Building Verification Condition Generators by Compositional Extensions

I.S.W.B. Prasetya  
Inst. of Inf. and Comp. Sciences, Utrecht Univ. Email: wishnu@cs.uu.nl

A. Azurat  
Fakultas Ilmu Komp., Univ. Indonesia. Email: ade@cs.ui.ac.id

T.E.J. Vos  
Inst. Tecnológico de Inf., Univ. Politécnica de Valencia. Email: tanja@iti.upv.es

A. van Leeuwen  
Inst. of Inf. and Comp. Sciences, Utrecht Univ. Email: arthurvl@cs.uu.nl

Abstract

This paper describes a technique that combines algebraic datatypes and monads to build derivative verification condition generators (VCGs) by extending a base VCG. Extensions are compositional and can be stacked while the base VCG is left unchanged. The technique can be used to build a set of weaker VCGs to do light weight verification. Moreover, it enables us to add an ability to generate validation traces. The paper explains the technique through an example that extends a simple language \( L_0 \) with new constructs to handle exceptions. To deal with exceptions, not only the logic of \( L_0 \) has to be extended with new rules, its structure also needs to be changed. We show that using our technique the extension can be implemented in a simple and compositional way, without any change to the underlying logic.

**Keywords**: verification tool, verification technique, modular verification

1 Introduction

Maintenance is a problem when developing and implementing a realistic programming logic. During the process features may be changed or added. At the early stage, even the object languages may be changed. The implementation has to be changed as well and from experience we learn that this is a dangerous and error prone operation 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 [6]. Such a straightforward implementation however results in a monolithic program that cannot be altered or extended without tampering with the code.

The contribution of this paper is a technique that enables us to change and extend the implementation of a logic in a modular way, i.e. without directly tampering with the code of the old implementation. Our approach has a number of interesting advantages. First of all, it is safer. Second, it allows alterations and extensions to be engaged or disengaged at will. Third, it enables us to easily create a set of partial logics, each of which can be used in isolation for light weight verification.

A very safe way to implement a programming logic exists, namely by deeply embedding inside a theorem prover. Excellent examples are the embedding of Java in Isabelle by Huisman [7] and C in HOL by Norrish [16]. The approach allows the soundness of the embedded logic to be verified. However, changes to the embedded logic invalidate its soundness proof, so we have to rework it. In general, this is quite expensive, and, after several changes, the proof itself will have maintenance problems. Our technique can potentially be combined with deep embedding (in theorem provers) to alleviate the problem there as well. See Section 5 where we discuss conditions under which alterations on a logic can be made to preserve soundness and how to factorize the proof over modifiers so that a change in one of the extensions do not invalidate the entire proof.

Our technique uses a combination of algebras and
monads to represent a (syntax driven) logic. Algebras are modular structures used to abstractly specify recursive computation [14]. Higher order functions are used to implement extensions or alterations on algebras. Monads have been recognized to be a useful tool to build modular logics. More specifically, monads are used to hide certain aspects regarding the structure of a logic; exposing only the aspects that will remain unchanged across various instances of the logic. This enables us to keep the code of the base logic unchanged despite an extension that would actually alter the logic’s structure. The change can be delayed to the instantiation of the corresponding monad class.

1.1 Preliminary

We will explain our approach through an example logic for a simple imperative language $L_0$ shown in Figure 1. For simplicity, we will assume that all statements terminate. The construct $\text{inv } i \text{ while } g \text{ do } S$ is just a while-loop; $i$ is a candidate invariant specified by the programmer.

The implementation will be explained using a notation that resembles that of the functional programming language Haskell (www.haskell.org) [3]. In depth knowledge of Haskell syntax is not needed to read this paper, though we assume the reader is familiar with functional programming. Familiarity with the concept of Haskell class and monads will be quite helpful. An actual Haskell implementation of the example is available on request.

In the rest of the paper, the type variable $e$ is assumed to represent the type of expressions. Rather than imposing a concrete representation and/or syntax of expressions, we will assume $e$ to be an instance of a class called Expression supporting a minimum set of constants and operations: $\text{true}, \text{false}, 0, \land, \lor, \Rightarrow$ and $\neg$. And in addition, the following two operations:

\[
\begin{align*}
\text{subst} & : (\text{String}, e) \rightarrow e \rightarrow e \\
\text{safe} & : e \rightarrow e
\end{align*}
\]

The first is to perform a syntactical substitution. The second will be explained later.

The type variable $m$ is assumed to range over monads.

Haskell Class

We will use the notion of class as in Haskell. A class $C$ specifies a collection of types supporting a fixed set of operations. An example of a class specification is this:

\[
\begin{align*}
\text{class } \text{Eq } a \text{ where } (==) : a \rightarrow a \rightarrow \text{Bool}
\end{align*}
\]

which declares a class called Eq of all types that for which the operation $==$ is available. This allows the overloading of the symbol $==$. By intention, there may be some algebraic properties associated to the operations of a class (e.g. that $==$ should be commutative), though these cannot be specified within Haskell. Using the same notation as above we can also define a class of type constructors.

