A Generic Complete Dynamic Logic for Reasoning About Purity and Effects

Till Mossakowski[1,2], Lutz Schröder[1,2], and Sergey Goncharov[2]

[1] DFKI Laboratory, Bremen
[2] Department of Computer Science, University of Bremen

Abstract. For a number of programming languages, among them Eiffel, C, Java and Ruby, Hoare-style logics and dynamic logics have been developed. In these logics, pre- and postconditions are typically formulated using potentially effectful programs. In order to ensure that these pre- and postconditions behave like logical formulae (that is, enjoy some kind of referential transparency), a notion of purity is needed. Here, we introduce a generic framework for reasoning about purity and effects. Effects are modeled abstractly and axiomatically, using Moggi's idea of encapsulation of effects as monads. We introduce a dynamic logic (from which, as usual, a Hoare logic can be derived) whose logical formulae are pure programs in a strong sense. We formulate a set of proof rules for this logic, and prove it to be complete with respect to a categorical semantics. Using dynamic logic, we then develop a relaxed notion of purity which allows for observationally neutral effects such writing on newly allocated memory.

1 Introduction

Design and programming by contract reduces software errors by providing specifications of Hoare-style pre- and postconditions and invariants along with the program. Originating in the Eiffel language [12], this paradigm has become a guiding design principle for a number of languages, including Sather [17], Lisaac [24], Nice [3], and D [4]. Moreover, for many existing languages, among them C, C++, Java, JavaScript, Scheme, Perl, Python, and Ruby, libraries, preprocessors and other tools have been developed that support programming by contract; for example, the Java modeling language JML comes with extended static analysis [5] and verification [26] tools. [6] treats contracts in a higher-order functional setting.

Unlike Hoare's original logic [8] and unlike some known formalisations of Hoare-logics in theorem provers like Isabelle and PVS [9, 27, 16], most of these languages do not have a separate language for expressing pre- and postconditions. Instead, these are expressed in a *pure subset* of the programming language itself. The restriction to a pure subset is necessary since specifications should not have side-effects; this serves both conceptual clarity and the possibility of run-time testing (which could easily yield wrong results if specifications could change the behaviour of a program).

Purity in this sense is closely related to referential transparency. Informally, pure programs have the following properties:

1. discardability: pure computations can be left out from a sequence of computation steps without changing its behaviour;

2. determinism: pure programs always return the same value;
3. interchangeability: pure programs can be interchanged with each other, that is, the order does not matter.

Of course, these properties depend on a suitable notion of observational equivalence of programs that explains the term "without changing its behaviour". The observational equivalence in turn depends on the specific set of possible operations; for example, allocating a new memory cell and writing a value into it will not be observable in Java, while in C, it can be observed if the integer representation of the memory cell's address is guessed or known by chance and converted from an integer to a pointer. By contrast, the latter operation is not available in Java.

It becomes clear that the Hoare logics for the above mentioned languages, while all following essentially the same style, may vary in subtle but important details. This greatly complicates the study of properties of such logics. Consequently, the goal of this paper is to formalize these notions and provide logical rules for Hoare-style reasoning in a way that abstracts from the details of the particular languages. The abstraction is achieved by encapsulating notions of effect and computation as monads, as originally suggested by Moggi [13] and used in the design of the functional programming language Haskell [18] as well as in programming semantics, e.g. for Java [10]. In earlier work [21, 23], we have developed a dynamic logic for generic side effects whose semantics is defined in terms of monads; in particular, we have given a sound proof calculus for this logic.[1] Here, we extend the proof calculus to a calculus which is strongly complete for the generic part of the calculus and linear sequential programs[2]. This setting is sufficient for our treatment of observational purity. Our logic is related to Pitts' evaluation logic [19], which in turn essentially puts the language-specific logic of [2] on a generic basis. Both these logics omit loops, like the restricted logic considered here, but unlike the full logic of [23].

In more detail, the definition of our dynamic logic is based on a strict notion of purity where the above requirements for referential transparency (discardability, determinism, interchangeability) are postulated to hold as actual equalities. Once we have the logic available, we can define a weaker notion of *observational purity* where referential transparency holds only up to observational equivalence in the logic. We thus arrive at a practically usable notion of purity, which in particular allows harmless side-effects such as creating and writing on new references. This generic concept of observational purity is related to the programming language specific notion put forward in [15]. An interesting point here is that as a byproduct of our completeness proof, we obtain the existence of fully abstract categorical models, where observational equalities hold on the nose, and in particular all observationally pure functions are pure in the original strict sense.

Besides the specification and verification of monadic programs, our logic serves also the *specification* of monads, i.e. notions of side effect and purity. That is, effects may

[1] There are two dynamic logics for Java, one for the KIV prover [25], and on in the KeY project [1]. However, they neither address purity, nor are they generic.

[2] The original calculus does feature generic loop constructs; however, absolute completeness results such as the one proved here are impossible for dynamic logics over Turing complete languages: this would entail recursive enumerability of the set of non-terminating programs.

be described in an axiomatic manner, which abstracts away from particular details of the implementation, and moreover allows for loose specifications that deliberately leave open particular design decisions.

2 Monads for Computations

The presence of computational effects, e.g. state, store, exceptions, input, output, non-determinism, or backtracking, substantially complicates reasoning about programs. Dealing with such features has in the past typically required dedicated logics, designed specifically for a particular combination of effects. In seminal work by Moggi [13], monads have been established as a unifying framework for modeling computational effects (including in particular all effects mentioned above) in an elegant way.

Intuitively, a monad associates to each type A a type TA of computations of type A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts from particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can be carried out without commitment to a particular type of side effect.

A monad on a given category $\mathbf{C}$ can be defined as a *Kleisli triple* $\mathbb{T} = (T, \eta, \_^*)$, where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $\_^*$ assigns to each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ such that

$$\eta_A^* = id_{TA}, \quad f^*\eta_A = f, \quad \text{and} \quad g^*f^* = (g^*f)^*.$$

This description is equivalent to the more familiar one [11].

In order to support a language with finitary operations and multi-variable contexts (see below), one needs a further ingredient: a monad is called *strong* if it is equipped with a natural transformation

$$t_{A,B} : A \times TB \to T(A \times B)$$

called *strength*, subject to certain coherence conditions (see e.g. [13]).

Example 1 ([13]). Computationally relevant monads on $\mathbf{Set}$ (all monads on $\mathbf{Set}$ are strong [23, 13]) include the following.

