*x*←*p*; *y*←*q*; *r x y*} = *do* {*y*←*q*; *x*←*p*; *r x y*}
by (*rule commute-1-3*, *simp-all add*: *dsef-def*)

A formalisation of the essential fact that *dsef* programs are actually stable under sequencing; this has only been proposed in [34], but has not been shown.

theorem *dsef-seq*: ⟦*dsef p*; ∀*x*. *dsef* (*q x*)⟧ ⟹ *dsef* (*do* {*x*←*p*; *q x*})
proof −
 assume *a1*: *dsef p*
 assume *a2*: ∀*x*. *dsef* (*q x*)
 from *a1* **have** *disp*: *dis p* **by** (*rule dsef-dis*)
 from *a1* **have** *cpp*: *cp p* **by** (*rule dsef-cp*)
 from *a2* **have** *disq*: ∀*x*. *dis* (*q x*) **by** (*unfold dsef-def*, *blast*)
 from *a2* **have** *cpq*: ∀*x*. *cp* (*q x*) **by** (*unfold dsef-def*, *blast*)
 let *?s* = *do* {*x*←*p*; *q x*}
 — The proof proceeds in three parts, each one asserting some property stated in the definition of *dsef* terms. Firstly, *dsef* terms are discardable.
 have *dis ?s*
 proof −
  have *do* {*x*←*?s*; *ret* ()} = *do* {*x*←*p*; *q x*; *ret* ()} **by** (*simp add*: *seq-def*)
  also from *disp disq*
  have ... = *ret* () **by** (*simp add*: *dis-left dis-left2*)
  finally show *?thesis* **by** (*simp add*: *dis-def*)
 qed
 — *dsef* terms are also copyable. We unfold the definition and prove the required equation directly.
 moreover have *cp ?s*
 proof −
  have *do* {*x*←*?s*; *y*←*?s*; *ret* (*x*,*y*)} =
   do {*u*←*p*; *x*←*q u*; *v*←*p*; *y*←*q v*; *ret* (*x*,*y*)}
   by *simp*
  also have ... = *do* {*u*←*p*; *v*←*p*; *x*←*q u*; *y*←*q v*; *ret* (*x*,*y*)}
  proof −
   — This swapping step is a bit more difficult; we have to assist the simplifier by the following general statement:
   have ∀*u*. *do* {*x*←*q u*; *v*←*p*; *y*←*q v*; *ret* (*x*,*y*)} = *do* {*v*←*p*; *x*←*q u*; *y*←*q v*; *ret* (*x*,*y*)}
    (**is** ∀*u*. *?A u* = *?B u*)
   proof
    fix *u*
    from *a2* **have** *dsef* (*q u*) **by** (*rule spec*)
    from *this a1*

have ∀*r*::′*b*⇒′*a*⇒(′*b*∗′*b*) *T*. *do* {*x*←*q u*; *v*←*p*; *r x v*} = *do* {*v*←*p*; *x*←*q u*; *r x v*}
by (*rule commute-dsef*)
thus *?A u* = *?B u* **by** (*rule spec*)
qed
thus *?thesis* **by** *simp*
qed
also from *cpp cpq* **have** ... = *do* {*u*←*p*; *x*←*q u*; *ret* (*x*,*x*)}
by (*simp add*: *cp-arb*)
finally show *?thesis* **by** (*simp add*: *cp-def*)
qed
— The final step is that *p* ≫= *q* commutes with bool-valued programs:
moreover have ∀*q*::*bool T*. *cp q* ∧ *dis q* ⟶ *cp* (*do* {*x*←*?s*; *y*←*q*; *ret*(*x*,*y*)})
proof
— The proof is carried out by a so called raw proof block, where the succeeding application of blast spares us having to do the trivial proof steps.
fix *qa*
{ **assume** *cpqa*: *cp* (*qa*::*bool T*)
assume *disqa*: *dis qa*
have *cp* (*do* {*x*←*do*{*u*←*p*; *q u*}; *y*←*qa*; *ret* (*x*, *y*)})
proof −
let *?w* = *do* {*x*←*do*{*u*←*p*; *q u*}; *y*←*qa*; *ret* (*x*, *y*)}
have *do* {*x*←*?w*; *y*←*?w*; *ret* (*x*,*y*)} =
do {*u*←*p*; *x*←*q u*; *y*←*qa*; *u*′←*p*; *x*′←*q u*′; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))}
by (*simp del*: *bind-assoc add*: *mon-ctr*)
also from *a1 cpqa disqa*
have ... = *do* {*u*←*p*; *x*←*q u*; *u*′←*p*; *y*←*qa*; *x*′←*q u*′; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))}
by (*simp add*: *commute-bool*)
also from *a1 a2*
have ... = *do* {*u*←*p*; *u*′←*p*; *x*←*q u*; *y*←*qa*; *x*′←*q u*′; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))}
proof −
— This fact is needed to help the simplifier solve the goal
have ∀*u*. *do* {*x*←*q u*; *u*′←*p*; *y*←*qa*; *x*′←*q u*′; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))} =
do {*u*′←*p*; *x*←*q u*; *y*←*qa*; *x*′←*q u*′; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))}
(**is** ∀*u*. *?A u* = *?B u*)
proof
fix *u*
from *a2* **have** *dsef* (*q u*) **by** (*rule spec*)
from *this a1* **have** ∀*r*. *do* {*x*←*q u*; *u*′←*p*; *r x u*′} = *do* {*u*′←*p*; *x*←*q u*; *r x u*′}
by (*rule commute-dsef*)
thus *?A u* = *?B u* **by** (*rule spec*)
qed
thus *?thesis* **by** *simp*
qed
also from *a2 cpqa disqa*
have ... = *do* {*u*←*p*; *u*′←*p*; *x*←*q u*; *x*′←*q u*′; *y*←*qa*; *y*′←*qa*; *ret*((*x*,*y*),(*x*′,*y*′))}
by (*simp add*: *commute-bool*)
also from *cpp cpq cpqa* **have** ... = *do* {*u*←*p*; *x*←*q u*; *y*←*qa*; *ret*((*x*,*y*),(*x*,*y*))}
by (*simp add*: *cp-arb*)
finally show *?thesis* **by** (*simp del*: *bind-assoc add*: *mon-ctr cp-def*)
qed
}
thus *cp* (*qa*::*bool T*) ∧ *dis qa* ⟶ *cp* (*do* {*x*←*do* {*u*←*p*; *q u*}; *y*←*qa*; *ret* (*x*, *y*)})
by *blast*

qed
ultimately show *dsef ?s* **by** (*simp add*:*dsef-def*)
qed

Given that *dsef* programs are stable under sequencing, this weak form, which comes in handy sometimes, can easily be proved.

lemma *weak-dsef-seq*: $dsef\ p \Longrightarrow dsef\ (do\ \{x \leftarrow p;\ ret\ (f\ x)\})$
by(*simp add*: *dsef-seq*)

With the help of theorem *dsef-seq* the following proof is immediate.

lemma *dsef-liftM2*: $[\![dsef\ p;\ dsef\ q]\!] \Longrightarrow dsef\ (liftM2\ f\ p\ q)$
proof −
assume *a1*: *dsef p* **and** *a2*: *dsef q*
from *a1* **have** $dsef\ (do\ \{x \leftarrow p;\ y \leftarrow q;\ ret\ (f\ x\ y)\})$
proof (*rule dsef-seq*)
show $\forall x.\ dsef\ (do\ \{y \leftarrow q;\ ret\ (f\ x\ y)\})$
proof
fix *x* **from** *a2* **show** $dsef\ (do\ \{y \leftarrow q;\ ret\ (f\ x\ y)\})$
proof (*rule dsef-seq*)
show $\forall y.\ dsef\ (ret\ (f\ x\ y))$
proof
fix *y* **show** $dsef\ (ret\ (f\ x\ y))$ **by** (*rule dsef-ret*)
qed
qed
qed
qed
thus $dsef\ (liftM2\ f\ p\ q)$ **by** (*simp only*: *liftM2-def*)
qed

lemma *Abs-Dsef-inverse-liftM2* [*simp*]: $[\![dsef\ p;\ dsef\ q]\!] \Longrightarrow$
$\Downarrow (\Uparrow (liftM2\ f\ p\ q)) = liftM2\ f\ p\ q$
by (*simp add*: *Abs-Dsef-inverse Dsef-def dsef-liftM2*)

end

C.3 Introducing Propositional Connectives

theory *MonLogic* = *MonProp*:

C.3.1 Propositional Connectives

As usual in intuitionistic logics, we introduce conjunction, disjunction and implication independently of each other.

consts
Valid :: $bool\ D \Rightarrow bool$ $((\vdash\text{-})\ 15)$
$\wedge_D$ :: $[bool\ D,\ bool\ D] \Rightarrow bool\ D$ (**infixr** *35*)
$\vee_D$ :: $[bool\ D,\ bool\ D] \Rightarrow bool\ D$ (**infixr** *30*)
$\longrightarrow_D$ :: $[bool\ D,\ bool\ D] \Rightarrow bool\ D$ (**infixr** *25*)

According with the definition in [34], the connectives are simply lifted from HOL, and validity amounts to being equal to a program always returning *True*.

defs
Valid-def: $\vdash P \equiv \Downarrow P = do\ \{x \leftarrow (\Downarrow P);\ ret\ True\}$
conjD-def: $P \wedge_D Q \equiv \Uparrow (liftM2\ (op\ \wedge)\ (\Downarrow P)\ (\Downarrow Q))$
disjD-def: $P \vee_D Q \equiv \Uparrow (liftM2\ (op\ \vee)\ (\Downarrow P)\ (\Downarrow Q))$
impD-def: $P \longrightarrow_D Q \equiv \Uparrow (liftM2\ (op\ \longrightarrow)\ (\Downarrow P)\ (\Downarrow Q))$

constdefs
iffD $::\ [bool\ D, bool\ D] \Rightarrow bool\ D$ (**infixr** $\longleftrightarrow_D$ *20*)
$P \longleftrightarrow_D Q \equiv (P \longrightarrow_D Q) \wedge_D (Q \longrightarrow_D P)$
NotD $::\ bool\ D \Rightarrow bool\ D$ ($\neg_D$ - [*40*] *40*)
$\neg_D P \equiv P \longrightarrow_D Ret\ False$

Because of discardability, the definition of *Valid*, which was simply taken over from the definition of global validity of terms of type *bool T*, can be simplified.

lemma *Valid-simp*: $(\vdash p) = (\Downarrow p = ret\ True)$
proof
assume *vp*: $\vdash p$
show $\Downarrow p = ret\ True$
proof $-$
from *vp* **have** $\Downarrow p = do\ \{\Downarrow p;\ ret\ True\}$
by (*simp only*: *Valid-def seq-def*)
also have $\ldots = ret\ True$ **by** (*rule dis-left*, *rule dis-Rep-Dsef*)
finally show *?thesis* **.**
qed
next
assume $\Downarrow p = ret\ True$
hence $\Downarrow p = do\ \{x \leftarrow \Downarrow p;\ ret\ True\}$ **by** *simp*
thus $\vdash p$ **by** (*simp only*: *Valid-def*)
qed

lemma *Valid-simpD*: $(\vdash P) = (P = Ret\ True)$
apply(*simp add*: *Valid-simp Ret-ret Ret-def*)
apply(*induct-tac P rule*: *Abs-Dsef-induct*)
apply(*simp add*: *Dsef-def Abs-Dsef-inverse*)
apply(*rule Abs-Dsef-inject*[*symmetric*])
by (*simp-all add*: *Dsef-def*)

There is a notion of homomorphism associated with lifted operations. The formulation does not really make clear what is intended, but the subsequent lemmas should illuminate the idea.

theorem *lift-Ret-hom*: $(\Uparrow (liftM2\ f\ (\Downarrow (Ret\ a))\ (\Downarrow (Ret\ b))))$
$= Ret\ (f\ a\ b)$
proof $-$
have $\Uparrow (liftM2\ f\ (\Downarrow (Ret\ a))\ (\Downarrow (Ret\ b)))$
$= \Uparrow (do\ \{x \leftarrow (\Downarrow (Ret\ a));\ y \leftarrow (\Downarrow (Ret\ b));\ ret\ (f\ x\ y)\})$
by (*simp only*: *liftM2-def*)
also have $\ldots = \Uparrow (do\ \{x \leftarrow (\Downarrow (\Uparrow (ret\ a)));$
$y \leftarrow (\Downarrow (\Uparrow (ret\ b)));\ ret\ (f\ x\ y)\})$
by (*simp add*: *Ret-def*)

also have $\ldots = \Uparrow (do\ \{x{\leftarrow}ret\ a;\ y{\leftarrow}ret\ b;\ ret(f\ x\ y)\})$
by (*simp add*: *Dsef-def Abs-Dsef-inverse*)
also have $\ldots = \Uparrow (ret\ (f\ a\ b))$ **by** *simp*
also have $\ldots = Ret\ (f\ a\ b)$ **by** (*simp only*: *Ret-def*)
finally show *?thesis* **.**
qed

lemma *conjD-Ret-hom*: $Ret\ (a{\wedge}b) = ((Ret\ a) \wedge_D (Ret\ b))$
by (*simp add*: *lift-Ret-hom conjD-def*)
lemma *disjD-Ret-hom*: $Ret\ (a{\vee}b) = ((Ret\ a) \vee_D (Ret\ b))$
by (*simp add*: *lift-Ret-hom disjD-def*)
lemma *impD-Ret-hom*: $Ret\ (a{\longrightarrow}b) = ((Ret\ a) \longrightarrow_D (Ret\ b))$
by (*simp add*: *lift-Ret-hom impD-def*)

lemma *NotD-Ret-hom*: $Ret\ (\neg\ P) = (\neg_D\ (Ret\ P))$
by(*simp add*: *NotD-def impD-Ret-hom*[*symmetric*])

If a formula depending on variable x is valid for all x, then we may also 'substitute' it by a *dsef* term.

lemma *dsef-form*: $\forall x.\ \vdash P\ x \Longrightarrow \forall b.\ \vdash \Uparrow (do\ \{a{\leftarrow}\Downarrow b;\ \Downarrow (P\ a)\})$
proof
fix b
assume *a1*: $\forall x.\ \vdash P\ x$
hence $\Downarrow (\Uparrow (do\ \{a{\leftarrow}\Downarrow (b{::}'a\ D);\ \Downarrow (P\ a)\})) =$
$\Downarrow (\Uparrow (do\ \{a{\leftarrow}\Downarrow (b{::}'a\ D);\ ret\ True\}))$
by (*simp add*: *Valid-simp*)
also have $\ldots = do\ \{a{\leftarrow}\Downarrow b;\ ret\ True\}$
proof (*rule Abs-Dsef-inverse*)
have $dsef\ (do\ \{a{\leftarrow}\Downarrow b;\ ret\ True\})$
by (*simp add*: *dsef-ret dsef-Rep-Dsef dsef-seq*)
thus $do\ \{a{\leftarrow}\Downarrow b;\ ret\ True\} \in Dsef$ **by** (*simp add*: *Dsef-def*)
qed
also have $\ldots = ret\ True$ **by** (*simp add*: *dis-left2 dsef-dis*[*OF dsef-Rep-Dsef*])
finally show $\vdash \Uparrow (do\ \{a{\leftarrow}\Downarrow (b{::}'a\ D);\ \Downarrow (P\ a)\})$
by (*simp add*: *Valid-simp*)
qed

Every true formula may be injected into *bool D* by *Ret* to yield a valid formula of dynamic logic. And the converse also holds!

