Incremental Implementation of Syntax Driven Logics*

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 combining higher order functions, algebraic datatypes, and monads to incrementally implement syntax driven logics. Extensions can be compositionally stacked while the base logic is left unchanged. The technique can furthermore be used to build a set of weaker logics for light weight verification or to generate validation traces. The paper explains the technique through an example: a Hoare logic for a simple command language. The example also shows how exceptions can be treated as an extension, without having to change the underlying logic.**

***Index Terms*— syntax driven logic, algebraic data type, modular logic, verification tool.**

I 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 language may be changed; so the logic has to be changed as well. Logics underlying imperative languages are usually syntax driven and are typically implemented as recursive functions over the target program. This essentially weaves the logic into the code that drives the recursion. The result is a monolithic program that cannot be modified without tampering with the code. From experience we learn that this is a dangerous and error prone operation which can easily introduce inconsistencies in the logic.

This paper describes a technique to incrementally implement a sytax driven logic, such that variants and extensions can be added modularly, i.e. without directly tampering with the code of the old implementation. This has a number of interesting advantages: (1) it is safer; (2) it allows modifications to be engaged or disengaged like plug-ins; and (3) it is easy to create a set of partial logics, each of which can be used in isolation for light weight verification.

The technique uses a combination of algebras, higher order functions, and monads to represent a syntax driven logic. Algebras are modular structures used to abstractly specify recursive computation [1]. Higher order functions are used to implement modifications on algebras. Monads have been recognized to be a useful tool to build modular semantics [2]. 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 modify the logic's structure. The change can be delayed to the instantiation of the corresponding monad class.

The paper also addresses the soundness issue. It discusses conditions under which modifying a logic $\mathcal{L}$ to $\mathcal{L}'$ would preserve the soundness of $\mathcal{L}$ and shows an example of an incremental proof. The latter is an issue when the new logic often has to satisfy a modified no-

*Based on "Building Verification Condition Generators by Compositional Extensions" by I.S.W.B. Prasetya, A. Azurat, T.E.J. Vos, and A. van Leeuwen which appeared in the Proceedings of 3rd IEEE International Conference on Software Engineering and Formal Methods, Koblenz, Germany, 2005. ©2005 IEEE.

†RUTI research grant.
‡NWO research grant

tion of soundness. Rather than proving this from the scratch an incremental proof reuses facts known about $\mathcal{L}$. In particular, it does not tamper with the proof of $\mathcal{L}$. This may be interesting in the context of *deep embedding*. Deep embedding implements a (programming) logic by modelling its syntax and semantics in the logic of a theorem prover. Excellent examples are the embedding of Java in Isabelle by Huisman [3] and C in HOL by Norrish [4]. It is a very safe approach, since it allows the soundness of the embedded logic to be verified. Suppose we want to make variants of the embedded logic. To prove their soundness, we could rework the proof of the base logic. Usually, this is an expensive operation. Furthermore, after several changes, the proofs will have a maintenance problem. An incremental proof offers a more maintainable alternative.

A 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 `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`) [5]. 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 Haskel class and monads will be quite helpful. An actual Haskell implementation of the example is available on request.

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:

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

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:

$$f :: \texttt{Eq}\ a \Rightarrow a \to \texttt{Bool}$$

is used to say that f is a function of type $a \to$ `Bool`, but it is also required that the type a is a known instance of the class `Eq`. The use of $\Rightarrow$ in the above notation should not be confused with $\Rightarrow$ in the predicate logic. A class can also be defined to extend another class, like:

$$\texttt{class Eq}\ a \Rightarrow \texttt{Ord}\ a\ \ \texttt{where}\ (<) :: a \to a \to \texttt{Bool}$$

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.

