

# Monad-independent dynamic logic in HASCASL

Lutz Schröder      Till Mossakowski

BISS, Dept. of Math. and Comput. Sci., Universität Bremen, Germany

## *Contact:*

Lutz Schröder  
Dept. of Computer Science  
Universität Bremen  
P.O. Box 330 440  
28334 Bremen  
Germany  
Phone: +49-421-218-4683  
Fax: +49-421-218-3054  
E-Mail: lschrode@informatik.uni-bremen.de

## **Abstract**

Monads have been recognized by Moggi as an elegant device for dealing with stateful computation in functional programming languages. In previous work, we have introduced a Hoare calculus for partial correctness of monadic programs. All this has been done in an entirely monad-independent way. Here, we extend this to a monad-independent dynamic logic (assuming a moderate amount of additional infrastructure for the monad). Dynamic logic is more expressive than the Hoare calculus; in particular, it allows reasoning about termination and total correctness. The background formalism for these concepts is the logic of HASCASL, a higher-order language for functional specification and programming. As an example application, we develop a monad-independent Hoare calculus for total correctness based on our dynamic logic, and illustrate this calculus by a termination proof for Dijkstra's non-deterministic implementation of Euclid's algorithm.

*Keywords:* Dynamic logic; monads; algebraic specification; functional programming; CASL

## Introduction

One of the central concepts of modern functional programming is the encapsulation of side effects via monads following the seminal paper [21]. In particular, state monads are used to emulate an imperative programming style in the functional programming language Haskell [40]. Monads can be used to abstract from a particular notion of computation, since they model a wide range of computational effects: e.g., stateful computations, non-determinism, partiality, exceptions, input, and output can all be viewed as monadic computations, and so can various combinations of these concepts such as non-deterministic stateful computations.

Moggi [21] has suggested a Hoare calculus for a state monad with state interpreted as global store. We have generalized this in [36] to a monad-independent Hoare calculus for partial correctness, which, however, does not allow reasoning about termination or total correctness. The right framework for studying the latter is dynamic logic as introduced in [29]. Here, we examine the infrastructure that is needed in order to develop dynamic logic in a monad-independent way, and show that this does indeed make sense when instantiated to the usual monads mentioned above. Using the generic dynamic logic thus obtained, we introduce a monad-independent Hoare calculus for total correctness. We illustrate the use of this Hoare calculus in an example verification over a monad with dynamic references and non-determinism.

The formalism is embedded into the logic of HASCASL, a higher order language for functional specification and programming based on the first order algebraic specification language CASL [2, 23]. The HASCASL internal logic is intuitionistic, so that we have to do with an intuitionistic version of dynamic logic. HASCASL allows one to express programs *and* their expected properties within one and the same language, so that we obtain a unified interpretation of dynamic logic, where formulae consist of programs and logical expressions. The HASCASL logic is basically the internal logic of partial cartesian closed categories (pccc's) with equality, thus slightly more general than topos logic [15].

This work is an extended version of [35]. Related work includes Felleisen's syntactic theory of control and state [6], which is concerned with equational reasoning for programs with the mentioned specific side effects. Moreover, there is a close connection with Pitts' evaluation logic, for which two rather different semantics have been defined by Pitts [26] and Moggi [22], respectively. Pitts introduces a *local* semantics; however, this semantics is given in terms of hyperdoctrines, which constitute additional data beyond the given monad itself. E.g. the state-dependence of formulae in the state monad is explicitly put into the model by choosing a hyperdoctrine where the fibre over each object  $A$  consists of the subobjects of  $A \times S$ , with  $S$  the object of states. Thus, the actual interpretation of the logic is not so much monad-independent as rather delegated away to the model. A similar approach is taken in the more specialized fixed-point logic introduced in [4] (specialized in the sense that it uses a framework where all computations have fixed points, so that intuitionistic implication and existential quantification are unavailable); interestingly, the proof rules for the modal operators given in loc. cit. are rather weaker than expected in that they apply only to stateless computations.

Starting from a criticism of the hyperdoctrinal approach, Moggi has given a semantics defined entirely in terms of strong monads; however, this semantics is *global*

— even for the state monad, there is no state dependency in the validity of formulae (i.e. all formulae are universally quantified over states). Here, we give a purely monadic semantics that retains the local character of the modal operators, thus in a sense reconciling the approaches of Pitts and Moggi; in order to emphasize the local aspect, we use the term ‘dynamic logic’. There is a tradeoff: our approach is axiomatic, and unlike in the case of Moggi’s global semantics, our local semantics cannot be defined for every monad. However, we show that the relevant axioms are satisfied for a large class of monads: we introduce monad-independent notions of *state* and *state discloser* (a program that returns the present state as a value); monads that have a state discloser admit dynamic logic. In fact, most other computationally relevant monads also admit dynamic logic in our sense, with the notable exception of the continuation monad.

The paper is organized as follows: Section 1 gives an overview of the logic of HASCASL, i.e. essentially intuitionistic higher order logic of partial functions. In Section 2, we provide an introduction to the use of monads for encapsulation of side effects. Section 3 introduces generic notions of side-effect freeness and the like, as well as a calculus of ‘global’ modalities similar (but not identical) to evaluation logic with the global semantics of [22]. The core of the paper is Section 4, where we define generic dynamic logic and prove its soundness in monads where it exists. A sufficient existence criterion is then given in Section 5. The generic Hoare calculus for total correctness and the example proof can be found in Sections 6 and 7.

## 1 HASCASL

The language HASCASL has been introduced in [34, 37] as a higher order extension of CASL, based on the partial  $\lambda$ -calculus. We give a brief summary of the logic associated to the partial  $\lambda$ -calculus with equality and its use in basic HASCASL specifications; thanks to the institution-independent mechanisms of CASL, HASCASL then automatically inherits structuring features for specifications and implementations. For more details on both syntax and semantics, see [34, 37].

### 1.1 The Partial $\lambda$ -Calculus

The *partial  $\lambda$ -calculus* as introduced in [19, 20] is a variant of the  $\lambda$ -calculus where  $\lambda$ -terms are thought of as denoting (strict) partial functions, with a correspondingly adapted equational calculus. Of course, function types are then regarded as types of partial functions, denoted  $A \rightarrow? B$  (total function types  $A \rightarrow B$  may be recovered as subtypes; see below.) Importantly, one distinguishes *existential equality* (two terms are defined and equal) and *strong equality* (one term is defined iff the other is, and in this case, both values are equal). The partial  $\lambda$ -calculus automatically comes with a notion of predicates in the shape of partial functions into the unit type 1; in particular, the type  $\Omega = (1 \rightarrow? 1)$  can be regarded as a type of truth values, with  $\top$  being  $\lambda x \bullet *$ , where  $*$  is the unique inhabitant of 1.

Here, we use the partial  $\lambda$ -calculus with (*internal*) *equality*; i.e. one has a predicate representing existential equality, so that equations can be  $\lambda$ -abstracted. The internal equality allows *defining* the full set of logical operations and quantifiers of intuitionistic

logic along the lines of [15]; this is carried out in detail in [34] (e.g.,  $\forall x.\phi$  is coded as the equation  $(\lambda x \bullet \phi) = \lambda x \bullet \top$ ). One thus obtains an intuitionistic *internal logic*.

In [31], it is shown that partial  $\lambda$ -theories with equality are equivalent to *partial cartesian closed categories (pccc's) with equality*, which may be concisely defined as cartesian closed categories with representable extremal partial morphisms (i.e. partial morphisms whose domain is an extremal subobject) [1]. In particular, the partial  $\lambda$ -calculus with equality, i.e. intuitionistic higher order logic of partial functions, is the internal logic of pccc's with equality. One should note that this logic is weaker than topos logic in that it need not satisfy the unique choice axiom (cf. [31] for details); in fact, extended by the unique choice axiom, it becomes precisely the logic of toposes (which may incidentally be defined as cartesian closed categories with representable partial morphisms). Every quasitopos is a pccc with equality; a typical example is the category of pseudotopological spaces [1]. The extremal partial morphisms of a pccc with equality form a category, called the induced *category of partial morphisms*. It should be noted that the notion of pccc *without* equality [19, 31] is vastly more general than that of pccc with equality. It includes in particular typical categories of domains and even plain cartesian closed categories as a special case and moreover does not in general admit the interpretation of higher order logic; the latter is possible only in the presence of equality.

The following features of the internal logic are used in the development below: the symbol  $=$  is used for the strong version of internal equality (expressed by existential equalities with definedness guards) — confusion with external equality is not a problem, since internal and external equality coincide wherever both may be used, i.e. outside the scope of  $\lambda$ -terms. Projections of binary products are denoted by  $\pi_1$  and  $\pi_2$ , respectively. The projections allow coding *conditional terms*: if  $\phi : \Omega$  is a formula and  $\alpha$  is a (partial) term, then  $\alpha \text{ res } \phi$  is a term that is defined, and then equal to  $\alpha$ , iff  $\alpha$  is defined and  $\phi$  holds — i.e.  $\alpha \text{ res } \phi = \pi_1(\alpha, \phi^*)$ .

As laid out in [31], the internal logic of pccc's with equality supports subtype comprehension, i.e. types of the form  $\{x : A \mid \phi\}$ , where  $\phi$  is a formula; such types always denote extremal subobjects. A word of caution is in order: we shall sometimes use polymorphic formulae, i.e. formulae that are implicitly universally quantified over type variables. These are to be understood as infinite conjunctions. Subtypes, however, can in general be defined only by means of monomorphic formulae, not by infinite conjunctions. Defining subtypes by polymorphic formulae becomes possible in pccc's with equality that have arbitrary intersections of extremal subobjects (since extremal and strong monomorphisms coincide in pccc's with equality, and strong monomorphisms are always stable under intersections, such intersections are automatically extremal). 'Real' pccc's tend to satisfy this property — it is implied e.g. by completeness, since pccc's with internal equality are always extremally well-powered. Nevertheless, this completeness assumption will be made only for the purposes of the existence theorem for dynamic logic (Section 5); it is not needed for the main line of development.

There are two notions of model for partial  $\lambda$ -theories, shown to be equivalent in [31, 33]:

- *intensional Henkin models* that interpret types and terms by sets and functions, respectively, where however a function type need not contain all set-theoretic functions, and two functions that yield the same value on every input need not be equal.

- models in pccc’s as originally defined by Moggi [19].

**Convention 1** We will use the internal logic, i.e. intuitionistic HOL, in an informal way in the development below. In particular, when we speak of *elements* of types, we mean *abstract* elements as introduced by quantifiers in the internal logic rather than elements in Henkin models or closed terms — e.g. a phrase such as ‘there exists an element  $a$  such that  $\phi$ ’ is to be construed as the formula  $\exists a. \phi$  in the internal logic.

## 1.2 The Syntax of HASCASL

A HASCASL specification is essentially a convenient way to determine the signature and axioms of a partial  $\lambda$ -theory. The only actual additional language feature is that HASCASL has shallow type class polymorphism, which however is semantically coded out by collections of instances (w.r.t. the generic framework of CASL structured and architectural specifications, this has the effect that one obtains a so-called rps pre-institution rather than an institution; however, this is not relevant for the results presented here).

The constituents of a HASCASL signature are *classes*, *basic types* and *operators*. Basic types are introduced by means of the keyword **type**. Types may be parameterized by type arguments; e.g., we may write

```
var   a : Type
type List a
```

and obtain a unary type constructor *List*. There are built-in type constructors (with fixed interpretations)  $\_ \times \_$  for product types,  $\_ \rightarrow? \_$  and  $\_ \rightarrow \_$  for partial and total function types, respectively, *Pred* for predicate types, and a unit type *Unit*. A *type* is, then, anything that can be formed from the basic types and the existing type constructors.

Next, an *operator* is a constant  $f$  of some type  $t$ . The type  $t$  may contain type variables, making  $f$  an ML-style polymorphic operator. An operator is declared by

```
op   f : t
```

From the given operators, we may form higher order terms in the usual way: a term is either a variable, an application, a tuple, or a (multi-argument)  $\lambda$ -abstraction. Such terms may then be used in *axioms* which may be formulated in either an external logic, which is not relevant for the purposes of this paper, or in the internal logic described in the preceding sections; use of the internal logic is indicated by the keyword **internal**. Axioms may be explicitly or implicitly universally quantified over type variables at the outermost level.

Classes are declared in the form

```
class C
```

and are to be understood as subsets of the syntactical universe of all types. Types as well as type variables can be restricted to belong to an assigned class, e.g. by writing

```
type t : C
```

In particular, axioms and operators may be polymorphic over classes. Classes may be subclasses of each other, and they may have generic instances. By attaching polymorphic operators and axioms to a class, one achieves a similar effect as with Haskell’s type classes.

More generally, one also has *constructor classes*, i.e. classes of polymorphic types such as *List*. They are interpreted as predicates on the syntactical universe of ab-

stracted type expressions (also called *pseudotypes*), e.g.

$$\lambda a : \text{Type} \bullet a \rightarrow? \text{List } a$$

Constructor classes may have subclasses; types, operators, and axioms may be polymorphic over constructor classes. A typical example of a constructor class is the class of monads (see Figure 1 for a specification of a slightly adapted class).

The semantics of a HASCASL specification is then given by a translation into a partial  $\lambda$ -theory, where polymorphism of types, operators, and axioms is coded out by means of collections of instances; for definiteness, the chosen notion of model is that of intensional Henkin model — however, we will more often think of models as living in suitable pccc's.

By means of the internal logic, one can specify a class of complete partial orders and fixed point recursion in much the same style as in HOLCF [30]. On top of this, syntactical sugar is provided that allows recursive function definitions in the style used in functional programming, indicated by the keyword **program**.

In summary, HASCASL is a language that allows both property-oriented specification and functional programming; executable HASCASL specifications may easily be translated into Haskell programs. As to the generality of the results obtained, one should keep in mind that HASCASL is just syntactical sugar for a standard intuitionistic higher order logic of partial functions (the internal logic of pccc's with equality), so that the results of the paper carry over to any similarly powerful framework. We list the key requirements explicitly:

- Types of (strict) partial functions, including a type  $\Omega$  of truth values
- $\lambda$ -abstraction for partial functions
- $\lambda$ -abstractable formulae in a full intuitionistic logic with equality and existence
- Comprehension for subtypes

In particular, our results hold over any topos (even quasitopos). The total Hoare calculus moreover makes use of a notion of complete partial order, which is however defined *within* the basic logic and thus does not constitute a further requirement on the framework.

## 2 Monads for computations

On the basis of the seminal paper [21], monads are being used for encapsulating side effects in modern functional programming languages; in particular, this idea is one of the central concepts of Haskell [25]:

*Principle of encapsulation of side effects:* If you need a certain type of side-effects, turn it into a monad. The abstract study of side effects is then the development of monad-independent notions, notations, and reasoning.

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  $A \rightarrow TB$ . This approach abstracts away from

particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can in fact be carried out independently of the choice of such a notion.

More formally, a monad on a given category  $\mathbf{C}$  can be defined as a *Kleisli triple*  $\mathbb{T} = (T, \eta, \_*)$ , where  $T : \text{Ob } \mathbf{C} \rightarrow \text{Ob } \mathbf{C}$  is a function, the *unit*  $\eta$  is a family of morphisms  $\eta_A : A \rightarrow TA$ , and  $\_*$  assigns to each morphism  $f : A \rightarrow TB$  a morphism  $f^* : TA \rightarrow TB$  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 via an endofunctor with unit and multiplication [17]. A monad gives rise to a *Kleisli category* over  $\mathbf{C}$ , which has the same objects as  $\mathbf{C}$  and ‘functions with side effects’  $f : A \rightarrow TB$  as morphisms from  $A$  to  $B$ ; the composite of two such functions  $g$  and  $f$  is just  $g^* f$ .

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

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

called *strength*, subject to certain coherence conditions (see e.g. [21]); this is equivalent to enrichment of the monad over  $\mathbf{C}$  (see discussion and references in [21]).

Figure 1 shows a specification of monads in HASCASL. As an example of an instance for this type class, a specification of the state monad is shown in Figure 2. The notation is (almost) identical to the one used in Haskell, i.e. the unit is denoted by *ret*, and the operator  $\_ \gg= \_$  denotes, in the above notation, the function  $(x, f) \mapsto f^*(x)$ . Since the operations of the monad are functions in the model, the monads thus specified are automatically strong:

**Proposition 2** Every monad over a pccc with equality in the sense of Figure 1 is strong.

**Proof** Using the internal language of a pccc, we can define the strength as

$$t_{A,B} := \lambda(a, m) : A \times TB \bullet m \gg= (\lambda b : B \bullet \text{ret}(a, b)). \quad \square$$

The specification of monads is the basis for a built-in sugaring in the form of a Haskell-style do-notation: for monadic expressions  $e_1$  and  $e_2$ ,

$$\text{do } x \leftarrow e_1; e_2$$

abbreviates  $e_1 \gg= \lambda x \bullet e_2$ . Further details will be discussed below.

A slight complication concerning the axiomatization arises from the fact that partial functions are involved, i.e. the binding operator needs to bind computations to partial functions. Note that the second unit law  $f^* \eta = f$  has been replaced by two axioms, one stating that the said equation holds on the domain of  $f$ , and another one stating that  $f^* \eta$ , while possibly having a larger domain of definition than  $f$ , behaves like  $f$  under binding. This ensures that standard monads such as the state monad with its usual definition (under which  $f^*$ , and hence  $f^* \eta$ , are actually always total

functions; cf. Figure 2 and the recent discussion on [11]) are really subsumed, while leaving the essence of the proposed calculus untouched. Moreover, for the sake of simplicity of the further treatment, we have included the mono requirement (stating that  $ret$  is injective) in the specification. Finally, observe that the binding operator  $\gg=$  has an additional profile for total functions. The effect of this second profile is essentially the introduction of an axiom stating that binding a computation to a total function is always defined, thus ensuring that the given monad actually restricts to a monad on the category of total functions.

```

spec MONAD = INTERNALLOGIC then
  class Monad : Type → Type {
  vars T : Monad; A, B, C : Type
  ops  _ >>= _ : T A → (A →? T B) →? T B;
        _ >>= _ : T A → (A → T B) → T B;
        ret : A → T A
  }
  internal {
  forall x, z : A; y : T A; f : A →? T B; g : B →? T C; a : A →? B
    • def (f x) ⇒ ((ret x) >>= f) = f x
    • y >>= (λx : A • ret (a x) >>= g) = y >>= (λx : A • g (a x))
    • (y >>= ret) = y
    • ((y >>= f) >>= g) = (y >>= (λx : A • f x >>= g))
    • ret x = ret z ⇒ x = z
  }

```

Figure 1: The constructor class of monads

**Example 3** Computationally relevant monads on **Set** (since strength is equivalent to enrichment, all monads on **Set** are strong) are listed in [21]. Some of the examples have to be adjusted in order to fit the definition of monad given in Figure 1, i.e. in order to accommodate binding for partial functions. Mostly, this amounts to replacing total by partial function types, denoted here as in Figure 1 by  $\_ \rightarrow? \_$ ; one thus obtains the same monads as actually used in Haskell, since function types in programming languages are always implicitly partial. Typical effects are

- stateful computations with possible non-termination:  $TA = (S \rightarrow? (A \times S))$ , where  $S$  is a fixed set of states (cf. Figure 2);
- (finite) non-determinism:  $TA = \mathcal{P}_{fin}(A)$ , where  $\mathcal{P}_{fin}$  denotes the finite power set;
- exceptions:  $TA = A + E$ , where  $E$  is a fixed set of exceptions;
- interactive input:  $TA$  is the smallest fixed point of  $\gamma \mapsto A + (U \rightarrow? \gamma)$ , where  $U$  is an alphabet of input tokens; the computational interpretation is that  $p : TA$  may either return a value or read an input token and then continue in the same manner, depending on the token received.

- non-deterministic stateful computations:  $TA = (S \rightarrow \mathcal{P}_{fn}(A \times S))$ , 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);
- continuations:  $TA = (A \rightarrow? R) \rightarrow? R$ , where  $R$  is a type of *results*. This is equivalent to putting  $TA = (A \rightarrow \tilde{R}) \rightarrow \tilde{R}$ , with  $\tilde{R} = 1 \rightarrow? R$  (e.g. on **Set**,  $\tilde{R}$  is just  $R$  extended by an error value). We shall thus continue to think of continuation monads as being of the form  $TA = (A \rightarrow R) \rightarrow R$ , keeping in mind that  $R$  needs to be of the form  $\tilde{S}$  for some  $S$ .

All these monads make sense in an arbitrary topos with natural numbers object (this is a sufficiently strong setting for recursive datatypes [18]), i.e. essentially in an ambient intuitionistic higher order type theory; cf. [15]. (In a topos, the required partial function types do indeed exist; see for example [3].)

Other typical examples of monads are the list monad, where  $TA$  is the type of lists over  $A$ , and the free abelian group monad, where  $TA$  consists of expressions of the form  $\sum_{i=1}^m n_i a_i$ , with  $n_i \in \mathbb{Z}$  and  $a_i \in A$  for  $i = 1, \dots, m$ . Generally, monads with rank over **Set** can be presented by signatures and equations [12]; we will call monads with such a presentation *algebraic monads*.

