Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 191 (2007) 73–83

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Building Verification Condition Generators by Compositional Extension

A. J. van Leeuwen[1]

Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands

Abstract

Current mechanizations of programming logics are often in the form of verification condition generators. These front ends to a prover translate a program and assertions into conditions that state that the program fulfills its assertions. Traditional verification condition generators are monolithic encapsulations of a programming language's semantics. This makes it hard to build such verification generators when designing a new language, or when extending a language.

We propose a more compositional method of building verification condition generators, using ideas from monadic denotational semantics and from generic programming. Our technique allows us to extend an existing verification condition generator to handle new language constructs, but also to add extensions at another level, such as the ability to generate validation traces. We explain the technique through an example, extending a simple while language with a construct for exception handling. This construct not only needs an extension to the logic, but also a change of its structure.

Keywords: Hoare logic, monadic logic, programming language semantics

1 Introduction

Development and maintenance of implementations of realistic programming logics is known to be hard. There are few methods of dealing with the changes that are incurred by changing the language or the addition of features. Implementing these changes is a dangerous and error prone process that can easily introduce inconsistencies in the logic.

Logics underlying imperative languages are usually syntax driven and implemented as a recursive function over the target program [5]. Although this is a straightforward method of implementing these logics, it leads to programs that cannot be altered or extended easily.

We demonstrate a technique that enables us to change and extend the implementation of a logic in a modular way, that is without directly tampering with the

[1] Email: arthurvl@cs.uu.nl

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.08.052

$$\begin{array}{lll} \mathit{Stmt} & \rightarrow & \mathit{Variable} := \mathit{Expr} \\ & | & \mathit{Stmt}; \mathit{Stmt} \\ & | & \mathbf{if}\ \mathit{Expr}\ \mathbf{then}\ \{\mathit{Stmt}\}\ \mathbf{else}\ \{\mathit{Stmt}\} \\ & | & \mathbf{inv}\ \mathit{Expr}\ \mathbf{while}\ \mathit{Expr}\ \mathbf{do}\ \{\mathit{Stmt}\} \end{array}$$

Figure 1. Simple imperative language L

code of the existing implementation of a logic. Our approach has several advantages. First and foremost is the advantage that it is safer. Second, it allows enabling and disabling alterations at will. Thus we can always fall back on the existing implementation. Third, it enables us to easily create a set of related partial logics, each of which can be used separately for light weight verification.

Our technique uses ideas from generic programming and monadic denotational semantics to represent a syntax driven logic. Generic programming abstracts from algebraic datatypes, and these techniques can thus be used to abstract from the actual implementation of the abstract syntax used to represent programs. The main use we make is that of a generic fold, encapsulating the recursion over the datatype, which we assume to be easily specified. This allows us to decouple the recursion scheme from the calculation of the semantics.

Monadic denotational semantics has been introduced as a method of separating concerns in denotational semantics [11,1]. The key idea is to choose an abstraction for the domain of the semantics such that different computations in that domain can be combined in a standard fashion. By relying on that combination method, independence of the domain is achieved. Thus, given a monadic domain, we can hide those aspects that do not affect a specific part of the logic. This allows us to keep the code of the base logic unchanged despite adding to the underlying structure, which normally affects the implementation of each rule in the logic.

2 Implementing a logic

Consider the simple imperative language L shown in Figure 1. For simplicity, we assume that all statements terminate. The syntax **inv** i **while** g **do** S is for a simple while loop where i is a candidate invariant specified by the programmer.

We will specify a verification condition generator for the logic of this language in terms of a predicate transformer logic [3]. A predicate transformer is a function that takes a predicate and a statement in a language, and returns a predicate transformed according to the semantics of that statement. In the best known instantiation, this function takes postconditions into preconditions, such that $p = \text{pre}\ P\ q \Rightarrow \{p\}P\{q\}$ where $\{p\}P\{q\}$ is a Hoare-triple stating that pre- and postcondition p and q are valid for statement P. Figure 2 shows rules that specify how the value of pre can be computed. Note that the value is only valid if the additional conditions that occur in the calculation for the while rule also hold.

