Building Verification Condition Generators by Compositional Extensions

I.S.W.B. Prasetya

A. Azurat

T.E.J. Vos

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-054

www.cs.uu.nl

Building Verification Condition Generators by Compositional Extensions

I.S.W.B. Prasetya (UU) and A. Azurat (UI) and T.E.J. Vos (UPV)

jul. 2004

UU: Institute of Information and Computing Sciences, Utrecht University. P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. UI: Fakultas Ilmu Komputer, Universitas Indonesia. Kampus UI Depok, Indonesia. UPV: Instituto Tecnológico de Informática , Universidad Politécnica de Valencia.
Emails: wishnu@cs.uu.nl, ade@cs.ui.ac.id, tanja@iti.upv.es

Abstract

This paper describes a technique that combines algebraic specifications 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, which are useful to support 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 that 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.

1 Introduction

Maintenance is a problem when implementing a realistic programming logic. When a language changes, the underlying logic, and consequently its implementation, has to be upgraded as well. From experience we learn that this is a dangerous and error prone operation that can easily introduce inconsistencies in the logic. The problem also grows, since most programming languages are updated from time to time.

One solution to this problem is to embed the logic inside another, usually simpler and much more stable, logic. Excellent examples are the embedding of Java in Isabelle by Huisman [8] and C in HOL by Norrish [17]. The approach is very reliable since it allows the soundness of the embedded logic to be verified. Moreover, changes to an embedded logic will not be accepted unless we can reprove its soundness. However, reworking the soundness proof is usually expensive, and, after several changes, the proof itself will have maintenance problems.

Logics underlying imperative languages are usually syntax driven and implemented as a recursive function over the target program [7]. 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. Moreover, if combined with embeddings, it forces us to organize its soundness proof in a modular way[1], and thus minimizing the maintenance problem in the latter.

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 [15]. Higher order functions are used to implement extensions or alterations on algebras. Monad has been recognized to be a useful tool to build modular semantics [14]. Here we use monads 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.

This work is *primarily about an implementation technique*. It does not touch the theoretical issues, such as how to compositionally prove the soundness of the logics implemented using our technique. These are difficult issues of which we have so far no firm answer yet.

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 terminates. The construct `inv` i `while` g `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`) [2]. 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 monads and Haskell class will be helpful. An actual Haskell implementation of the example is available on-line (see Appendix D).

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 the class `Expression` given in the Appendix B. The class supports a minimum set of operations, such as $\wedge$ and $\Rightarrow$, needed to express the logic of L_0.

The type variable m is assumed to range over monads.

Brief about 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:

$$\texttt{class Eq}\ a\ \ \texttt{where}\ (==) :: a \rightarrow a \rightarrow \texttt{Bool}$$

which declares a class called `Eq` supporting $==$ as a single operation. The type notation f :: `Eq` $a \Rightarrow a \rightarrow$ `Bool` is used to say that f is a function of type $a \rightarrow$ `Bool`, but it is also required that a is a type which is a known instance of the class `Eq`. By intention, there may be some algebraic properties associated to the operations of a class, though these cannot be specified within Haskell. A class can also be defined to extend another class, like:

$$\texttt{class Eq}\ a \Rightarrow \texttt{Additive}\ a\ \ \texttt{where}\ (+) :: a \rightarrow a \rightarrow a$$

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

The above notion of class also extends to the type constructor level.