1. *Stateful computations with non-termination:* $TA = S \to ?\ (A \times S)$, where S is a fixed set of states and $\_ \to ?\ \_$ denotes the partial function type.
2. *Non-determinism:* $TA = \mathcal{P}(A)$, where $\mathcal{P}$ is the covariant power set functor.
3. *Exceptions*: $TA = A + E$, where E is a fixed set of exceptions.
4. a) *Interactive input:* TA is the smallest fixed point of $\gamma \mapsto A + (U \to \gamma)$, where U is a set of input values. b) *Interactive output:* TA is the smallest fixed point of $\gamma \mapsto A + (U \times \gamma)$, where U is a set of output values.
5. *Non-deterministic stateful computations:* $TA = (S \to \mathcal{P}(A \times S))$, where, again, S is a fixed set of states (here, we can use the total function arrow since the binding operator can treat undefinedness as the empty set of results).

6. These monads can also be combined. E.g. non-deterministic stateful computations are obtained as $TA = S \to \mathcal{P}(A \times S)$. A monad for Java has been defined as follows [10]

$$JA = S \to (A \times S + E \times S + 1),$$

with S the set of states and E the set of exceptions. A state typically will comprise a stack, a heap, and a heap pointer, that is,

$$S = V^* \times (\mathit{Loc} \mapsto V) \times \mathit{Loc} \times \cdots$$

where V is a set of values and Loc a set of locations. While Java will provide an operation $\mathit{new} : J\ \mathit{Loc}$ that allocates a new location, C will additionally provide a coercion operation $\mathit{int2loc} : \mathit{Int} \to \mathit{Loc}$ (written e.g. as "`(*char)`" in C, if characters are stored).

3 Monad-Based Dynamic Logic

In program specification, dynamic logic as introduced in [20] and extended to monadic computations in [23] has a number of advantages over less expressive formalisms such as Hoare logic, among them the ability to express both partial and total correctness in a natural way and the possibility of reusing a state, say for statements of the nature 'what would happen if'. Here, we examine the infrastructure that is needed in order to develop generic monad-based dynamic logic, and illustrate that this does indeed make sense when instantiated to typical concrete computational monads.

3.1 Syntax

We begin by fixing the syntax of *monad-based dynamic logic (MDL)*. Given a set Sort of *basic types*, the set of *types* over Sort is generated by

$$A ::= 1 \,|\, \Omega \,|\, PA \,|\, TA \,|\, A \times A \,|\, \mathit{Sort}.$$

A *signature* $\Sigma = (\mathit{Sort}, F)$ consists of a set Sort of basic types and a set F of operation symbols $f : A \to B$, where A and B are types over Sort. Examples of such operation symbols are $\mathit{new} : T\ \mathit{Loc}$ and $\mathit{int2loc} : \mathit{Int} \to \mathit{Loc}$ as explained above. We make the general assumption that the operations in the signature have T*-free* argument types, more precisely, types containing neither T nor P. The term language over a signature Σ and a context Γ of typed variables is given in Fig. 1. $\Gamma \rhd t : A$ means that from context Γ, one can drive that term t has type A. $\top$ stands for true, $\bot$ for false. We let metavariables t etc. range over terms, a and b over normal formulas (i.e. terms of type Ω), φ, ψ, χ over monadic formulas (i.e. terms of type $P\Omega$), and p, q etc. over programs, i.e. terms whose type is of the form TA. $\mathrm{ret}\ t$ is an effectless computation that just returns the value t. In the term $\mathrm{do}\ x \leftarrow p; q$, the variable x is locally bound in q (but not in p), the interpretation is "perform computation p, bind the result to x and then perform computation $q(x)$". Repeated bindings such as $\mathrm{do}\ x_1 \leftarrow p_1, \ldots, x_n \leftarrow p_n; q$ are somewhat inaccurately denoted in the form $\mathrm{do}\ \bar{x} \leftarrow \bar{p}; q$. Term fragments of the form $\bar{x} \leftarrow \bar{p}$ are called *program sequences*.

$$\textbf{(var)}\ \frac{x : A \in \Gamma}{\Gamma \rhd x : A} \qquad \textbf{(app)}\ \frac{f : A \to B \in \Sigma \quad \Gamma \rhd t : A}{\Gamma \rhd f(t) : B} \qquad \textbf{(1)}\ \frac{}{\Gamma \rhd * : 1}$$

$$\textbf{(pair)}\ \frac{\Gamma \rhd t : A \quad \Gamma \rhd u : B}{\Gamma \rhd \langle t, u \rangle : A \times B} \qquad \textbf{(fst)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathrm{fst}(t) : A} \qquad \textbf{(snd)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathrm{snd}(t) : A}$$

$$(\top)\ \frac{}{\Gamma \rhd \top : \Omega} \quad (\bot)\ \frac{}{\Gamma \rhd \bot : \Omega} \quad (\neg)\ \frac{\Gamma \rhd a : \Omega}{\Gamma \rhd \neg a : \Omega} \quad \text{similarly for } \wedge, \vee, \Rightarrow, \iff$$

$$\textbf{(do)}\ \frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TB}{\Gamma \rhd \mathrm{do}\ x \leftarrow p; q : TB} \qquad \textbf{(doP)}\ \frac{\Gamma \rhd p : PA \quad \Gamma, x : A \rhd q : PB}{\Gamma \rhd \mathrm{do}\ x \leftarrow p; q : PB}$$

$$\textbf{(ret)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathrm{ret}\ t : PA} \quad (\Box)\ \frac{\Gamma \rhd p : T\Omega}{\Gamma \rhd \Box p : P\Omega} \qquad \textbf{(P)}\ \frac{\Gamma \rhd p : PA}{\Gamma \rhd p : TA}$$

Fig. 1. Term language for monad-based dynamic logic

The operations fst and snd are the projection functions for binary products. We treat n-ary products as iterated binary products; then projections π_i^n can easily be defined in terms of fst and snd.