This specification is syntax driven: for each production rule in the syntax, there is one rule specifying how to calculate pre. Therefore, calculating the value of pre (S) q given a statement S and a post-condition q is a matter of recurring over

$$\text{pre}\ (x\ :=\ e)\ q\ =\ q[e/x]$$

$$\text{pre}\ (S_1;\ S_2)\ q\ =\ \text{pre}\ (S_1)\ (\text{pre}\ (S_2)\ q)$$

$$\text{pre}\ (\mathbf{if}\ g\ \mathbf{then}\ \{S_1\}\ \mathbf{else}\ \{S_2\})\ q\ =\ \mathit{if}\ g\ \mathit{then}\ \text{pre}\ (S_1)\ q\ \mathit{else}\ \text{pre}\ (S_2)\ q$$

$$\frac{\begin{array}{l} p\ =\ \text{pre}\ (S)\ i \\ \vdash\ i \wedge \neg g \Rightarrow q \\ \vdash\ i \wedge g \Rightarrow p \end{array}}{\text{pre}\ (\mathbf{inv}\ i\ \mathbf{while}\ g\ \mathbf{do}\ \{S\})\ q\ =\ i}$$

Figure 2. Derivation rules for pre for language L

the structure of S. Some rules emit extra verification conditions that should also be collected, thereby resulting in the term *verification condition generator* for the implementation of pre. The straightforward implementation of such a verification condition generator computes both the precondition and collects the verification conditions. Let us name the function *pvcg*. Given a program s and a post-condition q, *pvcg* s q would return a tuple (p, vcs) where vcs is a list of verification conditions and p is pre (s) q. Its type therefore is $pvcg :: Stmt \to Expr \to (Expr, [Expr])$. An example in the functional programming language Haskell is in Figure 3. We will later silently reuse and extend the datatype *Stmt*, but not specify *Expr*. The logical connectives are to be read as constructors of the type *Expr*.

Although straightforward, such code is not easily changed. Turning it into an explicit algebra over the provided algebraic data type makes changes easier. The way to implement the algebra as a fold over the abstract syntax can be easily seen: it involves making a separate function for each alternative in the datatype representing the abstract syntax, and passing those functions as arguments to the fold. The set of passed functions clearly specifies an algebra over the language, with the nonterminals in the language being the sorts, and the alternatives inducing the operations. Applying the generic fold gives us the unique homomorphism between the initial algebra and the algebra specified.

This decouples the recursion over the syntax from the calculation of the pre-condition and the other verification conditions. Unfortunately the code then still strongly depends on the exact structure of the domain. We can improve on that.

Looking at the type of the implementation, $Stmt \to Expr \to (Expr, [Expr])$, we see that it takes a statement and an expression and returns a new expression, if we ignore the fact that we also collect other verification conditions alongside the calculation. However, using monads, we can abstract from such collection.

Moggi introduced the use of monads in programming in recognizing that monads can represent a computation over a value [11], such as the process of collecting conditions alongside transforming a predicate. Within a monadic setting, values can be injected into computations, and monadic computations can be bound together to form new computations. We denote these operations on monads with *return* to inject values into computations, and $\gg\!=$ to sequentially combine a computation over a value with a function taking that value and returning a new computation. The type of *return* is $\mathit{return} :: a \to m\ a$ and that of $\gg\!=$ is $\gg\!= :: ma \to (a \to m\ b) \to m\ b$, where $m\ \ a$ represents the type of a monad m over values of type a. In these operations, *return* is a left- and right-identity of $\gg\!=$, that is $(\mathit{return}\ \ x) \gg\!= f \equiv f\ \ x$ and $m \gg\!= \lambda y \to \mathit{return}\ \ y \equiv m$. Furthermore, $\gg\!=$ is associative, that is $(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ \ x \gg\!= g)$. Most monads provide more operations than just *return* and $\gg\!=$, but the associativity and identity laws allow us to still combine them. The extra operations provide the functionality to collect verification conditions, for example.