theorem *Valid-Ret* [*simp*]: $(\vdash Ret\ P) = P$
proof
assume *p*: P
have $\Downarrow (Ret\ P) = do\ \{x{\leftarrow}\Downarrow (Ret\ P);\ ret\ True\}$
proof $-$
have $dsef\ (\Downarrow (Ret\ P))$ **by** (*rule dsef-Rep-Dsef*)
hence *ds*: $dis\ (\Downarrow (Ret\ P))$ **by** (*simp only*: *dsef-def*)
have $\Downarrow (Ret\ P) = ret\ P$ **by** (*rule Ret-ret*)
also from *p* **have** $\ldots = ret\ True$ **by** *simp*
also from *ds* **have** $\ldots = do\ \{\Downarrow (Ret\ P);\ ret\ True\}$ **by** (*rule dis-left*[*symmetric*])
finally show *?thesis* **by** (*simp only*: *seq-def*)
qed
thus $\vdash Ret\ P$ **by** (*simp only*: *Valid-def*)

next
 assume *rp*: $\vdash$ *Ret P*
 hence $\Downarrow$ (*Ret P*) = *ret True* **by** (*rule iffD1*[*OF Valid-simp*])
 hence *ret P* = *ret True*
  by (*simp add*: *Ret-def Dsef-def Abs-Dsef-inverse*)
 hence *P* = *True* **by** (*rule ret-inject*)
 thus *P* **by** *rules*
qed

A bit more tedious, but conversely to *Valid-simp* it is also true that every valid formula that is a negation equals *ret False*.

lemma *Valid-not-eq-ret-False*: ($\vdash \neg_D$ *b*) = ($\Downarrow$ *b* = *ret False*)
proof
 assume $\vdash \neg_D$ *b*
 hence *nt*: $\Downarrow$ ($\neg_D$ *b*) = *ret True* **by** (*simp add*: *Valid-simp*)
 show $\Downarrow$ *b* = *ret False*
 proof −
  have *dsef* (*do* {*x*←$\Downarrow$ *b*; *ret* ($\neg$ *x*)})
   by (*rule weak-dsef-seq*, *rule dsef-Rep-Dsef*)
  hence *bnnb*: *b* = ($\neg_D$ ($\neg_D$ *b*))
   by (*simp add*: *NotD-def impD-def liftM2-def*
      Ret-ret Abs-Dsef-inverse Dsef-def mon-ctr Rep-Dsef-inverse)
  from *nt* **have** $\Uparrow$ ($\Downarrow$ ($\neg_D$ *b*)) = *Ret True* **by** (*simp add*: *Ret-def*)
  hence ($\neg_D$ *b*) = *Ret True* **by** (*simp only*: *Rep-Dsef-inverse*)
  hence ($\neg_D$ ($\neg_D$ *b*)) = ($\neg_D$ (*Ret True*)) **by** *simp*
  with *bnnb* **have** *b* = *Ret* ($\neg$ *True*) **by** (*simp add*: *NotD-Ret-hom*[*symmetric*])
  thus *?thesis* **by** (*simp add*: *Ret-ret*)
 qed
next
 assume $\Downarrow$ *b* = *ret False*
 hence $\Uparrow$ ($\Downarrow$ *b*) = $\Uparrow$ (*ret False*) **by** *simp*
 hence *bf*: *b* = *Ret False* **by** (*simp add*: *Rep-Dsef-inverse Ret-def*)
 have $\Downarrow$ ($\neg_D$ *b*) = *ret True*
 proof −
  from *bf* **have** $\Downarrow$ ($\neg_D$ *b*) = $\Downarrow$ (*Ret False* $\longrightarrow_D$ *Ret False*)
   by (*simp add*: *NotD-def*)
  also have … = $\Downarrow$ (*Ret True*)
  proof −
   have (*Ret False* $\longrightarrow_D$ *Ret False*) = *Ret* (*False* $\longrightarrow$ *False*)
    by (*rule impD-Ret-hom*[*symmetric*])
   thus *?thesis* **by** *simp*
  qed
  also have … = *ret True* **by** (*rule Ret-ret*)
  finally show *?thesis* **.**
 qed
 thus $\vdash \neg_D$ *b* **by** (*simp only*: *Valid-simp*)
qed

Lemmas *Valid-simp*, *Valid-not-eq-ret-False* and *Valid-Ret* show that, since the classical type *bool* is taken as the carrier of truth values, the whole calculus is classical.

C.3.2 Setting up the Simplifier for Propositional Reasoning

Since natural deduction rules don't get us far in the calculus of global validity judgments (in particular, we do not have an analogon for the implication introduction rule), we algebraize it and perform proofs by term manipulation.

All these axioms are in fact provable; it is just the shortage of time that forces us to impose them directly.

constdefs
$xorD :: [bool\ D, bool\ D] \Rightarrow bool\ D \qquad (\textbf{infixr} \oplus_D\ 20)$
$xorD\ P\ Q \ \equiv\ (P \wedge_D \neg_D Q) \vee_D (\neg_D P \wedge_D Q)$

axioms
apl-and-assoc: $((P \wedge_D Q) \wedge_D R) = (P \wedge_D (Q \wedge_D R))$
apl-xor-assoc: $((P \oplus_D Q) \oplus_D R) = (P \oplus_D (Q \oplus_D R))$
apl-and-comm: $(P \wedge_D Q) = (Q \wedge_D P)$
apl-xor-comm: $(P \oplus_D Q) = (Q \oplus_D P)$
apl-and-LC: $(P \wedge_D (Q \wedge_D R)) = (Q \wedge_D (P \wedge_D R))$
apl-xor-LC: $(P \oplus_D (Q \oplus_D R)) = (Q \oplus_D (P \oplus_D R))$
apl-and-True-r: $(P \wedge_D Ret\ True) = P$
apl-and-True-l: $(Ret\ True \wedge_D P) = P$
apl-and-absorb: $(P \wedge_D P) = P$
apl-and-absorb2: $(P \wedge_D (P \wedge_D Q)) = (P \wedge_D Q)$
apl-and-False-l: $(Ret\ False \wedge_D P) = Ret\ False$
apl-and-False-r: $(P \wedge_D Ret\ False) = Ret\ False$
apl-xor-False-r: $(P \oplus_D Ret\ False) = P$
apl-xor-False-l: $(Ret\ False \oplus_D P) = P$
apl-xor-contr: $(P \oplus_D P) = Ret\ False$
apl-xor-contr2: $(P \oplus_D (P \oplus_D Q)) = Q$
apl-and-ldist: $(P \wedge_D (Q \oplus_D R)) = ((P \wedge_D Q) \oplus_D (P \wedge_D R))$
apl-and-rdist: $((P \oplus_D Q) \wedge_D R) = ((P \wedge_D R) \oplus_D (Q \wedge_D R))$
— Expressing the connectives by conjunction and exclusive or
apl-imp-xor: $(P \longrightarrow_D Q) = ((P \wedge_D Q) \oplus_D P \oplus_D Ret\ True)$
apl-or-xor: $(P \vee_D Q) = (P \oplus_D Q \oplus_D (P \wedge_D Q))$
apl-not-xor: $(\neg_D P) = (P \oplus_D Ret\ True)$
apl-iff-xor: $(P \longleftrightarrow_D Q) = (P \oplus_D Q \oplus_D Ret\ True)$

pdl-taut is the collection of all these rules, so that they can be handed over to the simplifier conveniently.

This set of rewrite rules is complete with respect to normalisation of propositional tautologies to their normal form *Ret True*. Hence, we can prove monadic tautologies in one fell swoop by applying the tactic (*simp only*: *pdl-taut Valid-Ret*).

lemmas *pdl-taut* = — ... all axioms above

lemmas *mon-prop-reason* = *Abs-Dsef-inverse dsef-liftM2*
Dsef-def conjD-def disjD-def impD-def NotD-def

A proof showing in what manner the above axioms may be proved.

lemma $(P \wedge_D (\neg_D P)) = Ret\ False$
apply(*simp add*: *mon-prop-reason*, *simp only*: *liftM2-def*)
apply(*unfold Ret-def*)

apply$(rule\ cong[of\ Abs\text{-}Dsef\ Abs\text{-}Dsef],\ rule\ refl)$
apply$(simp\ add: Abs\text{-}Dsef\text{-}inverse\ Dsef\text{-}def)$
apply$(simp\ add: mon\text{-}ctr\ del: bind\text{-}assoc)$
apply$(simp\ add: cp\text{-}arb\ dsef\text{-}cp[OF\ dsef\text{-}Rep\text{-}Dsef])$
apply$(rule\ dis\text{-}left2)$
apply$(rule\ dsef\text{-}dis[OF\ dsef\text{-}Rep\text{-}Dsef])$
done

And another one, following the same scheme, only that the simplifier now needs help from the classical reasoner to finish.

lemma $(P \oplus_D Q) = (Q \oplus_D P)$
apply$(simp\ add: disjD\text{-}def\ conjD\text{-}def\ NotD\text{-}def\ impD\text{-}def\ liftM2\text{-}def\ xorD\text{-}def\ Ret\text{-}def)$
apply$(simp\ add: Abs\text{-}Dsef\text{-}inverse\ Dsef\text{-}def\ dsef\text{-}seq\ dsef\text{-}Rep\text{-}Dsef\ mon\text{-}ctr\ del: bind\text{-}assoc)$
apply$(simp\ add: commute\text{-}dsef[of \Downarrow Q \Downarrow P])$
apply$(simp\ add: dsef\text{-}cp\ cp\text{-}arb)$
apply$(subgoal\text{-}tac\ \forall x\ y.\ (x \wedge \neg y \vee \neg x \wedge y) = (y \wedge \neg x \vee \neg y \wedge x),\ simp)$
by *blast*

C.3.3 Proof Rules

Proof rules, which can all be proved to be correct, since we have the semantics built into the logic (i.e. we can access it within HOL). Some proofs however simply employ the above tautology reasoner.

theorem *pdl-excluded-middle*: $\vdash P \vee_D (\neg_D P)$
by $(simp\ add: pdl\text{-}taut)$

theorem *pdl-mp*: $[\![\vdash P \longrightarrow_D Q; \vdash P]\!] \Longrightarrow \vdash Q$
by$(simp\ add: Valid\text{-}simp\ impD\text{-}def\ liftM2\text{-}def\ Rep\text{-}Dsef\text{-}inverse)$

Disjunction introduction

theorem *pdl-disjI1*: $\vdash P \Longrightarrow \vdash (P \vee_D Q)$
proof −
assume $\vdash P$
hence *pt*: $\Downarrow P = ret\ True$ **by** $(simp\ only: Valid\text{-}simp)$
have $\Downarrow (P \vee_D Q) = ret\ True$
proof −
have $\Downarrow (\Uparrow (liftM2\ op \vee (\Downarrow P)\ (\Downarrow Q))) = ret\ True$
proof −
have $\Downarrow (\Uparrow (do\ \{x \leftarrow \Downarrow Q; ret\ True\})) = ret\ True$
proof −
have $\Downarrow (\Uparrow (do\ \{x \leftarrow \Downarrow Q; ret\ True\})) =$
$do\ \{x \leftarrow \Downarrow Q; ret\ True\}$
by $(simp\ add: Abs\text{-}Dsef\text{-}inverse\ Dsef\text{-}def\ weak\text{-}dsef\text{-}seq)$
also have $\ldots = do\ \{\Downarrow Q; ret\ True\}$ **by** $(simp\ only{:}seq\text{-}def)$
also have $\ldots = ret\ True$ **by** $(simp\ add: dis\text{-}Rep\text{-}Dsef\ dis\text{-}left)$
finally show *?thesis* **.**
qed
with *pt* **show** *?thesis* **by** $(simp\ add: liftM2\text{-}def)$
qed
thus *?thesis* **by** $(simp\ only: disjD\text{-}def)$
qed

thus $\vdash (P \vee_D Q)$ **by** $($*simp only*: *Valid-simp*$)$
qed

Entirely analogous for this dual rule.

theorem *pdl-disjI2*: $\vdash Q \Longrightarrow\ \vdash (P \vee_D Q)$

The following proof proceeds by a standard pattern: First insert the assumptions into some specifically tailored do-term and then reduce this do-term to *ret True* with the simplifier.

theorem *pdl-disjE*: $[\![\vdash P \vee_D Q; \vdash P \longrightarrow_D R; \vdash Q \longrightarrow_D R]\!] \Longrightarrow\ \vdash R$
proof $-$
 assume *a1*: $\vdash P \vee_D Q \vdash P \longrightarrow_D R \vdash Q \longrightarrow_D R$
 note *copy* $=$ *dsef-cp*[*OF dsef-Rep-Dsef*]
 note *dsc* $=$ *dsef-dis*[*OF dsef-Rep-Dsef*]
 — 1st part: blow up program $\Downarrow R$ to some giant term:
 have $\Downarrow R = do\ \{u{\leftarrow}ret\ True;\ v{\leftarrow}ret\ True;\ w{\leftarrow}ret\ True;\ r{\leftarrow}\Downarrow R;\ ret(u{\longrightarrow}v{\longrightarrow}w{\longrightarrow}r)\}$
  by *simp*
 also from *a1* **have** $\ldots = do\ \{u{\leftarrow}(\Downarrow (P \vee_D Q));$
 $v{\leftarrow}(\Downarrow (P \longrightarrow_D R));$
 $w{\leftarrow}(\Downarrow (Q \longrightarrow_D R));$
 $r{\leftarrow}\Downarrow R;\ ret\ (u{\longrightarrow}v{\longrightarrow}w{\longrightarrow}r)\}$
  by $($*simp add*: *Valid-simp*$)$
 — 2nd part: reduce this giant program to *ret True* exploiting properties of dsef programs
 also have $\ldots =$ *ret True*
  apply$($*simp add*: *mon-prop-reason liftM2-def dsef-Rep-Dsef dsef-seq mon-ctr del*: *bind-assoc*$)$
  apply$($*simp add*: *commute-dsef*[*of* $\Downarrow Q \Downarrow P$]$)$
  apply$($*simp add*: *commute-dsef*[*of* $\Downarrow R \Downarrow Q$]$)$
  apply$($*simp add*: *dsef-cp*[*OF dsef-Rep-Dsef*] *cp-arb del*: *bind-assoc*$)$
  apply$($*simp add*: *dsef-dis*[*OF dsef-Rep-Dsef*] *dis-left2*$)$
  done
 finally show *?thesis* **by** $($*simp only*: *Valid-simp*$)$
qed

theorem *pdl-conjI*: $[\![\vdash P; \vdash Q]\!] \Longrightarrow\ \vdash P \wedge_D Q$
proof $-$
 assume *a*: $\vdash P \vdash Q$
 from *a* **have** $\Downarrow P =$ *ret True* **by** $($*simp add*: *Valid-simp*$)$
 moreover
 from *a* **have** $\Downarrow Q =$ *ret True* **by** $($*simp add*: *Valid-simp*$)$
 ultimately
 have $\Downarrow (P \wedge_D Q) =$ *ret True*
  by $($*simp add*: *mon-prop-reason liftM2-def*$)$
 thus *?thesis* **by** $($*simp add*: *Valid-simp*$)$
qed

Derived rules of inference

theorem *pdl-FalseE*: $\vdash$ *Ret False* $\Longrightarrow\ \vdash R$
proof $-$
 assume $\vdash$ *Ret False*
 hence *False* **by** $($*rule iffD1*[*OF Valid-Ret*]$)$
 thus $\vdash R$ **by** $($*rule FalseE*$)$
qed

lemma *pdl-notE*: $[\![\vdash P; \vdash \neg_D P]\!] \Longrightarrow \vdash R$
proof (*unfold NotD-def*)
  assume *p*: $\vdash P$ **and** *np*: $\vdash P \longrightarrow_D$ *Ret False*
  from *np p* **have** $\vdash$ *Ret False* **by** (*rule pdl-mp*)
  thus $\vdash R$ **by** (*rule pdl-FalseE*)
qed

lemma *pdl-conjE*: $[\![\vdash P \wedge_D Q; [\![\vdash P; \vdash Q]\!] \Longrightarrow \vdash R]\!] \Longrightarrow \vdash R$
proof –
  assume *a1*: $\vdash P \wedge_D Q$
  assume *a2*: $[\![\vdash P; \vdash Q]\!] \Longrightarrow \vdash R$
  have $\vdash P$
  proof (*rule pdl-mp*)
    show $\vdash P \wedge_D Q \longrightarrow_D P$ **by** (*simp add*: *pdl-taut*)
  qed
  moreover
  have $\vdash Q$
  proof (*rule pdl-mp*)
    show $\vdash P \wedge_D Q \longrightarrow_D Q$ **by** (*simp add*: *pdl-taut*)
  qed
  moreover note *a1 a2*
  ultimately
  show $\vdash R$ **by** (*rules*)
qed