The type notation like:

\[
f :: \text{Eq } a \Rightarrow a \rightarrow \text{Bool}
\]

is used to say that $f$ is a function of type $a \rightarrow \text{Bool}$, but it is also required that the type $a$ is a known instance of the class Eq. Notice the use of $\Rightarrow$ in the above notation, which should not be confused with $\Rightarrow$ in the predicate logic.

A class can also be defined to extend another class, like:

\[
\begin{align*}
\text{class } \text{Eq } a \Rightarrow \text{Ord } a \text{ where } (<) : a \rightarrow a \rightarrow \text{Bool}
\end{align*}
\]

This declares the class Ord. Its operations are $<$ and all operations of Eq. It also means that for a type to be an instance of Ord, it has to be an instance of Eq as well.

1.2 Paper Outline

Section 2 explains our representation. Section 3 shows examples of how a logic can be altered and extended modularly. Section 4 shows an experiment where we extend the language $L_0$ by adding new constructs to raise and handle exceptions. Not only that the old logic underlying $L_0$ has to be extended with new rules, but some alteration to the logic’s structure is needed as well. Normally this would require surgery on the implementation of the old logic. Section 4 shows how it can be done without. Section 5 discusses soundness issues. Finally, conclusions, related work and conclusion are given in Sections 6 and 7.

2 Representation of Logics

Hoare logic [5] is commonly used to specify and verify imperative programs. Usually, it is used in combination with predicate transformers, which are functions that take and return a predicate [4, 2, 6]. Figure 1 shows a simplistic command language $L_0$ and its underlying logic. In the logic shown in Figure 1, pre is a predicate transformer. In particular, given a statement $S$ and a post-condition $q$, pre returns a pre-condition
that is sufficient for $S$ to realize $q$. It can be shown that $\vdash p \Rightarrow \mathsf{pre} P q$ implies $\{p\} P \{q\}$. Hence, a Hoare triple specification can be reduced to a problem expressed in terms of $\mathsf{pre}$.

The inference rules of the logic underlying $L_0$ specify how $\mathsf{pre}$ computes its result. Some of the rules, such as the rule for $\mathsf{while}$, produce so-called verification conditions like the conditions $i \land \neg g \Rightarrow q$ and $i \land g \Rightarrow p$. The pre-condition returned by $\mathsf{pre}$, as specified by a rule, is only sufficient if the corresponding verification conditions can be shown to be valid.

In functional programming, data types are used to abstractly represent sentences of a language. In our case, the sentences are $L_0$ statements and below is a data type called $\mathsf{Stmt}$ which is sufficient to represent them.

**Definition 2.1**: $\mathsf{Stmt}$

<table>
<thead>
<tr>
<th>$\mathsf{Stmt}$</th>
<th>$\rightarrow$ Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mid$ if Expr then ${ Stmt }$ else ${ Stmt }$</td>
<td></td>
</tr>
<tr>
<td>$\mid$ inv Expr while Expr do ${ Stmt }$</td>
<td></td>
</tr>
<tr>
<td>$\mid$ Stmt ; Stmt</td>
<td></td>
</tr>
</tbody>
</table>

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

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

$\mathsf{pre} (if g then S_1 else S_2) q = if g then \mathsf{pre} S_1 q else \mathsf{pre} S_2 q$

$p = \mathsf{pre} S i$

$\vdash i \land \neg g \Rightarrow q$, $\vdash i \land g \Rightarrow p$

$\mathsf{pre} (\mathsf{inv} : \mathsf{while} g \mathsf{do} S) q = i$

**Figure 1. A simple command language $L_0$ and its logic.**

The logic of $L_0$, which specifies the calculation of $\mathsf{pre}$, is syntax driven: for each kind of statement there is exactly one inference rule. Consequently, given a statement $S$ and a post-condition $g$, $\mathsf{pre} S q$ can be calculated recursively over the structure of $S$. Some rules emit verification conditions, which should be collected. Collecting these verification conditions is usually done by another recursive function, called a verification condition generator or VCG. A fraction of a straightforward implementation of $\mathsf{pre}$ and the corresponding VCG is shown below. Their computations are merged into one recursive function $\mathsf{pvcg}$. When given a program $S$, a post-condition $q$, and an initially empty list of verification conditions, $\mathsf{pvcg} S q []$ returns a tuple $(V, p)$ where $V$ is a list of verification conditions generated along the way, and $p$ is $\mathsf{pre} S q$.