[1] Since now a new logic L' is obtained by applying an extension operation E on an old logic L, to prove the soundness of L' it is sufficient to show that E lifts the soundness of L to L', which in principle can be done without having to alter the proof of L.

$$\begin{array}{lcl} \mathit{Stmt} & \rightarrow & \mathit{Assignment} \\ & | & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \{\ \mathit{Stmt}\ \}\ \texttt{else}\ \{\ \mathit{Stmt}\ \} \\ & | & \texttt{inv}\ \mathit{Expr}\ \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \{\ \mathit{Stmt}\ \} \\ & | & \mathit{Stmt}\ ;\ \mathit{Stmt} \end{array}$$

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

$$\frac{\text{let}\ \ p_2 \ = \ \textsf{pre}\ S_2\ q \quad \text{and} \quad p_1 \ = \ \textsf{pre}\ S_1\ p_2}{\textsf{pre}\ (S_1;\ S_2)\ q \ = \ p_1}$$

$$\frac{\text{let}\ \ p_1 \ = \ \textsf{pre}\ S_1\ q \quad \text{and} \quad p_2 \ = \ \textsf{pre}\ S_2\ q}{\textsf{pre}\ (\texttt{if}\ g\ \texttt{then}\ S_1\ \texttt{else}\ S_2)\ q \quad = \quad (g \Rightarrow p_1) \wedge (\neg g \Rightarrow p_2)}$$

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

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

1.2 Paper Setup

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. Finally, conclusions, related work and some notes on future directions are given in Sections 5 and 6.

2 Representation of Logics

Hoare logic [6] 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 [3, 1, 7]. 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[2]. The predicate transformer pre satisfies the following law, which also describes how to reduce a Hoare triple specification to a problem expressed in terms of pre.

$$\frac{\vdash\ p \Rightarrow \textsf{pre}\ P\ q}{\{p\}\ P\ \{q\}} \tag{1}$$

The inference rules of the logic underlying L_0 specify how pre computes its result. Some of the rules, such as the rule for `while`, produce so-called *verification conditions* like the conditions $i \wedge \neg g \Rightarrow q$ and $i \wedge g \Rightarrow p$. The pre-condition returned by 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 `Stmt` which is sufficient to represent them.

[2] If S does not contain any loop, pre will return the weakest pre-condition. Otherwise it will just produce a suficient one.

Definition 2.1 : `Stmt`
Let e be a type variable representing the type of expressions. We define:

$$
\begin{array}{lcl}
\texttt{data Stmt}\ e & = & \texttt{String :=}\ e \\
 & | & \texttt{Stmt :>}\ \ Stmt \\
 & | & \texttt{IfElse}\ e\ \texttt{Stmt Stmt} \\
 & | & \texttt{While}\ e\ e\ \texttt{Stmt}
\end{array}
$$

`:=` and `:>` are data constructors representing assignment and sequential composition in L_0.
□

The logic of L_0, which specifies the calculation of pre, is syntax driven: for each kind of statement there is exactly one inference rule. Consequently, given a statement S and a post-condition q, 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 usualy done by another recursive function, called a *verification condition generator* or VCG. Appendix C shows how pre and the corresponding VCG can be implemented in a straightforward manner as a recursive function. As indicated before, such an implementation is too monolithic in that it cannot be altered or extended without tampering with the code. Later we will show how to do it differently. In the next two subsections we will first introduce some notation and underlying concepts. The introduction on monad (Subsection 2.2) is a very brief one. If the reader is already familiar with the concept, it can be skipped.

2.1 Algebras

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

$$
\begin{array}{lcl}
\texttt{foldStmt}\ (A_{\texttt{asg}}, A_{\texttt{seq}}, A_{\texttt{if}}, A_{\texttt{while}})\ S & = & fold\ S \\
\quad \texttt{where} & & \\
\quad fold\ (x\ \texttt{:=}\ e) & = & A_{\texttt{asg}}\ x\ e \\
\quad fold\ (S_1\ \texttt{:>}\ S_2) & = & A_{\texttt{seq}}\ (fold\ S_1)\ (fold\ S_2) \\
\quad fold\ (\texttt{IfElse}\ g\ S_1\ S_2) & = & A_{\texttt{if}}\ g\ (fold\ S_1)\ (fold\ S_2) \\
\quad fold\ (\texttt{While}\ i\ g\ S) & = & A_{\texttt{while}}\ i\ g\ (fold\ S)
\end{array}
\tag{2}
$$

