Monad-Independent Hoare Logic in HasCasl

Lutz Schröder and Till Mossakowski

BISS, Department of Computer Science, University of Bremen

Abstract. Monads have been recognized by Moggi as an elegant device for dealing with stateful computation in functional programming languages. It is thus natural to develop a Hoare calculus for reasoning about computational monads. While this has previously been done only for the state monad, we here provide a generic, monad-independent approach, which applies also to further computational monads such as exceptions, input/output, and non-determinism. All this is formalized within the logic of HasCasl, a higher-order language for functional specification and programming. Combination of monadic features can be obtained by combining their loose specifications. As an application, we prove partial correctness of Dijkstra's nondeterministic version of Euclid's algorithm in a monad with nondeterministic dynamic references.

1 Introduction

One of the central concepts of modern functional programming is the encapsulation of side effects via monads following the seminal paper [11]. In particular, state monads are used to emulate an imperative programming style in the functional programming language Haskell [15]. 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.

Here, we show how one can also build a generic *logical* environment for reasoning about generic monadic computations by providing a monad-independent Hoare calculus. These results are developed in the framework of the algebraic specification language HasCasl. In this way, we generalize the suggestions of [11], which were aimed purely at a state monad with state interpreted as global store.

HasCasl has been introduced in [14] as a higher order extension of the first order algebraic specification language Casl [3]. HasCasl is geared towards specification of functional programs, in particular in Haskell; in fact, HasCasl has an executable subset that corresponds quite closely to a large subset of Haskell. Features of HasCasl include partial and total higher order types, polymorphism, type classes, and general recursive functions. The technical requirement for a general treatment of monads is support for constructor classes, which are a straightforward extension of HasCasl's type classes. In the correspondingly

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 261–277, 2003.

extended language, one can easily specify not only the operations, but also the axioms associated to a monad.

Using HasCasl's internal logic, one can give a semantics to Hoare triples independently of the internal structure of the monad. The HasCasl logic then provides a Hoare calculus that allows reasoning about partial correctness as well as loosely specifying imperative programs. We provide both a generic kernel calculus and specialized calculi that provide additional rules dealing with monad-specific operations such as assignment. We end up with an environment that offers not only a combination of functional and imperative programming (as provided in Haskell), but also a surrounding logic that is rather effortlessly adapted to the specification of both functional and imperative aspects.

2 HasCasl

The language HasCasl has been introduced in [14] as a higher order extension of Casl, based on the partial λ-calculus. We give a brief summary of how basic HasCasl specifications are written and what they mean. For more details on both syntax and semantics, see [14].

Any HasCasl specification determines essentially two things: a *signature* consisting of classes, types, and operations, and associated *axioms* that the operations are required to satisfy. Basic types are introduced by means of the keyword **type**. Types may be parametrized by type arguments; e.g., we may write

var $a : \mathit{Type}$
type $\mathit{List}\ a$

and obtain a unary type constructor *List*. There are built-in type constructors (with fixed interpretations) $\_ \times \_$, $\_ \times \_ \times \_$ etc. for product types, $\_ \to ? \_$ and $\_ \to \_$ for partial and total function types, respectively, $\mathit{Pred}\ \_$ for predicate types, and a unit type *Unit*. In particular, the function types of Haskell are really partial function types.

Next, an *operator* is a constant of some type, declared by

op $f : t$

where t is a type. Since types may contain type variables, operators can be polymorphic in the style of ML.

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 λ-abstraction. Such terms may then be used in *axioms* formulated, to begin, in what we shall call the *external logic*. This external logic offers the usual logical connectives (conjunction, negation etc.) as well as universal and existential quantifiers, where the outermost universal quantifications may also be over type variables, strong and existential equality denoted by $=$ and $\overset{e}{=}$, respectively, and definedness assertions $\mathit{def}\ \alpha$ (the latter feature and the distinction between the various equalities are related to partial functions; cf. [12] for a detailed discussion). The notation used in the examples below is largely self-explanatory, so we shall refrain from listing the syntactic details here. It is important to note that formulas of the external

logic, including external equations, are not regarded as terms of a program and hence cannot be λ-abstracted. Partial functions are, unlike in Haskell, required to be strict; non-strict functions can be emulated by means of the procedural lifting method, for which suitable syntactical sugar is provided.