The type TA contains the monadic programs over A; PA is a subtype of TA containing pure programs. The type of Booleans is denoted Ω; consequently, we take $P\Omega$ as the type of formulae of monad-based dynamic logic. The term forming operation $\Box : T\Omega \to P\Omega$ is a closure operator. The formula $\Box p$ intuitively expresses that all terminating runs of p return $\top$. Note that $\Box$ does not behave like a modal operator; in particular, for $\varphi : P\Omega$, we will have $\varphi \iff \Box\varphi$. However, $\Box$ serves to define modalities: For $\varphi : P\Omega$, we let $[\bar{x} \leftarrow \bar{p}]\,\varphi$ abbreviate the formula $\Box\,\mathrm{do}\ \bar{x} \leftarrow \bar{p}; \varphi$, and omit $\bar{x}$ if it does not occur in φ. The formula $\langle \bar{x} \leftarrow \bar{p} \rangle \varphi$ abbreviates $\neg[\bar{x} \leftarrow \bar{p}]\,\neg\varphi$. *Both in* do*-terms and within modal operators, we implicitly identify terms up to α-equivalence.*

Besides the specification and verification of monadic programs, MDL serves also the (potentially loose) *specification* of monads, i.e. notions of side effect.

The running example of [21, 23] involves references and non-determinism; numerous further examples can be found in [28], including the Java monad and a parsing monad, as well as a queue monad over a fixed set U of entries. Here, we present the specification of a monad for dynamic references. It is axiomatized using Hoare triples for total correctness

$$[\varphi]\ p\ [\psi]$$

by interpreting them as partial correctness plus termination:

$$\varphi \Rightarrow (\langle p \rangle \top \wedge [p]\,\psi).$$

The implication $\varphi \Rightarrow \langle p \rangle \top$ is called the termination part, and the implication $\varphi \Rightarrow [p]\,\psi$ is called the partial correctness part of the Hoare triple.

Example 2. We assume, for any basic type a, a (basic) type $Ref\ a$ of references to objects of type a. The specification of dynamic references is then as follows.

$$
\begin{array}{lr}
*\_\_ : Ref\ a \to P\ a & \\
\_\_ := \_\_ : (Ref\ a) \times a \to T\ 1 & \\
new : a \to T(Ref\ a) & \\
[\,]r := x\ [x = *r] & \text{read-write} \\
[x = *r]\ s := y\ [x = *r \vee r = s] & \text{read-write-other} \\
[\,]\ r \leftarrow new\ x\ [x = *r] & \text{read-new} \\
[x = *r]\ s \leftarrow new\ y\ [x = *r \vee \neg r = s] & \text{read-new-other} \\
[\,]\ r \leftarrow new\ x; p; s \leftarrow new\ y\ [\neg r = s] & \text{new-distinct}
\end{array}
$$

The operation $*\_\_$ reads the value from the location specified by the reference, while $\_\_ := \_\_$ assigns a new value to the location, and new creates a new location.

Note that the profile of $*\_\_$ ensures that reading is pure. The first axiom expresses that after assignment, the assigned value can be read; the second one that assignment for a particular location does not affect the other locations. The next two axioms state similar properties for the new operation, and the last axiom ensures that two newly created locations are distinct.

$$\textbf{(var)}\ \frac{}{\llbracket x_1 : A_1, \dots, x_n : A_n \rhd x_i : A_i \rrbracket = \pi_i^n}$$

$$\textbf{(app)}\ \frac{f : A \to B \in \Sigma \quad \llbracket \Gamma \rhd t : A \rrbracket = h}{\llbracket \Gamma \rhd f(t) : B \rrbracket = \llbracket f \rrbracket \circ h} \qquad \textbf{(1)}\ \frac{}{\llbracket \Gamma \rhd * : 1 \rrbracket = !}$$

$$\textbf{(pair)}\ \frac{\llbracket \Gamma \rhd t : A \rrbracket = h_1 \quad \llbracket \Gamma \rhd u : B \rrbracket = h_2}{\llbracket \Gamma \rhd \langle t, u \rangle : A \times B \rrbracket = \langle h_1, h_2 \rangle}$$

$$\textbf{(fst)}\ \frac{\llbracket \Gamma \rhd t : A \times B \rrbracket = h}{\llbracket \Gamma \rhd \mathrm{fst}(t) : A \rrbracket = \pi_1 \circ h} \qquad \textbf{(snd)}\ \frac{\llbracket \Gamma \rhd t : A \times B \rrbracket = h}{\llbracket \Gamma \rhd \mathrm{snd}(t) : A \rrbracket = \pi_2 \circ h}$$

$$(\top)\ \frac{}{\llbracket \Gamma \rhd \top : \Omega \rrbracket = \top} \qquad (\bot)\ \frac{}{\llbracket \Gamma \rhd \bot : \Omega \rrbracket = \bot}$$

$$(\neg)\ \frac{\llbracket \Gamma \rhd a : \Omega \rrbracket = h}{\llbracket \Gamma \rhd \neg a : \Omega \rrbracket = \neg(h)} \quad \text{similarly for } \wedge, \vee, \Rightarrow, \iff$$

$$\textbf{(do)}\ \frac{\llbracket \Gamma \rhd p : TA \rrbracket = h_1 \quad \llbracket \Gamma, x : A \rhd q : TB \rrbracket = h_2}{\llbracket \Gamma \rhd \mathrm{do}\ x \leftarrow p; q : TB \rrbracket = h_2^* \circ t_{\llbracket \Gamma \rrbracket, \llbracket A \rrbracket} \circ \langle id, h_1 \rangle}$$

$$\textbf{(ret)}\ \frac{\llbracket \Gamma \rhd t : A \rrbracket = h}{\llbracket \Gamma \rhd \mathrm{ret}\ t : PA \rrbracket = \eta_{\llbracket A \rrbracket} \circ h}$$

$$(\Box)\ \frac{\llbracket \Gamma \rhd p : T\Omega \rrbracket = h}{\llbracket \Gamma \rhd \Box p : P\Omega \rrbracket = \Box \circ h} \qquad \textbf{(P)}\ \frac{\llbracket \Gamma \rhd p : PA \rrbracket = h}{\llbracket \Gamma \rhd p : TA \rrbracket = \iota_A \circ h}$$

Fig. 2. Semantics of monad-based dynamic logic

3.2 Semantics

MDL is interpreted over a strong monad $\mathbb{T}$ on a cartesian category $\mathbf{C}$[3] with additional structure as follows. There is a distinguished object Ω such that hom-sets into Ω coherently carry a Boolean algebra structure; i.e. the hom-functor $\mathrm{hom}(\_, \Omega)$ factors through Boolean algebras. In order to enforce a Boolean logic, we additionally require that $(id_A \times \top : A \times 1 \to A \times \Omega, id_A \times \bot : A \times 1 \to A \times \Omega)$ is an episink. Moreover, $\mathbb{T}$ needs to be equipped with a strong submonad $\mathbb{P}$, with functor part P (the inclusion is denoted $\iota : P \to T$); we require that PA consists of pure computations as defined further below.[4] Finally, we need a left inverse $\Box : T\Omega \to P\Omega$ of the inclusion $\iota : P\Omega \hookrightarrow T\Omega$. It will be axiomatized uniquely later on (cf. Prop. 11).