The tuple $(A_{\texttt{asg}}, A_{\texttt{seq}}, A_{\texttt{if}}, A_{\texttt{while}})$ 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 $\mathcal{T}$ is a data type, an algebra A which can be folded over $\mathcal{T}$ is also called a $\mathcal{T}$-algebra. So, the `Stmt`-algebra has the following type:

$$
\begin{array}{lcl}
\texttt{type StmtAlgebra}\ e\ r & = & (\texttt{String} \rightarrow e \rightarrow r \\
 & & ,r \rightarrow r \rightarrow r \\
 & & ,e \rightarrow r \rightarrow r \rightarrow r \\
 & & ,e \rightarrow e \rightarrow r \rightarrow r)
\end{array}
\tag{3}
$$

A $\mathcal{T}$-algebra whose operations operate on the type r is also called a $\mathcal{T}$-algebra of r, or simply an algebra of r if the choice of $\mathcal{T}$ is clear from the context.

In the algebraic theories of data type, e.g. [15, 12], 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. In Appendix A we briefly outline how to look at the techniques explained here from an algebraic point of view.

Let us introduce some more notation. If A is an tuple, The notation $f|A$ extends A with f; for example, $f|(g,h) = (f,g,h)$. If A is a $\mathcal{T}$-algebra, and C is a data constructor of $\mathcal{T}$, A_C denotes the component of A which corresponds to C, and $A\{C = f\}$ denotes the algebra obtained by replacing A_C with f.

We will alter the behavior of an algebra by post-processing the results of the functions that constitute an algebra. This post-processing is formalized by what we call a *modifier* of an algebra. In particular, we consider the following type of modifiers for `Stmt`-algebras:

$$\begin{array}{lcl} \texttt{type Modifier}\ e\ r & = & (\texttt{String} \rightarrow e \rightarrow (r \rightarrow r) \\ & & ,r \rightarrow r \rightarrow (r \rightarrow r) \\ & & ,e \rightarrow r \rightarrow r \rightarrow (r \rightarrow r) \\ & & ,e \rightarrow e \rightarrow r \rightarrow r \rightarrow (r \rightarrow r)) \end{array} \tag{4}$$

We introduce an operator `<$>` to apply a modifier of the above type to an algebra.

Definition 2.2 : Applying a Modifier

$$\begin{array}{lcllcl} M \ \texttt{<\$>}\ A & = & ((\lambda\ x\ e & \rightarrow & (M_0\ x\ e)\ (A_0\ x\ e)) \\ & & ,(\lambda\ r_1\ r_2 & \rightarrow & (M_1\ r_1\ r_2)\ (A_1\ r_1\ r_2)) \\ & & ,(\lambda\ g\ r_1\ r_2 & \rightarrow & (M_2\ g\ r_1\ r_2)\ (A_2\ g\ r_1\ r_2)) \\ & & ,(\lambda\ i\ g\ r & \rightarrow & (M_3\ i\ g\ r)\ (A_3\ i\ t\ g\ r))) \end{array}$$

□

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`. A monad is a type constructor m equipped with some operations which can be used to mimic an imperative program in a functional language. Without monads, imperative programs can be imitated by explicitly threading the state the imperative program operates on. However, this results in complicated and ugly code. A monad can be made to carry a state. With proper syntactical sugaring (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:

$$\texttt{do}\ \{\ q \leftarrow S\ r\ ;\ p \leftarrow T\ q\ ;\ \texttt{return}\ p\ \}$$

which calls two imperative functions S and T in sequence. The code is translated to a purely functional composition of S and T. In Haskell, a class `Monad` is also provided. Instances of this class are monads. Via the Haskell class mechanism, it is possible to overload the do-notation on any 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.

```
type Transformer m e  =  e → m e
```

□

In particular, we will use *recorder monads* from the class $\texttt{Monad}^R$ that are explained below. A recorder monad extends an ordinary monad with an operation `record`. Notice that the class specification of $\texttt{Monad}^R$ 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

```
class Monad m ⇒ Monad^R 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:

```
pre  ::  Monad^R 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:

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

This looks cleaner than the straightforward implementation as a recursive function from Appendix C.