In the rest of the paper, type variable m is assumed to range over monads, and the type variable e represents 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: $\texttt{true}, \texttt{false}, 0, \wedge, \vee, \Rightarrow$ and $\neg$. And in addition, the following two operations:

$$\begin{array}{lcl} \texttt{subst} & :: & (\texttt{String}, e) \to e \to e \\ \texttt{safe} & :: & e \to e \end{array}$$

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

B Paper Outline

Section II explains our representation. Section III shows examples of how a logic can be altered and extended modularly. Section IV 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 modification to the logic's structure is needed as well. Normally this would require surgery on the implementation of the old logic. Section IV shows how it can be done without. Sections V and VI discuss soundness issues. Finally, related work and conclusion are given in Sections VII and VIII.

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

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

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

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

$$\frac{p \ = \ \texttt{pre}\ S\ i \qquad \vdash\ i \wedge \neg g \Rightarrow q \quad , \quad \vdash\ i \wedge g \Rightarrow p}{\texttt{pre}\ (\texttt{inv}\ i\ \texttt{while}\ g\ \texttt{do}\ S)\ q \ = \ i}$$

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

II Representating Syntax Driven 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 [7, 8, 9]. 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[1]. It can be shown that $\vdash\ p \Rightarrow \texttt{pre}\ P\ q$ implies $\{p\}\ P\ \{q\}$. Hence, a Hoare triple specification can be reduced to a problem expressed in terms of `pre`.

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.

Definition II.1 : `Stmt`

```
data Stmt e  =  String := e
             |  Stmt :> Stmt
             |  IfElse e Stmt Stmt
             |  While e e Stmt
```

`:=` 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. A fraction of a straightforward implementation of `pre` and the corresponding VCG is shown below. Their computations are merged into one recursive function `pvcg`. When given a program S, a post-condition q, and an initially empty list of verification conditions, `pvcg` S q [] returns a tuple (V, p) where V is a list of verification conditions generated along the way, and p is `pre` S q.

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

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

A Algebras

Any data type $\mathcal{T}$ induces a so-called fold function: a higher order function that defines a recursive pattern over $\mathcal{T}$. For the data type `Stmt` from Definition II.1, the corresponding fold function is:

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

The tuple $A = (A_{\texttt{asg}}, A_{\texttt{seq}}, A_{\texttt{if}}, A_{\texttt{while}})$ 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*. So, a tuple A such as above is also called an `Stmt`-algebra over r; and if the target data type is $\mathcal{T}$ rather than `Stmt`, then it is called a $\mathcal{T}$-algebra. Notice that via a fold function, an algebra can be said to *abstractly* specify a recursive computation. The notation $(\!|A|\!)$ is used to denote the function obtained by folding the algebra A as above. So, with respect to `Stmt`, we have:

$$(\!|A|\!) \ = \ \texttt{foldStmt}\ A$$

Algebraic theories of data types, e.g. [1, 10], use Category Theory to abstractly and generically express properties of algebras, e.g. without having to be explicit about the structure of the underlying data type. In this paper we stick to a more classical (non-categorical) approach.

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 $\mathcal{T}$-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.

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

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

by applying what we will call a *modifier*. For `Stmt`-algebras, the modifiers will have the following type:

$$\begin{array}{lll} \texttt{type Modifier}\ e\ r & = & (\texttt{String} \to e \to (r \to r) \\ & & ,(r \to r) \\ & & ,e \to (r \to r) \\ & & ,e \to e \to (r \to r)) \end{array} \quad (1)$$

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

Definition II.2 : Applying a Modifier

$$\begin{array}{lll} M\ \texttt{<\$>}\ A & = & ((\lambda\ x\ e & \to M_{\texttt{asg}}\ x\ e\ (A_{\texttt{asg}}\ x\ e)) \\ & & ,(\lambda\ r_1\ r_2 & \to M_{\texttt{seq}}\ (A_{\texttt{seq}}\ r_1\ r_2)) \\ & & ,(\lambda\ g\ r_1\ r_2 & \to M_{\texttt{if}}\ g\ (A_{\texttt{if}}\ g\ r_1\ r_2)) \\ & & ,(\lambda\ i\ g\ r & \to M_{\texttt{while}}\ i\ g\ (A_{\texttt{while}}\ i\ t\ g\ r))) \end{array}$$