The semantics of a HasCasl specification is the class of its (set-theoretic) *intensional Henkin models*: 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; see [14] for a discussion of the rationale behind this. If desired, extensionality of models may be forced by means of an axiom expressible within the language.

A consequence of the intensional semantics is the presence of an intuitionistic *internal logic* that lives within λ-terms. One can specify an *internal equality* (for which the symbol $=$ is built-in syntactical sugar) to be used within λ-terms, which then allows *specifying* the full set of logical operations and quantifiers of intuitionistic logic; this is carried out in detail in [14]. There is built-in syntactical sugar for the internal logic, invoked by means of the keyword **internal** which signifies that formulas in the following block are to be understood as formulas of the internal logic.

By means of the internal logic, one can then specify a class of complete partial orders and fixed point recursion in much the same style as in HOLCF [13]. 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**. Similarly, the no-junk-no-confusion axioms associated to datatypes are implicitly coded by means of the internal logic.

HasCasl supports *type classes*. These 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.

In a similar vein, one can add *constructor classes* to HasCasl. They can be interpreted as predicates on the syntactical universe of abstracted type expressions (also called *pseudotypes*), e.g.

$$\lambda a : Type \bullet a \to ? \; List \; a$$

As for type classes, there are constructor subclasses; types, operators, axioms may be polymorphic over constructor classes; and this polymorphism is semantically coded by collections of instances. A typical example of a constructor class is the class of monads (see Fig. 1); an example of a constructor subclass can be found in Fig. 3.

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.

3 Monads for Computations

On the basis of the seminal paper [11], 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 [8]. Intuitively, a monad associates to each type A a type TA of computations of type A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts 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* $(T, \eta, \_^*)$, where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $\_^*$ assigns to each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ such that

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

This description is equivalent to the more familiar one via an endofunctor with unit and multiplication [10]. A monad gives rise to a *Kleisli category* over $\mathbf{C}$, which has the same objects as $\mathbf{C}$ and 'functions with side effects' $f : A \to TB$ as morphisms from A to B; the composite of two such functions g and f is just g^*f. This composite will also be denoted $f; g$.

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

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

called *tensorial strength*, subject to certain coherence conditions (see e.g. [11]); this is equivalent to enrichment of the monad over $\mathbf{C}$ (see discussion and references in [11]).

Example 1 ([11]). Computationally relevant monads on **Set** (since strength is equivalent to enrichment, all monads on **Set** are strong) include

- stateful computations with possible non-termination: $TA = (S \to ? (A \times S))$, where S is a fixed set of states and $\_ \to ? \_$ denotes the partial function type;
- (finite) non-determinism: $TA = \mathcal{P}_{fin}(A)$, where $\mathcal{P}_{fin}$ denotes the finite power set functor;
- exceptions: $TA = A + E$, where E is a fixed set of exceptions;
- interactive input: TA is the smallest fixed point of $\gamma \mapsto A + (U \to \gamma)$, where U is a set of input values.

– non-deterministic stateful computations: $TA = (S \to \mathcal{P}_{fin}(A \times S))$, where, again, S is a fixed set of states;

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 Fig. 2. Since the operations of the monad are functions in the model, the monads thus specified are automatically strong, strength being equivalent to enrichment. The notation is (almost) identical to the one used in Haskell, i.e. the unit is denoted by *ret*, and the type constructor by m; the operator $\_ >>= \_$ denotes, in the above notation, the function $(x, f) \mapsto f^*(x)$. This specification 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 $e_1 >>= \lambda x \bullet e_2$. A slight complication concerning the axiomatization arises from the fact that partial functions are involved. Note that the first axiom has been equipped with a definedness guard. This ensures that standard monads such as the state monad with its usual definition (cf. Fig. 2 and the recent discussion on [7]) are actually subsumed, while leaving the essence of the proposed calculus untouched.

```
spec  MONAD = INTERNALLOGIC then
      class  Monad : Type → Type {
      vars   m : Monad;  a, b, c : Type
      ops    _ >>= __ : m a → (a →? m b) →? m b;
             ret : a → m a                          }
      internal {
      forall x, y : a;  y : m a;  f : a →? m b;  g : b →? m c
             • def (f x) ⇒ ((ret x) >>= f) = f x
             • (y >>= ret) = y
             • ((y >>= f) >>= g) = (y >>= (λ x : a • f x >>= g))    }
```