We can state the implementation's type as $\mathit{Stmt} \to \mathit{Expr} \to m\ \mathit{Expr}$, where the monad m abstractly represents the act of collecting the side conditions. Using a fold and monads, the code for the verification condition generator looks like that in Figure 4.

We assume a function *record* that records verification conditions in values of type $m\ ()$, that is, of a monad over the unit type. By using the abstract sequential combination of computations as opposed to explicit threading of values, we need not specify the exact monad m. We merely need the associativity of subsequent *record* computations.

The structure of the functions defined in the example is simple. For each constructor in the datatype *Stmt* we have a function that, given the predicate transformers for recursive occurrences of *Stmt* and the other arguments to the construc-

```
data Stmt = String := Expr            -- Variable := Expr
          | Stmt :> Stmt              -- Stmt; Stmt
          | IfElse Expr Stmt Stmt     -- if Expr then {Stmt} else {Stmt}
          | While Expr Expr Stmt      -- inv Expr while Expr do {Stmt}
pvcg' :: Stmt → Expr → (Expr, [Expr])
pvcg' (x := e)           q           = (subst (x, e) q, [ ])
pvcg' (s1 :> s2)         q           = (p, vcs2 ++ vcs1)
          where          (p', vcs2)  = pvcg' s2 q
                         (p, vcs1)   = pvcg' s1 p'
pvcg' (IfElse e s1 s2)   q           = (e → s1q ∧ ¬ e → s2q, vcs1 ++ vcs2)
          where          (s1p, vcs1) = pvcg' s1 q
                         (s2p, vcs2) = pvcg' s2 q
pvcg' (While i g body)   q           = (i, c1 : c2 : vcs')
          where          c1          = i ∧ ¬ g → q
                         c2          = i ∧ g → p
                         (p, vcs')   = pvcg' body i
```

Figure 3. Simple implementation of rules in Figure 2

$$
\begin{array}{lll}
\mathit{pvcg\_Assign}\ x\ e & = \lambda q \rightarrow & \mathit{return}\ (\mathit{subst}\ (x, e)\ q) \\
\mathit{pvcg\_Seq}\ \mathit{p\_s1}\ \mathit{p\_s2} & = \lambda q \rightarrow & \mathit{p\_s2}\ q \gg\!= \lambda p' \rightarrow \\
 & & \mathit{p\_s1}\ p' \gg\!= \lambda p \rightarrow \\
 & & \mathit{return}\ p \\
\mathit{pvcg\_IfElse}\ g\ \mathit{p\_s1}\ \mathit{p\_s2} & = \lambda q \rightarrow & \mathit{p\_s1}\ q \gg\!= \lambda p \rightarrow \\
 & & \mathit{p\_s2}\ q \gg\!= \lambda p' \rightarrow \\
 & & \mathit{return}\ (g \rightarrow p \wedge \neg\, g \rightarrow p') \\
\mathit{pvcg\_While}\ i\ g\ \mathit{p\_body} & = \lambda q \rightarrow & \mathit{p\_body}\ i \gg\!= \lambda p \rightarrow \\
 & & \mathit{record}\ (i \wedge \neg\, g \rightarrow q) \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{record}\ (i \wedge g \quad \rightarrow p) \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{return}\ i
\end{array}
$$

$$
\begin{array}{l}
\mathit{pvcg} = \mathit{fold}_{\mathit{Stmt}}\ (\mathit{pvcg\_Assign}, \\
\qquad\qquad\qquad \mathit{pvcg\_Seq}, \\
\qquad\qquad\qquad \mathit{pvcg\_IfElse}, \\
\qquad\qquad\qquad \mathit{pvcg\_While})
\end{array}
$$

Figure 4. Folded monadic implementation of a vcg for L

tor, returns a function of type $\mathit{Expr} \rightarrow m\ \mathit{Expr}$. Each function is composed of computations in a monad, bound together with $\gg\!=$. As the right-hand argument of $\gg\!=$ is a function taking a value and returning a computation, we explicitly state λ-abstractions. In some cases the value is not interesting, such as in the monads resulting from *record*. The notation $\lambda\_$ is for binding to a value but immediately discarding it. The type of the λ-abstractions on the right-hand sides of $\gg\!=$ in the example is either $\mathit{Expr} \rightarrow m\ \mathit{Expr}$ or $() \rightarrow m\ \mathit{Expr}$, with the latter only for the λ-abstractions that discard their binding.