Some further typical rules.

lemma *pdl-notI*: $[\![\vdash P; \vdash$ *Ret False*$]\!] \Longrightarrow \vdash \neg_D P$
by(*rule pdl-FalseE*)

lemma *pdl-conjunct1*: $\vdash P \wedge_D Q \Longrightarrow \vdash P$
proof –
  assume $\vdash P \wedge_D Q$
  thus $\vdash P$
  proof (*rule pdl-conjE*)
    assume $\vdash P$
    thus *?thesis* **.**
  qed
qed

lemma *pdl-conjunct2*: **assumes** *pq*: $\vdash P \wedge_D Q$ **shows** $\vdash Q$
proof –
  from *pq* **show** $\vdash Q$
  proof (*rule pdl-conjE*)
    assume $\vdash Q$
    thus *?thesis* **.**
  qed
qed

lemma *pdl-iffI*: $[\![\vdash P \longrightarrow_D Q; \vdash Q \longrightarrow_D P]\!] \Longrightarrow \vdash P \longleftrightarrow_D Q$
proof (*unfold iffD-def*)

assume a: $\vdash P \longrightarrow_D Q$ **and** b: $\vdash Q \longrightarrow_D P$
show $\vdash (P \longrightarrow_D Q) \wedge_D (Q \longrightarrow_D P)$
by (*rule pdl-conjI*)
qed

lemma *pdl-iffE*: $[\![\vdash P \longleftrightarrow_D Q; [\![\vdash P \longrightarrow_D Q; \vdash Q \longrightarrow_D P]\!] \Longrightarrow \vdash R]\!] \Longrightarrow \vdash R$
apply(*unfold iffD-def*)
apply(*erule pdl-conjE*)
by *blast*

lemma *pdl-sym*: $(\vdash P \longleftrightarrow_D Q) \Longrightarrow (\vdash Q \longleftrightarrow_D P)$
apply(*erule pdl-iffE*)
by(*rule pdl-iffI*)

lemma *pdl-iffD1*: $\vdash P \longleftrightarrow_D Q \Longrightarrow \vdash P \longrightarrow_D Q$
by(*erule pdl-iffE*)

lemma *pdl-iffD2*: $\vdash P \longleftrightarrow_D Q \Longrightarrow \vdash Q \longrightarrow_D P$
by (*erule pdl-iffE*)

lemma *pdl-conjI-lifted*:
assumes $\vdash P \longrightarrow_D Q$ **and** $\vdash P \longrightarrow_D R$ **shows** $\vdash P \longrightarrow_D Q \wedge_D R$
proof −
have $\vdash (P \longrightarrow_D Q) \longrightarrow_D (P \longrightarrow_D R) \longrightarrow_D (P \longrightarrow_D Q \wedge_D R)$
by (*simp add*: *pdl-taut*)
thus *?thesis* **by** (*rule pdl-mp*[*THEN pdl-mp*])
qed

lemma *pdl-eq-iff* : $[\![P = Q]\!] \Longrightarrow \vdash P \longleftrightarrow_D Q$
by (*simp only*: *pdl-taut Valid-Ret*)

lemma *pdl-iff-sym*: $\vdash P \longleftrightarrow_D Q \Longrightarrow \vdash Q \longleftrightarrow_D P$
by (*simp only*: *pdl-taut Valid-Ret*)

lemma *pdl-imp-wk*: $\vdash P \Longrightarrow \vdash Q \longrightarrow_D P$
proof −
assume $\vdash P$
have $\vdash P \longrightarrow_D Q \longrightarrow_D P$ **by** (*simp add*: *pdl-taut*)
thus *?thesis* **by** (*rule pdl-mp*)
qed

lemma *pdl-False-imp*: $\vdash$ *Ret False* $\longrightarrow_D P$
by (*simp add*: *pdl-taut*)

lemma *pdl-imp-trans*: $[\![\vdash A \longrightarrow_D B; \vdash B \longrightarrow_D C]\!] \Longrightarrow \vdash A \longrightarrow_D C$
proof −
assume *a1*: $\vdash A \longrightarrow_D B$ **and** *a2*: $\vdash B \longrightarrow_D C$
have $\vdash (A \longrightarrow_D B) \longrightarrow_D (B \longrightarrow_D C) \longrightarrow_D A \longrightarrow_D C$ **by** (*simp only*: *pdl-taut Valid-Ret*)
from *this a1 a2* **show** *?thesis* **by** (*rule pdl-mp*[*THEN pdl-mp*])
qed

Some applications of the enhanced simplifier, which is now capable of proving prop. tautologies immediately.

lemma $\vdash A \longrightarrow_D B \longrightarrow_D A$
by (*simp only*: *pdl-taut Valid-Ret*)

lemma $\vdash (P \wedge_D Q \longrightarrow_D R) \longleftrightarrow_D (P \longrightarrow_D Q \longrightarrow_D R)$
by (*simp only*: *pdl-taut Valid-Ret*)

lemma $\vdash (P \longrightarrow_D Q) \vee_D (Q \longrightarrow_D P)$
by (*simp only*: *pdl-taut Valid-Ret*)

lemma $\vdash (P \longrightarrow_D Q) \wedge_D (\neg_D P \longrightarrow_D R) \longleftrightarrow_D (P \wedge_D Q \vee_D \neg_D P \wedge_D R)$
by (*simp only*: *pdl-taut Valid-Ret*)

end

C.4 Monadic Equality

theory *MonEq* = *MonLogic*:

constdefs
MonEq :: $['a\ D, 'a\ D] \Rightarrow bool\ D$ (**infixl** $=_D$ *60*)
$MonEq\ a\ b \equiv \Uparrow (liftM2\ (op =)\ (\Downarrow a)\ (\Downarrow b))$

lemma *MonEq-Ret-hom*: $((Ret\ a) =_D (Ret\ b)) = (Ret\ (a{=}b))$
by (*simp add*: *lift-Ret-hom MonEq-def*)

Transitivity of monadic equality.

lemma *mon-eq-trans*: $[\![\vdash a =_D b; \vdash b =_D c]\!] \Longrightarrow \vdash a =_D c$
proof −
assume *ab*: $\vdash a =_D b$ **and** *bc*: $\vdash b =_D c$
have $\vdash (a =_D b) \longrightarrow_D (b =_D c) \longrightarrow_D (a =_D c)$
apply(*simp add*: *MonEq-def impD-def liftM2-def*)
apply(*simp add*: *Abs-Dsef-inverse dsef-Rep-Dsef Dsef-def dsef-seq mon-ctr del*: *bind-assoc*)
apply(*simp add*: *cp-arb dsef-cp*[*OF dsef-Rep-Dsef*])
apply(*simp add*: *commute-dsef*[*of* $\Downarrow c \Downarrow a$])
apply(*simp add*: *commute-dsef*[*of* $\Downarrow b \Downarrow a$])
apply(*simp add*: *cp-arb dsef-cp*[*OF dsef-Rep-Dsef*] *del*: *bind-assoc*)
apply (*simp add*: *dsef-dis*[*OF dsef-Rep-Dsef*] *dis-left2*)
apply(*subst Ret-def*[*symmetric*])
by *simp*
from *this ab bc* **show** *?thesis* **by** (*rule pdl-mp*[*THEN pdl-mp*])
qed

Reflexivity of monadic equality.

lemma *mon-eq-refl*: $\vdash a =_D a$

apply(*simp add*: *MonEq-def liftM2-def*)
apply(*simp add*: *cp-arb dsef-cp*[*OF dsef-Rep-Dsef*])
apply(*simp add*: *dis-left2 dsef-dis*[*OF dsef-Rep-Dsef*])
apply(*subst Ret-def*[*symmetric*])
by (*simp*)

Auxiliary lemma, just to help the simplifier.

lemma *sym-subst-seq2*: $\forall x\, y.\ c\ x\ y = c\ y\ x \Longrightarrow$
$(\Uparrow (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ c\ x\ y\})) = (\Uparrow (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ c\ y\ x\}))$
by *simp*

Symmetry of monadic equality. The simplifier gets into trouble here, for it must apply symmetry of real equality inside the scope of lambda terms. We circumvent this problem by extracting the essential proof obligation through *sym-subst-seq2* and then working by hand.

lemma *mon-eq-sym*: $(a =_D b) = (b =_D a)$
apply(*simp add*: *MonEq-def liftM2-def*)
apply(*simp add*: *commute-dsef*[*of* ⇓ *a* ⇓ *b*])
apply(*rule sym-subst-seq2*)
apply(*clarify*)
apply(*rule arg-cong*[**where** *f* = *ret*])
by (*rule eq-sym-conv*)

end

C.5 The Proof Calculus of Monadic Dynamic Logic

theory *PDL* = *MonLogic*:

C.5.1 Types, Rules and Axioms

Types, rules and axioms for the box and diamond operators of PDL formulas.

consts
Box :: $'a\ T \Rightarrow ('a \Rightarrow bool\ D) \Rightarrow bool\ D$ ([# -]- [0, 100] 100)
Dmd :: $'a\ T \Rightarrow ('a \Rightarrow bool\ D) \Rightarrow bool\ D$ (⟨-⟩- [0, 100] 100)

Syntax translations that let you write e.g. [# *x*←*p*; *y*←*q*](*ret* (*x*=*y*)) for *Box* (*do* {*x*←*p*; *y*←*q*; *ret* (*x*,*y*)}) (λ(*x*,*y*). *ret* (*x*=*y*)) Essentially, these translations collect all bound variables inside the boxes and return them as a tuple. The lambda term that constitutes the second argument of Box will then also take a tuple pattern as its sole argument.

nonterminals
bndseq bndstep

syntax (*xsymbols*)
-pdlbox :: [*bndseq*, *bool D*] ⇒ *bool D* ([# -]- [0, 100] 100)
-pdldmd :: [*bndseq*, *bool D*] ⇒ *bool D* (⟨-⟩- [0, 100] 100)
-pdlbnd :: [*idt*, *'a T*] ⇒ *bndstep* (-←-)
-pdlseq :: [*bndstep*, *bndseq*] ⇒ *bndseq* (-;/ -)
:: *bndstep* ⇒ *bndseq* (-)
-pdlin :: [*pttrn*, *bndseq*] ⇒ *bndseq*
-pdlout :: [*pttrn*, *bndseq*] ⇒ *bndseq*

translations

-pdlbox (*-pdlseq* (*-pdlbnd x p*) *r*) *phi*
 $\rightharpoonup$ *Box* (*-pdlseq* (*-pdlbnd x p*) (*-pdlin x r*)) *phi*
-pdlbox (*-pdlbnd x p*) *phi* $\rightharpoonup$ *Box p* ($\lambda x.$ *phi*)
-pdldmd (*-pdlseq* (*-pdlbnd x p*) *r*) *phi*
 $\rightharpoonup$ *Dmd* (*-pdlseq* (*-pdlbnd x p*) (*-pdlin x r*)) *phi*
-pdldmd (*-pdlbnd x p*) *phi* $\rightharpoonup$ *Dmd p* ($\lambda x.$ *phi*)
-pdlin tpl (*-pdlseq* (*-pdlbnd x p*) *r*)
 $\rightharpoonup$ *-pdlseq* (*-pdlbnd x p*) (*-pdlin* (*tpl*, *x*) *r*)
-pdlin tpl (*-pdlbnd x p*)
 $\rightharpoonup$ *-pdlout* (*tpl*,*x*) (*do* {*x*←*p*; *ret*(*tpl*,*x*)})
-pdlseq (*-pdlbnd x p*) (*-pdlout tpl r*)
 $\rightharpoonup$ *-pdlout tpl* (*do* {*x*←*p*; *r*})
Box (*-pdlout tpl r*) *phi*
 $\rightharpoonup$ *Box r* (λ*tpl*. *phi*)
Dmd (*-pdlout tpl r*) *phi*
 $\rightharpoonup$ *Dmd r* (λ*tpl*. *phi*)

The axioms of the proof calculus for propositional dynamic logic.

axioms

pdl-nec: $(\forall x. \vdash P\ x) \Longrightarrow \vdash [\#\ x{\leftarrow}p](P\ x)$
pdl-mp-: $[\![\vdash (P \longrightarrow_D Q); \vdash P]\!] \Longrightarrow \vdash Q$ — Only repeated here for completeness.

pdl-k1: $\vdash [\#\ x{\leftarrow}p](P\ x \longrightarrow_D Q\ x) \longrightarrow_D [\#\ x{\leftarrow}p](P\ x) \longrightarrow_D [\#\ x{\leftarrow}p](Q\ x)$
pdl-k2: $\vdash [\#\ x{\leftarrow}p](P\ x \longrightarrow_D Q\ x) \longrightarrow_D \langle x{\leftarrow}p\rangle(P\ x) \longrightarrow_D \langle x{\leftarrow}p\rangle(Q\ x)$
pdl-k3B: $\vdash Ret\ P \longrightarrow_D [\#\ x{\leftarrow}p](Ret\ P)$
pdl-k3D: $\vdash \langle x{\leftarrow}p\rangle(Ret\ P) \longrightarrow_D Ret\ P$
pdl-k4: $\vdash \langle x{\leftarrow}p\rangle(P\ x \vee_D Q\ x) \longrightarrow_D (\langle x{\leftarrow}p\rangle(P\ x) \vee_D \langle x{\leftarrow}p\rangle(Q\ x))$
pdl-k5: $\vdash (\langle x{\leftarrow}p\rangle(P\ x) \longrightarrow_D [\#\ x{\leftarrow}p](Q\ x)) \longrightarrow_D [\#\ x{\leftarrow}p](P\ x \longrightarrow_D Q\ x)$
pdl-seqB: $\vdash [\#\ x{\leftarrow}p;\ y{\leftarrow}q\ x](P\ x\ y) \longleftrightarrow_D [\#\ x{\leftarrow}p][\#\ y{\leftarrow}q\ x](P\ x\ y)$
pdl-seqD: $\vdash \langle x{\leftarrow}p;\ y{\leftarrow}q\ x\rangle(P\ x\ y) \longleftrightarrow_D \langle x{\leftarrow}p\rangle\langle y{\leftarrow}q\ x\rangle(P\ x\ y)$
pdl-ctrB: $\vdash [\#\ x{\leftarrow}p;\ y{\leftarrow}q\ x](P\ y) \longrightarrow_D [\#\ y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}](P\ y)$
pdl-ctrD: $\vdash \langle y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}\rangle(P\ y) \longrightarrow_D \langle x{\leftarrow}p;\ y{\leftarrow}q\ x\rangle(P\ y)$
pdl-retB: $\vdash [\#\ x{\leftarrow}ret\ a](P\ x) \longleftrightarrow_D P\ a$
pdl-retD: $\vdash \langle x{\leftarrow}ret\ a\rangle(P\ x) \longleftrightarrow_D P\ a$
pdl-dsefB: $dsef\ p \Longrightarrow \vdash \Uparrow (do\ \{a{\leftarrow}p;\ \Downarrow (P\ a)\}) \longleftrightarrow_D [\#\ a{\leftarrow}p](P\ a)$
pdl-dsefD: $dsef\ p \Longrightarrow \vdash \Uparrow (do\ \{a{\leftarrow}p;\ \Downarrow (P\ a)\}) \longleftrightarrow_D \langle a{\leftarrow}p\rangle(P\ a)$