We illustrate the axiomatics of Figure 1 for the case of the state monad (all other examples work similarly): for  $p : A \rightarrow? (S \rightarrow? (B \times S))$  and  $a : A$ , we have  $x \leftarrow \text{ret } a; px = \lambda s : S \bullet pa s$ . The latter is equal to  $pa$  only if  $pa$  is defined; however, the two terms have the same behaviour under binding: in general, partial (!) terms  $p_1, p_2 : S \rightarrow? (B \times S)$  behave identically under binding if the strong equation  $p_1 s = p_2 s$  holds for all  $s$ .

<pre> <b>spec</b> STATE = MONAD <b>then</b>   <b>type instance</b> <math>ST : Monad</math>   <b>vars</b> <math>A, B : Type</math>   <b>internal</b> {   <b>types</b> <math>S</math>;     <math>ST\ A := S \rightarrow? (A \times S)</math>   <b>forall</b> <math>x : A; y : ST\ A; f : A \rightarrow? ST\ B</math>     {     • <math>\text{ret } x = \lambda s : S \bullet (x, s)</math>     • <math>(y \gg= f) = \lambda s1 : S \bullet \text{let } (z, s2) = y\ s1 \text{ in } f\ z\ s2</math>     } </pre>
---

Figure 2: Specification of the state monad

**Remark 4** Categorically, the notion of monad as defined in Figure 1 deviates in some respects from what might be expected: the fact that  $f^*\eta$  may differ from  $f$  for partial maps means that the concept defined coincides neither with that of monads on the associated category of partial functions (cf. Section 1), nor with that of a cartesian monad on the category of total functions (e.g. the state monad fails to be cartesian). Instead, we work with monads on the total category which extend to the partial category, but with the extension being only a ‘weak’ monad in the sense that the unit law holds only in the weak form explained above (note that, on the other

hand, the unit is required to be total!). It is a typical phenomenon that categorical definitions change shape when transferred to categories of partial maps; thus, the notion of ‘partial monad’ as used here deserves independent investigation.

Reasoning about a category equipped with a strong monad is greatly facilitated by the fact that proofs can be conducted in a *meta-language* introduced in [21], which we here adapt to deal with partial functions. Although we do not a priori work in a category (this would require working out the details of how the specification of monads induces a monad on the classifying category as constructed in [31, 32]), the meta-language is still applicable here, simply because its logic can be obtained from the axioms in Figure 1. The crucial features of this language are

- A type constructor  $T$ ;  $TA$  is the type of  $A$ -valued *programs* or *computations*;
- a polymorphic operator  $\text{ret} : A \rightarrow TA$  corresponding to the unit;
- a binding construct, which we here denote in Haskell’s do style instead of by  $\text{let}$ : terms of the form

$$\text{do } x \leftarrow p; q$$

are interpreted by means of the tensorial strength and Kleisli composition (See [21] for details. This is equivalent the do-notation introduced above.) Intuitively,  $\text{do } x \leftarrow p; q$  first computes  $p$ , whose result is assigned to the bound variable  $x$ , and then computes  $q$  (which may depend on  $x$ ). Binding is *associative*, i.e. one has

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

if  $r$  does not contain  $x$ . We denote nested do expressions like  $\text{do } x \leftarrow p; \text{do } y \leftarrow q; \dots$  by  $\text{do } x \leftarrow p; y \leftarrow q; \dots$ . Repeated nestings such as  $\text{do } x_1 \leftarrow p_1, \dots, x_n \leftarrow p_n; q$  are somewhat inaccurately denoted in the form  $\text{do } \bar{x} \leftarrow \bar{p}; q$ . Term fragments of the form  $\bar{x} \leftarrow \bar{p}$  are called *program sequences*. Variables  $x_i$  that do not appear later on may be omitted from the notation. Terms that differ only in the names of bound variables are equal ( $\alpha$ -equivalence). The scope of a binding extends as far as possible. Binding is strict in its first argument, i.e. definedness of  $\text{do } x \leftarrow p; q$  implies definedness of  $p$ . Due to the second profile of the binding operator  $\gg=$ , which states that binding is total on total functions, the converse implication holds when  $\lambda x \bullet q$  is total.

- besides the associative law mentioned above, *unit laws* stating that

$$\begin{aligned} (\text{do } x \leftarrow y; \text{ret } x) &= y, \\ (\text{do } x \leftarrow \text{ret } a; p) &= p[a/x], && \text{if } a \text{ and } p[a/x] \text{ are defined, and} \\ (\text{do } x \leftarrow p; y \leftarrow \text{ret } a; q) &= \text{do } x \leftarrow p; q[a/y], && \text{if } a \text{ is defined.} \end{aligned}$$

The second of these laws implies in particular that

$$(\text{do } \text{ret } a; p) = p, \quad \text{if } p \text{ is defined.}$$

Terms are generally formed in a context of variables with assigned types; however, we will omit contexts from the notation. Much of the logical syntax introduced below will be defined in terms of program sequences. In all such cases, a program sequence  $\bar{x} \leftarrow \bar{p}$  can be regarded as syntactical sugar for the program  $\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \bar{x}$ . Avoiding this form of ‘result packaging’ helps making formulations clearer — see for instance the sequencing axioms in Figure 5. However, the idea of packaging is sometimes useful in proofs, e.g. in the following lemma, which we note for later reference:

**Lemma 5** If  $\bar{x} \leftarrow \bar{p}$  and  $\bar{x} \leftarrow \bar{q}$  are program sequences such that

$$(\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \bar{x}) = \text{do } \bar{x} \leftarrow \bar{q}; \text{ret } \bar{x},$$

then

$$(\text{do } \bar{x} \leftarrow \bar{p}; r) = \text{do } \bar{x} \leftarrow \bar{q}; r,$$

for every program  $r$ .

**Proof** We can assume that  $\bar{x}$  is not the empty sequence. Let  $\pi_i$  denote the  $i$ -th projection operator  $\lambda \bar{x} \bullet x_i$ , and let  $\sigma$  be the substitution that replaces  $x_i$  by  $\pi_i z$  for all  $i$ . Then we have

$$\begin{aligned} (\text{do } \bar{x} \leftarrow \bar{p}; r) &= \text{do } \bar{x} \leftarrow \bar{p}; z \leftarrow \text{ret } \bar{x}; r\sigma \\ &= \text{do } z \leftarrow (\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \bar{x}); r\sigma \\ &= \text{do } z \leftarrow (\text{do } \bar{x} \leftarrow \bar{q}; \text{ret } \bar{x}); r\sigma \\ &= \dots = \text{do } \bar{x} \leftarrow \bar{q}; r, \end{aligned}$$

where the first step uses the third unit law. □

On top of a monad, one can generically define control structures such as a while loop. Such definitions require general recursion, which is realized in HASCASL by means of fixed point recursion on  $\text{cpos}$ . Thus, one has to restrict to monads that allow lifting a  $\text{cpo}$  structure on  $A$  to a  $\text{cpo}$  structure on the type  $TA$  of computations in such a way that the monad operations become continuous. This is laid out in detail in [36].

**Remark 6** Filinski has shown that any  $\lambda$ -definable monad  $\mathbb{T}$  can be simulated in the continuation monad, using an appropriate continuation-style reformulation of the definition of  $\mathbb{T}$  [7]. Such an approach is interesting for programming languages like ML that can be considered to live within one fixed monad that is left implicit and captures all effects. It is not clear at present whether it is possible to define a continuation-style generic computational logic analogous to the monadic-style generic logic developed here (the latter does not apply to the continuation monad; cf. Example 49). From the point of view of computational logic the monadic approach has precisely the advantage that it avoids forcing everything into the powerful framework of continuations, but rather encapsulates effects and thus in particular limits their nature and scope.

### 3 The generic approach to side effects

We now discuss monad-independent notions of program properties such as side-effect freeness and determinism, as well as logical aspects of boolean-valued programs with effects. The notions of deterministic side-effect freeness and validity of formulae with effects have been introduced in [36]; the former notion is related to the concepts of discardability and copyability defined in [39] (see also [8, 9]). We will phrase all definitions in terms of the meta-language for monads, using the fixed notation  $\mathbb{T}$  for the monad,  $T$  for the associated type constructor etc. throughout.

**Definition 7** A program  $p$  is called *discardable* [39] if

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

where  $*$  is the unique element of the unit type.

(Discardable morphisms have been called *total* in [7].) Note that discardable programs are necessarily defined, due to strictness of the binding operator in the first argument.

**Example 8** A program  $p$  is discardable

- in the state monad iff  $p$  terminates and does not change the state;
- in the non-determinism monad iff  $p$  always has at least one possible outcome;
- in the exception monad iff  $p$  terminates normally;
- in the interactive input monad iff  $p$  never reads any input;
- in the non-deterministic state monad iff  $p$  does not change the state and always has at least one possible outcome (i.e. never gets stuck);
- in the continuation monad iff  $p : (A \rightarrow R) \rightarrow R$  maps constant continuations  $\lambda a \bullet r$  to their constant result  $r$ .

**Definition 9** A program  $p$  is called *stateless* if it factors through  $\text{ret}$ , i.e. if it is just a value inserted into the monad.

For example, in the state monad, statelessness means that the program neither changes nor reads the state ( $p$  is stateless iff  $p$  *exists* in the sense of [21]). Stateless programs are discardable, but not vice versa. However, discardable programs of type 1 *are* stateless:

**Lemma 10** A program  $p : T1$  is discardable iff  $p = \text{ret } *$ .

**Proof** Of course,  $\text{ret } *$  is discardable. Conversely, for  $p : T1$  we can always write  $p = \text{do } y \leftarrow p; \text{ret } y = \text{do } y \leftarrow p; \text{ret } *$ , and the latter term equals  $\text{ret } *$  if  $p$  is discardable.  $\square$

**Lemma 11** If  $p$  is discardable, then

$$(\text{do } p; q) = q$$

for each program  $q$ , provided that  $q$  (as a term) is defined. Moreover, if  $p, q : T1$ , then

$$(\text{do } q; p) = q.$$

**Proof** By the second unit law (cf. the preceding section),  $(\text{do } \text{ret } *; q) = q$ , so that

$$\begin{aligned} \text{do } p; q &= \text{do } p; \text{ret } *; q \\ &= \text{do } \text{ret } *; q \\ &= q, \end{aligned}$$

which proves the first claim. To prove the second claim, let  $z$  be a fresh variable of type 1. Then  $z = *$  and hence  $p = \text{ret } z$  by Lemma 10. Thus,

$$(\text{do } q; p) = (\text{do } z \leftarrow q; p) = (\text{do } z \leftarrow q; \text{ret } z) = q. \quad \square$$

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’  $\boxplus$ :

**Definition 12** Given a term  $\varphi$  of type  $T\Omega$ , where  $\Omega$  denotes the type of internal truth values,  $\boxplus\varphi$  abbreviates

$$\varphi = \text{do } \varphi; \text{ret } \top,$$

read as a formula of the internal logic.

If  $\varphi$  is discardable, then  $\boxplus\varphi$  simplifies to  $\varphi = \text{ret } \top$ ; otherwise, the formula above ensures that the right hand side has the same side-effect as  $\varphi$ .

**Remark 13** Note that the equality symbol  $=$  in the definition of the formula  $\boxplus\varphi$  above is *strong equality* (cf. Section 1.1). In particular, in the classical case  $\boxplus\varphi$  is true if  $\varphi$  is undefined.

**Example 14** In the monads of Example 3, satisfaction of  $\boxplus\varphi$  amounts to the following:

- in the state monad: successful execution of  $\varphi$  from any initial state yields  $\top$ ;
- in the non-determinism monad:  $\varphi$  yields at most the value  $\top$  (or none at all)
- in the exception monad:  $\varphi$  yields  $\top$  whenever it terminates normally.
- in the interactive input monad: the value eventually produced by  $\varphi$  after some combination of inputs is always  $\top$ ;
- in the non-deterministic state monad: execution of  $\varphi$  from any initial state yields at most the value  $\top$ ;
- in the continuation monad (over **Set**): for  $k : \Omega \rightarrow R$ ,  $\varphi k$  depends only on  $k \top$ .

In order to perform proofs about the logic introduced below, we require an auxiliary calculus for *global dynamic judgements* of the form  $[\bar{x} \leftarrow \bar{p}] \phi$ , which intuitively state that  $\phi$  holds after  $\bar{x} \leftarrow \bar{p}$ , where  $\phi$  is an actual formula of the internal logic (i.e.  $\phi : \Omega$ ), and  $\lambda \bar{x} \bullet \phi$  is a total function. 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.

$$\begin{array}{c}
(\wedge\mathbf{I}) \frac{[\bar{x} \leftarrow \bar{p}] \phi}{[\bar{x} \leftarrow \bar{p}] (\phi \wedge \xi)} \quad (\mathbf{wk}) \frac{\forall \bar{x}. \phi \Rightarrow \xi}{[\bar{x} \leftarrow \bar{p}] \phi} \quad (\mathbf{eq}) \frac{[\bar{x} \leftarrow \bar{p}] q_1 = q_2}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}] \phi} \\
(\mathbf{app}) \frac{[\bar{x} \leftarrow \bar{p}] \phi}{y \notin FV(\phi)} \quad (\mathbf{dis}_0) \frac{[\bar{x} \leftarrow \bar{p}; q] \phi}{q \text{ discardable}} \quad (\mathbf{pre}) \frac{\forall x. [\bar{y} \leftarrow \bar{q}] \phi}{[x \leftarrow p; \bar{y} \leftarrow \bar{q}] \phi} \\
(\eta) \frac{[\bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a; \bar{z} \leftarrow \bar{q}] \phi}{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{q}[a/y]] \phi[a/y]} \quad (\mathbf{ctr}) \frac{[\dots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}] \phi}{x \notin FV(\phi) \cup FV(\bar{r})}}{[\dots; y \leftarrow (\text{do } x \leftarrow p; q); \bar{z} \leftarrow \bar{r}] \phi}
\end{array}$$

Figure 3: Proof calculus for global dynamic judgements

**Definition 15** Given a program sequence  $\bar{x} \leftarrow \bar{p}$  and a formula  $\phi$  of type  $\Omega$ , the notation  $[\bar{x} \leftarrow \bar{p}] \phi$  abbreviates (slightly deviating from [36])

$$(\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \phi)) = \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \top)$$

(again, a strong equation). The degenerate case  $[\ ] \phi$  is (by the mono requirement) equivalent to  $\phi$ ; we shall in fact silently identify the two formulae.

Note that  $[\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] \phi$  is *not* equivalent to (more precisely, properly weaker than)  $[\bar{x} \leftarrow \bar{p}] [\bar{y} \leftarrow \bar{q}] \phi$ . We follow the convention that the equality sign binds stronger than modalities like  $[x \leftarrow p]$ , which in turn bind stronger than logical connectives. E.g.  $[x \leftarrow p] a = b$  means  $[x \leftarrow p] (a = b)$ , and  $[x \leftarrow p] \phi_1 \wedge \phi_2$  means  $([x \leftarrow p] \phi_1) \wedge \phi_2$ .

**Remark 16** The intuition behind this definition is the same as for  $\boxtimes$ . Indeed, in many cases,  $[\bar{x} \leftarrow \bar{p}] \phi$  is semantically equivalent to  $\boxtimes \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \phi$ ; monads for which this equivalence holds will be called *simple*. In an algebraic monad, we think of  $p : TA$  as being a term over the ‘supply of variables’  $A$ , taken modulo the relevant equality; then,  $[x \leftarrow p] \phi$  says that  $p$  depends only on those variables  $a$  that satisfy  $\phi[a/x]$ . Contrastingly,  $\boxtimes \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \phi$  means that substituting every variable  $a$  in  $p$  by  $\phi[a/x]$  yields, modulo equations, the same result as substituting with  $\top$ . An algebraic monad is simple if, in each of its equations, the two sides contain the same variables; this covers e.g. the exception monad, the non-determinism monad, and the list monad. Moreover, the usual state monads are simple, although their known equational presentation [27] does not satisfy the variable requirement.

In general, however,  $\boxtimes \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \phi$  is properly weaker than  $[\bar{x} \leftarrow \bar{p}] \phi$  (it follows from  $[\bar{x} \leftarrow \bar{p}] \phi$  by Lemma 19 below). E.g., in the free abelian group monad, one has  $\boxtimes \text{do } x \leftarrow (a - b); \text{ret } \perp$ , but not  $[x \leftarrow (a - b)] \perp$ . Similarly, the continuation monad fails to be simple: over  $\mathbf{Set}$ ,  $[x \leftarrow p] \phi$  means that, for  $k : A \rightarrow R$ ,  $pk$  depends

only on the values of  $k$  at those  $a$  for which  $\phi[a/x]$  holds, while  $\boxtimes \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } \phi$  implies this only for  $k$  of the form  $\lambda a : A \bullet l(\phi[a/x])$ , with  $l : \Omega \rightarrow R$ .

**Remark 17** The meaning of global dynamic judgements is close to the global semantics given by Moggi [22] to the modal operators of evaluation logic (but not at all to the semantics as defined in [26]). Compare, however, the above remark to Theorem 2.12 in [22].

**Remark 18** The global dynamic judgements  $[\bar{x} \leftarrow \bar{p}] \phi$  should not be confused with the similar-looking formulae  $[\bar{x} \leftarrow \bar{p}] \varphi$  of the dynamic logic that we introduce in Section 4 below. As the name suggests,  $[\bar{x} \leftarrow \bar{p}] \phi$  has a (rather simple) *global* semantics, and indeed, one of the central challenges of this work is to equip formulae  $[\bar{x} \leftarrow \bar{p}] \varphi$  with a *local* semantics.

We collect a few facts concerning global dynamic judgements:

**Lemma 19** If  $[\bar{x} \leftarrow \bar{p}] \phi$ , then

$$(\text{do } \bar{x} \leftarrow \bar{p}; q[\phi/y]) = \text{do } \bar{x} \leftarrow \bar{p}; q[\top/y]$$

for each program  $q$  in a context that includes  $y : \Omega$ .

**Proof** We can assume that  $\bar{x} \leftarrow \bar{p}$  is not the empty program sequence. Then the claim is equivalent to

$$(\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } \phi; q) = \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } \top; q$$

by the third unit law, and thus follows from the assumption  $[\bar{x} \leftarrow \bar{p}] \phi$  by Lemma 5.  $\square$

**Lemma 20** For  $\varphi : T\Omega$ ,  $[a \leftarrow \varphi] a$  is equivalent to  $\boxtimes \varphi$ .

**Proof** The left-to-right implication follows from Lemma 19, since  $\varphi = \text{do } a \leftarrow \varphi; \text{ret } a$  by the first unit law. Conversely, assume  $\boxtimes \varphi$ . Then we have

$$\begin{aligned} (\text{do } a \leftarrow \varphi; \text{ret } (a, a)) &= \text{do } a \leftarrow \varphi; b \leftarrow \text{ret } a; \text{ret } (b, a) \\ &= \text{do } a \leftarrow (\text{do } \varphi; \text{ret } \top); b \leftarrow \text{ret } a; \text{ret } (b, a) \\ &= \text{do } \varphi; a \leftarrow \text{ret } \top; b \leftarrow \text{ret } a; \text{ret } (b, a) \\ &= \text{do } \varphi; b \leftarrow \text{ret } \top; \text{ret } (b, \top) \\ &= \text{do } b \leftarrow (\text{do } \varphi; \text{ret } \top); \text{ret } (b, \top) \\ &= \text{do } b \leftarrow \varphi; \text{ret } (b, \top), \end{aligned}$$

i.e.  $[a \leftarrow \varphi] a$ .  $\square$

**Lemma 21** If  $[\bar{x} \leftarrow \bar{p}] q_1 = q_2$  for  $q_1, q_2 : B$ , then

$$(\text{do } \bar{x} \leftarrow \bar{p}; r[q_1/y]) = \text{do } \bar{x} \leftarrow \bar{p}; r[q_2/y]$$

for each program  $r$  in a context that includes  $y : B$ .

**Proof** Using Lemma 19, we have

$$\begin{aligned}
(\text{do } \bar{x} \leftarrow \bar{p}; r[q_1/y]) &= \text{do } \bar{x} \leftarrow \bar{p}; r[(q_1 \text{ res } \top)/y] \\
&= \text{do } \bar{x} \leftarrow \bar{p}; r[(q_1 \text{ res } (q_1 = q_2))/y] \\
&= \text{do } \bar{x} \leftarrow \bar{p}; r[(q_2 \text{ res } (q_1 = q_2))/y] \\
&= \dots = \text{do } \bar{x} \leftarrow \bar{p}; r[q_2/y]
\end{aligned}$$

(cf. Section 1 for an explanation of the restriction operator res).  $\square$

A converse of the preceding lemma holds for stateless terms:

**Lemma 22** Let  $\bar{x} \leftarrow \bar{p}$  be a program sequence, and let  $a_1, a_2 : A$ . Then

$$(\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, a_1)) = \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, a_2) \quad (*)$$

implies  $[\bar{x} \leftarrow \bar{p}] a_1 = a_2$ .