Note that although the code is not shorter, the structure of the rules in Figure 2 is more closely followed, and there is less visible administrative overhead. Also note that we could have written $\mathit{pvcg\_Seq}\ \mathit{p\_s1}\ \mathit{p\_s2} = \lambda q \rightarrow \mathit{p\_s2}\ q \gg\!= \mathit{p\_s1}$, which means the same, but hides that the result of the computation of *p_s2* is fed into *p_s1*, whose result is then returned.

3 Modifying a logic

Modifying a straightforward implementation as a recursive function involves changing the entire function. Modifying the implementation in Figure 4 involves changing only those functions affected. Furthermore, as we abstract from the structure of the domain in Figure 4, it is possible to change that independently.

Suppose we add exception handling to our language by adding the syntactical constructs **raise** and **try** {*Stmt*} **catch** {*Stmt*}. This involves adding appropriate alternatives to the *Stmt* datatype, and the need to write new rules to handle these new constructs. As the generic fold is independent of the exact alternatives present in the datatype, we need not rewrite it. We assume that expressions do not in themselves raise exceptions.

Now, if we naively impose that no valid program may execute **raise**, we can

$$
\begin{array}{l}
\mathit{pvcg\_Raise}' \qquad\qquad\qquad\quad = \lambda q \rightarrow \mathit{return\ false} \\
\mathit{pvcg\_TryCatch}'\ \mathit{p\_s1}\ \mathit{p\_s2} = \lambda q \rightarrow \mathit{p\_s1}\ q \\
\mathit{pvcg} = \mathit{fold}_{\mathit{Stmt}}\ (\mathit{pvcg\_Assign}, \\
\qquad\qquad\qquad\quad \mathit{pvcg\_Seq}, \\
\qquad\qquad\qquad\quad \mathit{pvcg\_IfElse}, \\
\qquad\qquad\qquad\quad \mathit{pvcg\_While}, \\
\qquad\qquad\qquad\quad \mathit{pvcg\_Raise}', \\
\qquad\qquad\qquad\quad \mathit{pvcg\_TryCatch}')
\end{array}
$$

Figure 5. Initial extension of pvcg from L to L_1

simply take the precondition of a **try** $\{$ *s1* $\}$ **catch** $\{$ *s2* $\}$ block to be that of *s1*. The resulting logic is implemented in Figure 5.

This naive code is not entirely satisfactory, but it already shows reuse of the existing code. A more satisfactory logic actually allows exceptions, not only their syntax, and handles them. This requires making a distinction between the postcondition for normal execution, and the postcondition for exceptional execution, and a method of specifying both. Figure 6 shows a set of derivation rules to be used for the extended language. It changes the predicate transformer so that it does not just take predicates, but tuples of predicates, one for the postcondition for normal execution and one for the postcondition for exceptional execution. In this specification of the derivation rules adding exception handling changes all previously existing rules. This is precisely what we wish to avoid.

Our technique can do that, with ease! Assume our computational monad m not only records verification conditions, but also stores the current postcondition for the exceptional case, with functions *getPostE* and *setPostE* to access that postcondition. Then we can write our verification condition generator as in Figure 7. As can be seen, we have needed no additional changes to the existing code at all, even though we have changed the domain. Note that adding the possibility for expressions to raise an exception involves adding invocations to *getPostE* in the appropriate places, but that such changes cannot be avoided.