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:

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

Notice that such an algebra fully specifies the transformer logic of L_0, and hence we use the first to represent the latter. We define this type abbreviation as follows:

Definition 2.5 : Family of L_0-logics

$$\texttt{type}\ \texttt{L}_0\texttt{Logic}\ m\ e \quad = \quad \texttt{StmtAlgebra}\ e\ (\texttt{Transformer}\ e\ m)$$

In particular, below we define an instance of such a logic which corresponds to the pre-logic of L_0 as in Figure 2.

Definition 2.6 : The Standard L_0-logic

$$_{\texttt{std}}\texttt{logic} \quad = \quad (\texttt{ruleAsg},\ \texttt{ruleSeq},\ \texttt{ruleIfElse},\ \texttt{ruleWhile})$$

where the rules are defined as in Figure 2.

```
ruleAsg $x$ $e$ $q$  =  return (subst $(x, e); q$)

 ruleSeq $t_1$ $t_2$ $q$  =
    do
    $p_2 \leftarrow t_2\ q$
    $p_1 \leftarrow t_1\ p_2$
    return $p_1$

 ruleIfElse $g$ $t_1$ $t_2$ $q$  =
    do
    $p_1 \leftarrow t_1\ q$
    $p_2 \leftarrow t_2\ q$
    return $((g \Rightarrow p_1) \wedge (\neg g \Rightarrow p_2))$

 ruleWhile $i$ $g$ $t_{body}$ $q$  =
    do
    $p \leftarrow t_{body}\ i$
    record $(i \wedge \neg g \Rightarrow q)$
    record $(i \wedge g \Rightarrow p)$
    return $i$
```

Figure 2: *The representation of L_0 inference rules.*

From now on, we will not distinguish a value of type $_{\texttt{std}}$`logic` from the actual logic it represents. We use the term 'logic' for both.

Since a logic is now an algebra, it can be folded over `Stmt`. Folding essentially comes down to applying the inference rules recursively down a given statement. For example folding the logic $_{\texttt{std}}$`logic` defined above will construct the transformer `pre`. Furthermore, if the underlying monad `m` is chosen properly, the transformer will record the generated verification conditions in its monad state and, hence, we also have a VCG.

Definition 2.7 : Representation of VCG

type VCG m e $=$ Stmt $e \rightarrow$ Transformer m e

The standard VCG that generates verification conditions while calculating the weakest precondition for a statement can now easily be defined as follows:

Definition 2.8 : The Standard VCG for L_0
Let $_{\texttt{std}}$`vcg` :: `Monad`$^{\mathrm{R}}$ `e m` $\Rightarrow$ `VCG m e`. We define:

$_{\texttt{std}}$`vcg` $=$ `foldStmt` $_{\texttt{std}}$`logic`

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. For example, consider the following weaker variants of the `while` rule:

Definition 3.1 : The b-rule

```
b_ruleWhile $i\ g\ t_{body}\ q$ =
    do
    $p \leftarrow t_{body}$ true
    record $(i \wedge \neg g \Rightarrow q)$
    record $(i \wedge g \Rightarrow p)$
    return true
```

Definition 3.2 : The i-rule

```
i_ruleWhile $i\ g\ t_{body}\ q$ =
    do
    $p \leftarrow t_{body}\ i$
    record $(i \wedge g \Rightarrow p)$
    return true
```

We can easily construct the corresponding logics by replacing the standard `while` rule with the variants above, and then construct the VCGs:

Definition 3.3 : The b and i Logics and VCGs

$$\begin{aligned}
{}_{\texttt{b}}\texttt{logic} &= {}_{\texttt{std}}\texttt{logic}\{\texttt{While} = \texttt{b\_ruleWhile}\} \\
{}_{\texttt{i}}\texttt{logic} &= {}_{\texttt{std}}\texttt{logic}\{\texttt{While} = \texttt{i\_ruleWhile}\} \\
{}_{\texttt{b}}\texttt{vcg} &= \texttt{foldStmt}\ {}_{\texttt{b}}\texttt{logic} \\
{}_{\texttt{i}}\texttt{vcg} &= \texttt{foldStmt}\ {}_{\texttt{i}}\texttt{logic}
\end{aligned}$$

When given a program P, ${}_{\texttt{b}}\texttt{vcg}$ 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 `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, ${}_{\texttt{b}}\texttt{vcg}$ will produce less verification conditions, which may be more suitable for limited budget verification. The other VCG, ${}_{\texttt{i}}\texttt{vcg}$, 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.

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 precondition 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 now assume that the class also offers a function `unsafe` $:: e \rightarrow [e]$, that, when given an expression e, will return a list $[c_1, \ldots, c_n]$ such that $c_1 \vee \ldots \vee c_n$ specifies an upper bound of unsafe states for the evaluation of e. Consequently, to guarantee safe evaluation of e, it is sufficient to guarantee that it is evaluated in a state satisfying $\neg c_1 \wedge \ldots \wedge \neg c_n$. We will assume a function `safe`, such that `safe` e will return $\neg c_1 \wedge \ldots \wedge \neg c_n$.

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 ${}_{\texttt{SE}}\texttt{extend} :: e \rightarrow \texttt{Transformer}\ m\ e \rightarrow \texttt{Transformer}\ m\ e$. We define:

$${}_{\texttt{SE}}\texttt{extend}\ e\ T\ q \quad = \quad \texttt{do}\ \{p \leftarrow T\ q\ ;\ \texttt{return}\ (\texttt{safe}\ e\ \wedge\ p)\}$$

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

$$_{\mathtt{SE}}\mathtt{ruleAsg}\; x\; e \quad = \quad {}_{\mathtt{SE}}\mathtt{extend}\; e\; (\mathtt{ruleAsg}\; x\; e)$$

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 guanrantee 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

$$\begin{array}{lcllcl} M_{\mathtt{SE}} & = & (\;(\lambda\; x\; e & & \rightarrow & {}_{\mathtt{SE}}\mathtt{extend}\; e) \\ & & ,\;(\lambda\; t_1\; t_2 & & \rightarrow & \mathsf{id}) \\ & & ,\;(\lambda\; g\; t_1\; t_2 & & \rightarrow & {}_{\mathtt{SE}}\mathtt{extend}\; g) \\ & & ,\;(\lambda\; i\; g\; t & & \rightarrow & {}_{\mathtt{SE}}\mathtt{extendWhile}\; i\; g)) \end{array}$$

where

$$_{\mathtt{SE}}\mathtt{extendWhile}\; i\; g\; T\; q \quad = \quad \mathsf{do}\; \{T\; q;\; \mathtt{record}(i \Rightarrow \mathtt{safe}\; g)\}$$

Now we can apply the modifier to a logic. For example, $M_{\mathtt{SE}}$ `<$>` $_{\mathtt{std}}\mathtt{logic}$ will result in the standard L_0 logic with the SE extensions. For more lightweight verification, we can construct $M_{\mathtt{SE}}$ `<$>` $_{\mathtt{b}}\mathtt{logic}$, 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, 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 [8, 19] 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 $\mathtt{Monad}^R$. 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 $\mathtt{Monad}^R$ with a new class, called $\mathtt{Monad}^D$ that adds an operation `recordDebugInfo` for inserting new information to the validation trace. We call instances of the class $\mathtt{Monad}^D$ *debugger monads.*

Definition 3.6 : Debugger Monad

$$\begin{array}{l} \mathtt{class}\; \mathtt{Monad}^R\; e\; m \Rightarrow \mathtt{Monad}^D\; e\; m \\ \quad \mathtt{where} \\ \quad \mathtt{recordDebugInfo} :: \mathtt{String} \rightarrow e \rightarrow m() \end{array}$$