Note that, for  $\bar{p}$  consisting of only one program  $p : T1$ , Condition  $(*)$  simplifies to

$$(\text{do } p; \text{ret } a_1) = \text{do } p; \text{ret } a_2.$$

**Proof** We can assume that  $\bar{x} \leftarrow \bar{p}$  is non-empty. Then the assumption implies

$$(\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a_1; \text{ret } (\bar{x}, y)) = \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a_2; \text{ret } (\bar{x}, y).$$

By Lemma 5, we obtain

$$(\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a_1; \text{ret } (\bar{x}, y = a_2)) = (\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a_2; \text{ret } (\bar{x}, y = a_2)).$$

The claim then follows by the third unit law.  $\square$

Figure 3 shows a proof calculus for global dynamic judgements. For the time being, this calculus should be regarded as a convenient way of stating a collection of lemmas; we are not committing ourselves to reasoning about global dynamic judgements solely by means of the given rules. Double lines indicate that a rule works in both directions. The rules are meant to be read in natural deduction style, with one caveat attached to the first premise of rule (wk): this premise should be regarded, similarly as in the weakening rule of the usual Hoare calculus, as a side condition to be established outside the calculus. The set of free variables of  $p$  is denoted by  $FV(p)$ . The rules (pre) and (wk) use explicit quantification to enforce the usual variable condition stating that certain variables do not occur freely in assumptions; a quantified global dynamic judgement such as the premise of (pre) is proved in the ambient higher order logic by deriving the enclosed judgement, observing the variable condition. The calculus is sound:

**Theorem 23** If a global dynamic judgement  $\phi$  is deducible from global dynamic judgements  $\xi_1, \dots, \xi_n$  by the rules of Figure 3, then  $(\bigwedge \xi_i) \Rightarrow \phi$  holds.

**Proof** We prove the rules as lemmas in the internal logic:

$(\wedge I)$ : Apply Lemma 19 to  $\phi$  and  $\xi$  in the term

$$\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \phi \wedge \xi).$$

(*wk*): The first premise implies that  $\phi \wedge \xi$  and  $\phi$  are equal truth values. Using this, we obtain by Lemma 19

$$\begin{aligned} (\text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \xi)) &= \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \top \wedge \xi) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \phi \wedge \xi) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \phi) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; \text{ret } (\bar{x}, \top), \end{aligned}$$

thus proving the conclusion.

(*eq*): By Lemma 21, we have

$$\begin{aligned} &\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}; \text{ret } (\bar{x}, y, \bar{z}, \phi) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}; \text{ret } (\bar{x}, y, \bar{z}, \phi) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}; \text{ret } (\bar{x}, y, \bar{z}, \top) \\ &= \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}; \text{ret } (\bar{x}, y, \bar{z}, \top), \end{aligned}$$

using the second premise in the second step.

(*app*): By Lemma 19, we have

$$(\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q; \text{ret } (\bar{x}, y, \phi)) = \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow q; \text{ret } (\bar{x}, y, \top),$$

which is just the conclusion (the side condition  $y \notin FV(\phi)$  is needed in order to avoid capture of  $y$  in the substitution according to the lemma).

(*dis*<sub>0</sub>): The conclusion is equivalent to

$$\text{do } \bar{x} \leftarrow \bar{p}; q; \text{ret } (\bar{x}, \phi) = \text{do } \bar{x} \leftarrow \bar{p}; q; \text{ret } (\bar{x}, \top)$$

by Lemma 11 and thus follows from the premise by Lemma 19.

(*pre*): By Lemma 19, the premise implies

$$(\lambda x \bullet \text{do } \bar{y} \leftarrow \bar{q}; \text{ret } (x, \bar{y}, \phi)) = \lambda x \bullet \text{do } \bar{y} \leftarrow \bar{q}; \text{ret } (x, \bar{y}, \top).$$

By substituting this equation for  $z$  in the term  $\text{do } x \leftarrow p; z x$ , we obtain

$$(\text{do } x \leftarrow p; \bar{y} \leftarrow \bar{q}; \text{ret } (x, \bar{y}, \phi)) = \text{do } x \leftarrow p; \bar{y} \leftarrow \bar{q}; \text{ret } (x, \bar{y}, \top),$$

i.e. the conclusion.

(*ctr*): By Lemma 19 and associativity of binding.

( *$\eta$* ): We first do the case that  $\bar{x} \leftarrow \bar{p}$  is non-empty. By the third unit law, we have, for any formula  $\xi$  (in particular for  $\phi$  and  $\top$ ),

$$\begin{aligned} (\text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a; \bar{z} \leftarrow \bar{q}; \text{ret } (\bar{x}, y, \bar{z}, \xi)) &= \\ &\text{do } \bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{q}[a/y]; \text{ret } (\bar{x}, a, \bar{z}, \xi[a/y]). \end{aligned}$$

By Lemma 19, this implies soundness of the upwards direction of the rule. The converse direction follows from

$$(\text{do } \bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{q}[a/y]; \text{ret } (\bar{x}, \bar{z}, \xi[a/y])) = \text{do } \bar{x} \leftarrow \bar{p}; y \leftarrow \text{ret } a; \bar{z} \leftarrow \bar{q}; \text{ret } (\bar{x}, \bar{z}, \xi),$$

$$\boxed{
\begin{array}{c}
\text{(tau)} \frac{\forall \bar{x}. \phi}{[\bar{x} \leftarrow \bar{p}] \phi} \quad \text{(rp)} \frac{\forall \bar{x}. q_1 = q_2 \quad [\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}] \phi}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}] \phi} \quad \text{(dis)} \frac{[\bar{x} \leftarrow \bar{p}; q; \bar{z} \leftarrow \bar{r}] \phi \quad q \text{ discardable}}{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi}
\end{array}
}$$

Figure 4: Derived rules for global dynamic judgements

again by Lemma 19.

The remaining case, namely that  $\bar{x} \leftarrow \bar{p}$  is the empty program sequence, is reduced to the non-empty case by observing that

$$[\bar{y} \leftarrow \bar{r}] \xi \iff [\text{ret } *; \bar{y} \leftarrow \bar{r}] \xi \quad (*)$$

for all program sequences  $\bar{y} \leftarrow \bar{r}$  and all  $\xi$ . To prove (\*), first note that we can assume that  $\bar{y} \leftarrow \bar{r}$  is non-empty — otherwise, both sides of (\*) are equivalent to  $\xi$  (concerning the right hand side, recall that binding is total on total functions). The left-to-right implication is just rule (pre). To prove the converse implication, we have to deduce the *strong* equation

$$(\text{do } \bar{y} \leftarrow \bar{r}; \text{ret } (\bar{y}, \xi)) = \text{do } \bar{y} \leftarrow \bar{r}; \text{ret } (\bar{y}, \top). \quad (**)$$

By strictness of binding in the first argument, definedness of one of the sides of (\*\*) implies definedness of  $r_1$ ; hence,  $(\text{do } \text{ret } *; r_1) = r_1$  by the second unit law. The right-to-left implication of (\*) is then an instance of rule (ctr).  $\square$

We will refer to use of the rules ( $\wedge$ I) and (wk) as *propositional reasoning*. A few derived rules for global dynamic judgements are shown in Figure 4.

**Proposition 24** The rules of Figure 4 are derivable from the rules of Figure 3.

**Proof** (*tau*): Since  $\phi$  is by convention identified with  $\square \phi$ , this rule is derivable by repeated application of (pre).

(*rp*): By (tau), we obtain  $[\bar{x} \leftarrow \bar{p}] q_1 = q_2$ ; the conclusion then follows by (eq).

(*dis*): By (dis<sub>0</sub>) if  $\bar{z} \leftarrow \bar{r}$  is empty; otherwise by (ctr), Lemma 11, and (rp).  $\square$

**Proposition 25** In simple monads (Remark 16), the converse rule of (ctr)

$$\text{(ctr-)} \frac{[\dots; y \leftarrow (\text{do } x \leftarrow p; q); \bar{z} \leftarrow \bar{r}] \phi}{[\dots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}] \phi}$$

is sound, hence also the converse rule of (dis)

$$\text{(dis-)} \frac{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi \quad q \text{ discardable}}{[\bar{x} \leftarrow \bar{p}; q; \bar{z} \leftarrow \bar{r}] \phi}.$$

$\square$

We recall the notion of copyability from [9, 39]:

**Definition 26** A program  $p$  is called *copyable* if

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

For discardable programs, copyability amounts to determinism:

**Proposition 27** A discardable program  $p$  is copyable iff

$$[x \leftarrow p; y \leftarrow p] x = y.$$

**Proof** By discardability of  $p$ , the right-hand side of Definition 26 equals  $\text{do } x \leftarrow p; y \leftarrow p; \text{ret } (x, x)$ . Thus, the equality in Definition 26 and the formula displayed above are equivalent by Lemmas 21 and 22.  $\square$

**Lemma 28** Let  $r$  be a program, and let  $p$  be copyable. Then

$$(\text{do } x \leftarrow p; y \leftarrow p; r) = \text{do } x \leftarrow p; r[x/y]$$

**Proof** Analogous to Lemma 5.  $\square$

It is not the case that copyable programs are stable under sequential composition [9]; however, we have

**Lemma 29** Let  $p$  be copyable; then  $q := \text{do } x \leftarrow p; \text{ret } a$  is copyable.

**Proof** We have

$$\begin{aligned} & (\text{do } u \leftarrow q; v \leftarrow q; \text{ret } (u, v)) \\ &= \text{do } x \leftarrow p; u \leftarrow \text{ret } a; y \leftarrow q; v \leftarrow \text{ret } a[y/x]; \text{ret } (u, v) \\ &= \text{do } x \leftarrow p; y \leftarrow p; \text{ret } (a, a[y/x]) && \text{(unit law)} \\ &= \text{do } x \leftarrow p; \text{ret } (a, a) && \text{(Lemma 28)} \\ &= \text{do } x \leftarrow p; u \leftarrow \text{ret } a; \text{ret } (u, u) && \text{(unit law)} \\ &= \text{do } u \leftarrow q; \text{ret } (u, u). \end{aligned}$$

$\square$

**Proposition and Definition 30** Let  $p, q$  be copyable and discardable programs with  $y \notin FV(p)$ ,  $x \notin FV(q)$ . The following are equivalent:

- (i) The program  $\text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, y)$  is copyable.
- (ii)  $(\text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, y)) = \text{do } y \leftarrow q; x \leftarrow p; \text{ret } (x, y)$ .
- (iii) For each program  $r$  (possibly containing  $x$  and  $y$ ),

$$(\text{do } x \leftarrow p; y \leftarrow q; r) = \text{do } y \leftarrow q; x \leftarrow p; r.$$

- (iv)  $[x \leftarrow p; y \leftarrow q; z \leftarrow p] x = z$ .

In this case we say that  $p$  *commutes* with  $q$ .

(Note that by Condition (ii), the relation ‘commutes with’ is symmetric.)

**Proof** Pick fresh variables  $u, v, w, z$ , put  $s = \text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, y)$ , and let  $\pi_1$  and  $\pi_2$  denote the first and second projections.

(i)  $\Rightarrow$  (ii):  $s$  is discardable and, by (i), copyable. Thus,

$$\begin{aligned}
s &= \text{do } z \leftarrow s; \text{ret } z && \text{(unit law)} \\
&= \text{do } w \leftarrow s; z \leftarrow s; \text{ret } z && \text{(Lemma 11)} \\
&= \text{do } w \leftarrow s; z \leftarrow s; \text{ret } (\pi_1(z), \pi_2(z)) \\
&= \text{do } w \leftarrow s; z \leftarrow s; \text{ret } (\pi_1(z), \pi_2(w)) && \text{(Lemma 28)} \\
&= \text{do } u \leftarrow p; v \leftarrow q; w \leftarrow \text{ret } (u, v); x \leftarrow p; y \leftarrow q; z \leftarrow \text{ret } (x, y); \text{ret } (\pi_1(z), \pi_2(w)) \\
&= \text{do } u \leftarrow p; v \leftarrow q; x \leftarrow p; y \leftarrow q; \text{ret } (x, v) && \text{(unit law)} \\
&= \text{do } v \leftarrow q; x \leftarrow p; \text{ret } (x, v) && \text{(Lemma 11 for } p, q),
\end{aligned}$$

and the last term is equal to the right hand side of the claim by  $\alpha$ -equivalence.

(ii)  $\Rightarrow$  (iii): Immediate by Lemma 5.

(iii)  $\Rightarrow$  (i): Using (iii) in the second step and Lemma 28 for  $p$  and  $q$  in the third, we have

$$\begin{aligned}
\text{do } w \leftarrow s; z \leftarrow s; \text{ret } (w, z) &= \text{do } u \leftarrow p; v \leftarrow q; x \leftarrow p; y \leftarrow q; \text{ret } ((u, v), (x, y)) \\
&= \text{do } u \leftarrow p; x \leftarrow p; v \leftarrow q; y \leftarrow q; \text{ret } ((u, v), (x, y)) \\
&= \text{do } u \leftarrow p; v \leftarrow q; \text{ret } ((u, v), (u, v)) \\
&= \text{do } w \leftarrow s; \text{ret } (w, w).
\end{aligned}$$

(iii)  $\Rightarrow$  (iv): By copyability of  $p$ , Proposition 27 and rule (app), we have

$$[x \leftarrow p; z \leftarrow p; y \leftarrow q] x = z.$$

By (iii), this is equivalent to (iv).

(iv)  $\Rightarrow$  (iii): Using discardability and Lemma 21, we have

$$\begin{aligned}
(\text{do } x \leftarrow p; y \leftarrow q; r) &= \text{do } x \leftarrow p; y \leftarrow q; z \leftarrow p; r \\
&= \text{do } x \leftarrow p; y \leftarrow q; z \leftarrow p; r[z/x] \\
&= \text{do } y \leftarrow q; z \leftarrow p; r[z/x],
\end{aligned}$$

and the last term is  $\alpha$ -equivalent to the right hand side of (iii).  $\square$

**Lemma 31** Let  $p, q$  be copyable and discardable programs with  $y \notin FV(p)$ ,  $x \notin FV(q)$ , and put  $\varphi = (\text{do } v \leftarrow q; \text{ret } (x = v)) : T\Omega$ . Then

- (i)  $\varphi$  is discardable and copyable for each  $x$ ;
- (ii) if  $p$  commutes with  $\varphi$  for all  $x$ , then  $p$  commutes with  $q$ .

**Proof** (i): Discardability is clear; copyability holds by Lemma 29.

(ii): We establish Condition (iv) of Proposition 30, i.e. (exploiting symmetry)

$$[x \leftarrow q; y \leftarrow p; z \leftarrow q] x = z,$$

by a direct calculation (using Lemma 28 for  $q$ ):

$$\begin{aligned}
& \text{do } x \leftarrow q; y \leftarrow p; z \leftarrow q; \text{ret } (x, y, z, x = z) \\
&= \text{do } x \leftarrow q; y \leftarrow p; z \leftarrow q; u \leftarrow q; \text{ret } (x, y, u, x = z) && (q \text{ copyable}) \\
&= \text{do } x \leftarrow q; y \leftarrow p; a \leftarrow (\text{do } z \leftarrow q; \text{ret } (x = z)); u \leftarrow q; \text{ret } (x, y, u, a) && (\text{unit law}) \\
&= \text{do } x \leftarrow q; a \leftarrow (\text{do } z \leftarrow q; \text{ret } (x = z)); y \leftarrow p; u \leftarrow q; \text{ret } (x, y, u, a) && (\text{assumption}) \\
&= \text{do } x \leftarrow q; z \leftarrow q; y \leftarrow p; u \leftarrow q; \text{ret } (x, y, u, x = z) && (\text{unit law}) \\
&= \text{do } x \leftarrow q; z \leftarrow q; y \leftarrow p; u \leftarrow q; \text{ret } (x, y, u, \top) && (q \text{ copyable}) \\
&= \text{do } x \leftarrow q; y \leftarrow p; u \leftarrow q; \text{ret } (x, y, u, \top) && (q \text{ discard.}) \\
&= \text{do } x \leftarrow q; y \leftarrow p; z \leftarrow q; \text{ret } (x, y, z, \top) && (\alpha\text{-equiv.}).
\end{aligned}$$

□

**Corollary and Definition 32** Let  $p$  be copyable and discardable. Then  $p$  commutes with all discardable and copyable programs iff  $p$  commutes with all discardable and copyable  $\Omega$ -valued programs. In this case,  $p$  is called *deterministically side-effect free* (dsef). The subtype of  $TA$  formed by the deterministically side-effect free computations will be denoted by  $DA$  throughout.

**Remark 33** In order to be able to define  $DA$  as a subtype of  $TA$  without further assumptions, it is essential that the fact that  $p$  is dsef can be expressed by a single formula in the internal logic (namely, the second of the two equivalent conditions in Corollary 32, as opposed to the first which is implicitly an infinite conjunction, taken over all types).

Stateless programs are deterministically side-effect free. In most of the running examples, all discardable programs are deterministically side-effect free, with the unsurprising exception of the monads where non-determinism is involved. In these cases, a discardable program is deterministically side-effect free iff it is deterministic. Another exception is the continuation monad; see Example 37 below.

**Proposition 34** In simple monads (Remark 16), each copyable and discardable program is already dsef.

**Proof** Let  $p$  and  $q$  be copyable and discardable with  $x \notin FV(q)$ ,  $y \notin FV(p)$ . From copyability of  $p$ , we obtain by Proposition 25

$$[x \leftarrow p; y \leftarrow q; z \leftarrow p] x = z,$$

i.e.  $p$  commutes with  $q$ . □

**Proposition 35 (Structural rules)** Let  $\phi : \Omega$  be a formula, and let  $p$  and  $q$  be dsef. Then

- (i)  $[\dots; x \leftarrow p; y \leftarrow p; \bar{z} \leftarrow \bar{r}] \phi \iff [\dots; x \leftarrow p; \bar{z} \leftarrow \bar{r}[x/y]] \phi[x/y]$ ;
- (ii)  $[\dots; x \leftarrow p; y \leftarrow q; \dots] \phi \iff [\dots; y \leftarrow q; x \leftarrow p; \dots] \phi$ ;
- (iii)  $[\dots; x \leftarrow p] \phi \iff [\dots] \phi$ ; (if  $x \notin FV(\phi)$ ).

**Proof** The contraction and interchange rules (i) and (ii) follow immediately from Lemma 28 and Proposition 30. The weakening rule (iii) (which of course works for any discardable program and is mentioned here only for the sake of completeness) follows from rules (dis) and (app) of Figure 3.  $\square$

Moreover, we have context weakening for dsef programs not only at the end of program sequences, but also at the beginning:

**Proposition 36** Let  $\phi : \Omega$  be a formula and let  $\bar{z} \leftarrow \bar{r}$  be a program sequence with  $y \notin FV(\phi) \cup FV(\bar{r})$ , let  $\bar{x} \leftarrow \bar{p}$  be a sequence of dsef programs, and let  $q$  be dsef. Then

$$[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi \iff [\bar{x} \leftarrow \bar{p}; y \leftarrow q; \bar{z} \leftarrow \bar{r}] \phi.$$

**Proof** The right-to-left implication holds by rule (dis) of Figure 4. Conversely, from  $[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi$  we obtain  $[y \leftarrow q; \bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi$  by rule (pre) (this would work even under the weaker assumption  $\forall y. [\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi$  instead of  $y \notin FV(\phi)$ ), which implies the right-hand side by the interchange rule (Proposition 35 (ii)).  $\square$

The following example shows that discardability and copyability do not already imply deterministic side-effect freeness:

**Example 37** Let  $T$  be the continuation monad with result type  $R$  on **Set**. Since  $\Omega$  is two-valued,  $p : T\Omega = (\Omega \rightarrow R) \rightarrow R$  can be thought of as a binary operation on  $R$ ; we shall write  $p(x, y)$  for  $pk$  with  $k\perp = x$ ,  $k\top = y$ . In this notation, do  $x \leftarrow p; y \leftarrow q; \text{ret}(x, y)$  is, for  $p, q : T\Omega$  with  $x \notin FV(q)$ , the quaternary operation

$$\lambda x, y, z, w \bullet p(q(x, y), q(z, w)), \quad (*)$$

with  $x, y, z, w$  representing in this order the values  $l(\perp, \perp)$ ,  $l(\perp, \top)$ ,  $l(\top, \perp)$ ,  $l(\top, \top)$  of a continuation  $l : \Omega \times \Omega \rightarrow R$ . Then

- (i)  $p$  is discardable iff  $p(x, x) = x$  for all  $x$ ;
- (ii)  $p$  is copyable iff  $p(p(x, y), p(z, w)) = p(x, w)$  for all  $x, y, z, w$ ;
- (iii)  $p$  is discardable and copyable iff  $p$  is of the form

$$p(x, y) = (\pi_1(x), \pi_2(y)),$$

where the projections  $\pi_1, \pi_2$  and the pairing operation are w.r.t. a decomposition  $R \cong R_1 \times R_2$ ;

- (iv)  $p$  is dsef iff  $p$  is stateless, i.e. iff  $p$  is one of the two projections.