B Brief on Monads

We will give a brief overview of monads that is sufficient to understand the rest of this paper. A more inspiring introduction on monads can be found in [11]. There are also plenty of texts at `www.haskell.org`. In Haskell `monad` is a class of type constructors supporting these two operations:

$$\begin{array}{l} \texttt{class Monad}\ m\ \texttt{where} \\ \quad \texttt{return} :: a \to m\ a \\ \quad \texttt{>>=} :: m\ a \to (a \to m\ b) \to m\ b \end{array}$$

There are some algebraic properties which the operations should satisfy —see [11]. 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; c `>>=` f executes c then passes the value v it returns to f and executes $f\ v$. For example we can use $m\ a = \texttt{Int} \to (\texttt{Int}, a)$ to represent computations whose state is a single integer. We can define `return` and `>>=` as follows:

$$\begin{array}{lcl} \texttt{return}\ x\ s & = & (s, x) \\ (c\ \texttt{>>=}\ f)\ s & = & f\ v\ t \text{ where } (t, v) = c\ s \end{array}$$

In this way we can mimic an imperative program in a functional language. 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:

$$\texttt{do}\ \{\ q \leftarrow t_1\ r\ ;\ p \leftarrow t_2\ q\ ;\ \texttt{return}\ p\ \}$$

or equivalently like:

$$\begin{array}{ll} \texttt{do} & q \leftarrow t_1\ r \\ & p \leftarrow t_2\ q \\ & \texttt{return}\ p \end{array}$$

The code will be translated to:

$$t_1\ r\ \texttt{>>=}\ (\lambda q \to t_2\ q\ \texttt{>>=}\ (\lambda p \to \texttt{return p}))$$

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

C 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:

$$e \to [e] \to ([e], e)$$

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 II.3 : Transformers
Let m be a monad.

$$\texttt{type Transformer}\ m\ e \quad = \quad e \to m\ e$$

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 II.4 : Recorder Monad

$$\texttt{class Monad}\ m \Rightarrow \texttt{Monad}^R\ e\ m \quad \texttt{where record} :: e \to m()$$

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:

$$\texttt{pre} :: \texttt{Monad}^R\ e\ m \ \Rightarrow\ \texttt{Stmt}\ e \to \texttt{Transformer}\ m\ e$$

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:

$$\begin{array}{lll} \texttt{ruleWhile} & (\texttt{While}\ i\ g\ \mathit{body})\ q & \\ & = \texttt{do}\ p \leftarrow \texttt{pre}\ \mathit{body}\ q & \\ & \quad \texttt{record}\ (i \wedge \neg g \Rightarrow q) & \\ & \quad \texttt{record}\ (i \wedge g \Rightarrow p) & \\ & \quad \texttt{return}\ i & \end{array}$$

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

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:

$$\texttt{StmtAlgebra}\ e\ (\texttt{Transformer}\ e\ m)$$

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 II.5 : Family of L_0-logics

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

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 II.6 : The Standard L_0-logic

$$\begin{array}{l}{}_{\texttt{std}}\texttt{logic}\\ =\\ (\texttt{ruleAsg},\ \texttt{ruleSeq},\ \texttt{ruleIfElse},\ \texttt{ruleWhile})\end{array}$$

where the rules are defined as in Figure 2.

From now on, we will not distinguish a value of type ${}_{\texttt{std}}\texttt{logic}$ 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 ${}_{\texttt{std}}\texttt{logic}$ 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 II.7 : Representation of VCG

$$\texttt{type}\ \texttt{VCG}\ m\ e\ \ =\ \ \texttt{Stmt}\ e \rightarrow \texttt{Transformer}\ m\ e$$