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

$$M_{\mathtt{VT}} \quad = \quad (m_{\mathtt{asg}},\; (\lambda\; t_1\; t_2 \rightarrow \mathsf{id}),\; (\lambda\; g\; t_1\; t_2 \rightarrow \mathsf{id}),\; (\lambda\; i\; g\; t \rightarrow \mathsf{id}))$$

where:

$$\begin{array}{lll} m_{\texttt{asg}}\; x\; e\; r\; q & = & \textsf{do} \\ & & p \rightarrow r\; q \\ & & \texttt{recordDebugInfo}\; ""\; q \\ & & \texttt{recordDebugInfo}\; (x + \!\! + \; "\; :=\; " + \!\! + \; \texttt{show}\; e)\; p \\ & & \texttt{return}\; p \end{array}$$

□

We can use this modifier on any logic. For example $M_{\texttt{VT}}$ `<$>` ${}_{\texttt{std}}$`logic` will extend the standard logic with the above trace validation feature; $M_{\texttt{VT}}$ `<$>` ($M_{\texttt{SE}}$ `<$>` ${}_{\texttt{std}}$`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<=i }

{ 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: `raise` and `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, `try` $\texttt{S}_1$ `catch` $\texttt{S}_2$, 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 minimalists 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 `raise` and try:

Definition 4.1 : CONSERVATIVE `raise` RULE

$$\texttt{ruleRaise}\; q \;=\; \texttt{return ff}$$

Definition 4.2 : CONSERVATIVE `try` RULE

$$\texttt{ruleTry}\; T_{\texttt{try}}\; T_{\texttt{catch}}\; q \;=\; T_{\texttt{try}}\; q$$

In particular, `ruleRaise` returns an `ff` as the pre-condition, which means that the rule actually wants to forbid an execution leading to `raise`. Consider $M_{\texttt{SE}}$ `<$>` ${}_{\texttt{std}}$`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 `catch` part can be igored, which what `ruleTry` above does. So, the new logic for the extended L_0 can be built by:

$$\texttt{ruleRaise} \;\mid\; \texttt{ruleTry} \;\mid\; (M_{\texttt{SE}}\; \texttt{<\$>}\; {}_{\texttt{std}}\texttt{logic})$$

A more reasonable extension, however, will really deal with exceptions rather than simply excluding them. Borrowing ideas from [13, 8], 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:

$$\texttt{ruleRaise}\ (q, q') \ = \ \texttt{return}\, q'$$

$$\texttt{ruleTry}\ T_{\texttt{try}}\ T_{\texttt{catch}}\ (q, q') \quad = \quad \texttt{do}\ \{\ p' \leftarrow T_{\texttt{catch}}\ (q, q')\ ;\ T_{\texttt{try}}\ (q, p')\ \}$$

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 *threading* the exceptional post-condition, though this is not the way post-conditions are normally treated[3]. 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 information in the exceptional post-condition remains most of the time constant, except when we encounter a `try` structure. We can still handle it, without too much loss of abstraction, as a threaded parameter; thus as part of the state of the used monad.

Below we introduce a class $\texttt{Monad}^E$ which extends $\texttt{Monad}^R$ 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.

Definition 4.3 :

$$\begin{array}{l}
\texttt{class Monad}^R\ e\ m \Rightarrow \texttt{Monad}^E\ e\ m \\
\quad \texttt{where} \\
\quad \texttt{getPostE} :: m\ e \\
\quad \texttt{setPostE} :: e \rightarrow m()
\end{array}$$

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

Definition 4.4 : `raise` Rule

$$_{\texttt{E}}\texttt{ruleRaise}\ q \ = \ \texttt{getPostE}$$

Definition 4.5 : `try` Rule

$$\begin{array}{lll}
_{\texttt{E}}\texttt{ruleTry}\ T_{\texttt{try}}\ T_{\texttt{catch}}\ q & = & \texttt{do} \\
 & & q' \leftarrow \texttt{getPostE} \\
 & & p' \leftarrow T_{\texttt{catch}}\ q \\
 & & \texttt{setPostE}\ p'\ ;\ p \leftarrow T_{\texttt{try}}\ q \\
 & & setPostE\ q' \\
 & & \texttt{return}\ p
\end{array}$$