$\mathsf{pvcg} : \mathsf{Expression} e \Rightarrow \mathsf{Stmt} \rightarrow e \rightarrow [e] \rightarrow ([e], e)$

$p = \mathsf{pre} S i$

$\mathsf{pvcg} (x := e) q \mathsf{vcs} = (\mathsf{vcs}, \mathsf{subst} (x, e) q)$

$\mathsf{pvcg} (s_1 :> s_2) q \mathsf{vcs} = \ldots$

$\mathsf{pvcg} (\mathsf{while} i g \mathsf{body}) q \mathsf{vcs} = (c_1 : c_2 : \mathsf{vcs}', i)$

where

$c_1 = i \land \neg g \Rightarrow q$

$c_2 = i \land g \Rightarrow p$

$(\mathsf{vcs}', p) = \mathsf{pvcg} i \mathsf{vcs}$

$\mathsf{pvcg} (\mathsf{ifElse} g s_1 s_2) \mathsf{vcs} q = \ldots$

Although straightforward to write, the code is also too monolithic in that it cannot be altered or extended without tampering with it. Later we will show how to do it differently. The next subsection will first introduce some notation and underlying concepts.

### 2.1 Algebras

Any data type $T$ induces a so-called fold function: a higher order function that defines a recursive pattern over $T$. For the data type $\mathsf{Stmt}$ from Definition 2.1, the corresponding fold function is:

$\mathsf{foldStmt} (A_{\mathsf{asg}}, A_{\mathsf{seq}}, A_{\mathsf{if}}, A_{\mathsf{while}}) S = \mathsf{fold} S$

where

$\mathsf{fold} (x := e) = A_{\mathsf{asg}} x e$

$\mathsf{fold} (S_1 :> S_2) = A_{\mathsf{seq}} (\mathsf{fold} S_1) (\mathsf{fold} S_2)$

$\mathsf{fold} (\mathsf{ifElse} g S_1 S_2) = A_{\mathsf{if}} g (\mathsf{fold} S_1) (\mathsf{fold} S_2)$

$\mathsf{fold} (\mathsf{while} i g S) = A_{\mathsf{while}} i g (\mathsf{fold} S)$

The tuple $(A_{\mathsf{asg}}, A_{\mathsf{seq}}, A_{\mathsf{if}}, A_{\mathsf{while}})$ consists of functions; each specifies how the results of the recursion are combined at the corresponding data constructor. If $r$ is the type of the result of the recursion, those functions can be seen as operations on $r$. In literature a tuple of operations is also called an algebra. Notice that via a fold function, an algebra can be said to abstractly specify a recursive computation.

If $T$ is a data type, an algebra $A$ which can be folded over $T$ is also called a $T$-algebra. So, the $\mathsf{Stmt}$-algebra has the following type:

---

1. If $S$ does not contain any loop, $\mathsf{pre}$ will return the weakest pre-condition. Otherwise it will just produce a sufficient one.
A $T$-algebra whose operations operate on the type $r$ is also called a $T$-algebra of $r$, or simply an algebra of $r$ if the choice of $T$ is clear from the context.

In the algebraic theories of data types, e.g. [14, 11], it is possible to talk about the properties of algebras in a more abstract way, e.g. without having to be explicit about the structure of the underlying data type. The use of this kind of abstraction is not really necessary here, and so we do not use it. The extended version of this paper [19] includes a brief outline on how to look at the techniques explained here from an algebraic point of view.

Let us introduce some more notation. If $A$ is a tuple, the notation $f[A]$ extends $A$ with $f$; for example, $f[(g, h)] = (f, g, h)$. If $A$ is a $T$-algebra, and $C$ is a data constructor of $T$, $AC$ denotes the component of $A$ which corresponds to $C$, and $A(C = f)$ denotes the algebra obtained by replacing $AC$ with $f$.

Later we will also consider modifications to some algebra $A$ by post-processing the results of the functions that constitute $A$. Such a modification is constructed by applying what we will call a modifier. For Stmt-algebras, the modifiers will have the following type:

\[
\text{type Modifier } e r = (\text{String} \rightarrow e \rightarrow (r \rightarrow r))
\]

The operator <$\$> defined below applies a modifier $M$ to an algebra $A$ and results in a new algebra.

**Definition 2.2 : Applying a Modifier**

\[
M <\$> A = ((\lambda \ x \ e \rightarrow M_{\text{seq}} x e (A_{\text{seq}} x e))
, (\lambda \ r_1 \ r_2 \rightarrow M_{\text{seq}} (A_{\text{seq}} r_1 r_2))
, (\lambda \ g \ r_1 \ r_2 \rightarrow M_{\text{seq}} g (A_{\text{seq}} g r_1 r_2))
, (\lambda \ i \ g \ r \rightarrow M_{\text{seq}} i g (A_{\text{seq}} i g r)))
\]

### 2.2 Brief on Monads

We will give a brief overview of monads that is sufficient to understand the remaining of this paper. A more inspiring introduction on monads can be found in [21]. There are also plenty of texts at [www.haskell.org](http://www.haskell.org). In Haswell, monad is a class of type constructors supporting these two operations:

```haskell
class monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

There are some algebraic properties which the operations should satisfy — see [21]. Members of the above class (the m's) are monads. Monads have a number of applications. For example, $m a$ can be made to represent state based computations returning values of type $a$. In this setup `return` $x$ is a computation that returns $x$ and does not change the underlying state; $e >>= f$ executes $e$ then passes the value $v$ it returns to $f$ and executes $f v$. For example we can use $m a = \text{Int} \rightarrow (\text{Int}, a)$ to represent computations whose state is a single integer. We can define `return` and `>>=` as follows:

\[
\text{return } x s = (s, x)
\]

\[
(c >>= f) s = f v t \text{ where } (t, v) = c s
\]

In this way we can mimic an imperative program in a functional language. With proper syntactical sugar (the do-notation), the state can be hidden and we can abstractly imitate imperative programs within a functional language. In Haskell we can write code like:

```haskell
do { q ← t_1 r ; p ← t_2 q ; return p }
```

or equivalently like:

```haskell
do { q ← t_1 r ; p ← t_2 q ; return p }
```

The code will be translated to:

\[
t_1 r >>= (\lambda q \rightarrow t_2 q >>= (\lambda p \rightarrow \text{return } p))
\]

Note that via the Haskell class mechanism, the do-notation is overloaded over all instances of `Monad`.

### 2.3 Predicate Transformer, Logic, and VCG

Recall that some rules of the predicate transformer `pre` generate verification conditions. In order to collect them, we can thread a list through the computation of `pre` such that whenever a verification condition is emitted, it is added into the list. Consequently, if $e$ is the type of expressions, we have to represent a predicate transformer by a function of type:

\[
e \rightarrow [e] \rightarrow ([e], e)
\]

where the $[e]$ in the second argument represents the threaded list of already generated verification conditions. Because the list is threaded, it can be seen as a state with respect to the computation of a transformer. Consequently, it can be represented by a monad, and we can change the representation of transformers as follows:

**Definition 2.3 : Transformers**

Let $m$ be a monad.

\[
\text{type Transformer } m e = e \rightarrow m e
\]
In particular, we will use *recorder monads* from the class `MonadR` that are explained below. A recorder monad extends an ordinary monad with an operation `record`. Notice that the class specification of `MonadR` leaves the exact implementation of the operation unspecified. For our purpose, `record c` will be an operation that somehow adds the verification condition `c` into the threaded list, now maintained as a state by the monad.

**Definition 2.4**: Recorder Monad

```haskell
class Monad m ⇒ MonadR e m where record :: e → m()
```

An inference rule can be represented by a function that takes a statement and returns a transformer. This means that an implementation of `pre`, will have the following type:

```haskell
pre :: MonadR e m ⇒ Stmt e → Transformer m e
```

Now we can benefit from the monad representation and can use the `do` notation. The rule for `while` can then, for example, be implemented in Haskell as follows:

```haskell
ruleWhile (While i g body) q = do p ← pre body q
    record (i ∧ ¬g ⇒ q)
    record (i ∧ g ⇒ p)
    return p
```

This looks cleaner than the straightforward implementation we had at the beginning of Section 2.

We will, however, use a slightly different implementation. Rather than passing a `while` statement as the first argument of `ruleWhile`, we pass the transformer for the body of the `while`, i.e. `pre body`. The resulting code for all rules is shown in Figure 2. The reason for passing the transformer (instead of the statement) is that now the type of a tuple containing the four rules matches that of an algebra, i.e., an algebra of transformers:

```haskell
StmtAlgebra e (Transformer e m)
```

Notice that such an algebra fully specifies the transformer logic of `L0`, and hence we use the first to represent the latter. We define this type abbreviation as follows:

**Definition 2.5**: Family of `L0`-logics