In particular, all discardable and copyable programs in  $T$  are dsef iff all decompositions  $R \cong R_1 \times R_2$  are trivial, i.e. iff  $R$  is finite of prime cardinality.

**Proof** (i): Immediate from Example 8.

(ii): Immediate from (\*).

(iii): By (i) and (ii), it is clear that  $p$  of the given form is discardable and copyable. Conversely, observe that product decompositions  $R \cong R_1 \times R_2$  are, modulo isomorphism, in bijective correspondence with pairs  $(\rho_1, \rho_2)$  of equivalence relations

on  $R$  such that  $\rho_1 \cap \rho_2$  is equality and for every pair of elements  $x, y \in R$ , there exists a (necessarily unique)  $z$  such that  $x\rho_1z$  and  $y\rho_2z$ . Now given  $p$  satisfying the equations of (i) and (ii), we define relations  $\rho_1, \rho_2$  by

$$\begin{aligned} x\rho_1y &\iff p(x, y) = y \\ x\rho_2y &\iff p(x, y) = x. \end{aligned}$$

Then  $\rho_1$  and  $\rho_2$  are equivalence relations: reflexivity holds by (i). If  $x\rho_1y$ , then

$$p(y, x) = p(p(x, y), p(x, x)) \stackrel{(ii)}{=} p(x, x) = x,$$

i.e.  $y\rho_1x$ . If  $x\rho_1y$  (hence also  $y\rho_1x$ ) and  $y\rho_1z$ , then

$$p(x, z) = p(p(y, x), z) = p(y, z) = z,$$

i.e.  $x\rho_1z$ . Symmetry and transitivity of  $\rho_2$  are shown analogously. It is clear that  $\rho_1 \cap \rho_2$  is equality. Finally, given  $x, y \in R$ ,  $z = p(x, y)$  satisfies  $x\rho_1z$  and  $y\rho_2z$ . This proves also that the decomposition  $R \cong (R/\rho_1) \times (R/\rho_2)$  indeed induces  $p$ .

(iv): Let  $p$  be dsef. We have to show that the associated decomposition  $R \cong R_1 \times R_2$  according to (iii) is trivial. Assume the contrary; then there exist  $x, y$  such that  $x, y, p(x, y)$ , and  $p(y, x)$  are pairwise distinct (put  $x = (u_1, v_1)$ ,  $y = (u_2, v_2)$  for distinct elements  $u_1, u_2 \in R_1$ ,  $v_1, v_2 \in R_2$ ). For any bijection  $\sigma : R \rightarrow R$ ,  $q$  defined by  $q(x, y) = \sigma^{-1}p(\sigma x, \sigma y)$  is, by (i) and (ii), discardable and copyable. Thus,  $p$  commutes with  $q$ , i.e.

$$p(q(x, y), q(z, w)) = q(p(x, z), p(y, w)) \quad (**)$$

for all  $x, y, z, w$ . Now take  $\sigma$  to be the 3-cycle  $[x, y, p(x, y)]$  and  $z = w = x$ . Then the left hand side of (\*\*) evaluates to  $p(\sigma^{-1}p(\sigma x, \sigma y), \sigma^{-1}p(\sigma x, \sigma x)) = p(\sigma^{-1}p(y, p(x, y)), x) = p(\sigma^{-1}y, x) = p(x, x) = x$ , and the right hand side is  $\sigma^{-1}p(\sigma p(x, x), \sigma p(y, x)) = \sigma^{-1}p(\sigma x, p(y, x)) = \sigma^{-1}p(y, p(y, x)) = \sigma^{-1}p(y, x) = p(y, x)$ , a contradiction.  $\square$

It is straightforward but tedious to generalize the above characterizations to arbitrary sets  $A$  in place of  $\Omega$ . One obtains in particular that discardable and copyable programs  $p : TA$  are in bijective correspondence with product decompositions of  $R$  into  $A$  factors, and that all dsef programs  $p : TA$  are stateless. The proofs are, modulo notation, essentially the same as above; we briefly sketch the required modifications: for  $a \in A$  and  $x, y \in R$ , let  $k_{x,y}^a : A \rightarrow R$  denote the continuation that maps  $b \in A$  to  $y$  if  $a = b$ , and  $x$  otherwise, and let  $k_x : A \rightarrow R$  be the constant map with value  $x$ . Then the family of relations  $(\rho_a)_{a \in A}$  determining the product decomposition associated to  $p$  as in (iii) is defined by  $x\rho_a y \iff pk_{x,y}^a = x$ . In the proof of (iv), one picks  $x, y \in R$  as before, exploiting  $R \cong \prod_{c \in A} R_c$  with  $R_a, R_b$  nontrivial and noting that  $\{x, y, pk_{x,y}^b\}$  is a three-element set containing neither  $pk_{x,y}^a$  nor  $pk_{y,x}^b$ . Then commutation of  $p$  with  $q = \lambda l. \sigma^{-1}p(\sigma \circ l)$  is applied for  $\sigma = [x, y, pk_{x,y}^b]$ ; the special case  $p(k_{qk_x, qk_{x,y}^b}^a) = q(k_{pk_x, pk_{x,y}^a}^b)$  of the commutation equation then yields the contradiction  $x = pk_{y,x}^b$ .

For dsef terms, we always have a generalized form of simplicity (cf. Remark 16).

**Lemma 38** Let  $\bar{y} \leftarrow \bar{q}$  be a sequence of dsef programs, and let  $\bar{x} \leftarrow \bar{p}$  be a program sequence. Then  $[\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] \phi$  iff  $[\bar{x} \leftarrow \bar{p}; a \leftarrow (\text{do } \bar{y} \leftarrow \bar{q}; \text{ret } \phi)] a$ .

(By Lemma 20, specializing to the case that  $\bar{x} \leftarrow \bar{p}$  is empty yields the equivalence of  $[\bar{y} \leftarrow \bar{q}] \phi$  with  $\boxtimes (\text{do } \bar{y} \leftarrow \bar{q}; \text{ret } \phi)$ .)

**Proof** The ‘only if’ direction holds universally due to Lemma 19. We prove the ‘if’ direction by a direct calculation. By rule ( $\eta$ ) of Figure 3, which allows insertion of  $\text{ret } *$ , we can assume that neither  $\bar{x} \leftarrow \bar{p}$  nor  $\bar{y} \leftarrow \bar{q}$  are empty sequences. By packaging (cf. Section 2), we can assume that both sequences consist of a single binding  $x \leftarrow p$  and  $y \leftarrow q$ , respectively. Let  $\pi_1$  and  $\pi_2$  denote the first and second projections. Then

$$\begin{aligned}
& (\text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, y, \phi)) \\
&= \text{do } x \leftarrow p; y \leftarrow q; z \leftarrow q; \text{ret } (x, z, \phi) && (q \text{ copyable}) \\
&= \text{do } x \leftarrow p; y \leftarrow q; a \leftarrow \text{ret } (x, \phi); z \leftarrow q; \text{ret } (\pi_1(a), z, \pi_2(a)) && (\text{unit law}) \\
&= \text{do } a \leftarrow (\text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, \phi)); z \leftarrow q; \text{ret } (\pi_1(a), z, \pi_2(a)) && (\text{associativity}) \\
&= \text{do } a \leftarrow (\text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, \top)); z \leftarrow q; \text{ret } (\pi_1(a), z, \pi_2(a)) && (\text{assumption}) \\
&= \dots = \text{do } x \leftarrow p; y \leftarrow q; \text{ret } (x, y, \top)
\end{aligned}$$

□

Because of their properties (Lemma 11, Lemma 28, Proposition 30, Proposition 35, Lemma 38), deterministically side-effect free subterms of programs can be handled notationally in a more relaxed way:

**Convention 39** We allow deterministically side effect free programs of type  $DA$  to occur in places where a term of type  $A$  is expected in programs and formulae. More precisely, if  $\bar{x} = (x_1, \dots, x_n)$  is a list of variables of types  $A_1, \dots, A_n$  and  $q$  is a program, then the program  $q[\bar{p}/\bar{x}]$  obtained by substituting terms  $p_i : DA_i$  for the  $x_i$  is defined as  $\text{do } \bar{x} \leftarrow \bar{p}; q$ , with well-definedness guaranteed by Lemma 11, Lemma 28, and Proposition 30. Similarly,  $[\bar{y} \leftarrow \bar{q}] \phi[\bar{p}/\bar{x}]$  abbreviates  $[\bar{y} \leftarrow \bar{q}; \bar{x} \leftarrow \bar{p}] \phi$ , with well-definedness guaranteed by Proposition 35 and Lemma 38 (the latter ensures that the two parts of the convention, for terms and for global dynamic judgements, do not lead to actual ambiguities). Note that this includes the case that  $\bar{y} \leftarrow \bar{q}$  is the empty sequence. Thus, since a further convention identifies  $\boxtimes \phi$  with  $\phi$ , e.g.  $\phi \Rightarrow \psi$  abbreviates  $[a \leftarrow \phi; b \leftarrow \psi] (a \Rightarrow b)$  for  $\phi, \psi : D\Omega$ . Ambiguities may arise from polymorphic predicates and operations such as equality, e.g. in the equation  $p = q$ , with  $p, q : DA$ . In such cases, we will disambiguate formulae by explicit type annotations where necessary; e.g.,  $p =_A q$  abbreviates  $[x \leftarrow p; y \leftarrow q] x = y$ , while  $p =_{DA} q$  is just equality of computations. Concerning the calculus of Figure 3, a single warning is required: rule ( $\text{app}$ ) is only valid if the formula  $\phi$  is really stateless.

A further abstraction concerns the *termination* of programs, based on the observation that a program  $p$  *fails* to terminate if  $[p] \perp$ .

**Definition 40** A program  $p$  *terminates* if

$$[\bar{x} \leftarrow \bar{q}; p] \phi \text{ implies } [\bar{x} \leftarrow \bar{q}] \phi$$

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

Note that both the value and the state resulting from the execution of  $p$  are ignored here: the truth value of the stateless formula  $\phi$  only depends on  $\bar{x}$ . Intuitively, the implication captures the fact that properties that necessarily hold after execution of  $p$  (but do not actually refer to  $p$ ) hold non-vacuously.

**Example 41** In the non-determinism monad,  $p$  terminates iff  $p \neq \emptyset$ . In the state monad,  $p : S \rightarrow? (A \times S)$  terminates iff  $p$  is total, and in the non-deterministic state monad,  $p : S \rightarrow \mathcal{P}_{fin}(A \times S)$  terminates iff  $p(s) \neq \emptyset$  for all  $s$ . (Thus, termination of a non-deterministic program means that there exists an execution sequence that terminates, not that *all* execution sequences terminate; cf. the discussion in [14]. In a denotational setting, this is up to a certain point unavoidable since e.g. non-deterministic choice between an infinite loop and a terminating program  $p$  is semantically indistinguishable from  $p$ . A solution is provided by the modal  $\mu$ -calculus as described e.g. in [13]; whether or not this is also possible in a general monadic setting is an open point of research.)

Curiously, in the continuation monad with result space  $R$  on **Set**, a program  $p : T1 \cong R \rightarrow R$  terminates iff  $p$  is surjective: by packaging (cf. Section 2), the former means that, whenever for all  $k : A \rightarrow R$ ,  $q(p \circ k)$  depends only on the restriction of  $k$  to  $\{a : A \mid \phi a\}$ , where  $q : TA$ , then so does  $qk$  for all  $k : A \rightarrow R$ . Thus surjectivity of  $k \mapsto p \circ k$  and hence of  $p$  is clearly necessary for termination of  $p$ . Conversely, if  $p$  is surjective, then it has a right inverse  $s$ , so that  $qk = q(p \circ (s \circ k))$ , which proves the required implication.

**Remark 42** The definition of termination given above is not entirely satisfactory in that it asserts soundness of a proof principle which one would prefer to deduce from a more basic definition. Of course, the definition unravels to a conditional equation; in fact, using Lemmas 21 and 22 one shows easily that  $p : TA$  terminates iff, for all  $q : TB$  and all  $a, b : B \rightarrow C$ ,

$$\text{do } x \leftarrow q; y \leftarrow p; \text{ret}(x, y, a x) = \text{do } x \leftarrow q; y \leftarrow p; \text{ret}(x, y, b x)$$

implies

$$\text{do } x \leftarrow q; \text{ret}(x, a x) = \text{do } x \leftarrow q; \text{ret}(x, b x).$$

However, this is obviously not much of an improvement and, on top of that, rather more lengthy to state. At any rate, if  $p$  terminates in the sense of Definition 40, then

$$[q; p] \perp \text{ implies } [q] \perp,$$

which in the classical case precisely captures termination. The intuition behind this is that global *non*-termination means that  $\perp$  holds ‘after’ execution, and that  $p$  terminates (globally) iff global non-termination of  $q; p$  is always the fault of  $q$ . (Note that  $\neg[p] \perp$  is weaker than termination of  $p$ : in the state monad, it just expresses that there is *some* state for which  $p$  terminates. Hence the need for  $q$ , which may lead to any state since it is arbitrary.)

Heuristically, it is clear that termination of  $p$  cannot be captured by an *unconditional* equation between simple do-terms similar to discardability or copyability: any such equation would have to contain  $p$  on both sides to account for possible side-effects of  $p$  besides non-termination, and thus would in the typical monads be trivially satisfied for globally non-terminating programs.

**Lemma 43** All discardable programs terminate.

**Proof** By rule (dis<sub>0</sub>) of Figure 3. □

## 4 A monad-independent dynamic logic

The approach to Hoare logic pursued in [36] was partly driven by the concept of interpreting formulae and programs within one and the same framework, that is, in the HASCASL internal logic, respectively in the meta-language over an arbitrary monad. This required giving a semantics to Hoare triples  $\{\varphi\} p \{\psi\}$ ; for this purpose, a notion of global validity was sufficient. By contrast, formulae of dynamic logic allow a nesting of modal operators of the nature ‘after execution of  $p$ ’ and the usual connectives of first order logic. This means informally that the state is changed according to the effect of  $p$  within the scope of the modal operator, but is ‘restored’ outside that scope. E.g., in a dynamic logic formula such as

$$[p] \varphi \implies [q] \psi,$$

the subformulae  $\varphi$  and  $\psi$  are evaluated in modified states, but  $[p] \varphi$  and  $[q] \psi$  are evaluated in the *same* state.

This means that the semantics of  $[p] \varphi$  must be deterministically side-effect free, although  $p$  may have side-effects that affect  $\varphi$ . Moreover, one will expect that a formula evaluates to a deterministic truth value (although  $p$  may be non-deterministic). Thus, it is reasonable to require that *formulae are interpreted as deterministically side effect-free  $\Omega$ -valued programs*. We state explicitly

**Definition 44** A *formula* (of dynamic logic) is a term  $\varphi : D\Omega$ . The formula  $\varphi$  is (globally) *valid* if  $\boxtimes \varphi$  holds, i.e. if  $\varphi = \text{ret } \top$ ; as usual, asserting a formula amounts to asserting its global validity.

The usual logical connectives  $\implies, \wedge, \vee, \neg$  etc. are then defined by

$$\varphi \implies \psi := (\text{do } a \leftarrow \varphi; b \leftarrow \psi; \text{ret } (a \implies b)) : D\Omega$$

etc.; recall from Section 3 that repetitions and the order of evaluation do not matter for dsef programs — actually, this is the formal reason why formulas of dynamic logic need to be dsef.

**Remark 45** At first sight, there appears to be a conflict between the notation just introduced and Convention 39, according to which e.g.  $\varphi \wedge \psi$  abbreviates also the global dynamic judgement  $[a \leftarrow \varphi; b \leftarrow \psi] (a \wedge b)$ , and indeed  $\varphi : D\Omega$  denotes also  $[a \leftarrow \varphi] a$ . However, by Lemmas 20 and 38, the two readings are compatible whenever both are possible, i.e. at the level of global validity: e.g.,  $\boxtimes \varphi$  iff  $[a \leftarrow \varphi] a$ , and  $\boxtimes (\text{do } a \leftarrow \varphi; b \leftarrow \psi; \text{ret } (a \wedge b))$  iff  $[a \leftarrow \varphi; b \leftarrow \psi] (a \wedge b)$ . Note also that a formula such as  $\varphi \implies \varphi$  decodes both as  $[a \leftarrow \varphi; b \leftarrow \varphi] (a \implies b)$  and as  $[a \leftarrow \varphi] (a \implies a)$ ; in particular,  $\varphi \implies \varphi$  is globally valid, similarly for other tautologies.

The question is now if  $D\Omega$  has enough structure to allow the interpretation of the diamond and box operators  $\langle p \rangle$  and  $[p]$  of dynamic logic. The interpretation can be introduced *axiomatically* in a rather straightforward manner. To begin, observe that  $D\Omega$  is made into a partial order by putting

$$\varphi \leq \psi \quad \text{iff} \quad (\varphi \Rightarrow \psi) \quad (\text{cf. Remark 45})$$

for  $\varphi, \psi : D\Omega$ . (Antisymmetry is proved using Lemma 21, noting that for truth values, equivalence is equality by construction of the internal logic.) The crucial requirement for dynamic logic is, then, the existence of lower and upper deterministically side-effect free approximations for  $\Omega$ -valued program sequences:

**Definition 46** We say that  $\mathbb{T}$  *admits (propositional) dynamic logic* if there exist, for each program sequence  $\bar{y} \leftarrow \bar{q}$  and each formula  $\varphi : D\Omega$ , a formula  $[\bar{y} \leftarrow \bar{q}] \varphi : D\Omega$  such that

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

for each program sequence  $\bar{x} \leftarrow \bar{p}$  containing  $x_i : \Omega$  and, dually, a formula  $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi : D\Omega$  such that

$$[\bar{x} \leftarrow \bar{p}] (\langle \bar{y} \leftarrow \bar{q} \rangle \varphi \Rightarrow x_i) \iff [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow x_i).$$

(Recall the convention that modalities bind stronger than logical connectives.) Since the internal logic is intuitionistic, one cannot simply define  $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$  as  $\neg[\bar{y} \leftarrow \bar{q}] \neg\varphi$ . Of course, the variable  $x_i$  in the definition can be equivalently replaced by any formula  $\psi : D\Omega$  in context  $\bar{x}$ . Note that, in analogy to the monad-independent Hoare logic of [36], we admit program *sequences* inside the modal operators in order to accommodate reasoning about intermediate results; technically, however, a single program would suffice due to the packaging principle discussed in Section 2. The formulae  $[\bar{y} \leftarrow \bar{q}] \varphi$  and, dually,  $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$  are uniquely determined (in contrast to the modalities in [26], whose interpretation is part of the model):

**Lemma 47** If  $\mathbb{T}$  admits dynamic logic, then  $[\bar{y} \leftarrow \bar{q}] \varphi$  is the greatest formula (w.r.t. the ordering introduced above)  $\psi : D\Omega$  such that

$$[a \leftarrow \psi; \bar{y} \leftarrow \bar{q}] (a \Rightarrow \varphi). \quad (*)$$

Dually,  $\langle \bar{y} \leftarrow \bar{q} \rangle \varphi$  is the smallest formula  $\psi$  such that  $[a \leftarrow \psi; \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a)$ .

**Proof** If  $\psi$  satisfies (\*), then the definition of  $[\bar{y} \leftarrow \bar{q}] \varphi$  implies  $[a \leftarrow \psi] (a \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi)$ , i.e.  $\psi \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi$ . That  $[\bar{y} \leftarrow \bar{q}] \varphi$  satisfies (\*) is seen as follows: we have  $[\bar{y} \leftarrow \bar{q}] \varphi \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi$  (cf. Remark 45), i.e.

$$[a \leftarrow [\bar{y} \leftarrow \bar{q}] \varphi] a \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi.$$

By the definition of  $[\bar{y} \leftarrow \bar{q}] \varphi$ , this is equivalent to  $[a \leftarrow [\bar{y} \leftarrow \bar{q}] \varphi; \bar{y} \leftarrow \bar{q}] (a \Rightarrow \varphi)$ .  $\square$

**Lemma 48** For each program sequence  $\bar{x} \leftarrow \bar{p}$  and each formula  $\varphi$ ,  $[\bar{x} \leftarrow \bar{p}] \varphi$  is equivalent to  $\boxtimes [\bar{x} \leftarrow \bar{p}] \varphi$ .