Fig. 1. The constructor class of monads

Reasoning about a category **C** equipped with a strong monad is greatly facilitated by the fact that proofs can be conducted in an *internal language* introduced in [11]. The crucial features of this language are

– A type operator T; terms of type TB for some B are called *programs*;
– an polymorphic operator ret : $A \to TA$ corresponding to the unit;
– a binding construct, which we here denote in Haskell's do style instead of by let: terms of the form

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

```
spec  STATE [type S] = MONAD then
      type instance ST : Monad
      vars  a, b : Type
      type  ST a := S →? (a × S)
      internal {
      forall x : a; y : ST; f : a →? ST b
            • ret x = λ s : S • (x, s)
            • (y >>= f) = λ s1 : S • let (z, s2) = y s1 in f z s2    }
```

Fig. 2. Specification of the state monad

are interpreted by means of the tensorial strength and Kleisli composition (See [11] for details. This is essentially equivalent the do-notation introduced above.). Intuitively, do $x \leftarrow p;\ q$ computes p and passes the results on to q. Nested do expressions like do $x \leftarrow p;$ do $y \leftarrow q;\ \ldots$ may also be denoted do $x \leftarrow p; y \leftarrow q;\ \ldots$. Repeated nestings such as do $x_1 \leftarrow p_1, \ldots, x_n \leftarrow p_n;\ q$ are somewhat inaccurately denoted in the form 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 lateron may be omitted from the notation.

Terms are generally formed in a context $\Gamma = (x_1 : s_1, \ldots, x_n : s_n)$ of variables with assigned types. Thanks to an equivalence theorem proved in [11], this language (with further term formation rules and a deduction system) can be used both in order to define morphisms in **C** and in order to prove equalities between them. For example, morphisms $f : A \to TB$ and $g : B \to TC$ may also be seen as terms $f : TB$ and $g : TC$ in context $x : A$ and $y : B$, respectively; the Kleisli composite $f; g$ is represented by do $y \leftarrow f;\ g$.

On top of a monad, one can generically define control structures such as a while loop. However, such definitions require 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 Fig. 3. Function types indicated by $\xrightarrow{c}$ indicate types of continuous functions [14]. The relevant examples including the ones given above belong to this subclass.

As an example of a recursively defined control structure we introduce an iteration construct which generalizes the while loop by extending it with a default return value (the while loop as programmed e.g. in the Haskell prelude returns only a unit value) which is fed through the iteration. This has the advantage that the construct makes sense also for 'stateless' monads; e.g., iteration in the nondeterminism 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 Fig. 4. Note that the while loop is just iteration ignoring the return value.

```
spec CpoMonad = Recursion and Monad then
     class CpoMonad < Monad {
     vars m : CpoMonad; a : Cpo
     type m a : Cpo
     ops _ >>= _ _ : m a ->c (a ->c? m b) ->c? m b;
         return : a ->c m a                          }
```

Fig. 3. The constructor subclass of cpo-monads

```
spec Iteration = CpoMonad and Bool then
     vars m : CpoMonad; a : Cpo
     op iter : (a ->c m Bool) ->c (a ->c? m a) ->c a ->c? m a
     program iter test f x =
                 do b <- test x
                    if b then
                         do y <- f x
                            iter test f y
                         else return x
     op while(b : m Bool)(p : m Unit) : m Unit = iter (λ x • b) (λ x • p) ()
```

Fig. 4. The iteration control structure

4 The Generic Hoare Calculus

We now proceed to describe a Hoare-calculus which is generic over the underlying monad. (Similar calculi discussed in [5,11] are specific for the state monad, where 'state' is additionally restricted to mean global store). We shall be using the notation for monads introduced in the previous section (T, η etc.) throughout, as well as the internal language discussed at the end of the previous section.

As usual, the calculus will be concerned with proving *Hoare triples* consisting of a stateful expression together with a pre- and a postcondition. Since in general we cannot undo changes to the 'state', we have to require the pre- and postconditions to 'leave the state unchanged' in a suitable sense in order to guarantee composability of Hoare triples, at the same time allowing the conditions to read the state.