A simpler notion of sequencing is often more practical in real programs. Essentially this boils down to admitting just one binding within the modal operators.

axioms

pdl-seqB-simp: $\vdash (\ [\#\ x{\leftarrow}p][\#\ y{\leftarrow}q\ x](P\ y)\ ) \longleftrightarrow_D (\ [\#\ y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}](P\ y)\ )$
pdl-seqD-simp: $\vdash (\ \langle x{\leftarrow}p\rangle\langle y{\leftarrow}q\ x\rangle(P\ y)\ ) \longleftrightarrow_D (\ \langle y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}\rangle(P\ y)\ )$

For simple monads [34] both rules can be derived from axiom *pdl-seqB* (or *pdl-seqD*). Simplicity is exploited through the use of the converse rule of *pdl-ctrB*.

lemma $\vdash [\#\ y{\leftarrow}do\ \{x{\leftarrow}p;\ q\ x\}](P\ y) \longrightarrow_D [\#\ x{\leftarrow}p;\ y{\leftarrow}q\ x](P\ y) \Longrightarrow$
 $\vdash (\ [\#\ p](\lambda x.\ [\#\ q\ x]P)\ ) \longleftrightarrow_D (\ [\#\ do\ \{x{\leftarrow}p;\ q\ x\}]P\ )$
apply(*rule pdl-iffI*)

apply(*rule pdl-imp-trans*)
 apply(*rule pdl-iffD2*[*OF pdl-seqB*])
 apply(*rule pdl-ctrB*) — dispose of the trailing ret expression
apply(*rule pdl-imp-trans*)
 apply(*assumption*) — this time dispose by the converse of *pdl-ctrB*
 apply(*rule pdl-iffD1*[*OF pdl-seqB*])
done

Further axioms satisfied by logically regular monads (which we deal with here). Cf. [34, Page 601]

axioms
pdl-eqB: $\vdash$ *Ret* $(p = q) \longrightarrow_D [\# x \leftarrow p](P\ x) \longrightarrow_D [\# x \leftarrow q](P\ x)$
pdl-eqD: $\vdash$ *Ret* $(p = q) \longrightarrow_D \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D \langle x \leftarrow q\rangle(P\ x)$

C.5.2 Derived Rules of Inference

'Multiple' modus ponens, provided for convenience.

lemmas
pdl-mp-2x = *pdl-mp*[*THEN pdl-mp*] **and**
pdl-mp-3x = *pdl-mp*[*THEN pdl-mp*, *THEN pdl-mp*]

First half of the classical relationship between diamond and box.

lemma *dmd-box-rel1*: $\vdash ([\# x \leftarrow p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \langle x \leftarrow p\rangle(P\ x)$
 (**is** $\vdash (?b \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?d$)
proof −
 — Show a classically equivalent statement
 have $\vdash (?d \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?b$
 proof −
  — The 'usual' axiomatic proof method
  have *f1*: $\vdash ((?d \longrightarrow_D [\# x \leftarrow p](\mathit{Ret\ False})) \longrightarrow_D ?b) \longrightarrow_D$
   $(?d \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D ?b$
   by (*simp add*: *pdl-taut*)
  have *f2*: $\vdash (?d \longrightarrow_D [\# x \leftarrow p](\mathit{Ret\ False})) \longrightarrow_D ?b$
   by (*rule pdl-k5*)
  from *f1 f2* **show** *?thesis* **by** (*rule pdl-mp*)
 qed
 thus *?thesis* **by** (*simp add*: *pdl-taut*)
qed

...and the second half.

lemma *dmd-box-rel2*: $\vdash \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D [\# x \leftarrow p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}$
proof −
 have $\vdash (\langle x \leftarrow p\rangle(\mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D$
  $([\# x \leftarrow p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D \langle x \leftarrow p\rangle(\mathit{Ret\ False})) \longrightarrow_D$
  $\langle x \leftarrow p\rangle(P\ x) \longrightarrow_D [\# x \leftarrow p](P\ x \longrightarrow_D \mathit{Ret\ False}) \longrightarrow_D \mathit{Ret\ False}$
  by (*simp add*: *pdl-taut*)
 from *this pdl-k3D pdl-k2* **show** *?thesis* **by** (*rule pdl-mp-2x*)
qed

Inheriting the classical theorems from Isabelle/HOL, one also obtains the classical equivalence between the diamond and box operator.

The proofs of *dmd-box-rel1* and *dmd-box-rel2* implicitly employ classical arguments through the use of the simplifier, since the algebraization of propositional logic behaves classically.

theorem *dmd-box-rel*: $\vdash \langle x \leftarrow p\rangle(P\ x) \longleftrightarrow_D \neg_D\ [\#\ x \leftarrow p](\neg_D\ P\ x)$
 apply(*rule pdl-iffI*)
 apply(*unfold NotD-def*)
 apply(*rule dmd-box-rel2*)
 apply(*rule dmd-box-rel1*)
done

Given *dmd-box-rel*, one easily obtains a dual one.

theorem *box-dmd-rel*: $\vdash [\#\ x \leftarrow p](P\ x) \longleftrightarrow_D \neg_D\ \langle x \leftarrow p\rangle(\neg_D\ P\ x)$
proof −
 have $\vdash (\ \langle x \leftarrow p\rangle(\neg_D\ P\ x) \longleftrightarrow_D \neg_D\ [\#\ x \leftarrow p](\neg_D \neg_D\ P\ x)\ ) \longrightarrow_D$
  $(\ [\#\ x \leftarrow p](P\ x) \longleftrightarrow_D \neg_D \neg_D\ [\#\ x \leftarrow p](\neg_D \neg_D\ P\ x)\ ) \longrightarrow_D$
  $(\ [\#\ x \leftarrow p](P\ x) \longleftrightarrow_D \neg_D\ \langle x \leftarrow p\rangle(\neg_D\ P\ x)\ )$
  by (*simp add*: *pdl-taut*)
 moreover
 have $\vdash\ \langle x \leftarrow p\rangle(\neg_D\ P\ x) \longleftrightarrow_D \neg_D\ [\#\ x \leftarrow p](\neg_D \neg_D\ P\ x)$
  by (*rule dmd-box-rel*)
 moreover
 have $\vdash [\#\ x \leftarrow p](P\ x) \longleftrightarrow_D \neg_D \neg_D\ [\#\ x \leftarrow p](\neg_D \neg_D\ P\ x)$
  by (*simp add*: *pdl-taut*)
 ultimately
 show *?thesis*
  by (*rule pdl-mp-2x*)
qed

A specialized form of the equality rule *pdl-eqD* that only requires the arguments of a program p to be equal.

theorem *pdl-eqD-ext*: $\vdash Ret\ (a = b) \longrightarrow_D \langle p\ a\rangle P \longrightarrow_D \langle p\ b\rangle P$ (**is** $\vdash ?ab \longrightarrow_D ?pa \longrightarrow_D ?pb$)
proof −
 have $\vdash (Ret\ (a = b) \longrightarrow_D Ret\ (p\ a = p\ b)) \longrightarrow_D$
  $(Ret\ (p\ a = p\ b) \longrightarrow_D ?pa \longrightarrow_D ?pb) \longrightarrow_D$
  $(?ab \longrightarrow_D ?pa \longrightarrow_D ?pb)$ **by** (*simp add*: *pdl-taut*)
 moreover
 have $\vdash Ret\ (a = b) \longrightarrow_D Ret\ (p\ a = p\ b)$
 proof (*subst impD-Ret-hom*[*symmetric*])
  show $\vdash Ret\ (a = b \longrightarrow p\ a = p\ b)$
  proof (*rule iffD2*[*OF Valid-Ret*])
   show $a = b \longrightarrow p\ a = p\ b$ **by** *blast*
  qed
 qed
 moreover
 have $\vdash Ret\ (p\ a = p\ b) \longrightarrow_D ?pa \longrightarrow_D ?pb$
  by (*rule pdl-eqD*)
 ultimately
 show *?thesis* **by** (*rule pdl-mp-2x*)
qed

The following are simple consequences of the axioms above; rather than monadic implication, they use Isabelle's meta implication (and hence represent rules).

lemma *box-imp-distrib*: $\vdash [\#\ x \leftarrow p](P\ x \longrightarrow_D Q\ x) \Longrightarrow \vdash [\#\ x \leftarrow p](P\ x) \longrightarrow_D [\#\ x \leftarrow p](Q\ x)$

by$($*rule pdl-k1*$[$*THEN pdl-mp*$])$

lemma *dmd-imp-distrib*: $\vdash [\#\ x \leftarrow p](P\ x \longrightarrow_D Q\ x) \Longrightarrow \vdash \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D \langle x \leftarrow p\rangle(Q\ x)$
 by $($*rule pdl-mp*$[$*OF pdl-k2*$])$

lemma *pdl-box-reg*: $\forall x. \vdash P\ x \longrightarrow_D Q\ x \Longrightarrow \vdash [\#\ x \leftarrow p](P\ x) \longrightarrow_D [\#\ x \leftarrow p](Q\ x)$
 apply$($*rule box-imp-distrib*$)$
 apply$($*rule pdl-nec*$)$
 apply *assumption*
done

lemma *pdl-dmd-reg*: $\forall x. \vdash P\ x \longrightarrow_D Q\ x \Longrightarrow \vdash \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D \langle x \leftarrow p\rangle(Q\ x)$
 apply$($*rule dmd-imp-distrib*$)$
 apply$($*rule pdl-nec*$)$
 apply *assumption*
done

theorem *pdl-wkB*: $[\![\vdash [\#\ x \leftarrow p](P\ x); \forall x. \vdash P\ x \longrightarrow_D Q\ x]\!] \Longrightarrow \vdash [\#\ x \leftarrow p](Q\ x)$
 apply$($*rule pdl-mp*$)$
 apply$($*rule box-imp-distrib*$)$
by$($*rule pdl-nec*$)$

theorem *pdl-wkD*: $[\![\vdash \langle x \leftarrow p\rangle(P\ x); \forall x. \vdash P\ x \longrightarrow_D Q\ x]\!] \Longrightarrow \vdash \langle x \leftarrow p\rangle(Q\ x)$
proof $-$
 assume *a*: $\vdash \langle x \leftarrow p\rangle(P\ x)$ **and** *b*: $\forall x. \vdash P\ x \longrightarrow_D Q\ x$
 from *b* **have** $\vdash [\#\ x \leftarrow p](P\ x \longrightarrow_D Q\ x)$ **by** $($*rule pdl-nec*$)$
 hence $\vdash \langle x \leftarrow p\rangle(P\ x) \longrightarrow_D \langle x \leftarrow p\rangle(Q\ x)$ **by** $($*rule pdl-k2*$[$*THEN pdl-mp*$])$
 from *this a* **show** $\vdash \langle x \leftarrow p\rangle(Q\ x)$ **by** $($*rule pdl-mp*$)$
qed

The following rule comes in handy when program sequences occur inside the box.

theorem *pdl-plugB*: $[\![\vdash [\#\ x \leftarrow p](P\ x); \forall x. \vdash P\ x \longrightarrow_D [\#\ y \leftarrow q\ x](C\ y)]\!] \Longrightarrow \vdash [\#\ do\ \{x \leftarrow p;\ q\ x\}]C$
 apply$($*drule pdl-wkB*, *assumption*$)$
 by $($*rule pdl-iffD1*$[$*OF pdl-seqB-simp*, *THEN pdl-mp*$])$

theorem *pdl-plugD*: $[\![\vdash \langle x \leftarrow p\rangle(P\ x); \forall x. \vdash P\ x \longrightarrow_D \langle y \leftarrow q\ x\rangle(C\ y)]\!] \Longrightarrow \vdash \langle do\ \{x \leftarrow p;\ q\ x\}\rangle C$
 apply$($*drule pdl-wkD*, *assumption*$)$
 by $($*rule pdl-iffD1*$[$*OF pdl-seqD-simp*, *THEN pdl-mp*$])$

lemma *box-conj-distrib1*: $\vdash [\#\ x \leftarrow p](P\ x) \wedge_D [\#\ x \leftarrow p](Q\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \wedge_D Q\ x)$
proof $-$
 have $\forall x. \vdash P\ x \longrightarrow_D Q\ x \longrightarrow_D P\ x \wedge_D Q\ x$
 proof
  fix *x* **show** $\vdash P\ x \longrightarrow_D Q\ x \longrightarrow_D P\ x \wedge_D Q\ x$
   by $($*simp only*: *pdl-taut Valid-Ret*$)$
 qed
 hence *a2*: $\vdash [\#\ x \leftarrow p](P\ x) \longrightarrow_D [\#\ x \leftarrow p](Q\ x \longrightarrow_D (P\ x \wedge_D Q\ x))$
  by $($*rule pdl-box-reg*$)$
 from *this pdl-k1* **have** $\vdash [\#\ x \leftarrow p](P\ x) \longrightarrow_D [\#\ x \leftarrow p](Q\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \wedge_D Q\ x)$
  by $($*rule pdl-imp-trans*$)$
 thus *?thesis* **by** $($*simp only*: *pdl-taut*$)$

qed

lemma *box-conj-distrib2*: $\vdash$ [# $x \leftarrow p$]($P\ x \wedge_D Q\ x$) $\longrightarrow_D$ [# $x \leftarrow p$]($P\ x$) $\wedge_D$ [# $x \leftarrow p$]($Q\ x$)
proof –
have $\forall x.\ \vdash P\ x \wedge_D Q\ x \longrightarrow_D P\ x$ **by** (*simp add*: *pdl-taut*)
hence *a1*: $\vdash$ [# $x \leftarrow p$] ($P\ x \wedge_D Q\ x$) $\longrightarrow_D$ [# $x \leftarrow p$]($P\ x$) **by** (*rule pdl-box-reg*)
have $\forall x.\ \vdash P\ x \wedge_D Q\ x \longrightarrow_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence *a2*: $\vdash$ [# $x \leftarrow p$] ($P\ x \wedge_D Q\ x$) $\longrightarrow_D$ [# $x \leftarrow p$]($Q\ x$) **by** (*rule pdl-box-reg*)
let *?P* = [# $x \leftarrow p$]($P\ x$) **and** *?Q* = [# $x \leftarrow p$]($Q\ x$) **and** *?PQ* = [# $x \leftarrow p$]($P\ x \wedge_D Q\ x$)
have $\vdash$ (*?PQ* $\longrightarrow_D$ *?P*) $\longrightarrow_D$ (*?PQ* $\longrightarrow_D$ *?Q*) $\longrightarrow_D$ (*?PQ* $\longrightarrow_D$ *?P* $\wedge_D$ *?Q*)
by (*simp only*: *pdl-taut Valid-Ret*)
from *this a1* **have** $\vdash$ (*?PQ* $\longrightarrow_D$ *?Q*) $\longrightarrow_D$ (*?PQ* $\longrightarrow_D$ *?P* $\wedge_D$ *?Q*) **by** (*rule pdl-mp*)
from *this a2* **show** *?thesis* **by** (*rule pdl-mp*)
qed

The box operator distributes over (finite) conjunction.

theorem *box-conj-distrib*: $\vdash$ [# $x \leftarrow p$]($P\ x \wedge_D Q\ x$) $\longleftrightarrow_D$ [# $x \leftarrow p$]($P\ x$) $\wedge_D$ [# $x \leftarrow p$]($Q\ x$)
apply (*rule pdl-iffI*)
apply (*rule box-conj-distrib2*)
apply (*rule box-conj-distrib1*)
done

Split and join rules for boxes and diamonds.