**Proof** The latter formula means that  $[\bar{x} \leftarrow \bar{p}] \varphi$  equals  $\text{ret } \top$ . By Lemma 47, this is equivalent to  $[a \leftarrow \text{ret } \top; \bar{x} \leftarrow \bar{p}] (a \Rightarrow \varphi)$  (because  $\text{ret } \top$  is the top element of  $D\Omega$ ), i.e. by rule  $(\eta)$  to  $[\bar{x} \leftarrow \bar{p}] \varphi$ .  $\square$

**Example 49** Most of our running example monads, with the exception of the continuation monad, admit dynamic logic. In monads without an explicit notion of state, dsef terms tend to be stateless, so that  $[y \leftarrow q] \varphi$  and  $\langle y \leftarrow q \rangle \varphi$  are just truth values. E.g., in the exception monad,  $[y \leftarrow q] \varphi$  is true iff  $q$  either throws an exception or returns a value that satisfies  $\varphi$ , while  $\langle y \leftarrow q \rangle \varphi$  is true iff  $q$  returns a value that satisfies  $\varphi$ . In the nondeterminism monad,  $[y \leftarrow q] \varphi$  holds iff all values in  $q$  satisfy  $\varphi$ , and  $\langle y \leftarrow q \rangle \varphi$  holds iff  $q$  contains a value that satisfies  $\varphi$ . In the interactive input monad,  $[\bar{x} \leftarrow \bar{p}] \phi$  means that the values produced by any given sequence of inputs satisfy  $\phi$ , and  $\langle \bar{x} \leftarrow \bar{p} \rangle \phi$  means that there exists a sequence of inputs that produces values satisfying  $\phi$ .

In the various state monads,  $[y \leftarrow q] \varphi$  and  $\langle y \leftarrow q \rangle \varphi$  depend on the state. E.g., in the non-deterministic state monad,  $[y \leftarrow q] \varphi$  holds in a state  $s$  iff all values obtained by execution of  $q$  from  $s$  satisfy  $\varphi$ ;  $\langle y \leftarrow q \rangle \varphi$  holds in  $s$  iff there is a possible return value of  $q$  that satisfies  $\varphi$ .

Finally, let  $T$  be the continuation monad with result space  $R$  on **Set**, where  $R$  admits a non-trivial decomposition  $R \cong R_1 \times R_2$ , and assume that  $T$  admits dynamic logic. Let  $\varphi : T\Omega$  be the discardable and copyable program induced by the decomposition  $R \cong R_1 \times R_2$ , i.e. in the notation of Example 37  $\varphi(x, y) = (\pi_1(x), \pi_2(y))$ . By Example 37, there are only two candidates for the dsef formula  $[a \leftarrow \varphi] a$ , namely  $\text{ret } \top$  and  $\text{ret } \perp$ . The former would mean that  $[a \leftarrow \varphi] a$  is valid, which by Lemma 48 would imply  $[a \leftarrow \varphi] a$  — i.e.  $\varphi(x, y)$  depends only on  $y$  (cf. Remark 16), which by construction is not the case.

We are left with  $[a \leftarrow \varphi] a = \text{ret } \perp$ . Now let  $r_0 \in R$ , and let  $p : T1 \cong R \rightarrow R$  be given by  $pk = \varphi(r_0, k)$  for  $k : R$ . Then for  $s := (\text{do } p; \varphi) : T\Omega$ , we have (in the notation of Example 37)  $s(x, y) = p(\varphi(x, y)) = \varphi(r_0, \varphi(x, y)) = \varphi(r_0, y)$ . In particular,  $s(x, y)$  depends only on  $y$ , i.e. we have  $[p; a \leftarrow \varphi] a$ . By Definition 46, this would imply  $[p] [a \leftarrow \varphi] a$ , i.e.  $[p] \perp$ , which means that  $pk$  does not depend on  $k$  at all, a contradiction.

In Section 5, we will provide a sufficient criterion for a monad to admit dynamic logic; this criterion covers all of the positive examples just given, except the interactive input monad.

Under a completeness assumption for  $D\Omega$ , the diamond modality can be defined in terms of the box modality:

**Definition 50** We say that  $\mathbb{T}$  is *complete* if  $D\Omega$  is a complete lattice, with infima denoted by  $\bigwedge$ , and that  $\mathbb{T}$  is *stably complete* if, moreover,

$$\forall j. [a \leftarrow \varphi_j; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a) \quad \text{implies} \quad [a \leftarrow \bigwedge_j \varphi_j; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a)$$

for each family of formulae  $\varphi_j : D\Omega$  and each program sequence  $\bar{x} \leftarrow \bar{p}$ .

(Again,  $x_i$  can be equivalently replaced by  $\psi : D\Omega$ .)

**Theorem 51** If  $\mathbb{T}$  admits box operators  $[-]$  in the sense of Definition 46 and is complete, then the following are equivalent:

- (i)  $\mathbb{T}$  admits dynamic logic;
- (ii)  $\mathbb{T}$  is stably complete.

In this case, the diamond operators are defined by

$$\langle \bar{y} \leftarrow \bar{q} \rangle \varphi \equiv \bigwedge_{a:\Omega} [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a.$$

**Proof**  $(i) \Rightarrow (ii)$ : Let  $[a \leftarrow \varphi_j; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a)$  for each  $j$ . By definition of the diamond operator, this is equivalent to

$$[a \leftarrow \varphi_j] (\langle \bar{x} \leftarrow \bar{p} \rangle x_i \Rightarrow a),$$

i.e. according to Convention 39 to  $\langle \bar{x} \leftarrow \bar{p} \rangle x_i \Rightarrow \varphi_j$ . By the definition of infimum, this implies  $\langle \bar{x} \leftarrow \bar{p} \rangle x_i \Rightarrow \bigwedge_j \varphi_j$ , which in turn is equivalent to  $[a \leftarrow \bigwedge_j \varphi_j; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a)$ .

$(ii) \Rightarrow (i)$ : By definition, we have to show

$$[\bar{x} \leftarrow \bar{p}] \left( \left( \bigwedge_{a:\Omega} [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a \right) \Rightarrow x_i \right) \iff [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow x_i); \quad (*)$$

by the definition of the box operator, the right hand side is equivalent to

$$[\bar{x} \leftarrow \bar{p}] [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow x_i). \quad (**)$$

‘ $\Leftarrow$ ’:  $(**)$  implies

$$[\bar{x} \leftarrow \bar{p}] (([\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow x_i) \Rightarrow x_i) \Rightarrow x_i).$$

The left hand side of  $(*)$  follows by propositional reasoning.

‘ $\Rightarrow$ ’: By rule (wk) of Figure 3, it suffices to show that for each  $c : \Omega$ ,

$$\left( \left( \bigwedge_{a:\Omega} [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a \right) \Rightarrow c \right) \Rightarrow [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow c).$$

By the definition of the box operator (and Convention 39), this is equivalent to

$$[b \leftarrow \left( \bigwedge_{a:\Omega} [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a \right); \bar{y} \leftarrow \bar{q}] ((b \Rightarrow c) \Rightarrow (\varphi \Rightarrow c)). \quad (\dagger)$$

By Lemma 47, we have for each  $a : \Omega$

$$[d \leftarrow ([\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a)); \bar{y} \leftarrow \bar{q}] (d \Rightarrow (\varphi \Rightarrow a)),$$

equivalently (replacing  $d \Rightarrow (\varphi \Rightarrow a)$  with  $\varphi \Rightarrow (d \Rightarrow a)$ )

$$[b \leftarrow ([\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a); \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow b).$$

By stable completeness, this implies

$$[b \leftarrow \left( \bigwedge_{a:\Omega} [\bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow a) \Rightarrow a \right); \bar{y} \leftarrow \bar{q}] (\varphi \Rightarrow b),$$

and hence  $(\dagger)$ . □

<b>Rules:</b>	
$\text{(nec)} \quad \frac{\varphi}{[\bar{x} \leftarrow \bar{p}] \varphi} \quad \bar{x} \text{ not free in assumptions}$	$\text{(mp)} \quad \frac{\varphi \Rightarrow \psi; \varphi}{\psi}$
<b>Axioms:</b>	
(K1)	$[\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi) \Rightarrow [\bar{x} \leftarrow \bar{p}] \varphi \Rightarrow [\bar{x} \leftarrow \bar{p}] \psi$
(K2)	$[\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi) \Rightarrow \langle \bar{x} \leftarrow \bar{p} \rangle \varphi \Rightarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi$
(K3 $\square$ )	$\text{ret } \phi \Rightarrow [p] \text{ret } \phi$
(K3 $\diamond$ )	$\langle p \rangle \text{ret } \phi \Rightarrow \text{ret } \phi$
(K4)	$\langle \bar{x} \leftarrow \bar{p} \rangle (\varphi \vee \psi) \Rightarrow (\langle \bar{x} \leftarrow \bar{p} \rangle \varphi \vee \langle \bar{x} \leftarrow \bar{p} \rangle \psi)$
(K5)	$(\langle \bar{x} \leftarrow \bar{p} \rangle \varphi \Rightarrow [\bar{x} \leftarrow \bar{p}] \psi) \Rightarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi)$
(seq $\square$ )	$[\bar{x} \leftarrow \bar{p}; y \leftarrow q] \varphi \iff [\bar{x} \leftarrow \bar{p}] [y \leftarrow q] \varphi$
(seq $\diamond$ )	$\langle \bar{x} \leftarrow \bar{p}; y \leftarrow q \rangle \varphi \iff \langle \bar{x} \leftarrow \bar{p} \rangle \langle y \leftarrow q \rangle \varphi$
(ctr $\square$ )	$[x \leftarrow p; y \leftarrow q] \varphi \implies [y \leftarrow (\text{do } x \leftarrow p; q)] \varphi \quad (x \notin FV(\varphi))$
(ctr $\diamond$ )	$\langle x \leftarrow p; y \leftarrow q \rangle \varphi \implies \langle y \leftarrow (\text{do } x \leftarrow p; q) \rangle \varphi \quad (x \notin FV(\varphi))$
(ret $\square$ )	$[x \leftarrow \text{ret } a] \varphi \iff \varphi[a/x]$
(ret $\diamond$ )	$\langle x \leftarrow \text{ret } a \rangle \varphi \iff \varphi[a/x]$

Figure 5: The generic proof calculus for propositional dynamic logic

**Remark 52** If  $\mathbb{T}$  is simple, then one can base dynamic logic on unparametrized modal operators  $\square$  and  $\diamond$ , where  $\square\varphi$  is, in the presentation given here, equivalent to  $[a \leftarrow \varphi] a$  for  $\varphi : T\Omega$ , correspondingly for  $\diamond$ . In this case, one can define  $[\bar{y} \leftarrow \bar{q}] \varphi$  as  $\square \text{do } \bar{y} \leftarrow \bar{q}; \varphi$ , dually for  $\langle \bar{y} \leftarrow \bar{q} \rangle$ .

**Convention 53** Actual truth values, i.e. terms of type  $\Omega$ , appearing in dynamic logic formulae are implicitly cast to  $D\Omega$  via  $\text{ret}$ ; it is easy to see that this is compatible with the interpretation of the logical connectives and hence does not lead to confusion.

Figure 5 shows a Hilbert style generic proof calculus for dynamic logic. That is, there are only two rules (necessitation and modus ponens), while the remainder of the calculus is given in the form of axioms (more precisely, axiom schemes). The necessitation rule is subject to the variable condition on  $\bar{x}$ , i.e.  $\bar{x}$  must not occur freely in assumptions — equivalently, we could have used the premise  $\forall \bar{x}. \varphi$ . The first five axioms are the standard axioms of the  $K$ -fragment of intuitionistic modal logic as given e.g. in [28, 38], with a slight variation of the third axiom — the usual form  $\neg\langle p \rangle \perp$  is a special case of the second of the two dual forms given. All intuitionistic propositional tautologies are implicitly included here. Moreover, there are four axioms concerning sequential composition of program sequences.

As in the case of the generic Hoare rules of [36], axioms that deal with monad-specific program constructs are necessarily missing here; typical examples include the nondeterministic choice and iteration constructs of standard dynamic logic. Such specific formulae come in via the specifications of particular monads; in fact, they often

turn out to be a good choice for actual axioms of the specification. Some examples are given in Section 6 below, where also axioms dealing with generic ‘user-defined’ control structures such as *if-then-else* or *while* are discussed.

The proof calculus is sound:

**Theorem 54** Let  $\mathbb{T}$  admit dynamic logic. If a formula  $\psi$  can be deduced from formulae  $\varphi_1, \dots, \varphi_n$  by means of the rules of Figure 5, then validity of the  $\varphi_i$  implies validity of  $\psi$ , i.e.  $\bigwedge \boxplus \varphi_i \implies \boxplus \psi$ .

Completeness of the calculus is the subject of further research. Any completeness result to be obtained will have to be based on a more restrictive definition of the logic than the rather generous definition given above (e.g. there are no axioms or rules to handle quantifiers, although technically nothing prevents the latter from appearing in formulae provided they can be expressed in the monad at hand). A good starting point would be a logic that admits only atomic programs, sequential composition of programs, the return operator, atomic predicates (but no equality), logical connectors (but no quantifiers), and the modal operators, with axioms restraining the atomic programs formulated also in this restricted logic. The relevant models are monads that admit dynamic logic, together with interpretations of the atomic programs and predicates.

**Proof** We have to show that the rules preserve validity, and that the axioms are valid. Validity of the intuitionistic tautologies follows from Remark 45.

*Rule (nec):* Assume  $\boxplus \varphi$ ; we have to show  $\boxplus [\bar{x} \leftarrow \bar{p}] \varphi$ . By substitution of the assumption, the latter formula reduces to  $\boxplus [\bar{x} \leftarrow \bar{p}] \top$ . Thus, we have to show  $\top \Rightarrow [\bar{x} \leftarrow \bar{p}] \top$ , which by the definition of  $[\bar{x} \leftarrow \bar{p}]$  amounts to proving the trivial formula  $[\bar{x} \leftarrow \bar{p}] (\top \Rightarrow \top)$ .

*Rule (mp):* By Remark 45, this can be proved by propositional reasoning.

*Axiom (K1):* By the definition of  $[\bar{x} \leftarrow \bar{p}] \psi$ , we have to show

$$[a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); b \leftarrow [\bar{x} \leftarrow \bar{p}] \varphi; \bar{x} \leftarrow \bar{p}] ((a \wedge b) \Rightarrow \psi).$$

By the definitions of  $[\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi)$  and  $[\bar{x} \leftarrow \bar{p}] \varphi$ , we have

$$\begin{aligned} [a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); \bar{x} \leftarrow \bar{p}] (a \Rightarrow (\varphi \Rightarrow \psi)) \quad \text{and} \\ [b \leftarrow [\bar{x} \leftarrow \bar{p}] \varphi; \bar{x} \leftarrow \bar{p}] (b \Rightarrow \varphi). \end{aligned}$$

By Proposition 36, this can be weakened to

$$\begin{aligned} [a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); b \leftarrow [\bar{x} \leftarrow \bar{p}] \varphi; \bar{x} \leftarrow \bar{p}] (a \Rightarrow (\varphi \Rightarrow \psi)) \quad \text{and} \\ [a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); b \leftarrow [\bar{x} \leftarrow \bar{p}] \varphi; \bar{x} \leftarrow \bar{p}] (b \Rightarrow \varphi). \end{aligned}$$

The claim then follows by propositional reasoning.

*Axiom (K2):* By the definition of  $\langle \bar{x} \leftarrow \bar{p} \rangle \varphi$ , it suffices to show

$$[a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); b \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi; \bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow (a \Rightarrow b)).$$

This is derived from

$$\begin{aligned} [a \leftarrow [\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi); \bar{x} \leftarrow \bar{p}] (a \Rightarrow (\varphi \Rightarrow \psi)) \quad \text{and} \\ [b \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi; \bar{x} \leftarrow \bar{p}] (\psi \Rightarrow b) \end{aligned}$$

by weakening and propositional reasoning as for (K1).

*Axiom (K3□)*: We have

$$\mathbf{[}a \leftarrow \text{ret } \phi; p; b \leftarrow \text{ret } \phi\mathbf{]} (a \Rightarrow b).$$

By Lemma 47, this implies the axiom.

*Axiom (K3◇)*: Dual to (K3□).

*Axiom (K4)*: According to the definition of  $\langle \bar{x} \leftarrow \bar{p} \rangle \varphi \vee \psi$ , we have to show

$$\mathbf{[}a \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \varphi; b \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi; \bar{x} \leftarrow \bar{p}\mathbf{]} ((\varphi \vee \psi) \Rightarrow (a \vee b)).$$

This follows from

$$\begin{aligned} \mathbf{[}a \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \varphi; \bar{x} \leftarrow \bar{p}\mathbf{]} (\varphi \Rightarrow a) \quad \text{and} \\ \mathbf{[}b \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi; \bar{x} \leftarrow \bar{p}\mathbf{]} (\psi \Rightarrow b) \end{aligned}$$

by weakening according to Proposition 36 and propositional reasoning.

*Axiom (K5)*: By the definition of  $[\bar{x} \leftarrow \bar{p}] (\varphi \Rightarrow \psi)$ , we have to show

$$\mathbf{[}a \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \varphi; b \leftarrow [\bar{x} \leftarrow \bar{p}] \psi; \bar{x} \leftarrow \bar{p}\mathbf{]} ((a \Rightarrow b) \Rightarrow (\varphi \Rightarrow \psi)).$$

This is obtained from

$$\begin{aligned} \mathbf{[}a \leftarrow \langle \bar{x} \leftarrow \bar{p} \rangle \varphi; \bar{x} \leftarrow \bar{p}\mathbf{]} (\varphi \Rightarrow a) \quad \text{and} \\ \mathbf{[}b \leftarrow [\bar{x} \leftarrow \bar{p}] \psi; \bar{x} \leftarrow \bar{p}\mathbf{]} (b \Rightarrow \psi) \end{aligned}$$

by weakening and propositional reasoning.

*Axiom (seq□)*: By Lemma 47, the right hand side is the greatest  $\psi : D\Omega$  such that

$$\mathbf{[}a \leftarrow \psi; \bar{x} \leftarrow \bar{p}\mathbf{]} a \Rightarrow [y \leftarrow q] \varphi.$$

By the definition of  $[y \leftarrow q] \varphi$ , this is equivalent to

$$\mathbf{[}a \leftarrow \psi; \bar{x} \leftarrow \bar{p}; y \leftarrow q\mathbf{]} (a \Rightarrow \varphi),$$

and the greatest  $\psi$  with this property is, again by Lemma 47, the left hand side of the axiom.

*Axiom (seq◇)*: Dual to (seq□).

*Axiom (ctr□)*: By Lemma 47, it suffices to show

$$\mathbf{[}a \leftarrow [x \leftarrow p; y \leftarrow q] \varphi; y \leftarrow (\text{do } x \leftarrow p; q)\mathbf{]} (a \Rightarrow \varphi).$$

By Rule (ctr) of Figure 3, this follows from

$$\mathbf{[}a \leftarrow [x \leftarrow p; y \leftarrow q] \varphi; x \leftarrow p; y \leftarrow q\mathbf{]} (a \Rightarrow \varphi),$$

which in turn holds by Lemma 47.

*Axiom (ctr◇)*: Dually to (ctr□).

*Axioms (ret□), (ret◇)*: by rule ( $\eta$ ) of Figure 3,  $\varphi[a/x]$  has the defining property of both  $[x \leftarrow \text{ret } a] \varphi$  and  $\langle x \leftarrow \text{ret } a \rangle \varphi$ .  $\square$

**Proposition 55** The following formulae are valid under the stated assumptions.

$$\begin{array}{ll}
(\text{dsef}\square) & [x \leftarrow p] \varphi \iff \varphi[p/x] \quad (\text{if } p \text{ is dsef}) \\
(\text{dsef}\diamond) & \langle x \leftarrow p \rangle \varphi \iff \varphi[p/x] \quad (\text{if } p \text{ is dsef}) \\
(\text{dis}\square) & [p] \varphi \implies \varphi \quad (\text{if } p \text{ is discardable}) \\
(\text{dis}\diamond) & \langle p \rangle \varphi \iff \varphi \quad (\text{if } p \text{ is discardable})
\end{array}$$

**Proof** (*dsef* $\square$ ): We have to show that  $\varphi[p/x]$  satisfies the defining property of  $[x \leftarrow p] \varphi$  according to Definition 46, i.e. that

$$[\bar{y} \leftarrow \bar{q}; x \leftarrow p] (y_i \Rightarrow \varphi) \iff [\bar{y} \leftarrow \bar{q}] (y_i \Rightarrow \varphi[p/x]).$$

This, however, is just the definition of  $[\bar{y} \leftarrow \bar{q}] (y_i \Rightarrow \varphi[p/x])$  according to Convention 39 (where we implicitly exploit Lemma 38 as explained in the convention).

(*dsef* $\diamond$ ): Dually to (*dsef* $\square$ ).

(*dis* $\square$ ): By Lemma 47,

$$[a \leftarrow [p] \varphi; p] (a \Rightarrow \varphi).$$

By Rule (*dis*) of Figure 4, we obtain  $[a \leftarrow [p] \varphi] (a \Rightarrow \varphi)$ , which by Convention 39 is the claim.