Remark 3. In a distributive category, one obtains the Boolean algebra structure by defining Ω as $1 + 1$. E.g., every topos is distributive, and in a classical topos, the subobject classifier is just $1 + 1$. In a category with equalizers, there is also a canonical choice for the subfunctor P, namely to take PA as the subobject of TA determined by purity as defined below. Then the $\Box$ arrow is uniquely determined if it exists (see Prop. 11 below).

We then interpret the basic sorts as objects in $\mathbf{C}$. This is easily extended to all types, giving an interpretation $[\![A]\!]$ for each type A. Basic operations $f : A \to B$ are interpreted as morphisms $[\![A]\!] \to [\![B]\!]$, and terms $x_1 : A_1, \ldots, x_n : A_n \rhd t : A$ as morphisms $[\![t]\!] : [\![A_1 \times \cdots \times A_n]\!] \to [\![A]\!]$, using the cartesian structure for pairing and projections, the monad for do and ret, and the Boolean algebra structure on Ω for the Boolean connectives as shown in Fig. 2. (Note that there is no particular rule for sequential composition within P: the fact that $\mathbb{P}$ is a submonad guarantees closedness under sequential composition). Observe, that logical connectives are also applicable to $P\Omega$. For instance $p \wedge q$ is an alias for $(\mathrm{do}\ x \leftarrow p; y \leftarrow q; \mathrm{ret}\ x \wedge y)$. The fact that $P\Omega$ inherits Boolean algebra structure from Ω strongly relies on the definition of P [23]. Equations between terms are interpreted as equations between the corresponding morphisms; this will be used in a series of definitions below (we often leave the context implicit).

3.3 Purity

MDL formulae will be interpreted as computations of type $P\Omega$ (i.e. morphisms from the object interpreting the context into the object $P\Omega$). They are expected to have no side-effect, although they may e.g. read the state (if a notion of state is present in the monad). The three conditions for purity listed in Sect. 1 are abstractly captured as follows.

Definition 4. In a monad given as above, a program $p : TA$ is called *discardable* if

$$(\mathrm{do}\ y \leftarrow p; \mathrm{ret}\ *) = \mathrm{ret}\ *,$$

and *copyable* if

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

[3] $\times$ is product with projections π_1, π_2, 1 the terminal object with $!_A : A \to 1$ the unique arrow.

[4] One obvious possibility is to let PA consist of *all* pure computations; this always gives a strong submonad. We will use this possibility in the examples below.

Finally, p is *pure* if it is both discardable and copyable, and *commutes* with all such programs q, that is,

$$(\mathrm{do}\ x \leftarrow p; y \leftarrow q; \mathrm{ret}\,(x, y)) = \mathrm{do}\ y \leftarrow q; x \leftarrow p; \mathrm{ret}\,(x, y).$$

The last condition follows from the first two for simple monads, on which we focus here; cf. Prop. 18.

For the monads described in Example 1, purity of p means the expected things:

1. State monad: p possibly reads the state, but does not change it;
2. Non-determinism: p is deterministic (i.e. returns precisely one value);
3. Exceptions: p terminates normally;
4. In-/Output: p does not read/write;
5. Nondeterministic state monad: p is deterministic and only reads the state, but does not change it.
6. Java monad: p only reads the state and terminates normally

The equation for discardability can be interpreted as a pair of arrows $dis_0, dis_1 : T[\![A]\!] \to T[\![1]\!]$; we require that $\iota_{[\![A]\!]} : P[\![A]\!] \to T[\![A]\!]$ equalizes this pair. (We do *not* require that $\iota_{[\![A]\!]}$ is the equalizer, for the reconstruction of this equalizer in the term model would require a coercion of provably pure terms of type TA to type PA, which means that term formation rules would need to interact with proof rules.) A similar requirement holds for copyability; i.e. we require that PA contains only pure programs.

3.4 The Logic

We will want to regard programs that return truth values as formulae with side effects in a modal logic setting. A basic notion we need for such formulae is that of global validity, which we denote explicitly by a 'global box' $\boxed{G}$:

Definition 5. Given a term $p : T\Omega$, $\boxed{G}p$ abbreviates the equation

$$p = \mathrm{do}\ p; \mathrm{ret}\ \top.$$

If p is discardable, then $\boxed{G}\,p$ simplifies to $p = \mathrm{ret}\ \top$; otherwise, the equation above ensures that the right hand side has the same side-effect as p. We say that an MDL formula φ is *valid* in a model $\mathbb{T}$, and write $\mathbb{T} \models \varphi$, if $\boxed{G}\varphi$; this is usually expressed by just writing φ. As usual, by $\mathbb{T} \models \Phi$ for a set Φ of MDL formulae we mean that $\mathbb{T} \models \varphi$ for all $\varphi \in \Phi$, and by $\Phi \models \psi$ that $\mathbb{T} \models \Phi$ implies $\mathbb{T} \models \psi$ for all $\mathbb{T}$.

A related notion is that of *global dynamic judgements* of the form $[\bar{x} \leftarrow \bar{p}]\ a$, which intuitively state that a holds after $\bar{x} \leftarrow \bar{p}$, where $a : \Omega$ is a truth-valued term in variables $\bar{x}$. The idea is to work with formulae that have all side effects shoved to the outside, so that the usual logical rules apply to the remaining part.