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

Definition II.8 : The Standard VCG for L_0
Let: ${}_{\texttt{std}}\texttt{vcg}\ ::\ \texttt{Monad}^{\texttt{R}}\ \texttt{e}\ \texttt{m}\ \Rightarrow\ \texttt{VCG}\ \texttt{m}\ \texttt{e}$. We define:

$${}_{\texttt{std}}\texttt{vcg}\ =\ (\!|{}_{\texttt{std}}\texttt{logic}|\!)$$

$$\begin{array}{l}\texttt{ruleAsg}\ x\ e\ q\ \ =\ \ \texttt{return}\ (\texttt{subst}\ (x,e)\ q)\\[1ex]
\texttt{ruleSeq}\ \ t_1\ t_2\ q\\ \quad =\texttt{do}\ \ p_2 \leftarrow t_2\ q\\ \qquad\quad p_1 \leftarrow t_1\ p_2\\ \qquad\quad \texttt{return}\ p_1\\[1ex]
\texttt{ruleIfElse}\ \ g\ t_1\ t_2\ q\\ \quad =\texttt{do}\ \ p_1 \leftarrow t_1\ q\\ \qquad\quad p_2 \leftarrow t_2\ q\\ \qquad\quad \texttt{return}\ ((g \Rightarrow p_1) \wedge (\neg g \Rightarrow p_2))\\[1ex]
\texttt{ruleWhile}\ \ i\ g\ t_{body}\ q\\ \quad =\texttt{do}\ \ p \leftarrow t_{body}\ i\\ \qquad\quad \texttt{record}\ (i \wedge \neg g \Rightarrow q)\\ \qquad\quad \texttt{record}\ (i \wedge g \Rightarrow p)\\ \qquad\quad \texttt{return}\ i\end{array}$$

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

III 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 III.1 : The b and i Logics and VCGs

$$\begin{array}{lcl}{}_{\texttt{b}}\texttt{logic} & = & {}_{\texttt{std}}\texttt{logic}\{\texttt{While} = \texttt{b\_ruleWhile}\}\\ {}_{\texttt{i}}\texttt{logic} & = & {}_{\texttt{std}}\texttt{logic}\{\texttt{While} = \texttt{i\_ruleWhile}\}\\ {}_{\texttt{b}}\texttt{vcg} & = & (\!|{}_{\texttt{b}}\texttt{logic}|\!)\\ {}_{\texttt{i}}\texttt{vcg} & = & (\!|{}_{\texttt{i}}\texttt{logic}|\!)\end{array}$$

The definition of `b_ruleWhile` and `i_ruleWhile` rules are given below. 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 fewer verification conditions; which is more suitable for 'light' 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.

Definition III.2 : The b-rule

$$\begin{array}{l}\texttt{b\_ruleWhile}\ \ i\ g\ t_{body}\ q\\ \quad =\texttt{do}\ \ p \leftarrow t_{body}\ \texttt{true}\\ \qquad\quad \texttt{record}\ (i \wedge \neg g \Rightarrow q)\\ \qquad\quad \texttt{record}\ (i \wedge g \Rightarrow p)\\ \qquad\quad \texttt{return}\ \texttt{true}\end{array}$$

Definition III.3 : THE I-RULE

$$\begin{array}{ll} \texttt{i\_ruleWhile} & i\ g\ t_{body}\ q \\ & = \texttt{do}\ \ p \leftarrow t_{body}\ i \\ & \qquad \texttt{record}\ (i \wedge g \Rightarrow p) \\ & \qquad \texttt{return true} \end{array}$$

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 $\texttt{safe} :: e \rightarrow e$. The idea is that given an expression e, `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 III.4 : SE RULE EXTENSION
Let:

$\texttt{SE}\texttt{extend} :: e \rightarrow \texttt{Transformer}\ m\ e \rightarrow \texttt{Transformer}\ m\ e$

We define:

$$\begin{array}{l} \texttt{SE}\texttt{extend}\ e\ t\ q \\ \quad = \texttt{do}\ \{\ p \leftarrow t\ q\ ;\ \texttt{return}\ (\texttt{safe}\ e\ \wedge\ p)\ \} \end{array}$$

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

$$\texttt{SE}\texttt{ruleAsg}\ x\ e \quad = \quad \texttt{SE}\texttt{extend}\ e\ (\texttt{ruleAsg}\ x\ e)$$