4 Constructing the domain

We have left the method of specifying the monad encapsulating the domain implicit. One way of specifying this monad is by directly choosing one. In our example, we need a monad that allows us to record predicates, and that keeps a distinct state. A direct specification of such a monad can be found in Figure 8. It specifies exactly the operations we need, and can be substituted for the monad m in both Figure 4 and 7. The new type *RecExc a* is specified as a tuple consisting of a list of expressions, an expression and a value of type a. The additional constructor is a technical requirement allowing one to distinguish the type *RecExc a* from the type $([\mathit{Expr}], \mathit{Expr}, a)$. For this type *RecExc a* the functions *return* and $\gg\!=$ are then defined. As $\gg\!=$ is an infix operator, it is enclosed in parentheses for the prefix definition.

Although this works, it involves replacing the entire monad upon changes. Much nicer would be the ability to just add aspects to the existing monad. Traditional approaches of combining monads are the use of distributive laws over monads [7] or the use of monad transformers [4,10]. We have chosen to use another approach, that of monadic coproducts, as described by Luth and Ghani [9].

A monad can be thought of as a computational layer over a certain value. If we want to combine computational actions from different monads, we can put multiple computational layers over a value. However, when choosing one particular layering of monads for the combined monad, in general one cannot directly find the $\gg\!=$ function. In a monadic coproduct we do not choose one particular layering but rather all possible layerings of two monads. Unfortunately, this denotes a largely redundant computational structure, as one is interested in computational actions rather than in layers of possible computational actions. The trick is to identify the actual computational actions, and quotient the coproduct's structure so that each actual computational action is only represented once in the coproduct.

Formally, the coproduct $m \oplus n$ of two monads m and n is just the coproduct in the category of monads. The coproduct has these useful properties: there are

$$\text{pre}\,(x\ :=\ e)\,(q,\ q') \;=\; (q[e/x],\ q')$$

$$\text{pre}\,(S_1;\ S_2)\,(q,\ q') \;=\; (\text{pre}\,(S_1)\,(\text{pre}\,(S_2)\,q),\ q')$$

$$\text{pre}\,(\textbf{if}\ g\ \textbf{then}\ \{S_1\}\ \textbf{else}\ \{S_2\})\,(q,\ q') \;=\; (\textit{if}\ g\ \textit{then}\ \text{pre}\,(S_1)\,q\ \textit{else}\ \text{pre}\,(S_2)\,q\ ,q')$$

$$\frac{\begin{array}{l} p \;=\; \text{pre}\,(S)\,i \\ \vdash\ i \wedge \neg g \Rightarrow q \\ \vdash\ i \wedge g \Rightarrow p \end{array}}{\text{pre}\,(\textbf{inv}\ i\ \textbf{while}\ g\ \textbf{do}\ \{S\})\,(q, q') \;=\; (i,\ q')}$$

$$\text{pre}\,(\textbf{raise})\,(q,\ q') \;=\; (q',\ q')$$

$$\frac{\begin{array}{l} p \;=\; \text{pre}\,(S_1)\,(q,\ p') \\ p' \;=\; \text{pre}\,(S_2)\,(q,\ q') \end{array}}{\text{pre}\,(\textbf{try}\ \{S_1\}\ \textbf{catch}\ \{S_2\})\,(q,\ q') \;=\; (p,\ q')}$$

Figure 6. Derivation rules for pre for extended language

$$
\begin{array}{lll}
\mathit{pvcg\_Raise} & = \lambda q \rightarrow & \mathit{getPostE} \\
\mathit{pvcg\_TryCatch}\ \mathit{p\_s1}\ \mathit{p\_s2} & = \lambda q \rightarrow & \mathit{getPostE} \gg\!= \lambda q' \rightarrow \\
 & & \mathit{p\_s2}\ q \gg\!= \lambda p' \rightarrow \\
 & & \mathit{setPostE}\ p' \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{p\_s1}\ q \gg\!= \lambda p \rightarrow \\
 & & \mathit{setPostE}\ q' \gg\!= \lambda\_ \rightarrow \\
 & & \mathit{return}\ p
\end{array}
$$

$$
\begin{array}{l}
\mathit{pvcg} = \mathit{fold}_{\mathit{Stmt}}\ (\mathit{pvcg\_Assign}, \\
\qquad\qquad\quad \mathit{pvcg\_Seq}, \\
\qquad\qquad\quad \mathit{pvcg\_IfElse}, \\
\qquad\qquad\quad \mathit{pvcg\_While}, \\
\qquad\qquad\quad \mathit{pvcg\_Raise}, \\
\qquad\qquad\quad \mathit{pvcg\_TryCatch})
\end{array}
$$