Definition 6. Given a program sequence $\bar{x} \leftarrow \bar{p}$ and a formula a of type Ω, the notation $[\bar{x} \leftarrow \bar{p}]\ a$ abbreviates the equation

$$(\mathrm{do}\ \bar{x} \leftarrow \bar{p}; \mathrm{ret}\,\langle \bar{x}, a\rangle) = \mathrm{do}\ \bar{x} \leftarrow \bar{p}; \mathrm{ret}\,\langle \bar{x}, \top\rangle.$$

Definition 7. A monad is called *simple* if $\boxed{G}$ do $\bar{x} \leftarrow \bar{p}; \mathrm{ret}\ a$ implies $\boldsymbol{[}\bar{x} \leftarrow \bar{p}\boldsymbol{]}\ a$. (The converse implication holds universally.) Roughly, an algebraic monad [11] is simple if, in each of its equations, the two sides contain the same variables. All monads of Example 1 are simple. The continuation monad and the abelian group monad are not simple. *In the sequel, all monads are assumed to be simple.*

We adapt the axiomatic definition of the dynamic modal operators from [23] to the simplified setting of simple monads:

Definition 8. $\mathbb{T}$ is said to *admit dynamic logic*, if for each $q : T\Omega$ and each $\bar{x} \leftarrow \bar{p}$ containing $x_i : \Omega$,

$$\boldsymbol{[}\bar{x} \leftarrow \bar{p}; a \leftarrow \Box q\boldsymbol{]}\ (x_i \Rightarrow a) \text{ iff } \boldsymbol{[}\bar{x} \leftarrow \bar{p}; a \leftarrow q\boldsymbol{]}\ (x_i \Rightarrow a).$$

This definition essentially axiomatizes $\Box$ via its interaction with global dynamic judgements. In order to gain some intuitive understanding, let us assume that we are working in a non-deterministic state monad. Note that purity of $\Box q$ is enforced by its type; this means that in a given state, $\Box q$ returns either $\top$ or $\bot$ (but not both — it is deterministic). Hence, the above definition expresses that in any given state ($\bar{p}$ can be used to move to that state), $\Box q$ holds iff all executions of q (starting from the given state) return true.

For the next proposition, it is important to recall the difference between the *local box* $[\bar{x} \leftarrow \bar{p}]\ \varphi$ (a monadic value of type $P\Omega$ that may be used in formation of monadic terms) and the *global box* $\boldsymbol{[}\bar{x} \leftarrow \bar{p}\boldsymbol{]}\ a$ (a global equation between certain monadic terms).

Proposition 9. *If a simple monad* $\mathbb{T}$ *admits dynamic logic, then*

$$\boldsymbol{[}\bar{x} \leftarrow \bar{p}; a \leftarrow [\bar{y} \leftarrow \bar{q}]\ \varphi\boldsymbol{]}\ (x_i \Rightarrow a) \textit{ iff } \boldsymbol{[}\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}; a \leftarrow \varphi\boldsymbol{]}\ (x_i \Rightarrow a),$$

i.e. $\mathbb{T}$ *admits dynamic logic in the sense of [23].*

Thus, the operator $[\bar{y} \leftarrow \bar{q}]$ in a sense serves to predict all *positive* statements to be made about the result $\bar{y}$ after executing the computations $\bar{q}$ *in a given state*; the formal content of the latter phrase is reflected in the quantification over program sequences $\bar{x} \leftarrow \bar{p}$ to be executed before $\bar{q}$.

Example 10. All monads described in Example 1 admit dynamic logic, with the meaning of the dynamic modal operators made explicit below. A typical example that fails to admit dynamic logic is the continuation monad $\lambda X.\ (X \to R) \to R$ [23].

1. *Stateful computations with non-termination* ($TA = (S \to ?\ (A \times S))$): Elements of $P\Omega$ are state-dependent truth values, i.e. functions $S \to \Omega$. A state s satisfies $[x \leftarrow p]\ \varphi$ if the value x, if any, returned by p after terminating execution starting in state s satisfies φ.
2. *Non-determinism* ($TA = \mathcal{P}(A)$): $P\Omega$ is isomorphic to the set Ω of truth values; $[x \leftarrow p]\ \varphi$ holds if all $x \in p$ satisfy φ.
3. *Exceptions* ($TA = A + E$): $P\Omega$ is essentially Ω; $[x \leftarrow p]\ \varphi$ holds if p is either an exception or a value x satisfying φ.

4. *Interactive input* ($TA = \mu\gamma.\, A + (U \to \gamma)$): $P\Omega$ is essentially Ω; $[x \leftarrow p]\,\varphi$ holds if the value returned by p after reading a number of inputs satisfies φ. *Interactive output* ($TA = \mu\gamma.\, A + (U \times \gamma)$): again, $P\Omega$ is essentially Ω, and $[x \leftarrow p]\,\varphi$ holds if the value returned by p satisfies φ (the output is ignored).

5. *Non-deterministic stateful computations* ($TA = \mathcal{P}(S \to (A \times S))$): Elements of $P\Omega$ are state-dependent truth values $S \to \Omega$. A state s satisfies $[x \leftarrow p]\,\varphi$ if all values x possibly returned by p after terminating execution starting in state s satisfy φ.

6. *Java monad* ($TA = S \to (A \times S + E \times S + 1)$: again, $P\Omega$ is $S \to \Omega$, and also $[x \leftarrow p]\,\varphi$ means that all values x returned by p after a terminating execution satisfy φ.

While at first sight, items 2–4 look uninteresting from the perspective of dynamic logic, it should be kept in mind that all these monads may be combined with ‘dynamic’ monads such as the state monad, as exemplified for the cases of the non-determinism monad in item 5 and the Java monad in item 6.

For $\varphi, \psi : P\Omega$, let $\varphi \leq \psi$ if

$$\boldsymbol{[}a \leftarrow \varphi; b \leftarrow \psi\boldsymbol{]}\,(a \Rightarrow b)$$

This is easily seen to be a partial order. The following two claims are proved similarly as in [23].

Proposition 11. *The formula $\Box p$ is the greatest formula $\varphi : P\Omega$ such that*

$$\boldsymbol{[}a \leftarrow \varphi; b \leftarrow p\boldsymbol{]}\,(a \Rightarrow b).$$

Proposition 12. $\boldsymbol{[}\bar{x} \leftarrow \bar{p}\boldsymbol{]}\,a$ *iff* $\boxed{\scriptstyle G}[\bar{x} \leftarrow \bar{p}]\,\mathrm{ret}\ a$

Proposition 11 implies that $\Box p$ is uniquely determined if it exists. Proposition 12 relates global dynamic judgements and local modal formulae. Note the difference between *global* dynamic judgements $\boldsymbol{[}\bar{x} \leftarrow \bar{p}\boldsymbol{]}\,a$ and the similar-looking MDL formulae $[\bar{x} \leftarrow \bar{p}]\,\varphi$ involving a *local* modality. From a technical point of view, a global dynamic judgement is an equation between terms (and the component formula a has type Ω), while a local modal formula is a term (and the component formula φ has type $P\Omega$). But the difference is more fundamental: local modalities can be nested, and e.g. in the state monad one can think of them as being evaluated relative to a local state. This is not possible with global dynamic judgements: they always quantify over *all* states.