lemma *pdl-seqB-split*: $\vdash$ [# *do* {$x \leftarrow p$; $y \leftarrow q\ x$; *ret* (x, y)}]($\lambda(x, y).\ P\ x\ y$)
$\Longrightarrow \vdash$ [# p]($\lambda x.$ [# $q\ x$]$P\ x$)
by (*rule pdl-seqB*[*THEN pdl-iffD1*, *THEN pdl-mp*])

lemma *pdl-seqB-join*: $\vdash$ [# p]($\lambda x.$ [# $q\ x$]$P\ x$)
$\Longrightarrow \vdash$ [# *do* {$x \leftarrow p$; $y \leftarrow q\ x$; *ret* (x, y)}]($\lambda(x, y).\ P\ x\ y$)
by (*rule pdl-seqB*[*THEN pdl-iffD2*, *THEN pdl-mp*])

lemma *pdl-seqD-split*: $\vdash$ $\langle$*do* {$x \leftarrow p$; $y \leftarrow q\ x$; *ret* (x, y)}$\rangle$($\lambda(x, y).\ P\ x\ y$)
$\Longrightarrow \vdash \langle p\rangle(\lambda x.\ \langle q\ x\rangle P\ x)$
by (*rule pdl-seqD*[*THEN pdl-iffD1*, *THEN pdl-mp*])

lemma *pdl-seqD-join*: $\vdash \langle p\rangle(\lambda x.\ \langle q\ x\rangle P\ x)$
$\Longrightarrow \vdash$ $\langle$*do* {$x \leftarrow p$; $y \leftarrow q\ x$; *ret* (x, y)}$\rangle$($\lambda(x, y).\ P\ x\ y$)
by (*rule pdl-seqD*[*THEN pdl-iffD2*, *THEN pdl-mp*])

Working in an axiomatic proof system requires a lot of auxiliary rules; especially the lack of an implication introduction rule ($(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$) cries for lots of lemmas that are essentially just basic lemmas lifted over some premiss.

lemma *pdl-wkB-lifted1*: $[\![\vdash A \longrightarrow_D$ [# p]B; $\forall x.\ \vdash B\ x \longrightarrow_D C\ x]\!] \Longrightarrow \vdash A \longrightarrow_D$ [# p]C
proof –
assume *a1*: $\vdash A \longrightarrow_D$ [# p]B **and** *a2*: $\forall x.\ \vdash B\ x \longrightarrow_D C\ x$
from *a2* **have** $\vdash$ [# p]$B \longrightarrow_D$ [# p]C **by** (*rule pdl-box-reg*)
with *a1* **show** *?thesis* **by** (*rule pdl-imp-trans*)
qed

lemma *pdl-wkD-lifted1*: $[\![\vdash A \longrightarrow_D \langle p\rangle B$; $\forall x.\ \vdash B\ x \longrightarrow_D C\ x]\!] \Longrightarrow \vdash A \longrightarrow_D \langle p\rangle C$
proof –

assume *a1*: $\vdash A \longrightarrow_D \langle p\rangle B$ **and** *a2*: $\forall x. \vdash B\ x \longrightarrow_D C\ x$
from *a2* **have** $\vdash \langle p\rangle B \longrightarrow_D \langle p\rangle C$ **by** (*rule pdl-dmd-reg*)
with *a1* **show** *?thesis* **by** (*rule pdl-imp-trans*)
qed

lemma *box-conj-distrib-lifted1*: $\vdash (A \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)) \longleftrightarrow_D ((A \longrightarrow_D [\# p]P) \wedge_D (A \longrightarrow_D [\# p]Q))$
proof (*rule pdl-iffI*)
show $\vdash (A \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)) \longrightarrow_D (A \longrightarrow_D [\# p]P) \wedge_D (A \longrightarrow_D [\# p]Q)$
proof −
have $\vdash ([\# p](\lambda x.\ P\ x \wedge_D Q\ x) \longrightarrow_D [\# p]P \wedge_D [\# p]Q) \longrightarrow_D$
$(A \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)) \longrightarrow_D$
$(A \longrightarrow_D [\# p]P) \wedge_D (A \longrightarrow_D [\# p]Q)$
by (*simp add*: *pdl-taut*)
from *this box-conj-distrib2* **show** *?thesis* **by** (*rule pdl-mp*)
qed
next
show $\vdash ((A \longrightarrow_D [\# p]P) \wedge_D (A \longrightarrow_D [\# p]Q)) \longrightarrow_D A \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)$
proof −
have $\vdash ([\# p]P \wedge_D [\# p]Q \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)) \longrightarrow_D$
$((A \longrightarrow_D [\# p]P) \wedge_D (A \longrightarrow_D [\# p]Q)) \longrightarrow_D$
$A \longrightarrow_D [\# p](\lambda x.\ P\ x \wedge_D Q\ x)$
by (*simp add*: *pdl-taut*)
from *this box-conj-distrib1* **show** *?thesis* **by** (*rule pdl-mp*)
qed
qed

lemma *pdl-seqB-lifted1*: $\vdash (\ A \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P)\ ) \longleftrightarrow_D (\ A \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P\ )$
proof (*rule pdl-iffI*)
show $\vdash (A \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P)) \longrightarrow_D A \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P$
proof −
have $\vdash ([\# p](\lambda x.\ [\# q\ x]P) \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P) \longrightarrow_D$
$(A \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P)) \longrightarrow_D$
$(A \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P)$
by (*simp add*: *pdl-taut*)
from *this pdl-iffD1*[*OF pdl-seqB-simp*] **show** *?thesis* **by** (*rule pdl-mp*)
qed
next
show $\vdash (A \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P) \longrightarrow_D A \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P)$
proof −
have $\vdash ([\# do\ \{x \leftarrow p;\ q\ x\}]P \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P)) \longrightarrow_D$
$(A \longrightarrow_D [\# do\ \{x \leftarrow p;\ q\ x\}]P) \longrightarrow_D$
$(A \longrightarrow_D [\# p](\lambda x.\ [\# q\ x]P))$
by (*simp add*: *pdl-taut*)
from *this pdl-iffD2*[*OF pdl-seqB-simp*] **show** *?thesis* **by** (*rule pdl-mp*)
qed
qed

lemma *pdl-seqD-lifted1*: $\vdash (\ A \longrightarrow_D \langle x \leftarrow p\rangle\langle q\ x\rangle P\ ) \longleftrightarrow_D (A \longrightarrow_D \langle do\ \{x \leftarrow p;\ q\ x\}\rangle P\ )$
proof (*rule pdl-iffI*)
show $\vdash (A \longrightarrow_D \langle p\rangle(\lambda x.\ \langle q\ x\rangle P)) \longrightarrow_D A \longrightarrow_D \langle do\ \{x \leftarrow p;\ q\ x\}\rangle P$
proof −
have $\vdash (\langle p\rangle(\lambda x.\ \langle q\ x\rangle P) \longrightarrow_D \langle do\ \{x \leftarrow p;\ q\ x\}\rangle P) \longrightarrow_D$

$(A \longrightarrow_D \langle p\rangle(\lambda x.\ \langle q\ x\rangle P)) \longrightarrow_D$
$(A \longrightarrow_D \langle do\ \{x{\leftarrow}p;\ q\ x\}\rangle P)$
by (*simp add*: *pdl-taut*)
from *this pdl-iffD1*[*OF pdl-seqD-simp*] **show** *?thesis* **by** (*rule pdl-mp*)
qed
next
show $\vdash (A \longrightarrow_D \langle do\ \{x{\leftarrow}p;\ q\ x\}\rangle P) \longrightarrow_D A \longrightarrow_D \langle p\rangle(\lambda x.\ \langle q\ x\rangle P)$
proof −
have $\vdash ((\langle do\ \{x{\leftarrow}p;\ q\ x\}\rangle P \longrightarrow_D \langle p\rangle(\lambda x.\ \langle q\ x\rangle P)) \longrightarrow_D$
$(A \longrightarrow_D \langle do\ \{x{\leftarrow}p;\ q\ x\}\rangle P) \longrightarrow_D$
$(A \longrightarrow_D \langle p\rangle(\lambda x.\ \langle q\ x\rangle P))$
by (*simp add*: *pdl-taut*)
from *this pdl-iffD2*[*OF pdl-seqD-simp*] **show** *?thesis* **by** (*rule pdl-mp*)
qed
qed

lemma *pdl-plugB-lifted1*: $[\![\vdash A \longrightarrow_D [\#\ p]B;\ \forall x.\ \vdash B\ x \longrightarrow_D [\#\ q\ x]C]\!] \Longrightarrow \vdash A \longrightarrow_D [\#\ do\ \{x{\leftarrow}p;\ q\ x\}]C$
proof −
assume *a1*: $\vdash A \longrightarrow_D [\#\ p]B$ **and** *a2*: $\forall x.\ \vdash B\ x \longrightarrow_D [\#\ q\ x]C$
from *a1 a2* **have** $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ [\#\ q\ x]C)$ **by** (*rule pdl-wkB-lifted1*)
thus *?thesis* **by** (*rule pdl-iffD1*[*OF pdl-seqB-lifted1*, *THEN pdl-mp*])
qed

lemma *pdl-plugD-lifted1*: $[\![\vdash A \longrightarrow_D \langle p\rangle B;\ \forall x.\ \vdash B\ x \longrightarrow_D \langle q\ x\rangle C]\!] \Longrightarrow \vdash A \longrightarrow_D \langle do\ \{x{\leftarrow}p;\ q\ x\}\rangle C$
proof −
assume *a1*: $\vdash A \longrightarrow_D \langle p\rangle B$ **and** *a2*: $\forall x.\ \vdash B\ x \longrightarrow_D \langle q\ x\rangle C$
from *a1 a2* **have** $\vdash A \longrightarrow_D \langle x{\leftarrow}p\rangle\langle q\ x\rangle C$ **by** (*rule pdl-wkD-lifted1*)
thus *?thesis* **by** (*rule pdl-iffD1*[*OF pdl-seqD-lifted1*, *THEN pdl-mp*])
qed

lemma *imp-box-conj1*: $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x) \Longrightarrow \vdash A \longrightarrow_D [\#\ p]B$
proof (*rule pdl-wkB-lifted1*)
assume $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$
show $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$ **.**
next
assume $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$
show $\forall x.\ \vdash B\ x \wedge_D C\ x \longrightarrow_D B\ x$
proof
fix *x* **show** $\vdash B\ x \wedge_D C\ x \longrightarrow_D B\ x$ **by** (*simp add*: *pdl-taut*)
qed
qed

lemma *imp-box-conj2*: $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x) \Longrightarrow \vdash A \longrightarrow_D [\#\ p]C$
proof (*rule pdl-wkB-lifted1*)
assume $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$
show $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$ **.**
next
assume $\vdash A \longrightarrow_D [\#\ p](\lambda x.\ B\ x \wedge_D C\ x)$

show $\forall x. \vdash B\ x \wedge_D C\ x \longrightarrow_D C\ x$
proof
fix *x* **show** $\vdash B\ x \wedge_D C\ x \longrightarrow_D C\ x$ **by** (*simp add*: *pdl-taut*)
qed
qed

The following lemmas show how one can split and join boxes freely with the help of axiom *pdl-seqB-simp*.

lemma *pdl-imp-id*: $\vdash A \longrightarrow_D A$
by (*simp add*: *pdl-taut*)

lemma $\vdash$ [# *do* {*x1*←*p1*; *x2*←*p2*; *x3*←*p3*; *r x1 x2 x3*}]*P* $\longrightarrow_D$
[# *x1*←*p1*][# *x2*←*p2*][# *x3*←*p3*][# *r x1 x2 x3*]*P*
apply(*rule pdl-imp-trans*, *rule pdl-iffD2*[*OF pdl-seqB-simp*], *rule pdl-box-reg* ,*rule allI*)+
by (*simp add*: *pdl-taut*)

lemma $\vdash$ [# *x1*←*p1*][# *x2*←*p2*][# *x3*←*p3*][# *x4*←*p4*][# *r x1 x2 x3 x4*]*P* $\longrightarrow_D$
[# *do* {*x1*←*p1*; *x2*←*p2*; *x3*←*p3*; *x4*←*p4*; *r x1 x2 x3 x4*}]*P*
apply(*rule pdl-plugB-lifted1*, *rule pdl-imp-id*, *rule allI*)+
by (*simp add*: *pdl-taut*)

C.5.3 Examples

Examples from [8, Theorem 6].

lemma $\vdash \langle x \leftarrow p \rangle (P\ x) \vee_D \langle x \leftarrow p \rangle (Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \vee_D Q\ x)$
proof −
have $\forall x. \vdash P\ x \longrightarrow_D P\ x \vee_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence *a1*: $\vdash \langle x \leftarrow p \rangle (P\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \vee_D Q\ x)$ **by** (*rule pdl-dmd-reg*)
have $\forall x. \vdash Q\ x \longrightarrow_D P\ x \vee_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence *a2*: $\vdash \langle x \leftarrow p \rangle (Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \vee_D Q\ x)$ **by** (*rule pdl-dmd-reg*)
let $?P = \langle x \leftarrow p \rangle (P\ x)$ **and** $?Q = \langle x \leftarrow p \rangle (Q\ x)$ **and** $?PQ = \langle x \leftarrow p \rangle (P\ x \vee_D Q\ x)$
have $\vdash (?P \longrightarrow_D ?PQ) \longrightarrow_D (?Q \longrightarrow_D ?PQ) \longrightarrow_D (?P \vee_D ?Q \longrightarrow_D ?PQ)$
by (*simp only*: *pdl-taut Valid-Ret*)
from *this a1* **have** $\vdash (?Q \longrightarrow_D ?PQ) \longrightarrow_D (?P \vee_D ?Q \longrightarrow_D ?PQ)$ **by** (*rule pdl-mp*)
from *this a2* **show** *?thesis* **by** (*rule pdl-mp*)
qed

lemma $\vdash \langle x \leftarrow p \rangle (P\ x) \wedge_D [\#\ x \leftarrow p](Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x)$
proof −
have $\forall x. \vdash Q\ x \longrightarrow_D P\ x \longrightarrow_D P\ x \wedge_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence $\vdash [\#\ x \leftarrow p](Q\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \longrightarrow_D P\ x \wedge_D Q\ x)$
by (*rule pdl-box-reg*)
moreover have $\vdash [\#\ x \leftarrow p](P\ x \longrightarrow_D P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x)$
by (*rule pdl-k2*)
ultimately have $\vdash [\#\ x \leftarrow p](Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x)$
by (*rule pdl-imp-trans*) — transitivity of implication
thus *?thesis* **by** (*simp only*: *pdl-taut*)
qed

lemma *pdl-conj-dmd*: $\vdash \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x) \wedge_D \langle x \leftarrow p \rangle (Q\ x)$
proof –
— first proving the 'P-part'
have *dp*: $\vdash \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x)$
proof –
have *fa*: $\forall x. \vdash P\ x \wedge_D Q\ x \longrightarrow_D P\ x$ **by** (*simp add*: *pdl-taut*)
thus *?thesis*
proof –
assume $\forall x. \vdash P\ x \wedge_D Q\ x \longrightarrow_D P\ x$
thus $\vdash \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (P\ x)$ **by** (*rule pdl-dmd-reg*)
qed
qed
— the same for Q
moreover
have *dq*: $\vdash \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (Q\ x)$
proof –
have *fa*: $\forall x. \vdash P\ x \wedge_D Q\ x \longrightarrow_D Q\ x$ **by** (*simp add*: *pdl-taut*)
thus *?thesis*
proof –
assume $\forall x. \vdash P\ x \wedge_D Q\ x \longrightarrow_D Q\ x$
thus $\vdash \langle x \leftarrow p \rangle (P\ x \wedge_D Q\ x) \longrightarrow_D \langle x \leftarrow p \rangle (Q\ x)$ **by** (*rule pdl-dmd-reg*)
qed
qed
— Now assemble the results to arrive at the main thesis
ultimately show *?thesis* **by** (*rule pdl-conjI-lifted*)
qed

lemma $\vdash [\#\ x \leftarrow p](P\ x) \vee_D [\#\ x \leftarrow p](Q\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \vee_D Q\ x)$
proof –
have $\forall x. \vdash P\ x \longrightarrow_D P\ x \vee_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence *a1*: $\vdash [\#\ x \leftarrow p](P\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \vee_D Q\ x)$ **by** (*rule pdl-box-reg*)
have $\forall x. \vdash Q\ x \longrightarrow_D P\ x \vee_D Q\ x$ **by** (*simp add*: *pdl-taut*)
hence *a2*: $\vdash [\#\ x \leftarrow p](Q\ x) \longrightarrow_D [\#\ x \leftarrow p](P\ x \vee_D Q\ x)$ **by** (*rule pdl-box-reg*)
let $?P = [\#\ x \leftarrow p](P\ x)$ **and** $?Q = [\#\ x \leftarrow p](Q\ x)$ **and** $?PQ = [\#\ x \leftarrow p](P\ x \vee_D Q\ x)$
have $\vdash (?P \longrightarrow_D ?PQ) \longrightarrow_D (?Q \longrightarrow_D ?PQ) \longrightarrow_D (?P \vee_D ?Q \longrightarrow_D ?PQ)$
by (*simp only*: *pdl-taut Valid-Ret*)
from *this a1 a2* **show** *?thesis* **by** (*rule pdl-mp-2x*)
qed

end

C.6 A Deterministic Parser Monad with Fall Back Alternatives

theory $Parsec = PDL + MonEq$:

In a typical implementation of this parser monad, T would have the form $T\ A = (S \Rightarrow (E + A) \times S)$, i.e. it would be a state monad (over states S) with exceptions of type E. The fall back alternative q in $p||q$ would then only be used if p failed to terminate.

consts
item :: *nat T* — Parses exactly one character (natural number)

$fail$ :: $'a\ T$ — Always fails
alt :: $'a\ T \Rightarrow 'a\ T \Rightarrow 'a\ T$ (**infixl** $\|$ 140) — Prefer first parser, but fall back on second if necessary

$getInput$ :: $nat\ list\ T$ — read the current state
$setInput$ :: $nat\ list \Rightarrow unit\ T$

constdefs
$eot :: bool\ T$
$eot \equiv (do\ \{i \leftarrow getInput;\ ret\ (null\ i)\})$
$Eot :: bool\ D$
$Eot \equiv \Uparrow eot$
$GetInput :: nat\ list\ D$
$GetInput \equiv \Uparrow getInput$

GetInput and *Eot* are the abstractions in $'a\ D$ of the resp. lower case terms in $'a\ T$.

axioms
$dsef\text{-}getInput$: $dsef\ getInput$
$fail\text{-}bot$: $\vdash [\#\ fail](\lambda x.\ Ret\ False)$
$eot\text{-}item$: $\vdash Eot \longrightarrow_D [\#\ x{\leftarrow}item](Ret\ False)$
$set\text{-}get$: $\vdash \langle setInput\ x\rangle(\lambda u.\ GetInput =_D Ret\ x)$
$get\text{-}item$: $\vdash GetInput =_D Ret\ (y\#ys) \longrightarrow_D \langle x{\leftarrow}item\rangle(Ret\ (x = y) \wedge_D GetInput =_D Ret\ ys)$
$altB\text{-}iff$: $\vdash [\#\ x{\leftarrow}p\|q](P\ x) \longleftrightarrow_D (\ [\#\ x{\leftarrow}p](P\ x) \wedge_D \langle x{\leftarrow}p\rangle(Ret\ True)\ ) \vee_D$
$(\ [\#\ x{\leftarrow}q](P\ x) \wedge_D [\#\ x{\leftarrow}p](Ret\ False)\ )$
$altD\text{-}iff$: $\vdash \langle x{\leftarrow}p\|q\rangle(P\ x) \longleftrightarrow_D \langle x{\leftarrow}p\rangle(P\ x) \vee_D (\langle x{\leftarrow}q\rangle(P\ x) \wedge_D [\#\ x{\leftarrow}p](Ret\ False))$
$determ$: $\vdash \langle x{\leftarrow}p\rangle(P\ x) \longleftrightarrow_D [\#\ x{\leftarrow}p](P\ x) \wedge_D \langle x{\leftarrow}p\rangle(Ret\ True)$

Axiom *determ* is the typical relationship between $\langle p\rangle P$ and $[\#\ p]P$ when no nondeterminism is involved. Axioms *altB-iff altD-iff* describe the fall back behaviour of the alternative operation.

dsef getInput implies *dsef eot*.

lemma *dsef-eot*: *dsef eot*
by (*simp add*: *eot-def dsef-seq dsef-ret dsef-getInput*)

Another way to state the properties of alternation (for the diamond operator).

axioms
$altD\text{-}left$: $\vdash \langle p\rangle P \longrightarrow_D \langle p\|q\rangle P$
$altD\text{-}right$: $\vdash \langle q\rangle P \longrightarrow_D \langle p\rangle(\lambda x.\ Ret\ True) \vee_D \langle p\|q\rangle P$

Proof that *Eot* actually is just an abbreviation.

lemma *Eot-GetInput*: $Eot = (GetInput =_D Ret\ [])$
proof −
have *null-eq-nil*: $\forall x.\ null\ x = (x = [])$
proof
fix x **show** $null\ x = (x = [])$
proof ($cases\ x$)
assume $x = []$ **thus** $null\ x = (x = [])$ **by** *simp*
next
fix $a\ list$ **assume** $x = (a\#list)$ **thus** $null\ x = (x = [])$ **by** *simp*
qed
qed

show *?thesis*
by(*simp add*: *Eot-def eot-def GetInput-def MonEq-def liftM2-def*
dsef-getInput Abs-Dsef-inverse Dsef-def Ret-def null-eq-nil)
qed

lemma *GetInput-item-fail*: $\vdash \mathit{GetInput} =_D \mathit{Ret}\ [] \longrightarrow_D [\#\ \mathit{item}](\lambda x.\ \mathit{Ret\ False})$
apply(*rule subst*[*OF Eot-GetInput*])
by (*rule eot-item*)

We can show that an alternative parser terminates iff one of its constituent parsers does.

lemma *par-term*: $\vdash \langle x \leftarrow p \| q\rangle(\mathit{Ret\ True}) \longleftrightarrow_D \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True})$
proof (*rule pdl-iffI*)
have $\vdash (\ \langle x{\leftarrow}p\|q\rangle(\mathit{Ret\ True}) \longrightarrow_D \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True}) \wedge_D [\#\ x{\leftarrow}p](\mathit{Ret\ False})\ ) \longrightarrow_D$
$\langle x{\leftarrow}p\|q\rangle(\mathit{Ret\ True}) \longrightarrow_D \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True})$
by (*simp add*: *pdl-taut*)
moreover note *pdl-iffD1*[*OF altD-iff*]
ultimately show $\vdash \langle p \parallel q\rangle(\lambda x.\ \mathit{Ret\ True}) \longrightarrow_D \langle p\rangle(\lambda x.\ \mathit{Ret\ True}) \vee_D \langle q\rangle(\lambda x.\ \mathit{Ret\ True})$
by (*rule pdl-mp*)
next
have $\vdash (\ \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True}) \wedge_D [\#\ x{\leftarrow}p](\mathit{Ret\ False}) \longrightarrow_D \langle x{\leftarrow}\ p \parallel q\rangle(\mathit{Ret\ True})\ ) \longrightarrow_D$
$(\ [\#\ x{\leftarrow}p](\mathit{Ret\ False}) \longleftrightarrow_D \neg_D \langle x{\leftarrow}p\rangle(\neg_D \mathit{Ret\ False})\ ) \longrightarrow_D$
$\langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True}) \longrightarrow_D \langle x{\leftarrow}\ p \parallel q\rangle(\mathit{Ret\ True})$
by (*simp add*: *pdl-taut*)
moreover
note *pdl-iffD2*[*OF altD-iff*]
moreover
note *box-dmd-rel*
ultimately
show $\vdash \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \vee_D \langle x{\leftarrow}q\rangle(\mathit{Ret\ True}) \longrightarrow_D \langle x{\leftarrow}\ p \parallel q\rangle(\mathit{Ret\ True})$
by (*rule pdl-mp-2x*)
qed

The following two lemmas are immediate from the axioms.

lemma *parI1*: $\vdash\ [\#\ x{\leftarrow}p](P\ x) \wedge_D \langle x{\leftarrow}p\rangle(\mathit{Ret\ True}) \longrightarrow_D [\#\ x{\leftarrow}p\|q](P\ x)$

lemma *parI2*: $\vdash [\#\ x{\leftarrow}p](\mathit{Ret\ False}) \wedge_D [\#\ x{\leftarrow}q](P\ x) \longrightarrow_D [\#\ x{\leftarrow}\ p\|q](P\ x)$

C.6.1 Specifying Simple Parsers in Terms of the Basic Ones

constdefs
$\mathit{sat} :: (\mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow \mathit{nat}\ T$
$\mathit{sat}\ p \equiv \mathit{do}\ \{x{\leftarrow}\mathit{item};\ \mathit{if}\ p\ x\ \mathit{then\ ret}\ x\ \mathit{else\ fail}\}$
$\mathit{digitp} :: \mathit{nat}\ T$
$\mathit{digitp} \equiv \mathit{sat}\ (\lambda x.\ x < 10)$

The intended semantics of *many* is that it maps a parser p into one that applies p as often as possible and collects the results (which may be none). *many1* requires at least one successful run of p.

consts
$\mathit{many} :: {}'a\ T \Rightarrow {}'a\ \mathit{list}\ T$

$many1 :: {'a}\ T \Rightarrow {'a}\ list\ T$

We cannot define *many*, since it is not primitive recursive and there is no termination measure.

axioms
$many\text{-}unfold$: $many\ p = ((do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\})\ \|\ ret\ [])$

defs
$many1\text{-}def$: $many1\ p \equiv (do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\})$

This is the most convenient and expressive rule we can hope for at the moment.

lemma $many\text{-}step$: $[\![\vdash \langle(do\ \{x \leftarrow p;\ xs \leftarrow many\ p;\ ret\ (x\#xs)\})\rangle P \vee_D$
$\langle ret\ []\rangle P \wedge_D [\#\ x{\leftarrow}p](Ret\ False)\ ]\!] \Longrightarrow \vdash \langle many\ p\rangle P$
constdefs
$natp :: nat\ T$
$natp \equiv do\ \{ns \leftarrow many1\ digitp;\ ret\ (foldl\ (\lambda r\ n.\ 10 * r + n)\ 0\ ns)\}$

The parser for natural numbers *natp* works on an input stream that conists of natural numbers and reads numbers between 0 and 9 (inclusive) until no such number can be read. Then it transforms its result list into a number by folding an appropriate function into the list. Of course, one might just as well consider an input stream of bounded numbers (e.g. ASCII characters in their numeric representation) and then read '0' to '9', but this would not provide any interesting further insight.

C.6.2 Auxiliary Lemmas

A convenient rendition of axiom *altD-iff* as a rule.

lemma $altD\text{-}iff\text{-}lifted1$: $[\![\vdash A \longrightarrow_D \langle x{\leftarrow}q\rangle(P\ x);\ \vdash A \longrightarrow_D [\#\ x{\leftarrow}p](Ret\ False)]\!] \Longrightarrow \vdash A \longrightarrow_D \langle x{\leftarrow} p\|q\rangle(P\ x)$
proof $-$
have $\vdash (\langle x{\leftarrow}p\|q\rangle(P\ x) \longleftrightarrow_D \langle x{\leftarrow}p\rangle(P\ x) \vee_D \langle x{\leftarrow}q\rangle(P\ x) \wedge_D [\#\ x{\leftarrow}p](Ret\ False)) \longrightarrow_D$
$(A \longrightarrow_D \langle x{\leftarrow}q\rangle(P\ x)) \longrightarrow_D (A \longrightarrow_D [\#\ x{\leftarrow}p](Ret\ False)) \longrightarrow_D$
$A \longrightarrow_D \langle x{\leftarrow}\ p\|q\rangle(P\ x)$
by $(simp\ add\colon pdl\text{-}taut)$
moreover
note *altD-iff*
moreover
assume $\vdash A \longrightarrow_D \langle x{\leftarrow}q\rangle(P\ x)$
moreover
assume $\vdash A \longrightarrow_D [\#\ x{\leftarrow}p](Ret\ False)$
ultimately
show *?thesis* **by** $(rule\ pdl\text{-}mp\text{-}3x)$
qed

The correctness of *natp* obviously relies on the correctness of *digitp*, which is proved first.

theorem $digitp\text{-}nat$: $\vdash GetInput =_D Ret\ (1\#ys) \longrightarrow_D \langle x{\leftarrow}digitp\rangle(Ret\ (x = 1) \wedge_D GetInput =_D Ret\ ys)$
(**is** $\vdash\ ?A \longrightarrow_D \langle digitp\rangle(\lambda x.\ ?C\ x \wedge_D\ ?D)$)
apply$(unfold\ digitp\text{-}def\ sat\text{-}def)$
apply$(rule\ pdl\text{-}plugD\text{-}lifted1)$
apply$(rule\ get\text{-}item)$

apply(*rule allI*)
apply(*simp add*: *split-if*)
apply(*safe*)
apply(*rule pdl-iffD2*[*OF pdl-retD*])
by (*simp add*: *pdl-taut*) — For the else-branch we obtain a contradiction, since the input was 1

On empty input, *digitp* will fail.

theorem *digitp-fail*: $\vdash$ *GetInput* $=_D$ *Ret* [] $\longrightarrow_D$ [# *digitp*](λx. *Ret False*)
apply(*simp add*: *digitp-def sat-def*)
apply(*rule pdl-plugB-lifted1*)
apply(*rule GetInput-item-fail*)
apply(*rule allI*)
apply(*rule pdl-False-imp*)
done

lemma *ret-nil-aux*: $\vdash A \wedge_D B \longrightarrow_D$
$\langle$*ret* []$\rangle$(λxs. $A \wedge_D B \wedge_D$ *Ret* (xs = []))

lemma *ret-one-aux*: $\vdash A \longrightarrow_D$
$\langle$*ret* (*Suc 0*)$\rangle$(λn. *Ret* (n = *Suc 0*) $\wedge_D A$)

lemma *pdl-eqD-aux1*: $\vdash (B \wedge_D C \longrightarrow_D \langle p\ b\rangle P) \longrightarrow_D$ *Ret* $(a = b) \wedge_D B \wedge_D C \longrightarrow_D \langle p\ a\rangle P$
lemma *pdl-eqD-aux2*: $\vdash (A \longrightarrow_D \langle\ p\ b\rangle P) \longrightarrow_D A \wedge_D$ *Ret* $(a = b) \longrightarrow_D \langle\ p\ a\rangle P$

lemma *pdl-imp-strg1*: $\vdash A \longrightarrow_D C \Longrightarrow\ \vdash A \wedge_D B \longrightarrow_D C$
lemma *pdl-imp-strg2*: $\vdash B \longrightarrow_D C \Longrightarrow\ \vdash A \wedge_D B \longrightarrow_D C$

C.6.3 Correctness of the Monadic Parser

The following is a major theorem, more because of its complexity and since it involves most of the axioms given for the monad, than because of its theoretical insight. Essentially, it states that *natp* behaves totally correct for a given input.