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:

$$_{\texttt{E}}\texttt{ruleRaise}\ \mid\ \texttt{ruleTry}\ \mid\ (M_{\texttt{SE}}\ \texttt{<\$>}\ _{\texttt{std}}\texttt{logic})$$

And if we prefer a more lightweight logic, we can, for example, replace $_{\texttt{std}}\texttt{logic}$ above with $_{\texttt{b}}\texttt{logic}$.

[3] In attribute grammars, post-conditions are most naturally represented by an inherited attribute. Such an attribute is however awkward to mimic with a monad.

5 Related Works

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 [10] 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 [11]. 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 [9]. 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 modularly build semantics has actually been realized much earlier —see for example the work of Moggi [16] and Liang and Hudak [14]. As opposed to using monads in the (executional) 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 [18]. Attribute grammar provides an abstract and convenient way to specify recursive computation over a parsing tree. Essentially, such a specification is an algebra, which is specified in terms of computation on attributes being passed 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. The approach offers a number of useful abstraction tools, which are quite complementary to the monad approach. To specify a generic monadic logic in an attribute grammar style would require polymorphic attributes though; and not all attribute grammar tools support polymorphic attributes.

6 Conclusion

We have described a technique to modularly change and extend a logic to obtain derivative verification condition generators. It has been implemented on Haskell using a small case study as a proof of principle. From the small experiment that we did, the technique seems to have a number of practical advantages that make the implementation of programming logics safer and much easier to maintain. We believe that it is worth further investigation.

Research Direction

This paper only describes a construction technique. Having constructed a new logic, for example via an extension on a base logic, one is still confronted by the question of whether the new logic is sound. We would want the compositionality to extend to the proof level. That is, want to be able to lift the soundness result of the old logic to the new one with minimal effort, and to be able to do so without having to tamper with the old proof. In particular, this will be a very useful in embedding. We have not conducted any experiment in this direction yet. There are some works addressing the issue, for example that of Liang and Hudak [14] and Harisson [4]. These works show examples suggesting that proof compositionality via monads is attainable. However, the examples do not specifically deal with logic extension, so we feel that more experiments are still needed.

We also have not tried the technique on a real case study, so there also lies a possible continuation. Also, Haskell, which we use in our prototype implementation, is actually a bit short to express some of the generic concepts we have. One may want to investigate combination with generic programming, e.g. like is done in [5].

References

[1] R. C. Backhouse. *Program Construction and Verification*. Prentice Hall, London, 1986.

[2] Richard J. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

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

[4] W.L. Harrison. Proof Abstraction for Imperative Languages, 2003. draft, URL: `www.cs.missouri.edu/~harrison`.

[5] Ralf Hinze. A new approach to generic functional programming. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, 2000.

[6] C.A.R. Hoare. An axiomatic basis for computers programs. *Commun. Ass. Comput. Mach.*, 12:576–583, 1969.

[7] P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In *LNCS: Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859, pages 269–284, 1994.

[8] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands, 2001.

[9] Joe Hurd. Formal verification of probabilistic algorithms. Technical Report UCAM-CL-TR-566, University of Cambridge, Computer Laboratory, May 2003.

[10] Bart Jacobs and Erik Poll. A monad for basic Java semantics. *Lecture Notes in Computer Science*, 1816:150–??, 2000.

[11] Bart Jacobs, Joachim van den Berg, Huisman Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes: preliminary report. *ACM SIGPLAN Notices*, 33(10):329–340, October 1998.

[12] J.T. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.

[13] K. R. M. Leino. Toward reliable modular programs. Technical Report cs-tr-95-03, California Institute of Technology, 1995.