4 A Calculus for Dynamic Logic

Figure 3 shows a proof calculus for MDL. The calculus differs from the one in [23] in that the rules for the diamond are omitted, because in classical logic, $\Diamond$ can be defined as $\neg\Box\neg$. Moreover, Axioms $(\Box)$ (resembling the implicit definition of $\Box$ in the formula for admission of dynamic logic), (dis) and (copy) (expressing purity of terms of type PA), (unit) and (CC) have been added. The axiom schema (CC) throws in all equations $CC \vdash t = u$ derivable using only the standard equations for tupling, projections, and

$*$, i.e. the internal equations of cartesian categories. The usual logical connectives are lifted to $P\Omega$ by defining e.g.

$$\varphi \Rightarrow \psi := [a \leftarrow \varphi; b \leftarrow \psi]\,\mathrm{ret}\,(a \Rightarrow b) : P\Omega,$$

which by ($\Box$MDL) below is equivalent to

$$\mathrm{do}\ a \leftarrow \varphi; b \leftarrow \psi; \mathrm{ret}\,(a \Rightarrow b).$$

We write $\Phi \vdash \psi$ if a formula is derivable in the calculus from a set Φ of axioms.

Rules:

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

Axioms:

(K)	$[\bar{x} \leftarrow \bar{p}]\,(\varphi \Rightarrow \psi) \Rightarrow [\bar{x} \leftarrow \bar{p}]\,\varphi \Rightarrow [\bar{x} \leftarrow \bar{p}]\,\psi$	
(seq$\Box$)	$[\bar{x} \leftarrow \bar{p}; y \leftarrow q]\,\varphi \iff [\bar{x} \leftarrow \bar{p}]\,[y \leftarrow q]\,\varphi$	
(ctr$\Box$)	$[x \leftarrow p; y \leftarrow q]\,\varphi \iff [y \leftarrow (\mathrm{do}\ x \leftarrow p; q)]\,\varphi$	$(x \notin FV(\varphi))$
(ret$\Box$)	$[x \leftarrow \mathrm{ret}\ t]\,\varphi \iff \varphi[t/x]$	
(dis)	$[x \leftarrow p]\,\psi \iff \psi$	$p : PA$, x not free in ψ
(copy)	$[x \leftarrow p; y \leftarrow p]\,\psi \iff [x \leftarrow p]\,\psi[x/y]$	$p : PA$
(cong-ret)	$\mathrm{ret}\,(t \Leftrightarrow u) \Rightarrow (\varphi[t/x] \iff \varphi[u/x])$	
($\Box$)	$[a \leftarrow \Box p]\,\mathrm{ret}\,(t \Rightarrow a) \iff [a \leftarrow p]\,\mathrm{ret}\,(t \Rightarrow a)$	$p : T\Omega$
(unit)	$[x \leftarrow \varphi]\ \mathrm{ret}\ x \iff \varphi$	
(CC)	$\varphi[t/x] \iff \varphi[u/x]$	for $CC \vdash t = u$
(taut)	$\mathrm{ret}\ a$	$a : \Omega$ a tautology

$$CC = \{\mathrm{fst}(\langle x, y\rangle) = x;\ \mathrm{snd}(\langle x, y\rangle) = y;\ \langle \mathrm{fst}(x), \mathrm{snd}(x)\rangle = x;\ x : 1 = *\}$$

Fig. 3. The generic proof calculus for monad-based dynamic logic

Proposition 13 (Soundness of MDL). *If $\Phi \vdash \psi$, then $\Phi \models \psi$.*

We now discuss some structural properties of the calculus.

Definition 14. A term is in *product β-normal form* if it does not contain subterms of the form $\mathrm{fst}\langle t, u\rangle$ or $\mathrm{snd}\langle t, u\rangle$.

Theorem 15. *The system of reduction rules*

$$\begin{array}{lcl}
\mathrm{fst}(\langle t, u\rangle) & \rightarrowtail & t \\
\mathrm{snd}(\langle t, u\rangle) & \rightarrowtail & u \\
\mathrm{do}\ x \leftarrow \mathrm{ret}\ t; p & \rightarrowtail & p[t/x] \\
\mathrm{do}\ x \leftarrow (\mathrm{do}\ y \leftarrow p; q); r & \rightarrowtail & \mathrm{do}\ y \leftarrow p; x \leftarrow q; r \\
\Box\Box p & \rightarrowtail & \Box p
\end{array}$$

is confluent and strongly normalizing.

In the sequel, the terms *rewriting* and *normal form* will always refer to the above rule system.

Proposition 16. *Let φ be a formula in product β-normal form having a subterm $r : TA$. Then there is a provably equivalent formula ψ with $\varphi \rightarrowtail \psi$ (ψ is even a subformula of φ) such that ψ has one of the following forms:*
1. p, where $p : P\Omega$, *2. $\Box p$, where $p : T\Omega$,*
3. $\mathrm{do}\ \bar{x} \leftarrow \bar{p}; q$, *4. $\Box(\mathrm{do}\ \bar{x} \leftarrow \bar{p}; q)$.*

Using this result, we can apply a leftmost-outermost reduction strategy to obtain

Proposition 17. *An MDL formula φ is provably equivalent to its normal form.*

This result in turn leads to further admissible rules:

Proposition 18. *The following rules are admissible:*

$$\textbf{(cong)}\ \frac{\varphi \Leftrightarrow \psi}{\chi[\varphi/x] \Leftrightarrow \chi[\psi/x]} \qquad \textbf{(subst)}\ \frac{\varphi}{\varphi[t/x]}.$$

The following equivalence is derivable:

$$\textbf{(comm)} \qquad [x \leftarrow \varphi; y \leftarrow \psi]\,\chi \iff [y \leftarrow \psi; x \leftarrow \varphi]\,\chi.$$

For a tautology a, $\mathrm{ret}\ a$ can be derived.

5 Observational Purity

Our notion of purity has some practical limitations: The program $\mathrm{do}\ r \leftarrow \mathit{new}\ x; \mathrm{ret}\ r$ is *not* pure, since it modifies the state. In particular, it is not discardable, since it generally makes a difference whether a new memory cell is allocated or not. However, we generally cannot *observe* whether a memory cell is allocated (unless we program in C, where a function *int2loc* is available). This leads to the following notions:

Given a background MDL theory Φ (that axiomatizes a given combination of effects), two programs p and q are *observationally equivalent*, $p \approx q$, iff for all φ,

$$\Phi \vdash [x \leftarrow p]\,\varphi \iff [x \leftarrow q]\,\varphi.$$

A program is *observationally pure*, if the equations for discardability, copyability and commutation hold up to observational equivalence. For example, in the case of discardability, this means that

$$\Phi \vdash [x \leftarrow (\mathrm{do}\ y \leftarrow p; \mathrm{ret}\ *)]\,\varphi \iff [x \leftarrow \mathrm{ret}\ *]\,\varphi,$$

which amounts to $\Phi \vdash \varphi \iff [p]\,\varphi$.