(*dis* $\diamond$ ): Dually to (*dis* $\square$ ). □

As a first application of the calculus, one can show

**Proposition 56**

$$\langle p \rangle \top \text{ iff } p \text{ terminates}$$

**Proof** ‘Only if’: Assume that  $\langle p \rangle \top$  is valid. According to Definition 40, termination of  $p$  means that  $[\bar{x} \leftarrow \bar{q}; p] \phi$  implies  $[\bar{x} \leftarrow \bar{q}] \phi$  (where  $\phi : \Omega$  is a stateless formula). By Lemma 48, we can rephrase these formulae as validities of the corresponding dynamic logic formulae, i.e. we have to show that, under  $\langle p \rangle \top$ ,  $[\bar{x} \leftarrow \bar{q}] \text{ret } \phi$  is derivable from  $[\bar{x} \leftarrow \bar{q}; p] \text{ret } \phi$ . By (*seq* $\square$ ) and (K1) (and the two deduction rules), it suffices to prove

$$\langle p \rangle \top \Rightarrow [p] \text{ret } \phi \Rightarrow \text{ret } \phi.$$

By (K2), we have (noting that  $\text{ret } \phi \iff (\top \Rightarrow \text{ret } \phi)$ )

$$[p] \text{ret } \phi \Rightarrow \langle p \rangle \top \Rightarrow \langle p \rangle \text{ret } \phi.$$

The claim is then proved by propositional reasoning with (K3 $\diamond$ ).

‘If’: By Lemma 47,  $\langle p \rangle \top$  is the smallest  $\psi$  such that  $[a \leftarrow \psi; p] (\top \Rightarrow a)$ . In particular,

$$[a \leftarrow \langle p \rangle \top; p] a$$

holds for arbitrary  $p$ . If  $p$  terminates, then this implies  $[a \leftarrow \langle p \rangle \top] a$ , i.e. by Lemma 20 validity of  $\langle p \rangle \top$ . □

Both for a pair of additional axioms (Remark 60) which are needed to prove some of the rules in the Hoare calculus described in Section 6, and for the sufficient criterion for dynamic logic given in Section 5, we need a mild notion of well-behavedness of monads w.r.t. global dynamic judgements:

**Definition 57** A monad is *logically regular* if

$$[\bar{x} \leftarrow \bar{p}] (([\bar{y} \leftarrow \bar{p}] \phi[\bar{y}/\bar{x}]) \Rightarrow \phi)$$

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

The following is immediate:

**Lemma 58**  $\mathbb{T}$  is logically regular iff, for each  $c : \Omega$ , each formula  $\phi : \Omega$ , and each program sequence  $\bar{x} \leftarrow \bar{p}$ ,

$$c \Rightarrow [\bar{x} \leftarrow \bar{p}] \phi \text{ implies } [\bar{x} \leftarrow \bar{p}] (c \Rightarrow \phi).$$

(The converse implication holds universally:  $c$  implies  $[\bar{x} \leftarrow \bar{p}] c$  by rule (app) of Figure 3, whence  $[\bar{x} \leftarrow \bar{p}] \phi$  by propositional reasoning with  $[\bar{x} \leftarrow \bar{p}] (c \Rightarrow \phi)$ .)

**Proof** Logical regularity follows from the condition above by taking  $c$  to be  $[\bar{y} \leftarrow \bar{p}] \phi[\bar{y}/\bar{x}]$ . Conversely,  $c \Rightarrow [\bar{x} \leftarrow \bar{p}] \phi$  implies  $[\bar{x} \leftarrow \bar{p}] (c \Rightarrow [\bar{y} \leftarrow \bar{p}] \phi[\bar{y}/\bar{x}])$  by  $\alpha$ -equivalence and rule (app) of Figure 3, whence  $[\bar{x} \leftarrow \bar{p}] (c \Rightarrow \phi)$  by propositional reasoning with logical regularity.  $\square$

The intuitionistically valid descriptions of the meaning of global dynamic judgements given in Example 14 for all our running example monads except the continuation monad (for which the given description holds only over **Set**) immediately imply logical regularity. If the internal logic is classical, i.e. if  $\top$  and  $\perp$  are the only truth values, then a case distinction over whether or not  $[\bar{y} \leftarrow \bar{p}] \phi[\bar{y}/\bar{x}]$  holds shows that *all monads are logically regular*. However, this does not hold in the intuitionistic case:

**Example 59** The set  $\Omega = \{\perp, *, \top\}$  ordered by  $\perp \leq * \leq \top$  is a complete Heyting algebra. Hence we can form the topos  $\Omega\text{-Set}$  of  $\Omega$ -sets; we briefly recall the definitions from [10]: an  $\Omega$ -set is a set  $X$  equipped with a fuzzy partial equivalence relation  $\sim$  on  $X$  (i.e. a morphism  $\sim : X \times X \rightarrow \Omega$  such the axioms of a partial equivalence hold in the intuitionistic logic given by  $\Omega$ ). Morphisms are fuzzy relations between  $\Omega$ -sets that are functional in the logic of  $\Omega$ . The subobject classifier of  $\Omega\text{-Set}$  is  $\Omega$  itself. Let  $T$  be the continuation monad on  $\Omega\text{-Set}$  defined by

$$T A = (A \rightarrow \Omega) \rightarrow \Omega.$$

( $\Omega = 1 \rightarrow ? 1$  satisfies the restriction mentioned in Example 3.) Define  $\varphi : T\Omega$  as

$$\varphi = \lambda k : \Omega \rightarrow \Omega. k \perp \vee \neg k \perp.$$

Then we have equations (to be understood externally, i.e. as meta-theoretic equality on  $\Omega$ )

$$\begin{aligned} [b \leftarrow \varphi] b &= (\text{do } b \leftarrow \varphi; \text{ret } (b, b)) \sim (\text{do } b \leftarrow \varphi; \text{ret } (b, \top)) \\ &= \lambda k : \Omega \times \Omega \rightarrow \Omega. \varphi(\lambda b : \Omega. k(b, b)) \sim \lambda k. \varphi(\lambda b. k(b, \top)) \\ &= \bigwedge_{k : \Omega \times \Omega \rightarrow \Omega} \varphi(\lambda b : \Omega. k(b, b)) \sim \varphi(\lambda b. k(b, \top)) \\ &= \bigwedge_{k : \Omega \times \Omega \rightarrow \Omega} (k(\perp, \perp) \vee \neg k(\perp, \perp)) \sim (k(\perp, \top) \vee \neg k(\perp, \top)) \\ &= * \end{aligned}$$

(where, although morphisms are fuzzy relations, we use functional application for them, thereby implicitly moving back into the internal logic — e.g.  $kx \sim kz$  abbreviates the truth value of the formula  $kx = kz$ ). The last equation holds because  $p \vee \neg p$  is either  $*$  or  $\top$ , and hence its comparison with  $q \vee \neg q$  is either  $*$  or  $\top$  as well. Moreover, for  $k_0$  with

$$k_0(a, b) = \begin{cases} \perp, & \text{if } b = \perp \\ *, & \text{otherwise} \end{cases}$$

we have

$$\begin{aligned} (k_0(\perp, \perp) \vee \neg k_0(\perp, \perp)) &\sim (k_0(\perp, \top) \vee \neg k_0(\perp, \top)) = (\perp \vee \neg \perp) \sim (* \vee \neg *) \\ &= \top \sim * \\ &= *, \end{aligned}$$

hence the meet is  $*$  as well. Finally, we have (with the last step being the same as above)

$$\begin{aligned} &[a \leftarrow \varphi] (([b \leftarrow \varphi] b) \Rightarrow a) \\ &= [a \leftarrow \varphi] (* \Rightarrow a) \\ &= (\text{do } a \leftarrow \varphi; \text{ret } (a, * \Rightarrow a)) \sim (\text{do } a \leftarrow \varphi; \text{ret } (a, \top)) \\ &= \lambda k. \varphi(\lambda a. k(a, * \Rightarrow a)) \sim \lambda k. \varphi(\lambda a. k(a, \top)) \\ &= \bigwedge_{k: \Omega \times \Omega \rightarrow \Omega} \varphi(\lambda a. k(a, * \Rightarrow a)) \sim \varphi(\lambda a. k(a, \top)) \\ &= \bigwedge_{k: \Omega \times \Omega \rightarrow \Omega} (k(\perp, * \Rightarrow \perp) \vee \neg k(\perp, * \Rightarrow \perp)) \sim (k(\perp, \top) \vee \neg k(\perp, \top)) \\ &= \bigwedge_{k: \Omega \times \Omega \rightarrow \Omega} (k(\perp, \perp) \vee \neg k(\perp, \perp)) \sim (k(\perp, \top) \vee \neg k(\perp, \top)) \\ &= * \neq \top. \end{aligned}$$

**Remark 60** If  $\mathbb{T}$  is logically regular, then we can prove two further axioms:

$$(\text{eq}\square) \quad p = q \Rightarrow [x \leftarrow p] \varphi \Rightarrow [x \leftarrow q] \varphi$$

$$(\text{eq}\diamond) \quad p = q \Rightarrow \langle x \leftarrow p \rangle \varphi \Rightarrow \langle x \leftarrow q \rangle \varphi.$$

E.g., to prove (eq□), we have to show by Lemma 47

$$[a \leftarrow \text{ret } (p = q); b \leftarrow [x \leftarrow p] \varphi; x \leftarrow q] (a \Rightarrow (b \Rightarrow \varphi)),$$

which becomes by rule ( $\eta$ ) of Figure 4 and logical regularity

$$(p = q) \Rightarrow [b \leftarrow [x \leftarrow p] \varphi; x \leftarrow q] (b \Rightarrow \varphi). \quad (*)$$

This implication is proved as follows: by Lemma 47, we have

$$[b \leftarrow [x \leftarrow p] \varphi; x \leftarrow p] (b \Rightarrow \varphi),$$

and  $p = q$  then implies the right hand side of (\*).

**Remark 61** If  $\mathbb{T}$  is simple, then the converses of Axioms (ctr $\square$ ), (ctr $\diamond$ ) hold; the proof is analogous to that of (ctr $\square$ ) given above, using Proposition 25 instead of rule (ctr).

An interesting consequence of the presence of dynamic logic is that we obtain a converse of rule (dis) of Figure 4 for dsef terms:

**Definition 62** A program  $q$  is called *logically neutral* if  $[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi$  implies  $[\bar{x} \leftarrow \bar{p}; q; \bar{z} \leftarrow \bar{r}] \phi$  for all program sequences  $\bar{p}, \bar{r}$  and for each formula  $\phi$ .

All stateless programs are logically neutral. In simple monads, all discardable programs are logically neutral by Proposition 25. Moreover, in most monads without an explicit notion of state, including the exception monad, the non-determinism monad, and the free abelian group monad, *all* programs are logically neutral. On the other hand, in the continuation monad even discardable and copyable programs may fail to be logically neutral.

**Proposition 63** Let  $\mathbb{T}$  admit dynamic logic. Then all dsef programs are logically neutral.

**Proof** Let  $\bar{x} \leftarrow \bar{p}$  and  $\bar{z} \leftarrow \bar{r}$  be program sequences, let  $\phi$  be a formula such that  $[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}] \phi$ , and let  $q$  be a dsef program. Then  $[\bar{x} \leftarrow \bar{p}; q; \bar{z} \leftarrow \bar{r}] \phi$  is by Definition 46 equivalent to

$$[\bar{x} \leftarrow \bar{p}; q] [\bar{z} \leftarrow \bar{r}] \phi,$$

which by Convention 39 abbreviates the same formulas as  $[\bar{x} \leftarrow \bar{p}] [\bar{z} \leftarrow \bar{r}] \phi$ ; the latter formula is equivalent to the assumption by Definition 46.  $\square$

## 5 Generic states and dynamic logic

Having introduced dynamic logic on an axiomatic basis, we now give a sufficient criterion for a monad to admit dynamic logic. This criterion depends on an abstract notion of state. The type of states will be defined as a subtype by means of a polymorphic formula; for this reason, we shall assume for this section only that we can take intersections of families of extremal subobjects (cf. Section 1.1).

**Definition 64** A *state* is a program  $s : T1$  such that  $s$  terminates and such that, for each deterministically side-effect free program  $p : DA$ , there exists  $a : A$  (due to termination and Lemma 22 necessarily uniquely) such that

$$(\text{do } s; p) = \text{do } s; \text{ret } a.$$

A state  $s$  is called *forcible* if

$$s = \text{do } p; s$$

for each terminating program  $p$ .

Intuitively, a dsef program  $p$  can read, but not modify the state, and hence a program  $s$  delivers a unique state iff each such  $p$  reads a unique value  $a$ . The (unique) state can then be identified with  $s$ . A state is forcible if it can be reached from any other state.

**Example 65** In our running examples, the notions of state and forcible state are made explicit as follows:

- the (abstract) states of the state monad are the constant state transformers  $\lambda s : S \bullet (*, t)$ , where  $t \in S$ , and  $S$  is the set of concrete states from the definition of the state monad (see Definition 3). All of these states are forcible. The set of constant state transformers is isomorphic to the set  $S$ , i.e. the definition of abstract states does indeed capture the original concrete states. The situation is essentially the same in the non-deterministic state monad, where the states are the constant deterministic state transformers  $\lambda s : S \bullet \{(*, t)\}$ .
- Both the exception monad and the non-determinism monad have only one state ('running'), namely the unique terminating element of  $T1$  — i.e.  $*$  in the exception monad, and  $\{*\}$  in the non-determinism monad. In both cases, this state is forcible.
- The states of the interactive input monad are the elements of  $T1$ , i.e. the  $U$ -branching trees with trivially labelled leaves. None of these states are forcible, in accordance with the intuition that one cannot unread input.
- Since in the continuation monad over **Set**, all dsef terms are stateless (Example 37), the states of the continuation monad are the terminating elements of  $T1 \cong R \rightarrow R$ , i.e. by Example 41 the surjective maps  $R \rightarrow R$ . None of these states are forcible: forcibility of  $s$  amounts to  $t \circ s = s$  (sic) for all terminating  $t : R \rightarrow R$ , i.e. for all surjective  $t$ , which is impossible unless  $R$  is trivial.

We will henceforth denote the type of forcible states by  $S$ . (By taking infinite intersections,  $S$  can be defined as a subtype of  $T1$ . Note that the only place where polymorphism really appears here is the unique evaluation of arbitrary dsef terms in Definition 64; the combination of termination and forcibility can be easily expressed monomorphically. Indeed, we conjecture that it suffices to postulate unique evaluation of dsef terms of type  $D\Omega$ ; if this turns out to be correct, the extra assumption made at the beginning of this section could be dropped. Independently of this,  $TS$  is defined as a subtype of  $TT1$  by the formula  $[x \leftarrow x'] x \in S$ .) For the state monad, the set of concrete states has also been denoted by  $S$ , but since the set of abstract and the set of concrete states in this monad are isomorphic, this overloading should not be a problem.

We now proceed to prove a structure theorem concerning the type  $DA$  of deterministically side-effect free programs. The result depends on the existence of a program that 'gives away' the present state. Such a program can generally be defined only in a monad-specific way; however, we can *axiomatize* it in a completely monad-independent way, in the spirit of the encapsulation principle laid out in Section 2. The structure theorem will then be applied to the case of dynamic logic formulae to obtain the sufficient criterion for dynamic logic.

**Definition 66** A deterministically side effect free program  $d : DS$  (recall that  $DS$  is a subtype of  $TT1$ ) is called a *state discloser* if the term

$$\text{do } x \leftarrow d; x$$

of type  $T1$  is discardable (i.e. equal to  $\text{ret } *$ , thus even stateless; cf. Lemma 10).

**Example 67** In monads with only one forcible state  $s$ , such as the non-determinism monad or the exception monad, there is, trivially, a state discloser in the shape of the term  $\text{ret } s$ . In the state monad, we may identify the set of forcible states with the originally given set  $S$  of states (cf. Example 65); then, the element

$$\lambda s : S \bullet (s, s)$$

of  $DS$  is a state discloser (without the said identification, the state discloser is  $\lambda s : S \bullet ((\lambda t : S \bullet (*, s)), s)$ , where  $S$  denotes the originally given set of states). This works analogously for the non-deterministic state monad. In this sense, the notion of state discloser axiomatizes the lookup operator mentioned in [22]. The interactive input monad and the continuation monad (with non-trivial result space) do *not* have state disclosers, since here  $S$  and, hence,  $DS$  are empty.

The relationship between the forcible states and the state discloser is confirmed by the following statement:

**Lemma 68** If  $d$  is a state discloser and  $s$  is a forcible state, then

$$(\text{do } s; d) = \text{do } s; \text{ret } s.$$

**Proof** Since  $s$  is a state and  $d : DS$ , there exists by Definition 64 a forcible state  $t : S$  such that

$$(\text{do } s; d) = \text{do } s; \text{ret } t. \quad (*)$$

For this  $t$ , we have

$$\begin{aligned} t &= \text{do } s; t && (t \text{ forcible}) \\ &= \text{do } s; v \leftarrow \text{ret } t; v \\ &= \text{do } s; v \leftarrow d; v && (\text{by } (*) \text{ above}) \\ &= s && (d \text{ state discloser; Lemma 11, second claim}), \end{aligned}$$

which proves the claim.  $\square$

A consequence of this lemma is

**Proposition 69** State disclosers are uniquely determined.

**Proof** Let  $d_1, d_2$  be state disclosers. Then

$$\begin{aligned} d_1 &= \text{do } x \leftarrow d_2; x; d_1 \\ &= \text{do } x \leftarrow d_2; x; \text{ret } x \\ &= \text{do } y \leftarrow d_2; x \leftarrow d_2; x; \text{ret } y \\ &= \text{do } y \leftarrow d_2; \text{ret } y = d_2, \end{aligned}$$

where we have used discardability of  $\text{do } x \leftarrow d_2; x$  (twice), copyability of  $d_2$ , and Lemma 68 for  $d_1$ .  $\square$

A final prerequisite is a *unique description operator*, i.e. terms of the form  $\iota a : A \bullet \phi$  which denote the element  $a$  that satisfies  $\phi$  if a unique such element exists, and are otherwise undefined. As stated in Section 1, such an operator need not be available for all types. We shall call a type  $A$  *indiscrete* [31] if it admits a unique description operator (which is equivalent to the principle of unique choice); importantly, the type  $\Omega$  of truth values is indiscrete: we can put

$$\iota a : \Omega \bullet \phi := (\forall a : \Omega. \phi \Rightarrow a) \text{ res } (\exists! a. \phi).$$

Note that a topos satisfies unique choice, and hence each type is indiscrete.

The announced structure theorem confirms the intuition that deterministically side-effect free computations are essentially state-dependent values, at least for indiscrete types:

**Theorem 70** If  $\mathbb{T}$  has a state discloser  $d$ , then for each indiscrete type  $A$ ,

$$DA \cong (S \rightarrow A),$$

where the isomorphism  $\kappa$  maps  $y : DA$  to

$$\kappa(y) := \lambda s : S \bullet \iota a : A \bullet ((\text{do } s; y) = (\text{do } s; \text{ret } a))$$

and its inverse maps  $f : S \rightarrow A$  to

$$\kappa^{-1}(f) := \text{do } x \leftarrow d; \text{ret } (f x).$$

**Proof** That the first of the two maps does yield an element of  $S \rightarrow A$  (i.e. a *total function*) follows from the definition of state. The term defining the second map is deterministically side-effect free since  $d$  and  $\text{ret } (f x)$  are dsef. It remains to be shown that the two given maps are inverses of each other.

Indeed we have, for  $y : DA$ ,

$$\begin{aligned} \kappa^{-1}(\kappa(y)) &= \text{do } x \leftarrow d; \text{ret } (\iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)) \\ &= \text{do } x \leftarrow d; w \leftarrow d; w; \text{ret } (\iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)) \\ &= \text{do } x \leftarrow d; w \leftarrow d; x; \text{ret } (\iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)) \\ &= \text{do } x \leftarrow d; x; \text{ret } (\iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)) \\ &= \text{do } x \leftarrow d; x; y \\ &= y, \end{aligned}$$

using discardability of  $(\text{do } w \leftarrow d; w)$ ,  $d$ , and  $(\text{do } x \leftarrow d; x)$ , as well as, in the third step, copyability of  $d$ . The fifth step is just the definition of  $\iota$ : as mentioned above, the element  $\iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)$  indeed exists and thus has its defining property, i.e.  $\text{do } x; y = \text{do } x; \text{ret } \iota a : A \bullet (\text{do } x; y = \text{do } x; \text{ret } a)$ .

Conversely, we have to show that, for  $f : S \rightarrow A$  and  $s : S$ ,

$$f s = \iota a : A \bullet (\text{do } s; x \leftarrow d; \text{ret } (f x) = \text{do } s; \text{ret } a),$$

i.e. that  $f s$  has the defining property of the right hand side (which exists by the above and thus is uniquely determined by its defining property):

$$\text{do } s; x \leftarrow d; \text{ret } (f x) = \text{do } s; \text{ret } (f s).$$

This follows immediately from Lemma 68. □

**Example 71** By Example 67, this theorem applies to most of our running example monads, except the interactive input monad and the continuation monad, which do not have state disclosers (and for which indeed the structure theorem fails to hold, since there are no forcible states).