Definition 2. A program p is called *side-effect free* if

$$(\text{do } y \leftarrow p;\ \text{ret} *) = \text{ret} * \qquad (\text{shorthand: } \mathit{sef}(p)),$$

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

Note that $\mathit{sef}(p)$ implies that p is always defined. Properties such as side-effect freeness are said to hold for a program sequence iff they hold for each of the component programs.

Lemma 3. *If p is side-effect free, then*

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

for each program q that does not contain x.

Example 4. A program p is side-effect free

- 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).

A program p is called *stateless* if it factors through η, i.e. if it is just a value inserted into the monad ('*p exists*' in the terminology of [11]) – otherwise, it is called *stateful*. E.g. in the state monad, p is stateless iff it neither changes nor reads the state. Stateless programs are side-effect free, but not vice versa.

We will want to regard programs that return truth values as formulas with side effects. We equip such formulas with a notion of global validity, denoted explicitly by a 'global box' $\boxed{\mathrm{G}}$:

Definition 5. Given a term ϕ of type $T\Omega$, where Ω denotes the type of internal truth values, $\boxed{\mathrm{G}}\,\phi$ abbreviates

$$\phi = \text{do } x \leftarrow \phi;\ \text{ret}\,\top,$$

read as a formula of the internal logic.

If ϕ is side-effect free, then $\boxed{\mathrm{G}}\,\phi$ simplifies to $\phi = \text{ret}\,\top$; otherwise, the formula above ensures that the right hand side has the same side-effect as ϕ.

Remark 6. Note that the equality in the definition of $\boxed{\mathrm{G}}\ \phi$ above is strong equality. In particular, in the classical case $\boxed{\mathrm{G}}\,\phi$ is true if ϕ is undefined.

Example 7. In the monads of Example 1, $\boxed{\mathrm{G}}\,\phi$ amounts to the following:

- in the state monad: successful execution of ϕ from any initial state yields $\top$;
- in the non-determinism monad: ϕ yields at most the value $\top$ (or none at all)
- in the exception monad: ϕ yields $\top$ whenever it terminates normally.
- in the interactive input monad: the value eventually produced by ϕ after some combination of inputs is always $\top$;
- in the non-deterministic state monad: execution of ϕ from any initial state yields at most the value $\top$.

For meta-proofs about the Hoare logic, we require an auxiliary calculus (Fig. 5) for judgements of the form $[\bar{x} \leftarrow \bar{p}]_G\, \phi$, which intuitively state that the formula $\phi : \Omega$, which may contain $\bar{x}$, holds after $\bar{x} \leftarrow \bar{p}$. The idea is to shove all state-dependence to the outside, so that the usual logical rules apply to the remaining part. Formally, $[\bar{x} \leftarrow \bar{p}]_G\, \phi$ abbreviates

$$\boxed{G}\ \text{do}\ \ \bar{x} \leftarrow \bar{p};\ \ \text{ret}\, \phi$$

The set of free variables of p is denoted by $FV(p)$. The calculus is sound:

Theorem 8. *If $[\bar{y} \leftarrow \bar{q}]_G\, \psi$ is deducible from $[\bar{x} \leftarrow \bar{p}]_G\, \phi$ by the rules of Fig. 5, then $([\bar{x} \leftarrow \bar{p}]_G\, \phi) \Rightarrow ([\bar{y} \leftarrow \bar{q}]_G\, \psi)$ holds in the internal logic.*

$$\textbf{(mp)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi_i,\ i = 1, \ldots, n \qquad \phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi}{[\bar{x} \leftarrow \bar{p}]_G\, \psi} \qquad \textbf{(eq)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi \Rightarrow q_1 = q_2}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}]_G\, \phi \Rightarrow \psi} \frac{}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}]_G\, \phi \Rightarrow \psi}$$

$$\textbf{(app)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi \qquad y \notin FV(\phi)}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q]_G\, \phi} \qquad \textbf{(pre)}\ \frac{[\bar{y} \leftarrow \bar{q}]_G\, \phi \qquad x \notin FV(\phi)}{[x \leftarrow p; \bar{y} \leftarrow \bar{q}]_G\, \phi} \qquad (\eta)\ \frac{}{[x \leftarrow \text{ret}\, a]_G\, x = a}$$

$$\textbf{(ctr)}\ \frac{[\ldots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}]_G\, \phi}{[\ldots; y \leftarrow (\text{do}\ \ x \leftarrow p;\ \ q); \bar{z} \leftarrow \bar{r}]_G\, \phi}\ (x \notin FV(\phi) \cup FV(\bar{r}))$$