Figure 7. Full pvcg for L_1 using code from L

monad morphisms, natural transformations commuting with *return* and *bind*, called *inl* and *inr* taking m respectively n into $m \oplus n$, and for any other monad r and two given monad morphisms $\alpha\ : m \rightarrow r$ and $\beta\ : n \rightarrow r$ there is a unique monad morphism $\mathit{coproduct}(\alpha, \beta) : m \oplus n \rightarrow r$ such that $\mathit{coproduct}(\alpha, \beta) \circ \mathit{inl} = \alpha$ and $\mathit{coproduct}(\alpha, \beta) \circ \mathit{inr} = \beta$.

To actually construct a monadic coproduct, one needs finitary layered monads, that is monads for which the behaviour on infinite objects is determined by its behaviour on finite objects, and for which it is possible to determine if the monadic value is in the range of the *return* function. The latter requirement allows us to distinguish between layers in which an action has been performed and layers which can be safely ignored, and therefore allows us to actually calculate the required quotient. Fortunately, many practical monads have these properties.

Given two finitary layered monads, we specify the coproduct as a datatype encapsulating the monads separate from the values under the monads. That is, two monads m and n can be added together to form a coproduct *Plus m n* that is defined as

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Plus}\ m\ n\ a = \mathit{PlusM}\ (m\ (\mathit{Plus}\ m\ n\ a)) \\
\qquad\qquad\qquad\quad\ \mid\ \mathit{PlusN}\ (n\ (\mathit{Plus}\ m\ n\ a)) \\
\qquad\qquad\qquad\quad\ \mid\ \mathit{PlusVar}\ a
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{newtype}\ \mathit{RecExc}\ a = \mathit{RecExc}\ ([\mathit{Expr}], \mathit{Expr}, a)} \\
\mathit{return}\ x & = \mathit{RecExc}\ ([\,], \mathit{tt}, x) \\
(\gg\!=)\ (\mathit{RecExc}\ (\mathit{xs}, p, v))\ f & = \mathit{RecExc}\ (\mathit{xs} +\!\!+ \mathit{ys}, p, v') \\
\quad \mathbf{where}\ (\mathit{RecExc}\ (\mathit{ys}, p', v')) = f\ v & \\
\mathit{record}\ p & = \mathit{RecExc}\ ([p], \mathit{tt}, ()) \\
\mathit{getPostE}\ (\mathit{RecExc}\ (\mathit{xs}, p, v)) & = \mathit{RecExc}\ (\mathit{xs}, p, p) \\
\mathit{setPostE}\ p & = \mathit{RecExc}\ ([\,], p, v)
\end{array}
$$

Figure 8. A monad allowing recording and exceptional state operations

$$
\begin{array}{ll}
\textbf{newtype}\ \mathit{Record}\ a = \mathit{Record}\ ([\mathit{Expr}], a) & \\
\mathit{return}\ x & = \mathit{Record}\ ([\,], x) \\
(\gg\!=)\ (\mathit{Record}\ (\mathit{xs}, x))\ f & = \mathit{Record}\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}, y) \\
\quad \textbf{where}\ (\mathit{Record}\ (\mathit{ys}, y)) & = f\ x \\
\mathit{record}\ p & = \mathit{Record}\ ([p], ())
\end{array}
$$