If  $\mathbb{T}$  has a state discloser, then  $D\Omega$  is by the above Theorem isomorphic to  $S \rightarrow \Omega$ , and the isomorphism translates the ordering on  $D\Omega$  to the pointwise order on  $S \rightarrow \Omega$ . Hence,  $D\Omega$  is a complete lattice with joins and meets inherited from pointwise joins and meets in  $S \rightarrow \Omega$ . Under a mild additional condition,  $\mathbb{T}$  is even stably complete:

**Definition 72** The monad  $\mathbb{T}$  is called *logically continuous* if, for each stateless formula  $\phi : \Omega$  and each program sequence  $\bar{x} \leftarrow \bar{p}$  with  $y \notin FV(\bar{p})$ ,

$$(\forall y. [\bar{x} \leftarrow \bar{p}] \phi) \quad \text{implies} \quad [\bar{x} \leftarrow \bar{p}] \forall y. \phi.$$

(The converse implication holds universally by rule (wk) of Figure 3.) By the description of global dynamic judgements given in Remark 16, all algebraic monads and even the continuation monad on **Set** are logically continuous. Moreover, all our running example monads except the continuation monad are also intuitionistically logically continuous by virtue of Example 14 and Remark 16.

**Example 73** The following monad fails to be logically continuous. Let  $\Omega$  be the real unit interval  $[0, 1]$ . Under the standard ordering, it is a complete Heyting algebra, and hence we can form the topos  $\Omega\text{-Set}$  as in Example 59, with  $\perp$  being 0,  $\top$  being 1,  $\vee$  and  $\wedge$  being inf,  $\exists$  and  $\forall$  being sup, and

$$a \Rightarrow b = \begin{cases} \top, & \text{if } a \leq b; \\ b, & \text{otherwise.} \end{cases}$$

In particular,

$$a \vee \neg a = \begin{cases} \top, & \text{if } a = \perp \\ a, & \text{otherwise} \end{cases} \quad \text{and} \quad \neg \neg a = \begin{cases} \top, & \text{if } a = \top \\ \perp, & \text{otherwise} \end{cases}$$

and hence  $\forall y : \Omega. y \vee \neg y$  is equal to  $\perp$ . Let  $T$  be the continuation monad on  $\Omega\text{-Set}$  defined by

$$TA = (A \rightarrow \Omega) \rightarrow \Omega.$$

Let  $\phi : \Omega$  be the formula  $y \vee \neg y$ , and let  $p : T1$  be the program

$$p = \lambda k : 1 \rightarrow \Omega. \neg \neg (k *)$$

Then (ignoring the redundant value  $* : 1$ )

$$\begin{aligned} \forall y : \Omega. [p] \phi &= \forall y : \Omega. \text{do } p; \text{ret } \phi \sim \text{do } p; \text{ret } \top \\ &= \forall y : \Omega. \lambda k : \Omega \rightarrow \Omega. p(k \phi) \sim \lambda k. p(k \top) \\ &= \bigwedge_{y : \Omega, k : \Omega \rightarrow \Omega} \neg \neg k(y \vee \neg y) \sim \neg \neg k \top \\ &= \top \end{aligned}$$

(where we take the same notational liberties as in Example 59). The last step can be seen as follows. Since any morphism  $k : \Omega \rightarrow \Omega$  (including also  $\neg$ ) is monotone w.r.t.  $\sim$ , we have for any  $y, k$

$$\perp < y \vee \neg y = (y \vee \neg y \sim \top) \leq (k(y \vee \neg y) \sim k(\top)) \leq (\neg \neg k(y \vee \neg y) \sim \neg \neg k(\top)).$$

Since the latter value is classical and greater than  $\perp$ , it must be  $\top$ .

On the other hand, we have

$$\begin{aligned} [p] \forall y : \Omega. \phi &= \text{do } p; \text{ret } \forall y : \Omega. \phi \sim \text{do } p; \text{ret } \top \\ &= \lambda k : \Omega \rightarrow \Omega. p(k(\forall y : \Omega. y \vee \neg y)) \sim \lambda k. p(k \top) \\ &= \bigwedge_{k: \Omega \rightarrow \Omega} \neg \neg k \perp \sim \neg \neg k \top \\ &= \perp \end{aligned}$$

where the latter step follows because we may take  $k = id$ .

**Theorem 74** If  $\mathbb{T}$  is logically continuous and has a state discloser, then  $\mathbb{T}$  is stably complete.

**Proof** Under the isomorphism  $D\Omega \cong (S \rightarrow \Omega)$ , stable completeness translates into the statement that for each family  $(f_j)$  of functions  $S \rightarrow \Omega$  and each program sequence  $\bar{x} \leftarrow \bar{p}$ ,

$$\forall j. [s \leftarrow d; a \leftarrow \text{ret}(f_j s); \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a) \quad \text{implies} \quad [s \leftarrow d; a \leftarrow \text{ret}(\forall j. f_j s); \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow a),$$

where  $d$  is the state discloser (recall that  $(\bigwedge_j f_j) s = \forall j. f_j s$ ). By rule  $(\eta)$  of Figure 3, this is equivalent to saying that  $\forall j. [s \leftarrow d; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow f_j s)$  implies  $[s \leftarrow d; \bar{x} \leftarrow \bar{p}] (x_i \Rightarrow \forall j. f_j s)$ , which is immediate by logical continuity.  $\square$

Moreover, under the assumptions of Theorem 74, there exists, for each program sequence  $\bar{y} \leftarrow \bar{q}$  and each  $\varphi : D\Omega$ , a (unique) formula  $[\bar{y} \leftarrow \bar{q}] \varphi$  that satisfies Lemma 47. In order to establish that this formula satisfies also Definition 46, we need an additional assumption.

**Definition 75** A state  $s : S$  is called *logically splitting* if, for all program sequences  $\bar{x} \leftarrow \bar{p}$  and  $\bar{y} \leftarrow \bar{q}$  and for each formula  $\varphi$ ,

$$[\bar{x} \leftarrow \bar{p}; s; \bar{y} \leftarrow \bar{q}] \varphi \quad \text{implies} \quad [\bar{x} \leftarrow \bar{p}] [s; \bar{y} \leftarrow \bar{q}] \varphi.$$

(Note that the converse implication holds universally thanks to Lemma 21.) In the running examples, all forcible states are logically splitting; it is presently unclear whether or not this is universally the case.

**Theorem 76** If  $\mathbb{T}$  is logically regular and logically continuous and has a logically neutral state discloser, and if all forcible states of  $\mathbb{T}$  are logically splitting, then  $\mathbb{T}$  admits dynamic logic.

(Note that logical neutrality of the state discloser is a necessary condition by Proposition 63.)

**Proof** By Theorems 74 and 51, it suffices to show that  $\mathbb{T}$  has box operators. Let  $\bar{y} \leftarrow \bar{q}$  be a program sequence, and let  $\varphi : D\Omega$ . By Theorem 70, we can define  $[\bar{y} \leftarrow \bar{q}] \varphi$  as  $\kappa^{-1}(f)$  for a function  $f : S \rightarrow \Omega$ . For  $s : S$ , let

$$f s = [s; \bar{y} \leftarrow \bar{q}] \varphi$$

(recall Convention 39). Let  $d$  be the state discloser. Then we have equivalences:

$$\begin{aligned} & [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) \\ \iff & [\bar{x} \leftarrow \bar{p}; (\text{do } t \leftarrow d; t); \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) && (d \text{ state discloser}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d; (\text{do } t \leftarrow d; t); \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) && (d \text{ logically neutral}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d; (\text{do } t \leftarrow d; s); \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) && (d \text{ copyable}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d; s; \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) && (d \text{ discardable}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d] [s; \bar{y} \leftarrow \bar{q}] (x_i \Rightarrow \varphi) && (s \text{ logically splitting}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d] (x_i \Rightarrow [s; \bar{y} \leftarrow \bar{q}] \varphi) && (\text{logical regularity}) \\ \iff & [\bar{x} \leftarrow \bar{p}; s \leftarrow d; a \leftarrow \text{ret } (f s)] (x_i \Rightarrow a) && (\text{definition of } f) \\ \iff & [\bar{x} \leftarrow \bar{p}; a \leftarrow \kappa^{-1}(f)] (x_i \Rightarrow a) && (\text{definition of } \kappa^{-1}) \\ \iff & [\bar{x} \leftarrow \bar{p}; a \leftarrow [\bar{y} \leftarrow \bar{q}] \varphi] (x_i \Rightarrow a) && (\text{definition of } [\bar{y} \leftarrow \bar{q}] \varphi) \\ \iff & [\bar{x} \leftarrow \bar{p}] (x_i \Rightarrow [\bar{y} \leftarrow \bar{q}] \varphi) && (\text{Convention 39}). \end{aligned}$$

□

**Example 77** The above theorem applies to all our running examples except the interactive input monad and the continuation monad (cf. Example 71). The interactive input monad does admit dynamic logic, although it does not have a state discloser.

## 6 A Hoare calculus for total correctness

Classically as well is in the monad-independent setting, dynamic logic subsumes Hoare logic: The monad-independent Hoare triples

$$\{\varphi\} p \{\psi\}$$

introduced in [36] can be expressed in dynamic logic as

$$\varphi \Rightarrow [p] \psi.$$

An obvious expressive advantage of dynamic logic over Hoare logic is its local nature — i.e. formulae hold locally, in any given state. By contrast, Hoare triples only provide global axiomatizations, where each formula is separately quantified over all states. Beyond this, a crucial feature of dynamic logic is its ability to express termination.

In the Hoare calculus, *non-termination* can be expressed by stating that the post-condition  $\perp$  holds. However, Hoare logic is too weak to express termination, while by Proposition 56, the formula

$$\langle p \rangle \top$$

$$\begin{array}{c}
(\Omega) \frac{[\text{ret } \phi] p \top}{[\text{ret } \phi] p [\text{ret } \phi]} \quad (\text{dsef1}) \frac{p \text{ dsef}}{[\varphi] p [\varphi]} \quad (\text{dsef2}) \frac{p \text{ dsef}}{[] x \leftarrow p [x = p]} \quad (\perp) \frac{}{[\perp] p [\varphi]} \\
(\text{ctr}) \frac{[\varphi] \dots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r} [\psi] \quad x \notin FV(\bar{r}) \cup FV(\psi)}{[\varphi] \dots; y \leftarrow (\text{do } x \leftarrow p; q); \bar{z} \leftarrow \bar{r} [\psi]} \quad (\text{if}) \frac{\begin{array}{c} b \text{ dsef} \\ [\varphi \wedge b] x \leftarrow p [\psi] \\ [\varphi \wedge \neg b] x \leftarrow q [\psi] \end{array}}{[\varphi] x \leftarrow \text{if } b \text{ then } p \text{ else } q [\psi]} \\
(\text{iter}) \frac{\begin{array}{c} b \text{ dsef}; \quad t : A \rightarrow DB \\ \_ < \_ : B \times B \rightarrow \Omega \text{ is well-founded} \end{array} \quad [\varphi \wedge b x \wedge (t x =_B z)] y \leftarrow p x [\varphi[y/x] \wedge (t y < z)]}{[\varphi[e/x]] y \leftarrow \text{iter } b p e [\varphi[y/x] \wedge \neg(b y)]} \quad (\text{wk}) \frac{[\varphi] \bar{x} \leftarrow \bar{p} [\psi] \quad \varphi' \Rightarrow \varphi \quad \forall \bar{x}. \psi \Rightarrow \psi'}{[\varphi'] \bar{x} \leftarrow \bar{p} [\psi']} \\
(\text{seq}) \frac{[\varphi] \bar{x} \leftarrow \bar{p} [\psi] \quad [\psi] \bar{y} \leftarrow \bar{q} [\chi]}{[\varphi] \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} [\chi]} \quad (\text{conj}) \frac{[\varphi] \bar{x} \leftarrow \bar{p} [\psi] \quad [\varphi] \bar{x} \leftarrow \bar{p} [\chi]}{[\varphi] \bar{x} \leftarrow \bar{p} [\psi \wedge \chi]} \quad (\text{disj}) \frac{[\varphi] \bar{y} \leftarrow \bar{q} [\chi] \quad [\psi] \bar{y} \leftarrow \bar{q} [\chi]}{[\varphi \vee \psi] \bar{y} \leftarrow \bar{q} [\chi]}
\end{array}$$

Figure 6: The generic Hoare calculus for total correctness

of dynamic logic states that  $p$  terminates in the monad-independent sense defined above. We now can give a meaning to 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. A calculus for such Hoare triples is given in Fig. 6.

Note that in the weakening rule (wk), we have to take care that the weakening of the postcondition does not use undischarged assumptions about the results of the computation; this is ensured by the explicit quantification. The rules (if) and (iter) need some additional infrastructure to be meaningful. For (if), a two-valued type  $Bool$  of Booleans is needed, together with a polymorphic function  $\text{if\_then\_else\_} : Bool \rightarrow a \rightarrow a \rightarrow a$  satisfying the usual laws. (Actually, this just amounts to stating that  $Bool$  is the coproduct  $1 + 1$ .) Furthermore,  $Bool$  can be embedded into  $\Omega$  via

$$\lambda b : Bool \bullet \text{if } b \text{ then } \top \text{ else } \perp;$$

the image of this embedding behaves classically. In the rule, this embedding is implicitly used when truth-valued dsef programs  $b : D Bool$  are treated as formulae (i.e. values of type  $D\Omega$ ). Moreover, we make use of Convention 39 when we apply the *if* construct to dsef computations  $b : D Bool$  rather than values of type  $Bool$ . Both the

*if* rule and the loop rule (*iter*) are given in simplified versions that require the condition to be *dsef*; more general rules can be formulated for conditions with arbitrary side-effects.

The iteration construct in the (*iter*) rule is a control structure that can be defined on top of a monad provided that one has general recursion, which is realized in HASCASL by means of fixed point recursion on *cpos*. Thus, one has to restrict to monads that allow lifting a *cpo* structure on  $A$  to a *cpo* structure on the type  $TA$  of computations in such a way that the monad operations become continuous. This is an example of a *constructor subclass*; the corresponding specification of *cpo-monads* is shown in Figure 7. Function types indicated by  $\xrightarrow{c}$  indicate types of continuous functions [34]. The relevant examples including the ones given above belong to this subclass.

```

spec CPOMONAD = RECURSION and MONAD then
  class CpoMonad < Monad {
  vars  $T : CpoMonad; A, B : Cpo$ 
  type  $T A : Cpo$ 
  ops  $\_ \gg= \_ : T A \xrightarrow{c} (T \xrightarrow{c} ? T B) \xrightarrow{c} ? T B;$ 
       $ret : A \xrightarrow{c} T A$ 
  }

```

Figure 7: The constructor subclass of *cpo-monads*

The iteration construct is introduced as a generalization of the while loop which has a return value (the while loop as programmed e.g. in the Haskell prelude returns only a unit value) which is fed through the iteration; for the case that the body of the loop is never executed, a default return value must be provided as an argument. This has the advantage that the construct makes sense also for ‘stateless’ monads; e.g., iteration in the non-determinism monad results in a function that has all values as outcomes that can be reached by repeatedly applying the original function while a given condition holds. The (executable) specification of the iteration construct is shown in Figure 8. Note that the while loop is just iteration ignoring the return value.

```

spec ITERATION = CPOMONAD and BOOL then
  vars  $T : CpoMonad; A : Cpo$ 
  op  $iter : (A \xrightarrow{c} T Bool) \xrightarrow{c} (A \xrightarrow{c} ? T A) \xrightarrow{c} A \xrightarrow{c} ? T A$ 
  program iter test f x =
    do b ← test x
    if b then
      do y ← f x
      iter test f y
    else ret x
  op  $while(b : T Bool)(p : T Unit) : T Unit = iter (\lambda x \bullet b) (\lambda x \bullet p) ()$ 

```

Figure 8: The iteration control structure

The rules for total correctness formally correspond to the rules for partial correctness from [36], with the exception of the iteration rule which has been extended

by a termination measure along the lines of [16] ( $x \leq y$  stands as shorthand for  $x < y \vee x = y$ ). The intuitionistic notion of wellfoundedness is somewhat more subtle than in the classical case; see e.g. [24]. The iteration rule specializes to the following simpler rule for while:

$$\text{(while)} \frac{\begin{array}{c} t : DB \\ -- < -- : B \times B \rightarrow \Omega \text{ is well-founded} \\ [\varphi \wedge b \wedge t =_B z] p \quad [\varphi \wedge t < z] \end{array}}{[\varphi] \text{while } b \text{ } p \quad [\varphi \wedge \neg b]}$$

(Concerning explicitly typed equality  $=_B$  cf. Convention 39.)

There is no basic rule for stateless programs; however, the rule

$$\text{(ret)} \frac{}{[\phi[a/x]] \quad x \leftarrow \text{ret } a \quad [\phi]}$$

is derivable: by (dsef2), we have  $\square \quad x \leftarrow \text{ret } a \quad [x = a]$ , whence by (wk), (dsef1), and (conj) we obtain  $[\phi[a/x]] \quad x \leftarrow \text{ret } a \quad [x = a \wedge \phi[a/x]]$ ; the claim then follows by (wk).

The Hoare calculus for total correctness is sound:

**Theorem 78** Let  $\mathbb{T}$  admit dynamic logic. Then the rules of the Hoare calculus for total correctness can be derived in the calculus for dynamic logic (including Proposition 55 and Remark 60), where the rules (if) and (iter) depend on  $\mathbb{T}$  being logically regular.

**Proof** We leave propositional reasoning and applications of (mp) implicit.

( $\Omega$ ): We have to show  $\text{ret } \phi \Rightarrow (\langle p \rangle^\top \wedge [p] \text{ret } \phi)$ . The termination part follows from the premise of the rule, and the partial correctness part from Axiom (K3 $\square$ ).

(dsef1): For  $p$  deterministically side-effect free, we need to show

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

The termination part follows from formula (dis $\diamond$ ) and the partial correctness part from formula (dsef $\square$ ) in Proposition 55.

(dsef2): For  $p$  deterministically side-effect free, we need to show

$$\langle p \rangle^\top \wedge [x \leftarrow p] x = p.$$

This follows similarly as for (dsef1).

(seq): We have to show the rule

$$\frac{\begin{array}{c} \varphi \Rightarrow (\langle \bar{x} \leftarrow \bar{p} \rangle^\top \wedge [\bar{x} \leftarrow \bar{p}] \psi) \\ \psi \Rightarrow (\langle \bar{y} \leftarrow \bar{q} \rangle^\top \wedge [\bar{y} \leftarrow \bar{q}] \chi) \end{array}}{\varphi \Rightarrow (\langle \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \rangle^\top \wedge [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] \chi)}$$

to be valid. We proceed with the partial correctness part first. From the second premise of the rule, weakened to the partial correctness part, we obtain  $[\bar{x} \leftarrow \bar{p}] \psi \Rightarrow [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] \chi$  by (nec), (K1) and (seq $\square$ ). Using the first premise, we arrive at

$$\varphi \Rightarrow [\bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}] \chi.$$

Concerning the termination part, from the second premise of the rule weakened to the termination part, we obtain  $\langle \bar{x} \leftarrow \bar{p} \rangle \psi \Rightarrow \langle \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \rangle \top$  by (nec), (K2) and (seq  $\diamond$ ). By combining this with

$$[\bar{x} \leftarrow \bar{p}] (\top \Rightarrow \psi) \wedge \langle \bar{x} \leftarrow \bar{p} \rangle \top \Rightarrow \langle \bar{x} \leftarrow \bar{p} \rangle \psi$$

(obtained from (K2)), we prove

$$[\bar{x} \leftarrow \bar{p}] (\top \Rightarrow \psi) \wedge \langle \bar{x} \leftarrow \bar{p} \rangle \top \Rightarrow \langle \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \rangle \top.$$

This can now be combined with the first premise in order to arrive at

$$\varphi \Rightarrow \langle \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q} \rangle \top.$$

(ctr): Immediate from (ctr  $\square$ ), (ctr  $\diamond$ ).

(if): The premises of the rule translate as

$$\begin{aligned} \varphi \wedge b &\Rightarrow (\langle x \leftarrow p \rangle \top \wedge [x \leftarrow p] \psi) \text{ and} \\ \varphi \wedge \neg b &\Rightarrow (\langle x \leftarrow q \rangle \top \wedge [x \leftarrow q] \psi) \end{aligned}$$

Putting  $s = \text{if } b \text{ then } p \text{ else } q$ , we have to show

$$\varphi \Rightarrow (\langle x \leftarrow s \rangle \top \wedge [x \leftarrow s] \psi)$$

From the definition of  $\text{if } \dots \text{ then } \dots \text{ else } \dots$ , one easily obtains

$$b \Rightarrow s = p \text{ and } \neg b \Rightarrow s = q.$$

Since  $\mathbb{T}$  is logically regular, we can use (eq  $\square$ ) and (eq  $\diamond$ ) by Remark 60. Thus, the above two premises imply

$$\begin{aligned} \varphi \wedge b &\Rightarrow (\langle x \leftarrow s \rangle \top \wedge [x \leftarrow s] \psi) \\ \varphi \wedge \neg b &\Rightarrow (\langle x \leftarrow s \rangle \top \wedge [x \leftarrow s] \psi) \end{aligned}$$

Now  $\text{Bool}$  is classical, hence  $b \vee \neg b$ , and the conclusion follows.