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 III.5 : SE MODIFIER

$$\begin{array}{l} M_{\texttt{SE}} \\ = \\ ((\lambda\ x\ e \rightarrow \texttt{SE}\texttt{extend}\ e),\ \mathsf{id},\ \texttt{SE}\texttt{extend},\ \texttt{SE}\texttt{extendWhile}) \end{array}$$

where

$$\texttt{SE}\texttt{extendWhile}\ i\ g\ t\ q \quad = \ \texttt{do}\ \{\ t\ q\ ;\ \texttt{record}(i \Rightarrow \texttt{safe}\ g)\ \}$$

Now we can apply the modifier to a logic. For example, $M_{\texttt{SE}}$ `<$>` $_{\texttt{std}}\texttt{logic}$ will result in the standard L_0 logic with the SE extensions. For more lightweight verification, we can construct $M_{\texttt{SE}}$ `<$>` $_{\texttt{b}}\texttt{logic}$, 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, 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 [3, 12] 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 $\texttt{Monad}^R$. 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 $\texttt{Monad}^R$ with a new class, called $\texttt{Monad}^D$ that adds an operation `recordDebugInfo` for inserting new information to the validation trace. We call instances of the class $\texttt{Monad}^D$ *debugger monads*.

Definition III.6 : DEBUGGER MONAD

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

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

Definition III.7 : VT MODIFIER

$$M_{\texttt{VT}} \quad = \quad (m_{\texttt{asg}},\ \mathsf{id},\ (\lambda\ g \rightarrow \mathsf{id}),\ (\lambda\ i\ g \rightarrow \mathsf{id}))$$

where:

$$
\begin{array}{l}
m_{\texttt{asg}} x\ e\ r\ q \\
= \texttt{do}\ \ p \rightarrow r\ q \\
\qquad \texttt{recordDebugInfo}\ (\texttt{show}\ q) \\
\qquad \texttt{recordDebugInfo}\ (\texttt{show}\ (x{:=}e)) \\
\qquad \texttt{recordDebugInfo}\ (\texttt{show}\ p) \\
\qquad \texttt{return}\ p
\end{array}
$$

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.

IV 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 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 `raise` and try:

Definition IV.1 : Conservative `raise` Rule

$$\texttt{ruleRaise}\ q\ =\ \texttt{return false}$$

Definition IV.2 : Conservative `try` Rule

$$\texttt{ruleTry}\ t_{\texttt{try}}\ t_{\texttt{catch}}\ q\ =\ t_{\texttt{try}}\ q$$

In particular, `ruleRaise` returns an `false` 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, 3], 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:

$$
\begin{array}{lcl}
\texttt{ruleRaise}\ (q, q') & = & \texttt{return}\ q' \\
\texttt{ruleTry}\ t_{\texttt{try}}\ t_{\texttt{catch}}\ (q, q') & = & \texttt{do}\ \ p' \leftarrow t_{\texttt{catch}}\ (q, q') \\
 & & \qquad t_{\texttt{try}}\ (q, p')
\end{array}
$$

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 information 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 $\texttt{Monad}^E$ 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 IV.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 IV.4 : `raise` RULE

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

Definition IV.5 : `try` RULE

$$\begin{array}{l}{}_{\texttt{E}}\texttt{ruleTry}\ t_{\texttt{try}}\ t_{\texttt{catch}}\ q\\ \quad = \quad \texttt{do}\ \{\quad q' \leftarrow \texttt{getPostE}\ ;\ p' \leftarrow t_{\texttt{catch}}\ q\ ;\\ \qquad\qquad\quad \texttt{setPostE}\ p'\ ;\\ \qquad\qquad\quad p \leftarrow t_{\texttt{try}}\ q\ ;\\ \qquad\qquad\quad \texttt{setPostE}\ q'\ ;\\ \qquad\qquad\quad \texttt{return}\ p\}\end{array}$$

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:

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

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

V PRESERVING SOUNDNESS

Modifying a logic may introduce inconsistencies. This section discusses conditions sufficient for soundness preserving modifications.