Fig. 5. The auxiliary calculus

We can now define and give a semantics to Hoare triples:

Definition 9. A *Hoare triple*, written $\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi\}$, consists of a program sequence $\bar{x} \leftarrow \bar{p}$, a precondition $\phi : T\Omega$, and a postcondition $\psi : T\Omega$ (which may contain $\bar{x}$), where ϕ and ψ are side-effect free. This abbreviates the formula

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi]_G\, a \Rightarrow b.$$

The fact that Hoare triples as just defined mention program *sequences* (rather than just programs) reflects the need to actually reason about results of computations, including intermediate results, as opposed to just about state changes as in the traditional case.

Example 10. A Hoare triple $\{\phi\}\ x \leftarrow p\ \{\psi\}$ holds

- in the state monad iff, whenever ϕ holds in a state s, then ψ holds for x after successful execution of p from s with result x;
- in the non-determinism monad iff, whenever ϕ holds possibly, then ψ holds for all possible results x of p;
- in the exception monad iff, whenever ϕ holds and p terminates normally, returning x, then ψ holds for x;
- in the interactive input monad iff, whenever ϕ holds and p returns x after reading some sequence of inputs, then ψ holds for x.
- in the non-deterministic state monad iff, whenever ϕ holds possibly in a state s, then ϕ holds after execution of p for all possible results x.

Remark 11. It is clear that the main application domain of Hoare triples are monads where some sort of state is involved that gives meaning to notions of 'before' and 'after'. However, as can be seen in the combination of the state monad with non-determinism, considering Hoare-triple for 'stateless' monads does make sense inasmuch as it provides for a separation of concerns.

Lastly, one can capture determinacy at least for side-effect free programs:

Definition 12. A side-effect free program p is *deterministically side-effect free* if

$$[x \leftarrow p; y \leftarrow p]_G\, x = y \qquad (\text{shorthand: } \mathit{dsef}(p)).$$

Stateless programs are deterministically side-effect free. In most of the running examples, all side-effect free programs are deterministically side-effect free, with the unsurprising exception of the monads where non-determinism is involved. In these cases, a side-effect free program is deterministically side-effect free iff it is deterministic.

Having defined an interpretation of Hoare triples as formulas in the internal logic of HasCasl, we can now proceed to establish a set of monad-independent Hoare rules as shown in Fig. 6; the rules are lemmas in the internal logic.

The rule (wk) uses the notation $\phi \Rightarrow_T \psi$. This is just syntactic sugar for the Hoare triple $\{\phi\}\ \{\psi\}$; hence, (wk) is really a special case of the sequential rule (seq). The decoding of $\phi \Rightarrow_T \psi$ can be simplified to

$$(\text{do } a \leftarrow \phi; b \leftarrow \psi;\ \text{ret}\, a \Rightarrow b) = \text{ret}\, \top.$$

The rule (dsef) applies in particular to stateless programs $p = \text{ret}\, a$, for which the postcondition simplifies to $x = a$. Although the classical Hoare calculus does not require the usual introduction and elimination rules for logical connectives, such rules are sometimes convenient (see the example below); we have included introduction rules for conjunction and disjunction. Here, $\phi \wedge \psi$ abbreviates

$$\text{do } a \leftarrow \phi; b \leftarrow \psi;\ \text{ret}\, a \wedge b,$$

similarly for other logical connectives.