For the monads described in Example 1, observational purity of p of course depends on the observational equivalence, that is, on the number of observations that can be made. We have:

1. State monad: p possibly reads the state, and moreover possibly makes changes to the state *none of which are observable*. In particular, this means that operations such as *new* are observationally pure as long as there is no possibility to directly observe memory cells.

2. Non-determinism: p is deterministic (i.e. returns precisely one value) *up to observational equivalence*.

3. Exceptions: p terminates normally (note that non-termination is always observable, because $[p]\top$ holds for non-terminating p).

4. In-/Output: any p is observationally pure—this is caused by the non-observability of input and output (assuming there is no feedback between input and output). Informally, this means that setting breakpoints and outputting trace information is harmless.

5. Nondeterministic state monad: p is deterministic up to observational equivalence and changes the state only to an observationally equivalent one.

6. Java monad: p changes the state only to an observationally equivalent one and terminates normally.

6 Completeness

The completeness proof for MDL is based on a term model construction. Given a signature Σ and a set Φ of MDL formulae over Σ, we construct a category $\mathbf{C}_{\Sigma,\Phi}$ as follows: the objects of $\mathbf{C}_{\Sigma,\Phi}$ are the types of Σ, and morphisms $t : A \to B$ are terms in context

$$x : A \rhd t : B,$$

taken modulo *contextual equivalence*

$$t \sim u \text{ iff for all MDL formulae } \varphi,\ \Phi \vdash \varphi[t/x] \iff \varphi[u/x] \qquad (*)$$

(this is obviously a congruence). Identities are given by variables $[x : A \rhd x : A]_\sim$, and composition by substitution

$$[y : B \rhd u : C]_\sim \circ [x : A \rhd t : B]_\sim := [x : A \rhd u[t/y] : C]_\sim.$$

Using axiom (ret□), one easily proves that this is well-defined and obeys the identity and associativity laws of a category. The basic types of Σ are interpreted as themselves, and so are the basic operations:

$$[\![f : A \to B]\!] := [x : A \rhd f(x) : B]_\sim.$$

The category $\mathbf{C}_{\Sigma,\Phi}$ comes with a canonical cartesian structure, which on objects is just given by taking product types as categorical products, and the unit type as the terminal object. Projections are $\pi_1 := [x : A\times B \rhd \mathrm{fst}(x) : A]_\sim$ and $\pi_2 := [x : A\times B \rhd \mathrm{snd}(x) : B]_\sim$, pairing of morphisms is $\langle [t]_\sim, [u]_\sim\rangle := [\langle t, u\rangle]_\sim$, and the unique morphism into the terminal object is $!_A := [x : A \rhd * : 1]_\sim$. Axiom (CC) ensures that this does define a cartesian structure.

The problem with contextual equivalence is that, although its definition is simple and intuitive, it can be quite hard to prove that given monadic programs are contextually equivalent. The key result is

Theorem 19. *For terms of type TA, contextual equivalence $\sim$ coincides with observational equivalence $\approx$.*

We are now ready to complete the term model construction by constructing a monad $\mathbb{T}_{\Sigma,\Phi}$ on $\mathbf{C}_{\Sigma,\Phi}$. It is given by the following data:

$$\mathbb{T}_{\Sigma,\Phi}\, A := T\, A, \qquad \eta_A := [x : A \rhd \text{ret } x : TA]_{\sim},$$

and given $x : A \rhd q : TB$,

$$[x : A \rhd q : TB]^{*}_{\sim} := [p : TA \rhd \text{do } x \leftarrow p; q : TB]_{\sim}$$

Well-definedness follows easily by Theorem 19. Finally, the strength is given by

$$t_{A,B} := [p : A \times TB \rhd \text{do } x \leftarrow \text{snd}(p); \text{ret } \langle \text{fst}(p), x\rangle : T(A \times B)]_{\sim}$$

In $\mathbf{C}_{\Sigma,\Phi}$, $Hom(A, \Omega)$ is coherently turned into a Boolean algebra by defining e.g.

$$[x : A \rhd t : \Omega]_{\sim} \wedge [x : A \rhd u : \Omega]_{\sim} = [x : A \rhd t \wedge u : \Omega]_{\sim}$$

Lemma 20. *In $\mathbf{C}_{\Sigma,\Phi}$, $(id_A \times \top : A \times 1 \to A \times \Omega, id_A \times \bot : A \times 1 \to A \times \Omega)$ is an episink.*

To complete the construction of the term model, we put $\iota_A : PA \to TA := [p : PA \rhd p : TA]_{\sim}$, and $\Box : T\Omega \to P\Omega := [p : T\Omega \rhd \Box p : P\Omega]_{\sim}$.

Corollary 21 (Full abstractness). *In $\mathbb{T}_{\Sigma,\Phi}$, $[\![t]\!] = [\![u]\!]$ iff $t \sim u$.*

By Theorem 19, we also have

Corollary 22. *In $\mathbb{T}_{\Sigma,\Phi}$, for $p, q : TA$, $[\![p]\!] = [\![q]\!]$ iff $p \approx q$.*

The crucial properties of the term model construction are summarized in the next two results.

Proposition 23. *$\mathbb{T}_{\Sigma,\Phi}$ is a fully abstract simple strong monad admitting dynamic logic.*

Lemma 24 (Truth Lemma). *$\mathbb{T}_{\Sigma,\Phi} \models \varphi$ iff $\Phi \vdash \varphi$.*

The main result now follows straightforwardly:

Theorem 25 (Completeness of MDL). *If $\Phi \models \psi$, then $\Phi \vdash \psi$.*

7 Conclusion and Future Work

Inspired by the logics of design by contract languages, we have introduced monad-based dynamic logic (MDL) as a means of handling with effects, purity and observational equivalence in an abstract way that avoids the development of a specific theory for each language. Our generic notion of observational purity captures the same intuition as Naumann's notion of observational purity [15] when instantiated to his specific imperative language. However, besides state, store and non-termination, our logic can also handle effects like non-determinism, input/output, and many others. Moreover, our notions of observational and contextual equivalence (which coincide in the term model) come out of the logic in a more natural way, compared with the variety of notions in [15]. The sound and complete calculus allows for reasoning about Hoare style and dynamic logic assertions as well as about observational purity in a uniform way.