```haskell
type L0Logic m e = StmtAlgebra e (Transformer e m)
```

In particular, below we define an instance of such a logic which corresponds to the `pre`-logic of `L0` as in Figure 2.
3 Modifying Logics

Since now a logic is just a tuple of inference rules, we can easily construct a variant logic by replacing some of the rules. The corresponding VCG can be obtained simply by folding the new logic. Below are some examples of 'lighter' logics obtained by replacing the standard while rule with weaker ones.

Definition 3.1: The b and i logics and VCGs

\[
\begin{align*}
\text{blogic} & = \text{stdlogic}\{\text{While} = \text{b_ruleWhile}\} \\
\text{ilogic} & = \text{stdlogic}\{\text{While} = \text{i_ruleWhile}\} \\
\text{bvcg} & = \text{foldStmt}\_\text{blogic} \\
\text{ivcg} & = \text{foldStmt}\_\text{illogic}
\end{align*}
\]

The definition of b_ruleWhile and i_ruleWhile rules are given below. When given a program \( P \), \( bvcg \) will perform a reduction that assumes the invariance and reachability of all \( i \)’s that decorate the loops in \( P \). More precisely, if \( P \) contains a loop \( \text{inv} \ i \) while \( g \) do \( S \), the reduction will assume that \( S \) preserves \( i \) and that the state of \( P \) as it enters the loop will satisfy \( i \). Because these aspects of correctness are now assumed, \( bvcg \) will produce fewer verification conditions; which is more suitable for 'light' verification. The other VCG, \( ivcg \), is another example of a 'light' VCG. When given a program \( P \) with no nested loop, it will only produce the verification conditions that are needed to verify that all \( i \)'s decorating the loops in \( P \) are preserved by their respective loops’ body.

Definition 3.2: The b-rule

\[
\text{b_ruleWhile} \quad i \ g \ t_{\text{body}} \ q \quad = \text{do} \ p \leftarrow t_{\text{body}} \ \text{true} \\
\quad \text{record} \ (i \land \neg\ g \Rightarrow q) \\
\quad \text{record} \ (i \land \ g \Rightarrow p) \\
\quad \text{return} \ \text{true}
\]

Definition 3.3: The i-rule

\[
\text{i_ruleWhile} \quad i \ g \ t_{\text{body}} \ q \quad = \text{do} \ p \leftarrow t_{\text{body}} \ i \\
\quad \text{record} \ (i \land \ g \Rightarrow p) \\
\quad \text{return} \ \text{true}
\]

We can also easily extend a logic. Suppose we consider a more realistic variant of \( L_0 \) that has the ability to abort when an expression is evaluated inside a statement. This can come in handy when, for example, the evaluation of an expression causes a division by zero, or an attempt to read an array outside its range. To deal with this, the logic of \( L_0 \) will have to be strengthened accordingly. We can do this by modifying each affected inference rule so that the computed pre-condition is strengthened by a predicate sufficient to guarantee safe evaluation of the expressions in the target statement. We will call such an extension an SE (Safe Evaluation) extension.

Recall that the type \( e \) is an instance of the class Expression. We assumed that the class also offers a function \( \text{safe} :: e \rightarrow e \). The idea is that given an expression \( e \), \( \text{safe} \ e \) (symbolically) analyzes \( e \) and returns a predicate which is sufficient to guarantee that \( e \) can be evaluated safely (e.g. it will not raises a division by zero exception).

Now we can define the following higher order function to strengthen an inference rule. The resulting rule produces a strengthened pre-condition \( p \) such that evaluating an expression \( e \) in a state satisfying \( p \) is always safe:

Definition 3.4: SE Rule Extension

Let:

\[
\text{seextend} :: e \rightarrow \text{Transformer} \ m \ e \rightarrow \text{Transformer} \ m \ e
\]

We define:

\[
\text{seextend} \ e \ t \ q \quad = \text{do} \ \{ \ p \leftarrow t \ q ; \ \text{return} \ (\text{safe} \ e \land p) \}
\]

For example, we can apply it to extend the assignment rule:

\[
\text{seruleAsg} \ x \ e \ = \text{seextend} \ e \ (\text{ruleAsg} \ x \ e)
\]

This will strengthen the rule such that applying it to an assignment \( x:=e \) will result in a pre-condition that will guarantee the safe evaluation of the expression \( e \).

We can now define a modifier that will extend each inference rule of \( L_0 \) accordingly. The extension for the assignment has been shown above. The rules for IfElse and While have to be extended as well to guarantee the safe evaluation of their guards. The rule for sequential composition does not need any extension because it does not need to evaluate any expression (at the top level). Here is the SE-modifier:

Definition 3.5: SE Modifier

\[
M_{\text{se}} = ((\lambda \ e \ e \rightarrow \text{seextend} \ e), \text{id}, \text{seextend}, \text{seextendWhile})
\]

where

\[
\text{seextendWhile} \ i \ g \ t \ q \quad = \text{do} \ \{ \ t \ q ; \ \text{record}(i \Rightarrow \text{safe} \ g) \}
\]

Now we can apply the modifier to a logic. For example, \( M_{\text{se}} \langle \$\rangle \text{stdlogic} \) will result in the standard \( L_0 \) logic with the SE extensions. For more lightweight verification, we can construct \( M_{\text{se}} \langle \$\rangle \text{blogic} \), that, when
given true as the post-condition, will produce only the verification conditions related to the safe evaluation of the expressions in the target program, regardless of its functionality.

We can also use a modifier to extend the functionality of a VCG such that, besides generating verification conditions, it leaves a trace of information that can be used for debugging or validation.

For example, the inference rule that handles assignments in java-like OO languages is quite complicated [7, 18] and users would definitely benefit from a trace that can, for example, be sent to a third party tool for validation. Below, we will define a modifier that records the pre- and post-conditions of every assignment in order to generate a trace. Note that such a modifier needs to extend the state structure of a VCG. Normally, this would require surgery on the existing code of the VCG. In our case, however, no surgery is needed since we have specified the monad underlying a logic and its VCG using a Haskell class called MonadD. Such a specification lays down the general type of operations available to the class, but leaves the precise internal structure of the class instances unspecified. We can now simply extend the class MonadD with a new class, called MonadR that adds an operation recordDebugInfo for inserting new information to the validation trace. We call instances of the class MonadD debugger monads.

**Definition 3.6 : Debugger Monad**

\[
\text{class MonadR e m \Rightarrow MonadD e m} \\
\text{where} \\
\text{recordDebugInfo :: String \rightarrow m()} \\
\]

We can now define a modifier that extends the assignment rule so that it records its post-condition and the calculated pre-condition:

**Definition 3.7 : VT Modifier**

\[
M_{VT} = (m_{asg}, id, (\lambda g \rightarrow id), (\lambda i g \rightarrow id)) \\
\]

where:

\[
m_{asg} x e r q = \text{do } p \rightarrow r q \\
\hspace{1cm} \text{recordDebugInfo (show q)} \\
\hspace{1cm} \text{recordDebugInfo (show (x:=c))} \\
\hspace{1cm} \text{recordDebugInfo (show p)} \\
\hspace{1cm} \text{return } p \\
\]

We can use this modifier on any logic. For example \(M_{VT} \triangleleft \text{logic}\) will 'plug-in' the SE and validation trace extensions to the standard logic. After some beautification, the trace extension can produce a trace like:

```
TRACE:
{ 0<=0 } i:=0  { 0<=1 }
{ 0<=i+1 } i:=i+1  { 0<=i }
...
```

Notice that the validation trace extension can now be added without changing *anything* in the base logic. All we need to do is properly instantiate the monad used by the logic, in order to create a concrete instance of the logic that is needed to make a concrete VCG.

### 4 Extending Logics

We will now consider a situation where we extend the language \(L_0\). Let us add two constructs: \texttt{raise} and \texttt{try}. The first will enable us to raise an exception, for example if the evaluation of an expression within a statement causes a division by 0. The second construct, \texttt{try S \catch S}, will try to do \(S_1\). If \(S_1\) terminates normally then \(S_2\) is skipped, otherwise \(S_2\) is executed. Furthermore, evaluating an expression in a statement may now raise an exception.

In the following, we assume the representing type Stmt and its fold function are extended accordingly to accommodate the new constructs.

As the language grows, the logic supporting it should also be expanded accordingly. Basically, all we have to do is add the rules for the new constructs to the old logics of \(L_0\). Let us try a minimalist extension first. It is an extension of the \(L_0\) logic that will produce a pre-condition that will enforce normal execution of the target statement (that is, the pre-condition guarantees that at no point during its execution the statement will throw an exception). Consider now the following rules for \texttt{raise} and \texttt{try}:

**Definition 4.1 : Conservative raise Rule**

\[
\text{ruleRaise } q = \text{return } \text{false} \\
\]

**Definition 4.2 : Conservative try Rule**

\[
\text{ruleTry } t_{\text{try}} t_{\text{catch }} q = t_{\text{try }} q \\
\]

In particular, \texttt{ruleRaise} returns an \texttt{false} as the pre-condition, which means that the rule actually wants to forbid an execution leading to \texttt{raise}. Consider \(M_{SE} \triangleleft \text{logic}\) as the base logic. The SE extension makes sure that no expression in the target statement will cause an exception. Since exception is now excluded, the statement in the \texttt{catch} part can be ignored,
which what ruleTry above does. So, the new logic for
the extended \( L_0 \) can be built by:

\[
\text{ruleRaise \mid ruleTry \mid (MSE <\$> \text{std\_logic})}
\]

A more reasonable extension, however, will really deal
with exceptions rather than simply excluding them.
Borrowing ideas from [12, 7], the Hoare triple notation
is extended to:

\[
\{ p \} S \{(q, q')\}
\]

where \( q \), called normal post-condition, denotes the
post-condition of \( S \), if it terminates normally; and
\( q' \), called exceptional post-condition, denotes the
post-condition if \( S \) terminates via an exception. The rules
for raise and try are changed to:

\[
\begin{align*}
\text{ruleRaise} (q, q') & \quad = \quad \text{return} q' \\
\text{ruleTry} \; t_{\text{try}} \; t_{\text{catch}} (q, q') & \quad = \quad \text{do} \; \begin{array}{l}
p' \leftarrow t_{\text{catch}} (q, q') \\
t_{\text{try}} (q, p')
\end{array}
\end{align*}
\]

Notice that this requires the structure of the post-
condition in the old logic of \( L_0 \) to be extended to a pair.
Our representation can handle such an extension!
Recall that we represent post-conditions by a
type variable \( e \) that can take any structure, including
tuples. However we do require \( e \) to be an instance of
the class Expression. So, whatever the concrete choice
of \( e \) is, a proper instance of the class Expression
will have to be written, keeping in mind that the class has
to support quite a number of operations.

Another way to implement the extension is by
putting the exceptional post-condition in the state of
the used monad. This is not the way post-conditions
are normally treated. However the only rule that alters
the information in the exceptional post-condition is the
try rule, since this is the only place in \( L_0 \) where an
exception is handled. Consequently, as we recursively
apply the rules down a target statement, the informa-
tion in the exceptional post-condition remains most of
the time constant. So, we can get a better abstraction
by hiding it, which is what we do by making it part of
the monad’s state.

Below we introduce a class Monad\(^E\) which extends
Monad\(^R\) with two operations: getPostE which is
used to fetch the exceptional post-condition from the
monad’s state, and setPostE which is used to change it.

\[
\text{Definition 4.3 : Monad}\(^E\) = \text{Monad}\(^R\) \; e \; m \; \Rightarrow \quad \text{Monad}\(^E\) \; e \; m \; \text{where}
\]

\[
\begin{align*}
\text{getPostE} :: \; m \; e \\
\text{setPostE} :: \; e \rightarrow \; m()
\end{align*}
\]

Now we can redefine the \( \text{raise} \) and \( \text{try} \) rules to make
use of a monad from the class Monad\(^E\):

\[
\text{Definition 4.4 : raise Rule}
\]

\[
\varepsilon \text{ruleRaise } q \; = \; \text{getPostE}