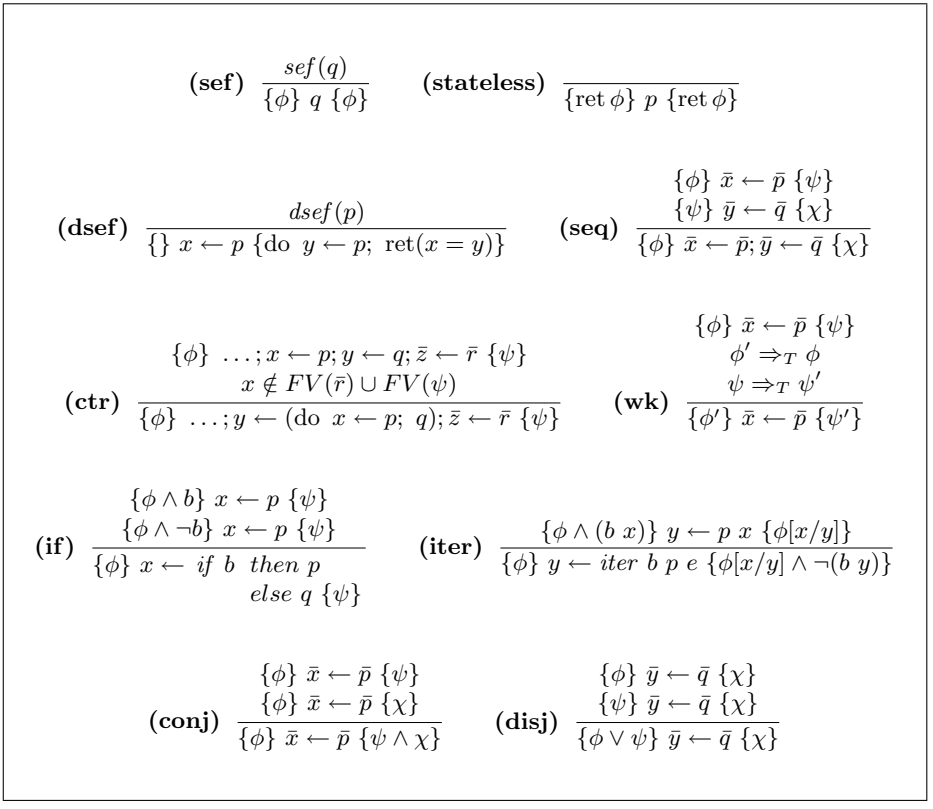

Fig. 6. The generic Hoare calculus

One typical Hoare rule that is missing here is the assignment rule; this rule only makes sense in a more specialized context where some sort of store is present. An example of an extension of the calculus by specialized rules for a particular monad is presented below.

As examples of rules for user-defined generic control-structures, we have included an invariant rule for the iteration construct introduced in Section 3 and a rule for an if-then-else construct defined in the obvious way; rules for similar control structures such as while work as usual. In the pre- and postconditions, boolean values b are implicitly converted to Ω as $b = \mathit{true}$, and *formulas of type Ω are implicity cast to $T\Omega$* via ret when needed.

The rules of the calculus are sound for arbitrary (cpo-)monads:

Theorem 13. *If a Hoare triple is derivable in a cpo-monad (monad) by the rules of Fig. 6 (excluding (iter)), then the corresponding formula is derivable in the internal language.*

It is clear that completeness can only be expected in combination with suitable monad-specific rules; e.g., the calculus becomes the usual (complete) Hoare cal-

culus when extended with an assignment rule specific to the store monad. In this sense, the calculus may be regarded as a generic framework for computational deduction systems.

5 Example: Reasoning about Dynamic References

We now apply the general machinery developed so far to the (slightly extended) domain of the classical Hoare calculus, namely states consisting of creatable and 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 constructur 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 Fig. 7. We use the shorthand $\phi(*r)$ for do $x \leftarrow read\ r;\ \phi(x)$. Note that with this notation, $\mathrm{ret}(x = *r)$ is *not* stateless. Also note the difference between $\mathrm{ret}(r = s)$ (equality of references, a stateless formula) and $\mathrm{ret}(*r = *s)$ (equality of contents, a stateful formula). Moreover, recall that ret is inserted implicitly where needed.