Consider a programming language L and a syntax driven logic $\mathcal{L}$ for L. $\mathcal{L}$ is represented by an algebra A. So, $\mathcal{L} = (\!|A|\!)$. We also write $\mathcal{L}.\mathsf{alg}$ to denote A. Note that $\mathcal{L}$ is a function that maps sentences of L to some results domain. Let τ be the type of these results. For example, in the logics discussed in the previous sections, elements of τ are predicate transformers.

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

Definition V.1 : SOUNDNESS

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

For example, below we show a soundness criterion ($C_{\texttt{std}}$) for the ${}_{\texttt{std}}\texttt{logic}$ of L_0. The monad $m\ e$ in the definition of ${}_{\texttt{std}}\texttt{logic}$ is below concretely instantiated to $[e] \rightarrow ([e], e)$; σ ranges over states; for a statement S, $\mathcal{E}\ S$ is the semantics of S, which for simplicity is assumed to be a function that maps an initial state to S's terminal state; for a predicate q the notation $\sigma \vdash q$ means that the state σ satisfies q.

$$\begin{array}{l}C_{\texttt{std}}\ S\ t\\ =\\ (\forall q, \sigma ::\quad \bigwedge V \text{ is valid } \wedge\ \sigma \vdash p\\ \qquad\qquad\ \Rightarrow\\ \qquad\qquad\ \mathcal{E}\ S\ \sigma \vdash q,\ \text{where } (V, p) = t\ q)\end{array}$$

If $\leq$ and $\sqsubseteq$ are partial orders, a function f is $(\leq, \sqsubseteq)$-monotonic if for all x, y: $x \leq y \Rightarrow f\ x \sqsubseteq f\ x$. Consider a partial order relation $\leq$ over τ with the intention that $t_1 \leq t_2$ implies that it is in some sense safe to replace t_1 with t_2. A soundness criterion C is said to be $\leq$-monotonic if it is $(\leq, \Rightarrow)$-monotonic on its second argument. For such a criterion, replacing a logic with a 'bigger' one is safe:

Theorem V.2 : BIGGER IS SAFE
If $(L, \mathcal{L})$ is sound wrt C, C is $\leq$-monotonic, and $\mathcal{M}$ is such that $(\forall S : S \in L : \mathcal{L}\ S \leq \mathcal{M}\ S)$, then $(L, \mathcal{M})$ is also sound wrt C.

Proof: We have to prove $C\ S\ (\mathcal{M}\ S)$ for all $S \in L$. Since C is $\leq$-monotonic, it suffices to prove: (1) $C\ S\ (\mathcal{L}\ S)$ and (2) $\mathcal{L}\ S \leq \mathcal{M}\ S$. The first follows from the soundness of $(L, \mathcal{L})$ wrt C, the second is assumed.

For example, $C_{\texttt{std}}$ defined before is $\leq$-monotonic, where $\leq$ is this partial order over predicate transformers:

Definition V.3 : A POSSIBLE ORDERING

$$\begin{array}{l}t_1 \leq t_2 \;=\; (\forall q ::\quad (p_1 \Leftarrow p_2)\ \wedge\ (\bigwedge V_1 \Leftarrow \bigwedge V_2)\\ \qquad\qquad\qquad\quad \text{where}\\ \qquad\qquad\qquad\quad (V_1, p_1) = t_1\ q\\ \qquad\qquad\qquad\quad (V_2, p_2) = t_2\ q\quad\ )\end{array}$$

Furthermore, we call an algebra A $\leq$-monotonic if for all components f of A, f is $(\leq, \leq)$-monotonic on all its τ-arguments. For example, the logic ${}_{\texttt{std}}\texttt{logic}$ is monotonic with respect to the $\leq$ defined above.