\]

\[
\text{Definition 4.5 : try Rule}
\]

\[
\varepsilon \text{ruleTry } t_{\text{try}} \; t_{\text{catch}} \; q \; = \; \text{do} \; \begin{array}{l}
p' \leftarrow \text{getPostE}\; ; \; p' \leftarrow t_{\text{catch}} \; q' ; \\
\quad \text{setPostE} \; p' ; \\
\quad p \leftarrow t_{\text{try}} \; q' ; \\
\quad \text{setPostE} \; q' ; \\
\quad \text{return} \; p
\end{array}
\]

To obtain the new logic, we simply add the above rules
to the old logics of \( L_0 \) with the SE extension. For
example, a new standard logic can be built by:

\[
\varepsilon \text{ruleRaise \mid ruleTry \mid (MSE <\$> \text{std\_logic})}
\]

And if we prefer a more lightweight logic, we can, for
example, replace std\_logic above with std\_logic.

5 Soundness Issue

Modifying a logic may of course introduce inconsistencies.
We will give a sufficient condition so that a
modification is soundness preserving.

Consider a programming language \( L \) and its logic \( \mathcal{L} \),
which is syntax driven and is represented by an algebra
\( A \). We will write \( \mathcal{L}\text{\_alg} \) to denote \( A \) and \( \text{fold} \) \( A \) to
denote the function that results from folding \( A \) (so we
have this equality: \( \mathcal{L} = \text{fold} \mathcal{L}\text{\_alg} \)). Note that \( \mathcal{L} \)
is a function that maps sentences of \( L \) to some results
domain. Let \( \tau \) be the type \( \mathcal{L} \)’s results. For example, in
the logics discussed in the previous sections, values of
\( \tau \) are predicate transformers.

A soundness notion over \( \mathcal{L} \) can be expressed in terms
of a predicate \( C \), called soundness criterion.

\[
\text{Definition 5.1 : Soundness}
\]

\[
(L, \mathcal{L}) \text{ is sound wrt } C = (\forall S : \mathcal{L} S \in L : C \; S \; (\mathcal{L} \; S))
\]

For example, if \( \mathcal{L} \) is the std\_logic of \( L_0 \) its sound-
ness criterion \( C_{\text{std}} \) could be (instantiating \( m \; e \) to
\( [e] \rightarrow ([e], e) \)):

\[
C_{\text{std}} \; S \; t \quad = \quad (\forall q :: \; \mathcal{L} S \text{ valid} \; \Rightarrow \; \{ p \} S \; \{ q \}, \text{ where } (V, p) = t \; q)
\]

Consider a partial order relation \( \preceq \) over \( \tau \). The notion
of \( \preceq \)-monotonicity for functions is defined as standard.
Furthermore:
1. We call a soundness criterion \( C \) \( \leq \)-monotonic if it is \( \leq \)-monotonic on its second argument.

2. We call an algebra \( A \) \( \leq \)-monotonic if for all components \( f \) of \( A \), \( f \) is \( \leq \)-monotonic on all its \( \tau \)-arguments.

For example, \( C_{\text{std}} \) defined before is monotonic with respect to \( \equiv \) lifted to the transformer level:

\[
\begin{align*}
t_1 \equiv t_2 &= (\forall q :: (p_1 \equiv p_2) \land (\forall V_1 \equiv V_2) \\
&\text{where} \\
& (V_1, p_1) = t_1 q \\
& (V_2, p_2) = t_2 q
\end{align*}
\]

The logic \( \text{stdlogic} \) is \( \equiv \)-monotonic.

A modification to an algebra can be expressed in terms of a function that transform an algebra to another. In the categorical treatment of datatypes, e.g. [14, 11], such a transformer is called functor. We will use the same terminology here, though because the setup here is not categorical, a functor here is just a function from algebras to algebras. As we will see, to preserve soundness we will want the functors to be 'ascending'.

**Definition 5.2** : Ascending Functor

A functor \( F \) is \( \leq \)-ascending if:

\[
(\forall S :: \text{fold} \ A \ S \leq \text{fold} \ (F \ A) \ S)
\]

The previous section shows four ways to alter \( L \): we can replace some components of \( L, \text{alg} \), adding new components, instantiating the monad that parameterizes it, or apply a modifier to it. All these transformations are instances of functors. In general, proving that a functor is ascending requires an inductive proof. However for the above found kinds of alterations induction is not needed, if the target algebra is monotonic: (1) if we replace a component \( f \) of an algebra \( A, \text{alg} \) with \( f' \), it is sufficient to prove that when given the same arguments the result of \( f' \) is at least equal to that of \( f \) in terms of \( \leq \); (2) if we add a component \( g \) to \( L, \text{alg} \), it is sufficient to show that \( g \) is \( \leq \)-monotonic on all its \( \tau \)-arguments; (3) if we apply a modifier \( M \) to \( L, \text{alg} \), it is sufficient to show that each component of \( M \) is ascending on its last argument. Finally, (4) changing the underlying monad \( m \) is just an instance of (1). However, often the new monad is just a pure extension to state structure of the old monad (as in our examples). In this case, the alteration is always ascending.

As an example, the application of the SE-modifier (Definition 3.5) is ascending with respect to the \( \equiv \) defined above.