[14] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[15] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–280, October 1990.

[16] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[17] Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998.

[18] Jukka Paakki. Attribute grammar paradigms – A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[19] C. Pierik and F.S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI*, pages 64–78, 2003.

[20] S.D. Swierstra. Homepage of the AG system, 1999.
URL: `www.cs.uu.nl/groups/ST/Software/UU_AG/index.html`.

[21] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

A Algebraic Data Type View

Algebraic theories of data types, e.g. as by Malcolm [15], allow algebras on data types to be treated abstractly. This note just gives an outline of how to look at this paper from the algebraic angle. It does not explain the theories, nor the notation. A data type can be described by a functor F which specifies the structure of the data type's constructors. An algebra over r can be given a general type $F\ r \to r$. Thus, a logic has a general type:

$$F\ (\texttt{Transformer}\ e\ r) \to \texttt{Transformer}\ e\ r$$

A modifier M is a function of type $F\ r \to r \to r$. The operator `<$>` can be defined generally as:

$$M\ \texttt{<\$>}\ \psi = (\lambda\ a \to M\ a\ (\psi\ a))$$

For example, the SE modifier can be defined in this style:

$$\begin{array}{lcl} M_{\texttt{SE}} & = & (\lambda\ x\ e \to {}_{\texttt{SE}}\texttt{extend}\ e)\ +\ (\lambda\ t_1\ t_2 \to \mathsf{id})\ +\ (\lambda\ g\ t_1\ t_2 \to {}_{\texttt{SE}}\texttt{extend}\ g) \\ & & + \\ & & (\lambda\ i\ g\ t \to {}_{\texttt{SE}}\texttt{extendWhile}\ i\ g) \end{array}$$

B The Class Expression

Definition B.1 :

$$\begin{array}{l} \texttt{class Expression}\ e\ \texttt{where} \\ \quad \begin{array}{lcl} \mathsf{true} & :: & e \\ \mathsf{false} & :: & e \\ 0 & :: & e \\ \wedge & :: & e \to e \to e \\ \vee & :: & e \to e \to e \\ \Rightarrow & :: & e \to e \to e \\ \neg & :: & e \to e \\ \texttt{subst} & :: & (\texttt{String}, e) \to e \to e \\ \texttt{unsafe} & :: & e \to [e] \end{array} \end{array}$$

C Direct Implementation

Below is a direct implementation of a VCG for L_0 as a recursive function over `Stmt`. The type variable `e` represents the type of expressions; it has to be an instance of class `Expression` given above.

The function `vcg` takes a statement as an argument. Its third parameter is the post condition which is to be reduced. The result is a pair (`vcs`, `p`) where `p` is the resulting pre-condition and `vcs` is a list of collected verification conditions. To collect the verification conditions the list `vcs` is threaded through the function in its second argument.

```
vcg :: Expression e => Stmt -> [e] -> e -> ([e],e)
vcg (x:=e)     vcs q  = (vcs, subst (x,e) q)
vcg (s1 :> s2) vcs q  = (vcs1,p1)
    where
    (vcs2,p2) = vcg vcs  q
    (vcs1,p1) = vcg vcs2 p2
vcg (While i g body) vcs q = (c1:c2:vcs',i)
    where
    c1 = i /\ neg g ==> q
    c2 = i /\ g ==> p
    (vcs',p) = vcg vcs i
vcg (IfElse g s1 s2) vcs q = (vcs2,p)
    where
    (vcs1,p1) = vcg s1 vcs  q
    (vcs2,p2) = vcg s2 vcs1 q
    p = (g ==> p1) /\ (neg g ==> p2)
```

D Download

A Haskell implementation of the techniques mentioned by the paper on the language L_0 can be downloaded from our subversion server:

```
https://svn.cs.uu.nl:12443/repos/wpprojects/xmech/papers/pvcg/haskellmodels/
```

Notice the non-standard port number, as some site may put restrictions on accessing non-standard port numbers.