Figure 9. A simple recording monad

$$
\begin{array}{ll}
\textbf{newtype}\ \mathit{PostE}\ a = \mathit{PostE}\ (a, \mathit{Expr}) & \\
\mathit{return}\ x & = \mathit{PostE}\ (a, \mathit{tt}) \\
(\gg\!=)\ (\mathit{PostE}\ (x, e))\ f & = \mathit{PostE}\ (y, e) \\
\quad \textbf{where}\ (\mathit{PostE}\ (y, e')) & = f\ y \\
\mathit{getPostE}\ (\mathit{PostE}\ (x, e)) & = \mathit{PostE}\ (e, e) \\
\mathit{setPostE}\ e & = \mathit{PostE}\ ((), e)
\end{array}
$$

Figure 10. A monad encapsulating a second postcondition

and in this monad a single layer of a monad m over a value a would be represented using $\mathit{PlusM}\ (m\ (\mathit{PlusVar}\ a))$. For more details as to how to calculate the quotient or implement the *coproduct* function, see the paper by Luth and Ghani [9].

Suppose that we have the monad in Figure 9. This monad is finitary. It also is layered, as a monadic value is in the range of *return* if and only if the list of recorded expressions is empty. It provides a method of capturing side conditions, but not an exceptional postcondition. If we now take *Plus Record Identity* to be our monad, we can see that $\lambda p \rightarrow \mathit{inl} \circ \mathit{record}\ p$ forms the required operation corresponding to the occurrence *record* in Figure 4.

As we mentioned before, for a satisfactory semantics of the exception handling construct we added, we need to add a method of recording the exceptional postcondition. To do so, we take another monad, as in Figure 10. This monad is also finitary and layered. Filling in that monad in the place of *Identity* in the previous coproduct gives us *Plus Record PostE*. With this monad the operations $\mathit{inr} \circ \mathit{getPostE}$ and $\lambda e \rightarrow \mathit{inr} \circ \mathit{setPostE}\ e$ correspond to the required *getPostE* and *setPostE* in Figure 7. Furthermore, the existing operation for *record* is still valid.

The problem that remains is that of choosing a monad r that we map the operations on *Record* and *PostE* to. This monad r is the essential domain of the semantics. In this case we can use a general state carrying monad, with as state a tuple of expressions and an expression, that is $\mathit{State}\ ([\mathit{Expr}], \mathit{Expr})$. This monad has functions *get* and *put*, where *get* extracts the state from the monad, and *put* takes the state and puts it back under a monad over the unit type. Their types are $\mathit{get} :: \mathit{State}\ ([\mathit{Expr}], \mathit{Expr})\ ([\mathit{Expr}], \mathit{Expr})$ and $\mathit{put} :: ([\mathit{Expr}], \mathit{Expr}) \rightarrow \mathit{State}\ ([\mathit{Expr}], \mathit{Expr})\ ()$. The mapping is given in Figure 11. From the given functions *alpha* and *beta* we get $\mathit{coproduct}\ \mathit{alpha}\ \mathit{beta} :: \mathit{Plus}\ \mathit{Record}\ \mathit{PostE}\ a \rightarrow \mathit{State}\ ([\mathit{Expr}], \mathit{Expr})\ a$, which allows us to interpret the values created in the coproduct monad within the chosen essential domain of the semantics. Note that alpha and beta can be defined independently of each other.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Domain}\ e = \mathit{Plus}\ \mathit{Record}\ \mathit{PostE}\ e\\
&\mathit{alpha}\ (\mathit{Record}\ (xs, p)) = \mathit{get} \ggg \mathit{put} \circ (\lambda(ys, q) \to (ys + xs, q))\\
&\mathit{beta}\ (\mathit{PostE}\ (p, e)) \quad\ = \mathit{get} \ggg \mathit{put} \circ (\lambda(ys, q) \to (ys, e))
\end{aligned}
$$

Figure 11. Combining *Record* and *PostE*

5 Research direction

Given this implementation method for verification condition generators, several research paths remain. Not only do we want a more compositional way of implementing a verification condition generator, we also want to formally verify said generator, as for example Homeier and Martin described [5]. One hypothesis we have investigated is that our approach makes it possible to not only reuse existing code, but also to reuse proofs so that proving e.g. soundness for a changed verification condition generator need not be done from scratch. This does indeed seem possible for a simple language, given more care in constructing the underlying monad than exposed in section 4. The question remains if this scales to languages with function abstraction and application.