The following theorem states a sufficient condition under which an alteration will preserve the soundness of the base logic.

**Theorem 5.3** : Soundness Preserving Alteration

If: (1) \( (L, L) \) is sound wrt \( C \), (2) \( L, \text{alg} \) is \( \leq \)-monotonic, (3) \( C \) is \( \leq \)-monotonic, and (4) \( F \) is \( \leq \)-ascending, then \( (L, F \ L, \text{alg}) \) is also sound wrt \( C \).

For example, by the above theorem applying the SE-modifier on \text{stdlogic} will preserve \( C_{\text{std}} \) defined above. The proof of the theorem is quite easy, and is omitted.

Suppose \( L \) is sound with respect to \( C \). Suppose we alter \( L \) to \( L' \). Typically \( L' \) is intended to operate on a different language \( L' \) and has a different soundness notion \( C' \). If \( L' \) is an extension of \( L \) and \( C' \) is an extension of \( C \), the above theorem says that the alteration is at least safe with respect to \( C \). It does not unfortunately say anything about the soundness with respect to \( C' \), which in the end is what we want. However, if we can factorize \( C' \) to that it can be written as follows:

\[
C' S t = \begin{cases} S \in L & \text{then } C S t \text{ else } D S t \end{cases}
\]

for some criterion \( D \), then the soundness of \( (L', C') \) with respect to \( C' \) is equivalent to the conjunction of: (1) \( (L, C) \) is sound with respect to \( C \), and (2) \( (L'/L, C') \) is sound with respect to \( D \). The first one can be discharged via the above theorem, so only (2) need to be proven. This also modularizes the proofs. For example, now changing the semantics and the logic over \( L \) will not affect the validity of (2).

6 Related Work

One way monads can be useful for program verification is to use them in the semantics of the target language. In particular if the language has some extended features, monads can be used to abstractly represent the semantics of those features. For example, in [9] Jacobs and Poll show how monads can provide a useful level of abstraction and a means for organizing various complications in the denotational semantics of Java used in the verification tool LOOP [10]. Java has a complicated denotational semantics due to various abnormal termination schemes supported by the language. Another example is the work on Hurd in verification of probabilistic algorithms [8]. Hurd uses state-transformer monad in his semantical model of probabilistic programs to thread random bit generators over computation. In the language design community the usefulness of monads to build semantics in a modular way has actually been realized much earlier —see for example the work of Moggi [15] and Liang and Hudak [13]. As opposed to using monads in the (operational) semantics of a language \( L \), this paper discusses the use of monads to implement logics about \( L \). A syntax driven logic can of course be seen as a semantics.
of $L$, so general results about monadic semantics also applies to monadic logics. As for other works along the same line as ours, so far, we have not much success in tracking one in bibliography databases.

The use of algebras to implement a syntax directed logic is related to the attribute grammar approach [17]. Using an attribute grammar formalism we can abstractly specify recursive computation over a parsing tree. Essentially, such a specification is an algebra, which is specified in terms of how values, also called attributes, being passed and processed between parent and child nodes in a tree. A number of attribute grammar tools are available today, such as Swierstra’s AG system [20]. Traditionally these tools are used to build compiler related tools, such as type checkers and pretty printers, but it is also suitable to build any syntax directed tool such as VCGs (we tried this ourselves in our x-mech verification tool [1]). Intricate weavings of bottom-up and top-down computations are often found in the implementation of modern type checkers. Such a weaving can be conveniently specified in an attribute grammar formalism, whereas using a monad this would be awkward. All VCGs shown here do not require any weaving, hence monad implementation is sufficient. In a more realistic setting, this may change. For example, we may want to add a sub-logic to deal with aliases. To cleverly track down aliases, we may want to collect information in both bottom-up and top-down directions. For such a setting combining the monad and the attribute grammar approaches seems to be a good approach.

7 Conclusion

We have described a technique to change and extend a logic to obtain derivative verification condition generators in a modular manner. The technique should lead to safer and more maintainable implementation of programming logics. As a proof of principle, a small case study has been implemented in Haskell. Experiments seem to confirm the technique’s advantages. We believe that it is worth further investigation.

References