theorem *natp-corr*: $\vdash$ $\langle$*do* {*uu*$\leftarrow$*setInput* [*1*]; *natp*}$\rangle$(λn. *Ret* (n = *1*) $\wedge_D$ *Eot*)
proof –
have $\vdash$ $\langle$*uu*$\leftarrow$*setInput* [*1*]$\rangle$(*GetInput* $=_D$ *Ret* [*1*])
by (*rule set-get*)
moreover
have $\forall$*uu*::*unit*. $\vdash$ *GetInput* $=_D$ *Ret* [*1*] $\longrightarrow_D$ $\langle n\leftarrow$*natp*$\rangle$(*Ret* (n = *1*) $\wedge_D$ *Eot*)
proof
fix *uu*
— The actual proof starts here: from a given input, show that *natp* is correct
show $\vdash$ *GetInput* $=_D$ *Ret* [*1*] $\longrightarrow_D$ $\langle$*natp*$\rangle$(λn. *Ret* (n = *1*) $\wedge_D$ *Eot*)
proof –
— Prove the formula with defn. of *natp* unfolded
have $\vdash$ *GetInput* $=_D$ *Ret* [*1*] $\longrightarrow_D$ $\langle$*do* {*x*$\leftarrow$*digitp*; *xs*$\leftarrow$*many digitp*; *ret* (*foldl* (λr. *op* + (*10* $*$ *r*)) *x xs*)}$\rangle$(λn. *Ret* (n = *1*) $\wedge_D$ *Eot*) (**is** $\vdash$ *?a* $\longrightarrow_D$ *?b*)
proof – — Work out each atomic program separately
have $\vdash$ *GetInput* $=_D$ *Ret* [*1*] $\longrightarrow_D$ $\langle x\leftarrow$*digitp*$\rangle$(*Ret* (*x*=*1*) $\wedge_D$ *GetInput* $=_D$ *Ret* [])
by (*rule digitp-nat*)
moreover

```
    have ∀x. ⊢ (Ret (x=(1::nat)) ∧D GetInput =D Ret []) ⟶D
      (⟨do {xs←many digitp; ret (foldl (λr. op + (10 ∗ r)) x xs)}⟩(λn. Ret(n=1) ∧D Eot))
    proof — Here, digitp will fail, ie. many will return []
      fix x
      show ⊢ Ret (x = 1) ∧D GetInput =D Ret [] ⟶D
        ⟨do {xs←many digitp; ret (foldl (λr. op + (10 ∗ r)) x xs)}⟩(λn. Ret (n = 1) ∧D Eot)
      proof (rule pdl-plugD-lifted1[where B = λxs. Ret (x = 1) ∧D GetInput =D Ret [] ∧D
                                          Ret (xs = [])])
        show ⊢ Ret (x = 1) ∧D GetInput =D Ret [] ⟶D
          ⟨many digitp⟩(λxs. Ret (x = 1) ∧D GetInput =D Ret [] ∧D Ret (xs = []))
          apply(subst many-unfold)
          apply(rule altD-iff-lifted1)
          apply(rule ret-nil-aux)
          apply(rule pdl-plugB-lifted1)
          apply(rule pdl-imp-strg2)
          apply(rule digitp-fail)
          apply(rule allI)
          by (simp add: pdl-taut)
      next
        show ∀xs. ⊢ Ret (x = 1) ∧D GetInput =D Ret [] ∧D Ret (xs = []) ⟶D
                ⟨ret (foldl (λr. op + (10 ∗ r)) x xs)⟩(λn. Ret (n = 1) ∧D Eot)
          apply(rule allI)
          apply(rule pdl-eqD-aux1[THEN pdl-mp])
          apply(rule pdl-eqD-aux2[THEN pdl-mp])
          apply(simp)
          apply(subst Eot-GetInput)
          by (rule ret-one-aux)
      qed
    qed
    ultimately
    show ?thesis by (rule pdl-plugD-lifted1)
  qed
  thus ?thesis by (simp add: natp-def many1-def mon-ctr del: bind-assoc)
 qed
qed
ultimately show ?thesis by (rule pdl-plugD)
qed
end
```

C.7 A Simple Reference Monad with `while` and `if`

theory *State* = *PDL* + *MonEq*:

Read/write operations on references of arbitrary type, and a while loop.

```
typedecl 'a ref

consts
 newRef    :: 'a ⇒ 'a ref T
 readRef   :: 'a ref ⇒ 'a T
 writeRef  :: 'a ref ⇒ 'a ⇒ unit T        ((- := -) [100, 10] 10)
 monWhile  :: bool D ⇒ unit T ⇒ unit T     (WHILE (4-) /DO (4-) /END)
```

To make the dsef operation of reading a reference more readable (pun unintended), we introduce syntactical sugar: $*r$ stands for $\Uparrow$ *readRef r*.

syntax
$\textit{-readRefD} :: {'a}\ \textit{ref} \Rightarrow {'a}\ D \qquad (*\text{-}\ [100]\ 100)$

translations
$\textit{-readRefD}\ r \quad \rightleftharpoons \quad \Uparrow (\textit{readRef}\ r)$

This definition is rather useless as it stands, since one actually wants *oldref r* to be a formula in *bool D*. The quantifier is necessary to avoid introducing a fresh variable *a* on the right hand side of the definition.

The idea is appealing however, since it would provide a statement about the existence of *r* as a reference.

constdefs
$\textit{oldref} \quad :: {'a}\ \textit{ref} \Rightarrow \textit{bool}$
$\textit{oldref}\ r \equiv \forall a. \vdash [\#\ s{\leftarrow}\textit{newRef}\ a](\textit{Ret}\ (\neg(r{=}s)))$

The basic axioms of a simple while language with references. In the following we will not make use of operation *newRef* and hence neither of its axioms.

axioms
dsef-read: $\textit{dsef}\ (\textit{readRef}\ r)$
read-write: $\vdash [\#\ r := x](\lambda uu.\ *r =_D \textit{Ret}\ x)$
read-write-other-gen: $\vdash \Uparrow (\textit{do}\ \{u{\leftarrow}\textit{readRef}\ r;\ \textit{ret}\ (f\ u)\}) \longrightarrow_D$
$\quad [\#\ s := y](\lambda uu.\ \textit{Ret}\ (r{\neq}s) \longrightarrow_D \Uparrow (\textit{do}\ \{u{\leftarrow}\textit{readRef}\ r;\ \textit{ret}\ (f\ u)\}))$
while-par: $\vdash P \wedge_D b \longrightarrow_D [\#\ p](\lambda u.\ P) \Longrightarrow \vdash P \longrightarrow_D [\#\ \textit{WHILE}\ b\ \textit{DO}\ p\ \textit{END}](\lambda x.\ P \wedge_D \neg_D\ b)$
read-new: $\vdash [\#\ r{\leftarrow}\textit{newRef}\ a](\ \textit{Ret}\ a =_D *r)$
read-new-other: $\vdash (\textit{Ret}\ x =_D *r) \longrightarrow_D [\#\ s \leftarrow \textit{newRef}\ y]((\textit{Ret}\ x =_D *r) \vee_D \textit{Ret}\ (r{=}s))$

lemma *read-write-other*: $\vdash (\ *r =_D \textit{Ret}\ x) \longrightarrow_D [\#\ s := y](\lambda uu.\ \textit{Ret}\ (r{\neq}s) \longrightarrow_D (\ *r =_D \textit{Ret}\ x))$
proof −
have $\vdash \Uparrow (\textit{do}\ \{u{\leftarrow}\textit{readRef}\ r;\ \textit{ret}\ (u = x)\}) \longrightarrow_D$
$\quad [\#\ s := y](\lambda uu.\ \textit{Ret}\ (r{\neq}s) \longrightarrow_D \Uparrow (\textit{do}\ \{u{\leftarrow}\textit{readRef}\ r;\ \textit{ret}\ (u = x)\}))$
by (*rule read-write-other-gen*)
thus *?thesis*
by (*simp add*: *MonEq-def liftM2-def Dsef-def Ret-def Abs-Dsef-inverse dsef-read*)
qed

It is not really necessary to step back to the do-notation for *read-write-other-gen*.

lemma $\vdash *r =_D \textit{Ret}\ b \wedge_D \textit{Ret}\ (f\ b) \longrightarrow_D \Uparrow (\textit{do}\ \{a{\leftarrow}\textit{readRef}\ r;\ \textit{ret}\ (f\ a \wedge a = b)\})$

Definitions of oddity and evenness of natural numbers, as well as an algorithm for computing Russian multiplication *rumult*.

constdefs
$\textit{nat-even} :: \textit{nat} \Rightarrow \textit{bool}$
$\textit{nat-even}\ n \equiv 2\ \textit{dvd}\ n$
$\textit{nat-odd} :: \textit{nat} \Rightarrow \textit{bool}$
$\textit{nat-odd}\ n \equiv \neg\ \textit{nat-even}\ n$
$\textit{rumult} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{nat ref} \Rightarrow \textit{nat ref} \Rightarrow \textit{nat ref} \Rightarrow \textit{nat}\ T$
$\textit{rumult}\ a\ b\ x\ y\ r \equiv \textit{do}\ \{x{:=}a;\ y{:=}b;\ r{:=}0;$

$WHILE\ (\Uparrow\ (do\ \{u \leftarrow readRef\ x;\ ret\ (0 < u)\}))$
$DO\ do\ \{u \leftarrow readRef\ x;\ v \leftarrow readRef\ y;\ w \leftarrow readRef\ r;$
$if\ (nat\text{-}odd\ u)\ then\ (r := w + v)\ else\ ret\ ();$
$x := u\ div\ 2;\ y := v * 2\}\ END;\ readRef\ r\}$

C.7.1 General Auxiliary Lemmas

Following are several auxiliary lemmas which are not general enough to be placed inside the general theory files, but which are used more than once below – and thus justify their mere existence.

Some weakening rules.

lemma *pdl-conj-imp-wk1*: $\vdash A \longrightarrow_D C \Longrightarrow \vdash A \wedge_D B \longrightarrow_D C$
proof −
assume $\vdash A \longrightarrow_D C$
have $\vdash (A \longrightarrow_D C) \longrightarrow_D A \wedge_D B \longrightarrow_D C$
by (*simp add*: *pdl-taut*)
thus *?thesis* **by** (*rule pdl-mp*)
qed

lemma *pdl-conj-imp-wk2*: $\vdash B \longrightarrow_D C \Longrightarrow \vdash A \wedge_D B \longrightarrow_D C$
proof −
assume $\vdash B \longrightarrow_D C$
have $\vdash (B \longrightarrow_D C) \longrightarrow_D A \wedge_D B \longrightarrow_D C$
by (*simp add*: *pdl-taut*)
thus *?thesis* **by** (*rule pdl-mp*)
qed

The following can be used to prove a specific goal by proving two parts separately. It is similar to *pdl-iffD2* [*OF box-conj-distrib-lifted1* , *THEN pdl-mp*], which is

$\vdash (A\text{-}2 \longrightarrow_D [\#\ p\text{-}2]P\text{-}2) \wedge_D (A\text{-}2 \longrightarrow_D [\#\ p\text{-}2]Q\text{-}2) \Longrightarrow$
$\vdash A\text{-}2 \longrightarrow_D [\#\ p\text{-}2](\lambda x.\ P\text{-}2\ x \wedge_D Q\text{-}2\ x)$

.

lemma *pdl-conj-imp-box-split*: $[\![\vdash A \longrightarrow_D [\#\ p]C;\ \vdash B \longrightarrow_D [\#\ p]D]\!] \Longrightarrow \vdash A \wedge_D B \longrightarrow_D [\#\ x \leftarrow p](C\ x \wedge_D D\ x)$
proof (*rule pdl-iffD2*[*OF box-conj-distrib-lifted1*, *THEN pdl-mp*])
assume *a1*: $\vdash A \longrightarrow_D [\#\ p]C$ **and** *a2*: $\vdash B \longrightarrow_D [\#\ p]D$
show $\vdash (A \wedge_D B \longrightarrow_D [\#\ p]C) \wedge_D (A \wedge_D B \longrightarrow_D [\#\ p]D)$
proof (*rule pdl-conjI*)
show $\vdash A \wedge_D B \longrightarrow_D [\#\ p]C$
proof (*rule pdl-conj-imp-wk1*)
show $\vdash A \longrightarrow_D [\#\ p]C$ **.**
qed
next
show $\vdash A \wedge_D B \longrightarrow_D [\#\ p]D$
proof (*rule pdl-conj-imp-wk2*)
show $\vdash B \longrightarrow_D [\#\ p]D$ **.**
qed
qed
qed

Since dsef programs may be discarded, a formula is equal to itself prefixed by such a program.

lemma *dsef-form-eq*: $dsef\ p \Longrightarrow P = \Uparrow (do\ \{a \leftarrow p; \Downarrow P\})$
proof $-$
 assume *a1*: *dsef p*
 have *f1*: $do\ \{a \leftarrow p; \Downarrow P\} = \Downarrow P$
 proof (*rule dis-left2*)
  show *dis p*
   by (*rule dsef-dis*[*OF a1*])
 qed
 thus *?thesis*
 proof $-$
  have $P = \Uparrow (\Downarrow P)$
   by (*rule Rep-Dsef-inverse*[*symmetric*])
  with *f1* **show** *?thesis* **by** *simp*
 qed
qed

A rendition of *pdl-dsefB*.

lemma *dsefB-D*: $dsef\ p \Longrightarrow \vdash P \longrightarrow_D [\#\ x \leftarrow p]P$
by(*subst dsef-form-eq*[*of p P*], *assumption*, *rule pdl-iffD1*[*OF pdl-dsefB*])

An even number is equal to the sum of its div-halves.

lemma *even-div-eq*: $nat\text{-}even\ n = (n\ div\ 2 + n\ div\ 2 = n)$
 apply(*unfold nat-even-def*)
 by *arith*

Dividing *n* by two and adding the result to itself yields a number one less than *n*.

lemma *odd-div-eq*: $nat\text{-}odd\ (x{::}nat) = (x\ div\ 2 + x\ div\ 2 + 1 = x)$
 apply(*simp add*: *nat-odd-def nat-even-def*)
 by (*arith*)

A slight variant of *pdl-dsefB* for stateless formulas.

lemma *pdl-dsefB-ret*: $dsef\ p \Longrightarrow \vdash \Uparrow (do\ \{a \leftarrow p; ret\ (P\ a)\}) \longleftrightarrow_D [\#\ a \leftarrow p](Ret\ (P\ a))$
 apply(*subgoal-tac* $\forall a.\ ret\ (P\ a) = \Downarrow Ret\ (P\ a)$)
 apply(*simp*)
 apply(*rule pdl-dsefB*)
 apply(*assumption*)
 apply(*simp add*: *Ret-ret*)
done

C.7.2 Problem-Specific Auxiliary Lemmas

The following lemmas are required for the final correctness proof to go through, but are of rather limited interest in general.

lemma *var-aux1*: $\vdash (\ *y =_D Ret\ b \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D (Ret\ (x \neq y) \longrightarrow_D *x =_D Ret\ a)\ ) \longrightarrow_D$
  $(\ *x =_D Ret\ a \wedge_D *y =_D Ret\ b \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r)\ )$
 by (*simp add*: *conjD-Ret-hom pdl-taut*)

lemma *var-aux2*: $\vdash ((\ {*r} =_D Ret\ 0 \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r)) \wedge_D (Ret\ (x \neq r) \longrightarrow_D {*x} =_D Ret\ a)) \wedge_D$
$(Ret\ (y \neq r) \longrightarrow_D {*y} =_D Ret\ b) \longrightarrow_D$
$(\ {*x} =_D Ret\ a \wedge_D {*y} =_D Ret\ b \wedge_D {*r} =_D Ret\ (0{::}nat) \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r))$
by (*simp add*: *conjD-Ret-hom pdl-taut*)

The following proof it typical: since some formulas are built from do-terms and then lifted into *bool D*, the usual proof rules will not get us far. The standard scheme in this case is to proceed as documented in the following side remarks.