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

spec DYNAMICREFERENCE = REFERENCE **then**
  var a, b : *Type*
  op new : $a \xrightarrow{c} R(Ref\ a)$
forall x, y : a; r : $Ref\ a$; p : $R\ b$
• $\{\}\ r \leftarrow new\ x\ \{x = *r\}$ *%(read-new)%*
• $\{x = *r\}\ s \leftarrow new\ y\ \{\neg r = s \Rightarrow x = *r\}$ *%(read-new-other)%*
• $\{\}\ r \leftarrow new\ x;\ p;\ s \leftarrow new\ y\ \{\neg r = s\}$ *%(new-distinct)%*

Fig. 7. Specification of the reference and the dynamic reference monad

The axiomatization provides all that is really necessary in order to reason about references, i.e. one does not need to rely 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. The specification of *dynamic* references additionally provides an operation *new* for dynamically creating new references. *read-new* states that after initializing a reference, we can read the initial value. Moreover, creation of new references does not change the values of other references (*read-new-other*). Finally, two newly created references are distinct (*new-distinct*). Note that we do not say anything about reading from references that have not been created yet.

Starting from this axiomatization, properties such as

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{\neg r = s \wedge x = *r \wedge y = *s\} \quad (1)$$

are easily established using the Hoare rules of Fig. 6.

Another example is the nondeterminism monad, shown in Fig. 8. While *fail* yields no result and hence everything follows from it, *chaos* yields any result and hence nothing can be said about it. $[]$ is nondeterministic choice (i.e. takes the union of value sets), and *sync* synchronises two nondeterministic values (i.e. takes the intersection of value sets).

spec Nondeterminism = CpoMonad **then**
  var a : *Cpo*
  ops *fail*, *chaos* : $N\ a$;
    $\_\_[]\_\_, \_\_sync\_\_ : N\ a \xrightarrow{c} N\ a \xrightarrow{c} N\ a$
forall $x : a;\ p, q : N\ a;\ \varphi, \psi : N\Omega;\ \chi_1, \chi_2 : a \to N\Omega$
• $\{\}\ fail\ \{\psi\}$ *%(fail)%*
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p[]q\ \{\chi_1 x \vee \chi_2 x\}$ *%(join)%*
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p\ sync\ q\ \{\chi_1 x \wedge \chi_2 x\}$ *%(sync)%*

Fig. 8. The nondeterminism monad

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 (Fig. 9).

As an example, we prove the partial correctness of Dijkstra's nondeterministic version of Euclid's algorithm for computing the greatest common divisor [4] within this monad. Let *euclid* be the program sequence (over $NR\ Int$)

$r \leftarrow new\ x;$
$s \leftarrow new\ y;$
$while\ ret(\neg *r == *s)$
  $(if\ ret(*r > *s)\ then\ r := *r - *s\ else\ fail$
  $[]$
  $if\ ret(*s > *r)\ then\ s := *s - *r\ else\ fail)$

spec NondeterministicDynamicReference =
DynamicReference **with** $R \mapsto NR$
and Nondeterminism **with** $N \mapsto NR$

Fig. 9. The nondeterministic dynamic reference monad

Assuming that we have some specification of arithmetic, including *gcd* specified to be the greatest common divisor function, we now will try to prove

$$\{\}\ euclid\ \{*r = gcd(x, y)\}.$$

We proceed as follows. Using (dsef), (sef), (seq), (stateless) and (conj), we can show

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\} \\ u \leftarrow read\ r; v \leftarrow read\ s \\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s \wedge u = *r \wedge v = *s\}.\end{array}$$

By arithmetic reasoning and (wk), we obtain

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\} \\ u \leftarrow read\ r; v \leftarrow read\ s \\ \{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\}.\end{array} \tag{3}$$

By (stateless), (read-write), (read-write-other), and (conj), we can show

$$\begin{array}{l}\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\} \\ r := u - v \\ \{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s \wedge u - v = *r\}.\end{array}$$

By arithmetic reasoning and (wk), we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\} \\ r := u - v \\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{array}$$

By (seq) with (3) and noting that $r := {}^*r - {}^*s$ is shorthand for $u \leftarrow read\ r; v \leftarrow read\ s; r := u - v$, we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\} \\ r := {}^*r - {}^*s \\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{array}$$

By (fail), we have

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge \neg {}^*r > {}^*s\} \\ fail \\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{array}$$

Hence by (if)

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\ \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\end{array}$$

In an entirely analogous way, we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{array}$$

From these, together with (join) and (wk), we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\ \quad \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{array}$$

Applying (wk) and (iter) leads to

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\ \mathit{while}\ \ \mathrm{ret}(\neg *r == *s)\\ (\quad \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ \ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail})\\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge *r == *s\}.\end{array}$$

Using the arithmetic fact that $gcd(x, x) = x$, by (wk) we obtain

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\ \mathit{while}\ \neg *r == *s\\ (\quad \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail} \qquad (4)\\ \ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail})\\ \{*r = gcd(x, y)\}.\end{array}$$

From (1) above, we get by arithmetic reasoning and (wk)

$$\{\}\ r \leftarrow \mathit{new}\ x; s \leftarrow \mathit{new}\ y\ \{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}, \quad (5)$$

and the result now follows by applying (seq) to (4) and (5).