Another research path is that of combining this method of specifying a logic with the existing methods of building monadic interpreters and compilers, so that when designing a language one can declaratively specify syntax and semantics, and get a complete implementation of the language, including a sound proof system, for free.

6 Related Work

The usefulness of monads for programming language semantics was already realized by Moggi and Cenciarelli [11,1], and then expanded upon by Espinosa, Liang and Hudak [4,8]. Monads were applied to proving properties over programs by Jacobs and Poll [6] and more recently by Schröder and Mossakowski [13].

Jacobs and Poll specify only the essential monad in which to interpret the semantics of Java, solving the problem that we discuss in the beginning of section 4 in the direct fashion. They do not specify how to deal with changes to the language however.

Our approach differs from the one by Schröder and Mossakowski in that instead of specifying a logic that is independent of the underlying monad, we use the monad to implement a logic. That is, our logic is built using monadic actions, and therefore the reasoning in the logic depends heavily on the exact monad used, whereas the logic of Schröder and Mossakowski is parameterized on the monad, and reasoning is valid in all possible monads.

The use of algebras and folds in our approach is reminiscent of attribute grammars [12]. For our purposes we would need a method of introspection on the grammar rules, so that we can perform the modifications that we described. The first class attribute grammars by De Moor, Backhouse and Swierstra [2] may well fit the bill. The methods of composing these grammars also suggests another approach to specifying the semantics.

7 Conclusion

We have shown a novel technique to deal with extensions to the implementation of a verification condition generator. This technique provides for a method of modular development of the verification condition generator in the face of changing requirements. Experiments have shown that the technique works.

References

[1] Cenciarelli, P. and E. Moggi, *A syntactic approach to modularity in denotational semantics*, in: *CTCS 5* (1993).

[2] de Moor, O., K. Backhouse and S. D. Swierstra, *First class attribute grammars*, Informatica: An International Journal of Computing and Informatics **24** (2000), pp. 329–341, special Issue: Attribute grammars and Their Applications.

[3] Dijkstra, E. W. and C. S. Scholten, "Predicate Calculus and Program Semantics," Texts and Monographs in Computer Science, Springer Verlag, Berlin, 1990.

[4] Espinosa, D., "Semantic Lego," Ph.D. thesis, Columbia University (1995).

[5] Homeier, P. V. and D. F. Martin, *Trustworthy tools for trustworthy programs: A verified verification condition generator*, in: *TPHOLS 1994*, number 859 in LNCS (1994), pp. 269–284.

[6] Jacobs, B. and E. Poll, *A monad for basic java semantics*, in: T. Rus, editor, *AMAST 2000*, number 1816 in LNCS (2000), pp. 150–165.

[7] King, D. J. and P. Wadler, *Combining monads*, in: J. Launchbury and P. M. Sansom, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, Workshops in Computing Series (1992).

[8] Liang, S., P. Hudak and M. Jones, *Monad transformers and modular interpreters*, in: *POPL'05* (1995), pp. 333–343.

[9] Lüth, C. and N. Ghani, *Composing monads using coproducts*, in: *ICFP'02* (2002), pp. 133–144.

[10] Moggi, E., *An abstract view of programming languages*, Technical Report ECS-LFCS-90-113, University of Edinburgh (1989).

[11] Moggi, E., *Computational lambda-calculus and monads*, in: *LICS'89*, IEEE Computer Society Press, Washington, DC, 1989 pp. 14–23.

[12] Paakki, J., *Attribute grammar paradigms — a high-level methodology in language implementation*, ACM Computing Surveys **27** (1995), pp. 196–255.

[13] Schröder, L. and T. Mossakowski, *Monad-independent dynamic logic in hascasl*, Journal of Logic and Computation **14** (2004), pp. 571–619.