lemma *derive-inv-aux*: $\vdash {*x} =_D Ret\ a \wedge_D {*y} =_D Ret\ b \wedge_D {*r} =_D Ret\ (0{::}nat) \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r)$
$\longrightarrow_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
$\Uparrow (do\ \{u{\leftarrow}readRef\ x;\ v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u{*}v{+}w = a{*}b)\})$
(**is** $\vdash\ ?x \wedge_D\ ?y \wedge_D\ ?r \wedge_D\ ?diff \longrightarrow_D\ ?diff \wedge_D\ ?seq$)
proof −
— Simplify the goal by proving something tautologously equivalent.
have $\vdash (?x \wedge_D\ ?y \wedge_D\ ?r \longrightarrow_D\ ?seq) \longrightarrow_D$
$(?x \wedge_D\ ?y \wedge_D\ ?r \wedge_D\ ?diff \longrightarrow_D\ ?diff \wedge_D\ ?seq)$ **by** (*simp add*: *pdl-taut*)
moreover
have $\vdash\ ?x \wedge_D\ ?y \wedge_D\ ?r \longrightarrow_D\ ?seq$
— Turn the formula into a straight program sequence
apply(*simp add*: *liftM2-def impD-def conjD-def MonEq-def dsef-read Abs-Dsef-inverse Dsef-def Ret-ret*)
apply(*simp add*: *dsef-read Abs-Dsef-inverse Dsef-def dsef-seq*)
apply(*simp add*: *mon-ctr del*: *bind-assoc*)
— Sort programs so that equal ones are next to each other
apply(*simp del*: *dsef-ret add*: *commute-dsef*[*of readRef r readRef x*] *dsef-read*)
apply(*simp del*: *dsef-ret add*: *commute-dsef*[*of readRef y readRef x*] *dsef-read*)
apply(*simp del*: *dsef-ret add*: *commute-dsef*[*of readRef r readRef y*] *dsef-read*)
— Remove duplicate occurrences of all programs
apply(*simp add*: *dsef-cp*[*OF dsef-read*[*of x*]] *cp-arb*)
apply(*simp add*: *dsef-cp*[*OF dsef-read*[*of y*]] *cp-arb*)
apply(*simp add*: *dsef-cp*[*OF dsef-read*[*of r*]] *cp-arb*)
— Finally prove the returned stateless formula and conclude by reducing the program to *ret True*
apply(*simp add*: *dsef-dis*[*OF dsef-read*] *dis-left2*)
apply(*simp add*: *Valid-simp Abs-Dsef-inverse Dsef-def*)
done
ultimately show *?thesis* **by** (*rule pdl-mp*)
qed

lemma *doterm-eq1-aux*: $do\ \{u{\leftarrow}readRef\ x;\ v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u * v + w = a * b)\} =$
$do\ \{u{\leftarrow}readRef\ x;\ \Downarrow (\Uparrow (do\ \{v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u * v + w = a * b)\}))\}$

lemma *doterm-eq2-aux*: $do\ \{v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u * v + w = a * b)\} =$
$do\ \{v{\leftarrow}readRef\ y;\ \Downarrow (\Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (u * v + w = a * b)\}))\}$

lemma *arith-aux*: $[\![nat\text{-}odd\ u;\ u * v + w = a * b]\!] \Longrightarrow (u\ div\ 2 + u\ div\ 2) * v + (w + v) = a * b$

lemma *rel1-aux*: $nat\text{-}odd\ u \Longrightarrow \vdash\ (\ Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D {*r} =_D Ret\ (w + v) \wedge_D Ret\ (u * v + w = a * b)\ ) \longrightarrow_D$
$Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ ((u\ div\ 2 + u\ div\ 2) * v + w = a * b)\})$

(**is** $?odd \Longrightarrow \vdash (?diff \wedge_D ?r \wedge_D ?ar) \longrightarrow_D ?diff \wedge_D ?seq$)

lemma *wrt-other-aux*: $\vdash Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (f\ w)\}) \longrightarrow_D$
$[\#\ x := a](\lambda uu.\ Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (f\ w)\}))$

lemma *wrt-other2-aux*: $\vdash Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (f\ w)\}) \longrightarrow_D$
$[\#\ y := b](\lambda uu.\ Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (f\ w)\}))$

lemma *rd-seq-aux*: $\vdash \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ (f\ a\ w)\}) \wedge_D *x =_D Ret\ a \longrightarrow_D$
$\Uparrow (do\ \{u{\leftarrow}readRef\ x;\ w{\leftarrow}readRef\ r;\ ret\ (f\ u\ w)\})$

lemma *arith2-aux*: $(u\ div\ (2{::}nat) + u\ div\ 2) * v + w = a * b \longrightarrow u\ div\ 2 * (v * 2) + w = a * b$

lemma *asm-results-aux*: $\vdash (Ret\ (x \neq y) \longrightarrow_D *x =_D Ret\ (u\ div\ (2{::}nat))) \wedge_D$
$*y =_D Ret\ (v * 2) \wedge_D$
$Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D \Uparrow (do\ \{w{\leftarrow}readRef\ r;\ ret\ ((u\ div\ 2 + u\ div\ 2) * v + w = a * b)\}) \longrightarrow_D$
$Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D \Uparrow (do\ \{u{\leftarrow}readRef\ x;\ v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u * v + w = a * b)\})$

Yet another dsef formula extension.

lemma *yadfe*: $[\![dsef\ p;\ dsef\ q;\ dsef\ r;\ \forall x\ y\ z.\ f\ x\ y\ z]\!] \Longrightarrow \vdash \Uparrow (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ z{\leftarrow}r;\ ret\ (f\ x\ y\ z)\})$
proof −
assume *ds*: *dsef p dsef q dsef r*
assume *a1*: $\forall x\ y\ z.\ f\ x\ y\ z$
hence $\Downarrow (\Uparrow (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ z{\leftarrow}r;\ ret\ (f\ x\ y\ z)\})) =$
$\Downarrow (\Uparrow (do\ \{x{\leftarrow}p;\ y{\leftarrow}q;\ z{\leftarrow}r;\ ret\ True\}))$
by (*simp*)
also from *ds* **have** $\ldots = ret\ True$
by (*simp add*: *Abs-Dsef-inverse Dsef-def dsef-seq dis-left2 dsef-dis*)
finally show *?thesis* **by** (*simp add*: *Valid-simp*)
qed

lemma *conclude-aux*: $\vdash (Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
$\Uparrow (do\ \{u{\leftarrow}readRef\ x;\ v{\leftarrow}readRef\ y;\ w{\leftarrow}readRef\ r;\ ret\ (u * v + w = (a{::}nat) * b)\})) \wedge_D$
$\neg_D \Uparrow (do\ \{u{\leftarrow}readRef\ x;\ ret\ (0 < u)\}) \longrightarrow_D$
$[\#\ readRef\ r](\lambda x.\ Ret\ (x = a * b))$

C.7.3 Correctness of Russian Multiplication

Equipped with all these prerequisites, the correctness proof of Russian multiplication is 'at your fingertips'™. We will not display the actual rule applications but only the important proof goals arising in between.

theorem *russian-mult*: $\vdash (Ret\ (x{\neq}y \wedge y{\neq}r \wedge x{\neq}r)) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
apply(*unfold rumult-def*) — First, unfold the definition of *rumult*
apply(*simp only*: *seq-def*)
apply(*rule pdl-plugB-lifted1*)

Establish the 'strongest postcondition' of the assignment to x

$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ x := a]?B$

From this postcondition proceed with assignment to y

$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge xa. \vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D *x =_D Ret\ a \longrightarrow_D [\#\ y := b]?B9\ xa$

After the final assignment to r all variables will have their initial values

$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge xa\ xaa.$
$\vdash *x =_D Ret\ a \wedge_D *y =_D Ret\ b \wedge_D Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D$
$[\#\ r := 0]?B27\ xa\ xaa$

Now we have arrived at the while-loop, with the invariant readily established.

$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge xa\ xaa\ xb.$
$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
$\Uparrow (do\ \{u \leftarrow readRef\ x;$
$v \leftarrow readRef\ y;\ w \leftarrow readRef\ r;\ ret\ (u * v + w = a * b)\}) \longrightarrow_D$
$[\#\ do\ \{x \leftarrow WHILE \Uparrow (do\ \{u \leftarrow readRef\ x;\ ret\ (0 < u)\})$
$DO\ do\ \{u \leftarrow readRef\ x;$
$v \leftarrow readRef\ y;$
$w \leftarrow readRef\ r;$
$xa \leftarrow if\ nat\text{-}odd\ u\ then\ r := w + v\ else\ ret\ ();$
$x \leftarrow x := u\ div\ 2;\ y := v * 2\}$
$END;$
$readRef\ r\}](\lambda x.\ Ret\ (x = a * b))$

apply(*rule pdl-plugB-lifted1*)
apply(*rule while-par*) — applied the while rule

After splitting off the while-loop as a single box formula, we can apply the while rule, so that we obtain the following proof goal, telling us to establish the invariant after one run of the loop body:

$\vdash Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge xa\ xaa\ xb.$
$\vdash (Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
$\Uparrow (do\ \{u \leftarrow readRef\ x;$
$v \leftarrow readRef\ y;\ w \leftarrow readRef\ r;\ ret\ (u * v + w = a * b)\})) \wedge_D$
$\Uparrow (do\ \{u \leftarrow readRef\ x;\ ret\ (0 < u)\}) \longrightarrow_D$
$[\#\ do\ \{u \leftarrow readRef\ x;$
$v \leftarrow readRef\ y;$
$w \leftarrow readRef\ r;$
$xa \leftarrow if\ nat\text{-}odd\ u\ then\ r := w + v\ else\ ret\ ();$
$x \leftarrow x := u\ div\ 2;$
$y := v *$
$2\}](\lambda u.\ Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
$\Uparrow (do\ \{u \leftarrow readRef\ x;$
$v \leftarrow readRef\ y;$
$w \leftarrow readRef\ r;\ ret\ (u * v + w = a * b)\}))$

After having worked off all read operations, we again have to establish the strongest postcondition that is required after the if-statement.

$\vdash$ *Ret* $(x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge u\ v\ w.$
 $\vdash$ *Ret* $(0 < u) \wedge_D$
 Ret $(x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
 Ret $(u * v + w = a * b) \wedge_D$
 $\Uparrow (do\ \{w \leftarrow readRef\ r;\ ret\ (u * v + w = a * b)\}) \longrightarrow_D$
 $[\#\ if\ nat\text{-}odd\ u\ then\ r := w + v\ else\ ret\ ()]?B111\ u\ v\ w$

Here we see what the just mentioned postcondition looks like: it says that the following relation (found in the premiss of the implication) holds:

$\vdash$ *Ret* $(x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge u\ v\ w\ xa.$
 $\vdash$ *Ret* $(x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
 $\Uparrow (do\ \{w \leftarrow readRef\ r;\ ret\ ((u\ div\ 2 + u\ div\ 2) * v + w = a * b)\}) \longrightarrow_D$
 $[\#\ x := u\ div\ 2]?B142\ u\ v\ w\ xa$

Now only the assignment to y remains.

$\vdash$ *Ret* $(x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge u\ v\ w\ xa\ xaa.$
 $\vdash *x =_D Ret\ (u\ div\ 2) \wedge_D$
 Ret $(x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
 $\Uparrow (do\ \{w \leftarrow readRef\ r;\ ret\ ((u\ div\ 2 + u\ div\ 2) * v + w = a * b)\}) \longrightarrow_D$
 $[\#\ y := v * 2]?B151\ u\ v\ w\ xa\ xaa$

We finally succeeded in re-establishing the loop invariant after one execution of the loop body. The final part is just to read reference r, which is easily done.

$\vdash$ *Ret* $(x \neq y \wedge y \neq r \wedge x \neq r) \longrightarrow_D [\#\ rumult\ a\ b\ x\ y\ r](\lambda x.\ Ret\ (x = a * b))$
1. $\bigwedge xa\ xaa\ xb\ xc.$
 $\vdash (Ret\ (x \neq y \wedge y \neq r \wedge x \neq r) \wedge_D$
 $\Uparrow (do\ \{u \leftarrow readRef\ x;$
 $v \leftarrow readRef\ y;\ w \leftarrow readRef\ r;\ ret\ (u * v + w = a * b)\})) \wedge_D$
 $\neg_D \Uparrow (do\ \{u \leftarrow readRef\ x;\ ret\ (0 < u)\}) \longrightarrow_D$
 $[\#\ readRef\ r](\lambda x.\ Ret\ (x = a * b))$

apply(*rule conclude-aux*) — … Just 124 straightforward proof steps later
done

end

Bibliography

[1] Peter B. Andrews. *An Introduction to Mathematical Logic: To Truth Through Proof*. Number 27 in Applied Logic Series. Kluwer Academic Publishers, 2002.

[2] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Clarendon, 1992.

[3] Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited – and Isabelle/Isar experience. In *Theorem Proving in Higher Order Logics*, number 2152 in LNCS. Springer-Verlag, 2001.

[4] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Number 53 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[5] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. ICFP, Montreal, Canada, 2000.

[6] Melvin Fitting. Basic modal logic. In *Handbook of logic in artificial intelligence and logic programming*, volume 1, pages 368–448. Oxford University Press, Inc., New York, NY, USA, 1993.

[7] Martin Fowler. *UML Distilled*. Object technology. Addison-Wesley, 3rd edition, 2004.

[8] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic, 2nd ed.*, volume 4. Kluwer Academic Publishers, Dordrecht, 2002.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[10] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

[11] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer-Verlag, 2000.

[12] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[13] B. Jacobs and E. Poll. A monad for basic Java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 150–164. Springer-Verlag, 2000.

[14] Simon P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr 2003.

[15] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Symposium on Principles of Programming Languages*. ACM Press, Jan 1993.

[16] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[17] Richard Kuhn, Ramaswamy Chandramouli, and Ricky Butler. Cost effective use of formal methods in verification and validation. In *Foundations 02 Workshop on Verification & Validation*. Columbia, MD, Oct 2002.

[18] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1998.

[19] E. Moggi. A semantics for evaluation logic. *Fund. Inform.*, 22:117–152, 1995.

[20] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[21] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 1(1), 1998.

[22] Tobias Nipkow. Structured proofs in Isar/HOL. In *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer-Verlag, 2003.

[23] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.

[25] Lawrence C. Paulson. *The Isabelle Reference Manual*, 2004. Available at `http://isabelle.in.tum.de/doc/ref.pdf`.

[26] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, 1991.

[27] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer-Verlag, Berlin, 1991.

[28] Andrew M. Pitts. Categorical logic. In *Handbook of Logic in Computer Science*, volume VI. Oxford University Press, May 1995.

[29] Gordon Plotkin. A structural approach to operational semantics, 1981. The Aarhus notes. Available at `http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps`.

[30] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[31] Stuart Russell and Peter Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Pearson Education, Inc., 2nd edition, 2003.

[32] Lutz Schröder and Till Mossakowski. Monad-independent Hoare logic in HASCASL. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 261–277. Springer, Berlin, 2003.

[33] Lutz Schröder and Till Mossakowski. Generic exception handling and the java monad. In *Algebraic Methodology and Software Technology*, volume 3116 of *LNCS*, pages 443–459, 2004.

[34] Lutz Schröder and Till Mossakowski. Monad-independent dynamic logic in HASCASL. *Journal of Logic and Computation*, 14(4):571–619, 2004.

[35] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[36] Ann E. Kelley Sobel and Michael R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3), Mar 2002.

[37] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[38] Dennis Walter, Lutz Schröder, and Till Mossakowski. Parametrized exceptions. In Jose Fiadeiro and Jan Rutten, editors, *Algebra and Coalgebra in Computer Science*, Lecture Notes in Computer Science. Springer; Berlin, 2005. To appear.

[39] Markus Wenzel. *Isabelle/Isar - a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, 2002.

[40] Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction*. The MIT Press, Cambridge, Massachusetts, 1993.