MDL is similar to Pitts' evaluation logic [19], but equipped with a different semantics which is induced directly by the underlying monad — rather than relying on an extra hyperdoctrine structure, which must in all likelihood be considered additional data (e.g. in the case of the state monad, the interpretation of formulae as state predicates is explicitly imposed by the chosen hyperdoctrine). Existing completeness results for monad-based logics rely on a *global* semantics [14, 7], which e.g. for the state monad means that a sequence of nested modalities leads to universal quantification over all states at each new nesting level - the (implicit) state is not passed across nesting levels. By contrast, our logic allows for reasoning in a truly local way about changes of state. So far, no completeness result has been proved for such logics.

A Hoare calculus has been built on top of MDL [21] and extended to a treatment of Java-style abrupt termination [22, 29]. Practical applications of MDL include reasoning about Haskell and the imperative fragment of Java. Numerous examples and a coding in the theorem prover Isabelle can be found in [28].

The completeness result now guarantees closedness of the deduction system (after the extension w.r.t. [23]) and paves the way for the future development of decision procedures. Of course, one cannot expect a decision procedure for arbitrary equational theories. In this respect, our calculus (unlike similar calculi [19, 14, 7]) has the advantage that it makes use only of a rather limited and efficiently decidable equational theory. In particular, equations between programs are avoided; instead, properties of programs are expressed in terms of their observable behaviour.

More practical experience is needed for evaluating how practical actual proofs of purity and observational purity are. Typically, such proofs will not be semantical like those of the claims made in Sect. 5, but rather rely on the types of basic operations that deliver pure results, and on suitable axioms. Indeed, our calculus can be instantiated with a variety of different effects by *axiomatizing* these effects; we have provided one sample axiomatization for the dynamic reference monad.

The generalization to non-simple monads is an important open question. Another important extension will be the treatment of control structures such as while loops (which currently happens at the meta-level) as well as of datatypes in the calculus proper.

Acknowledgements. This work forms part of the DFG-funded project HasCasl (KR 1191/7-1 and KR 1191/7-2). The authors wish to thank the anonymous referees for useful comments, and Erwin R. Catesbeiana for pointing out various pitfalls.

References

[1] Beckert, B.: A dynamic logic for the formal verification of Java Card programs. In: Attali, I., Jensen, T. (eds.) JavaCard 2000. LNCS, vol. 2041, pp. 6–24. Springer, Heidelberg (2001)
[2] Boehm, H.-J.: Side effects and aliasing can have simple axiomatic descriptions. ACM Trans. Program. Lang. Syst 7, 637–655 (1985)
[3] Bonniot, D., Keller, B.: The Nice user's manual (2003), `http://nice.sourceforge.net`
[4] Bright, W.: The D programming language. Dr. Dobb's Journal of Software Tools 27(2), 36–40 (2002)
[5] Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)

[6] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP, pp. 48–59 (2002)
[7] Goncharov, S., Schröder, L., Mossakowski, T.: Completeness of global evaluation logic. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 447–458. Springer, Heidelberg (2006)
[8] Hoare,: An axiomatic basis for computer programming. CACM 12 (1969)
[9] Huisman, M.: Java program verification in higher order logic with PVS and Isabelle. PhD thesis, University of Nijmegen (2001)
[10] Jacobs, B., Poll, E.: Coalgebras and Monads in the Semantics of Java. Theoret. Comput. Sci. 291, 329–349 (2003)
[11] Mac Lane, S.: Categories for the Working Mathematician. Springer, Heidelberg (1997)
[12] Meyer, B.: Eiffel: The Language. Prentice-Hall, Englewood Cliffs (1992)
[13] Moggi, E.: Notions of computation and monads. Inform. and Comput. 93, 55–92 (1991)
[14] Moggi, E.: A semantics for evaluation logic. Fund. Inform. 22, 117–152 (1995)
[15] Naumann, D.A.: Observational purity and encapsulation. Theoret. Comput. Sci 376, 205–224 (2007)
[16] Nipkow, T.: Hoare logics in Isabelle/HOL. In: Schwichtenberg, H., Steinbrüggen, R. (eds.) Proof and System-Reliability, pp. 341–367. Kluwer Academic Publishers, Dordrecht (2002)
[17] Omohundro, S.M.: The Sather language. Technical report, International Computer Science Institute, Berkeley (1991)
[18] Peyton-Jones, S. (ed.): Haskell 98 Language and Libraries — The Revised Report, Cambridge (2003), also: J. Funct. Programming 13 (2003)
[19] Pitts, A.: Evaluation logic. In: Higher Order Workshop, Workshops in Computing, pp. 162–189. Springer, Heidelberg (1991)
[20] Pratt, V.: Semantical considerations on Floyd-Hoare logic. In: Foundations of Conputer Science, FOCS 1976, pp. 109–121. IEEE, Los Alamitos (1976)
[21] Schröder, L., Mossakowski, T.: Monad-independent Hoare logic in HASCASL. In: Pezzè, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 261–277. Springer, Heidelberg (2003)
[22] Schröder, L., Mossakowski, T.: Generic Exception Handling and the Java Monad. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 443–459. Springer, Heidelberg (2004)
[23] Schröder, L., Mossakowski, T.: Monad-independent dynamic logic in HASCASL. J. Logic Comput. 14, 571–619 (2004)
[24] Sonntag, B., Colnet, D.: Lisaac: the power of simplicity at work for operating system. In: Technology of Object-Oriented Languages and Systems, TOOLS Pacific 2002. CRPIT, vol. 10, pp. 45–52. ACS (2002)
[25] Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
[26] van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
[27] von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience 13, 1173–1214 (2001)
[28] Walter, D.: Monadic dynamic logic: Application and implementation. Master's thesis, University of Bremen (2005), http://www.cs.chalmers.se/~denniswa
[29] Walter, D., Schröder, L., Mossakowski, T.: Parametrized exceptions. In: Fiadeiro, J.L., Harman, N.A., Roggenbach, M., Rutten, J. (eds.) CALCO 2005. LNCS, vol. 3629, pp. 424–438. Springer, Heidelberg (2005)