(wk): The first premise is

$$\varphi \Rightarrow (\langle \bar{x} \leftarrow \bar{p} \rangle \top \wedge [\bar{x} \leftarrow \bar{p}] \psi).$$

From the third premise, we obtain by (nec)

$$[\bar{x} \leftarrow \bar{p}] (\psi \Rightarrow \psi').$$

Then conclusion

$$\varphi' \Rightarrow (\langle \bar{x} \leftarrow \bar{p} \rangle \top \wedge [\bar{x} \leftarrow \bar{p}] \psi').$$

then follows by (K1).

(conj): As in all normal modal logics, we can prove the formula

$$[\bar{x} \leftarrow \bar{p}] \psi \Rightarrow [\bar{x} \leftarrow \bar{p}] \chi \Rightarrow [\bar{x} \leftarrow \bar{p}] (\psi \wedge \chi)$$

using (nec) and (K1). Correctness of (conj) is then immediate.

(*disj*), ( $\perp$ ): These rules translate into intuitionistic tautologies.

(*iter*): We show by well-founded induction over  $k : B$  that the premises imply

$$[\varphi \wedge (t x \leq k)] y \leftarrow \text{iter } b p x [\varphi[y/x] \wedge \neg(b y)]$$

for all  $x$ , with the inductive assumption being the same formula, but with  $\leq$  replaced by  $<$ . By definition of *iter*, the goal expands to

$$[\varphi \wedge (t x \leq k)] y \leftarrow \text{if } b x \text{ then do } x' \leftarrow p x; \text{iter } b p x' \text{ else ret } x [\varphi[y/x] \wedge \neg(b y)].$$

By the (if) rule, this reduces to showing

$$[\varphi \wedge \neg b x \wedge (t x \leq k)] y \leftarrow \text{ret } x [\varphi[y/x] \wedge \neg(b y)].$$

and

$$[\varphi \wedge b x \wedge (t x \leq k)] x' \leftarrow p x; y \leftarrow \text{iter } b p x' [\varphi[y/x] \wedge \neg(b y)].$$

The former holds by the derived rule (ret), while the latter follows by rule (seq), since the premise on  $p$  and the inductive assumption allow us to squeeze the assertion

$$[\varphi[x'/x] \wedge (t x' < k)]$$

between the two statements. □

## 7 Example: Reasoning about dynamic reference

We now apply the general machinery developed so far to the (slightly extended) domain of the classical Hoare calculus, namely states consisting of destructively updatable references (note that this is just one example of a state monad), later to be extended by non-determinism.

The reference monad  $R$  uses a type constructor  $Ref$ , where  $Ref a$  is the set of references to values of type  $a$ .  $R a$  is, then, the type of reference computations over  $a$ . The monad comes with operations for reading from and writing to references (besides the usual monad operations); see Figure 9. For  $r, s : Ref a$  note the difference between  $\text{ret}(r = s)$  (equality of references, a stateless formula) and  $\text{ret}(*r =_a *s)$  (equality of contents, a stateful formula); cf. also Convention 39. Moreover, recall from Convention 53 that formulae are implicitly wrapped in  $\text{ret}$  where needed.

The axiomatization provides the usual properties of references without relying on a particular implementation: rule *dsef-read* states that reading is deterministically side-effect free. *read-write* says that after writing to a reference, we can read the value. By contrast, writing to a reference does not change the values of *other* references (*read-write-other*). Note that nothing is said about the nature of references; they could e.g. be integers.

Another example is the nondeterminism monad. Using the monad-independent Hoare calculus, in [36] we have, for instance, axiomatized non-determinism as follows:

$\{\varphi\} \text{fail } \{\psi\}$	$\%(fail)\%$
$\{\varphi\} x \leftarrow p \{\chi_1 x\} \wedge \{\varphi\} x \leftarrow q \{\chi_2 x\} \Rightarrow$ $\{\varphi\} x \leftarrow p \parallel q \{\chi_1 x \vee \chi_2 x\}$	$\%(join)\%$

```

spec REFERENCE = CPOMONAD then
  var   a : Cpo
  types R : CpoMonad; Ref a : Flatcpo
  ops   *_: Ref a  $\xrightarrow{c}$  R a;
         _ := _: Ref a  $\xrightarrow{c}$  a  $\xrightarrow{c}$  R Unit
  internal {
  forall x, y : a; r, s : Ref a
  • dsef(*r)                                     %(dsef-read)%
  • [] r := x [x = *r]                           %(read-write)%
  • [x = *r] s := y [x = *r  $\vee$  r = s]           %(read-write-other)%
  }

```

Figure 9: Specification of the reference monad

With dynamic logic, a more fine-grained specification of non-determinism is possible:

$$\begin{aligned}
\langle p \parallel q \rangle \varphi &\iff (\langle p \rangle \varphi \vee \langle q \rangle \varphi) \\
[p \parallel q] \varphi &\iff ([p] \varphi \wedge [q] \varphi) \\
[fail] \perp &
\end{aligned}$$

From the second axiom, we easily get the Hoare rule (join) above by taking  $\varphi$  to be  $\chi_1 x \vee \chi_2 x$ . But we get more than that: the first axiom implies that  $p \parallel q$  terminates *if and only if*  $p$  or  $q$  terminates. This is not expressible in Hoare logic for partial correctness. Since  $\parallel$  behaves differently w.r.t. partial correctness and termination, this leads to a more complicated Hoare rule for total correctness:

$$\text{(join)} \frac{
\begin{array}{l}
[\varphi]x \leftarrow p[\chi_1 x] \\
\{\psi\}x \leftarrow p\{\chi_1 x\} \\
[\psi]x \leftarrow q[\chi_2 x] \\
\{\varphi\}x \leftarrow q\{\chi_2 x\}
\end{array}
}{[\varphi \vee \psi]x \leftarrow p \parallel q[\chi_1 x \vee \chi_2 x]}$$

The correctness of this rule can be seen as follows: Concerning termination, the first premise gives us  $\varphi \Rightarrow \langle p \rangle \top$ , hence by the above first property of  $\parallel$ ,  $\varphi \Rightarrow \langle p \parallel q \rangle \top$ . Similarly, the third premise gives us  $\psi \Rightarrow \langle p \parallel q \rangle \top$ ; so altogether  $\varphi \vee \psi \Rightarrow \langle p \parallel q \rangle \top$ . Concerning partial correctness, the first two premises lead to  $\varphi \vee \psi \Rightarrow [x \leftarrow p] \chi_1 x$ , hence  $\varphi \vee \psi \Rightarrow [x \leftarrow p](\chi_1 x \vee \chi_2 x)$ , while the remaining two lead to  $\varphi \vee \psi \Rightarrow [x \leftarrow q](\chi_1 x \vee \chi_2 x)$ . Combining these using the above second property of  $\parallel$  yields  $\varphi \vee \psi \Rightarrow [x \leftarrow p \parallel q](\chi_1 x \vee \chi_2 x)$ .

By contrast, the *fail*-rule remains as above, and can be expressed using only partial correctness:

$$\text{(fail)} \frac{}{\{\varphi\} fail \{\psi\}}$$

One advantage of the looseness of the specifications introduced so far is that we now can combine the specification of references and of nondeterminism and get a specification of nondeterministic reference computations.

As an example, we prove the termination of Dijkstra's nondeterministic version of Euclid's algorithm for computing the greatest common divisor [5] within this monad (we have already shown partial correctness in [36]). We assume that we have some specification of natural numbers and arithmetic, including an operation  $max$  computing the maximum of two natural numbers, and  $Bool$ -valued functions  $==$  and  $<$  for comparison of natural numbers (which, when composed with the inclusion of  $Bool$  into  $\Omega$ , are the usual predicates  $=$  and  $<$ ). We also use  $x \leq y$  as shorthand for  $x < y \vee x = y$ . We work with the monad of nondeterministic references, applied to the naturals. Let *euclid* be the program sequence:

$$\begin{aligned} & \text{while } \text{ret}(\neg *r == *s) \\ & \quad (\text{if } \text{ret}(*s < *r) \text{ then } r := *r - *s \text{ else } \text{fail} \\ & \quad \quad \perp \\ & \quad \text{if } \text{ret}(*r < *s) \text{ then } s := *s - *r \text{ else } \text{fail}) \end{aligned}$$

We now will try to prove that *euclid* terminates provided that  $r$  and  $s$  are distinct references, i.e.

$$[\neg r = s] \text{ euclid } [\top].$$

We proceed as follows. Using (dsef1), (dsef2), (seq), ( $\Omega$ ) and (conj), we can show

$$\begin{aligned} & [\neg r = s \wedge \text{max}(*r, *s) \leq z \wedge *s < *r] \\ & u \leftarrow *r; v \leftarrow *s \\ & [\neg r = s \wedge \text{max}(*r, *s) \leq z \wedge *s < *r \wedge u = *r \wedge v = *s]. \end{aligned}$$

By arithmetic reasoning and (wk), we obtain

$$\begin{aligned} & [\neg r = s \wedge \text{max}(*r, *s) \leq z \wedge *s < *r] \\ & u \leftarrow *r; v \leftarrow *s \\ & [\neg r = s \wedge \text{max}(u, v) \leq z \wedge v < u \wedge v = *s]. \end{aligned} \tag{1}$$

By (read-write), (read-write-other), ( $\Omega$ ), and (conj), we can show (noting that from  $A \vee B$  and  $\neg B$ , one can intuitionistically infer  $A$ )

$$\begin{aligned} & [\neg r = s \wedge \text{max}(u, v) \leq z \wedge v < u \wedge v = *s] \\ & r := u - v \\ & [\neg r = s \wedge \text{max}(u, v) \leq z \wedge v < u \wedge v = *s \wedge u - v = *r]. \end{aligned}$$

By arithmetic reasoning and (wk), we get

$$\begin{aligned} & [\neg r = s \wedge \text{max}(u, v) \leq z \wedge v < u \wedge v = *s] \\ & r := u - v \\ & [\neg r = s \wedge \text{max}(*r, *s) < z]. \end{aligned}$$

By (seq) with (1) and noting that  $r := *r - *s$  is shorthand for  $u \leftarrow *r; v \leftarrow *s; r := u - v$ , we get

$$\begin{aligned} & [\neg r = s \wedge \text{max}(*r, *s) \leq z \wedge *s < *r] \\ & r := *r - *s \\ & [\neg r = s \wedge \text{max}(*r, *s) < z]. \end{aligned}$$

From [*fail*]  $\perp$  we get by (wk) and ( $\perp$ )

$$\begin{aligned} & [\neg r = s \wedge \text{max}(*r, *s) \leq z \wedge *s < *r \wedge \neg *s < *r] \\ & \text{fail} \\ & [\neg r = s \wedge \text{max}(*r, *s) < z]. \end{aligned}$$

Hence by (if)

$$\begin{aligned} & [\neg r = s \wedge \max(*r, *s) \leq z \wedge *s < *r] \\ & \text{if ret } (*s < *r) \text{ then } r := *r - *s \text{ else fail} \\ & [\neg r = s \wedge \max(*r, *s) < z] \end{aligned} \quad (2)$$

By ( $\perp$ ) (and noting that total implies partial correctness),

$$\begin{aligned} & \{\neg r = s \wedge \max(*r, *s) \leq z \wedge *r < *s \wedge *s < *r\} \\ & r := *r - *s \\ & \{\neg r = s \wedge \max(*r, *s) < z\} \end{aligned}$$

and by (fail)

$$\begin{aligned} & \{\neg r = s \wedge \max(*r, *s) \leq z \wedge *r < *s \wedge \neg *s < *r\} \\ & \text{fail} \\ & \{\neg r = s \wedge \max(*r, *s) < z\} \end{aligned}$$

From these, with the rule (if) from the Hoare calculus for partial correctness, we get

$$\begin{aligned} & \{\neg r = s \wedge \max(*r, *s) \leq z \wedge *r < *s\} \\ & \text{if ret } (*s < *r) \text{ then } r := *r - *s \text{ else fail} \\ & \{\neg r = s \wedge \max(*r, *s) < z\} \end{aligned} \quad (3)$$

Analogously, we obtain

$$\begin{aligned} & [\neg r = s \wedge \max(*r, *s) \leq z \wedge *r < *s] \\ & \text{if ret } (*r < *s) \text{ then } s := *s - *r \text{ else fail} \\ & [\neg r = s \wedge \max(*r, *s) < z]. \end{aligned} \quad (4)$$

and

$$\begin{aligned} & \{\neg r = s \wedge \max(*r, *s) \leq z \wedge *s < *r\} \\ & \text{if ret } (*r < *s) \text{ then } s := *s - *r \text{ else fail} \\ & \{\neg r = s \wedge \max(*r, *s) < z\} \end{aligned} \quad (5)$$

From (2) to (5) and rule (join), we get

$$\begin{aligned} & [\neg r = s \wedge \max(*r, *s) \leq z \wedge (*s < *r \vee *r < *s)] \\ & \text{if ret } (*s < *r) \text{ then } r := *r - *s \text{ else fail} \\ & \parallel \text{if ret } (*r < *s) \text{ then } s := *s - *r \text{ else fail} \\ & [\neg r = s \wedge \max(*r, *s) < z]. \end{aligned}$$

Now  $\_ == \_$  and  $\_ < \_$  take values in *Bool* and hence behave classically. Therefore by (wk)

$$\begin{aligned} & [\neg r = s \wedge \neg *r == *s \wedge \max(*r, *s) = z] \\ & \text{if ret } (*s < *r) \text{ then } r := *r - *s \text{ else fail} \\ & \parallel \text{if ret } (*r < *s) \text{ then } s := *s - *r \text{ else fail} \\ & [\neg r = s \wedge \max(*r, *s) < z]. \end{aligned}$$

Noting that  $\_ < \_$  is well-founded, we are now set to apply the while rule, obtaining

$$\begin{aligned} & [\neg r = s] \\ & \text{while ret } (\neg *r == *s) \\ & ( \text{if ret } (*r > *s) \text{ then } r := *r - *s \text{ else fail} \\ & \parallel \text{if ret } (*s > *r) \text{ then } s := *s - *r \text{ else fail} \\ & [\neg r = s \wedge *r == *s]. \end{aligned}$$

In the last step, we arrive by rule (wk) at

$$\begin{array}{l}
 [\neg r = s] \\
 \text{while ret } (\neg *r == *s) \\
 ( \text{if ret } (*r > *s) \text{ then } r := *r - *s \text{ else fail} \\
 \quad \parallel \text{if ret } (*s > *r) \text{ then } s := *s - *r \text{ else fail} \\
 [\top].
 \end{array}$$

## 8 Conclusion

Building on results on monad-independent reasoning about program properties developed in [36], we have designed a monad-independent dynamic logic and a representation of this logic in the internal logic of HASCASL, i.e. essentially in intuitionistic partial higher order logic. The main problem here was to provide a monad-independent semantics of the dynamic modal operators. Our solution to this problem reconciles the approaches of Pitts [26] (who defines a local semantics that however needs additional structure on top of the monad) and Moggi [22] (who defines a monad-independent semantics that however is global) by giving a purely monadic semantics that retains the local character of the modal operators. To this end, we introduce an axiomatic method which imposes additional constraints on the monad. This method is complemented by results that, under suitable conditions, allow the extraction of abstract states from a given monad, giving rise to a structure theorem for ‘dynamic truth values’ which guarantees the interpretability of the modal operators. The structure theorem is both of independent interest and the basis for further research into the question of whether a monad can generally be decomposed into aspects of input, output, non-determinism and state.

Given the semantics of the modal operators, we have introduced a generic proof system for dynamic logic over an arbitrary monad, where the rules and axioms are proved as lemmas about the encoding. We have thus ended up with a logic that allows dynamic reasoning about computations with side effects, leaving the actual nature of the side effects open. In practice, one will aim at performing a large amount of verification in this generic setting, and switch to instantiations of the calculus for particular monads only in the more detailed analysis. As an example, we have laid out how operations in the non-determinism monad can be axiomatized by means of dynamic logic formulae. A library of monad definitions in HASCASL will make extensive use of this axiomatization principle; the compositionality of such axiomatizations w.r.t. monad combination is subject of further research.

One of the crucial features of dynamic logic is that it is able to express termination of programs. As an example application, we have shown how to obtain a termination rule for a generic iteration construct, and illustrated the use of this calculus in a termination proof for Dijkstra’s non-deterministic version of Euclid’s algorithm; i.e. not only partial correctness, but also total correctness lends itself to monad-independent reasoning. This corroborates Moggi’s claim [21] that the logic of monads is the right setting for reasoning about computations with effects.

## Acknowledgements

This work forms part of the DFG-funded project HasCASL (KR 1191/7-1). The authors wish to thank Christoph Lüth for useful comments and discussions, Erwin R. Catesbeiana for pointing out various pitfalls, and the anonymous referees for their detailed suggestions for improvement of the paper.

## References

- [1] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Wiley Interscience, 1990.
- [2] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the Common Algebraic Specification Language. *Theoret. Comput. Sci.*, 286:153–196, 2002.
- [3] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer, 1984.
- [4] R. L. Crole. *Programming Metalogics with a Fixpoint Type*. PhD thesis, University of Cambridge, 1991.
- [5] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.*, 103:235–271, 1992.
- [7] A. Filinski. Declarative continuations: an investigation of duality in programming language semantics. In *Category Theory and Computer Science*, volume 389 of *LNCS*, pages 224–229. Springer, 1989.
- [8] C. Führmann. *The Structure of Call-by-Value*. PhD thesis, University of Edinburgh, 2000.
- [9] C. Führmann. Varieties of effects. In *Foundations of Software Science and Computation Structures*, volume 2303 of *LNCS*, pages 144–158. Springer, 2002.
- [10] R. Goldblatt. *Topoi, The Categorical Analysis of Logic*. North-Holland, revised edition, 1984.
- [11] The Haskell mailing list. <http://www.haskell.org/maillinglist.html>, 2002.
- [12] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalizers, and presentations of finitary monads. *J. Pure Appl. Algebra*, 89:163–179, 1993.
- [13] D. Kozen. Results on the propositional mu-calculus. *Theoret. Comput. Sci.*, 27:333–354, 1983.
- [14] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. MIT, 1990.

- [15] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge, 1986.
- [16] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley, 1987.
- [17] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1997.
- [18] I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic*, 104:189–218, 2000.
- [19] E. Moggi. Categories of partial morphisms and the  $\lambda_p$ -calculus. In *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 242–251. Springer, 1986.
- [20] E. Moggi. *The Partial Lambda Calculus*. PhD thesis, University of Edinburgh, 1988.
- [21] E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93:55–92, 1991.
- [22] E. Moggi. A semantics for evaluation logic. *Fund. Inform.*, 22:117–152, 1995.
- [23] Peter D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.
- [24] L. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symbolic Comput.*, 2:325–355, 1986.
- [25] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: *J. Funct. Programming* **13** (2003).
- [26] A. Pitts. Evaluation logic. In *Higher Order Workshop*, Workshops in Computing, pages 162–189. Springer, 1991.
- [27] G. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.
- [28] G. Plotkin and C. Stirling. A framework for intuitionistic modal logic. In *Theoretical Aspects of Reasoning about Knowledge*. Morgan Kaufmann, 1986.
- [29] V. Pratt. Semantical considerations on Floyd-Hoare logic. In *Foundations of Computer Science*, pages 109–121. IEEE, 1976.
- [30] F. Regensburger. HOLCF: Higher order logic of computable functions. In *Theorem Proving in Higher Order Logics*, volume 971 of *LNCS*, pages 293–307, 1995.
- [31] L. Schröder. The HASCASL prologue: categorical syntax and semantics of the partial  $\lambda$ -calculus. Available as <http://www.informatik.uni-bremen.de/~lschrode/hascasl/plam.ps>
- [32] L. Schröder. Classifying categories for partial equational logic. In *Category Theory and Computer Science*, volume 69 of *ENTCS*, 2002.

- [33] L. Schröder. Henkin models of the partial  $\lambda$ -calculus. In *Computer Science Logic*, volume 2803 of *LNCS*, pages 498–512. Springer, 2003.
- [34] L. Schröder and T. Mossakowski. HASCASL: Towards integrated specification and development of functional programs. In *Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 99–116. Springer, 2002.
- [35] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HASCASL. In *Recent Developments in Algebraic Development Techniques, 16th International Workshop, WADT02*, volume 2755 of *LNCS*, pages 425–441. Springer, 2003.
- [36] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HASCASL. In M. Pezze, editor, *Fundamental Aspects of Software Engineering*, volume 2621 of *LNCS*, pages 261–277, 2003.
- [37] L. Schröder, T. Mossakowski, and C. Maeder. HASCASL – Integrated functional specification and programming. Language summary. Available at [http://www.informatik.uni-bremen.de/agbkb/forschung/formal\\_methods/CoFI/HasCASL](http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL)
- [38] A. K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.
- [39] H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.
- [40] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29:240–263, 1997.