6 Conclusion and Future Work

We have generalized Moggi's Hoare calculus for the state monad to a *monad-independent* Hoare calculus. To this end, we have used an extension of the wide-spectrum language HasCasl with Haskell-style imperative programming via monads, which in terms of additional language features required essentially no more than a rather straightforward encorporation of constructor classes.

We have illustrated this approach by several example monads, some of which we have described *axiomatically*, rather than via an implementation as in Haskell. Specific monads come with specific extensions to the generic Hoare calculus; it is even possible to axiomatize a monad by means of Hoare triples. Further work will include the production of a library of such monad specifications, thus providing a broad basis for formal reasoning about imperative functional programs. Moreover, the examples suggest that general results about monad combination [9] bear some relation to the combination of monad-specific Hoare calculi; ways in which axiomatizations of more complex monads can be compositionally obtained those of simpler ones are the subject of further investigation. Another interesting topic is the relation to the monad independent aspects of the testing tool QuickCheck [1].

The traditional Hoare calculus can be embedded into propositional dynamic logic [6], which allows for rather more flexibility. However, this is expected to work with monads only in special cases, since unlike as with Moggi's evaluation logic, the computation needs to be split (i.e. the 'state' needs to be duplicated) for evaluations of compound formulas, e.g. conjunctions such as $[p]\,\phi \wedge [q]\,\psi$ where the evaluation of the second conjunct requires 'resetting the state' (here, $[p]\,\phi$ reads 'ϕ holds after exection of p'). Several of the computationally relevant monads, among them the usual state monad, do admit such a state duplication; a general axiomatization of this concept is forthcoming.

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.

References

[1] K. Claessen and J. Hughes, *Testing monadic code with QuickCheck*, Haskell Workshop, ACM, 2002, pp. 65–77.
[2] CoFI, *The Common Framework Initiative for algebraic specification and development, electronic archives*, `http://www.brics.dk/Projects/CoFI`.
[3] CoFI Language Design Task Group, CASL – *The CoFI Algebraic Specification Language – Summary, version 1.0*, Documents/CASL/Summary, in [2], July 1999.
[4] E. W. Dijkstra, *A discipline of programming*, Prentice Hall, 1976.
[5] J.-C. Filliâtre, *Proof of imperative programs in type theory*, Types for Proofs and Programs, LNCS, vol. 1657, Springer, 1999, pp. 78–92.
[6] R. Goldblatt, *Logics of time and computation*, CSLI, 1992.
[7] *The Haskell mailing list*, `http://www.haskell.org/mailinglist.html`, May 2002.
[8] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, *Haskell 98: A non-strict, purely functional language*, (1999), http://www.haskell.org/onlinereport.
[9] C. Lüth and N. Ghani, *Monads and modularity*, Frontiers of Combining Systems, LNAI, vol. 2309, Springer, 2002, pp. 18–32.

[10] S. Mac Lane, *Categories for the working mathematician*, Springer, 1997.
[11] E. Moggi, *Notions of computation and monads*, Inform. and Comput. **93** (1991), 55–92.
[12] P. D. Mosses, Casl*: A guided tour of its design*, Workshop on Abstract Datatypes, LNCS, vol. 1589, Springer, 1999, pp. 216–240.
[13] F. Regensburger, *HOLCF: Higher order logic of computable functions*, Theorem Proving in Higher Order Logics, LNCS, vol. 971, 1995, pp. 293–307.
[14] L. Schröder and T. Mossakowski, HasCasl*: Towards integrated specification and development of Haskell programs*, Algebraic Methodology and Software Technology, LNCS, vol. 2422, Springer, 2002, pp. 99–116.
[15] Philip Wadler, *How to declare an imperative*, ACM Computing Surveys **29** (1997), 240–263.