A modification to an algebra can be expressed in terms of a function F that transforms an algebra to another. We will use the term *adaptor* for F; we define the notion of 'ascending adaptor':

Definition V.4 : ASCENDING ADAPTOR
An adaptor F is $\leq$-*ascending* if:

$$(\forall S ::\ (\!|A|\!)\ S\ \leq\ (\!|F\ A|\!)\ S)$$

An alteration via an ascending adaptor is *safe*; this is just a simple corollary of Theorem V.2:

Corollary V.5 : SOUNDNESS PRESERVING ALTERATION
If: (1) $(L, \mathcal{L})$ is sound wrt C, (2) C is $\leq$-monotonic, and (3) F is $\leq$-ascending, then $(L,\ F\ \mathcal{L}.\mathsf{alg})$ is also sound wrt C.

For example, the application of the SE-modifier (Definition III.5) is ascending with respect to the $\leq$ in Definition V.3. Therefore, applying the SE-modifier on $_{\texttt{std}}\texttt{logic}$ will preserve C_{std} defined above.

The previous section shows four ways to alter $\mathcal{L}$: we can replace some components of $\mathcal{L}.\mathsf{alg}$, adding new components, instantiating the monad that parameterizes it, or apply a modifier to it. All these transformations are instances of adaptors. In general, proving that an adaptor 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 $\mathcal{A}.\mathsf{alg}$ 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 $\mathcal{L}.\mathsf{alg}$, it is sufficient to show that g is $\leq$-monotonic on all its τ-arguments; (3) if we apply a modifier M to $\mathcal{L}.\mathsf{alg}$, 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). 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.

VI Incremental Soundness Proof

When we alter a logic $\mathcal{L}$, Theorem V.2 allows us to infer that the new logic $\mathcal{M}$ respects the old soundness criterion. However, $\mathcal{M}$ may have to satisfy a different soundness criterion. Ideally, the soundness of the new criterion is proven incrementally. That is, by reusing old results as much as possible. This is possible if for example the new criterion D is just a conjunctive extension of the old one C. That is:

$$D\ S\ t\ =\ C\ S\ t\ \wedge\ C'\ S\ t$$

If the alteration from $\mathcal{L}$ to $\mathcal{M}$ has been shown to be soundness preserving, then the soundness of $\mathcal{M}$ with respect to D follows from its soundness with repect to C' —so, we only need to prove the latter. But this is a rather trivial case. If the alteration is non-trivial, reuse is in general difficult.

Our investigation reveals that reuse is *possible* from an inductive proof. Essentially this is because each case of such a proof leads to a partial soundness result which is independent of how the other cases are handled. So, such a result remains valid even if we change the other cases, and thus can be reused.

The use of monad also helps. All the rules and logics given so far are parameterized by the used monad: they will *abstractly* behave as specified regardless of the used monad. Their concrete behavior still depends on the used (concrete) monad. When two logics sharing several rules are obtained using different concrete monads, we can expect that there is a close relation between the two concrete incarnations of the shared rules. By exploiting this relation, it is possible to reuse the soundness proof of one logic in proving the other.

$\mathcal{E}\ S = (\!|(\mathcal{E}_{\texttt{asg}}, \mathcal{E}_{\texttt{seq}})|\!)\ S$ where:

$$\begin{array}{lcl} \mathcal{E}_{\texttt{asg}}\ x\ e\ \sigma & = & \texttt{update}\ (x, e)\ \sigma \\ \mathcal{E}_{\texttt{seq}}\ r_1\ r_2\ \sigma & = & r_2\ (r_1\ \sigma) \end{array}$$

Figure 3: An evaluator semantics for L^-.

In the sequel we will illustrate the technique with an example. For simplicity, we will strip down L_0 to L^- which just contains assignment and sequential composition. As the logic we will take $_{\texttt{std}}\texttt{logic}$ restricted to L^-; so:

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

The rules are abstractly defined in Figure 2, which implicitly takes a monad m as a parameter. The concrete behavior of these rules still depends the concrete choice of m. Recall that a rule implements a predicate transformer, see Definition II.3, which is a function of type $e \to m\ e$ where e represents the post-condition, and $m\ e$ is a monadic wrapping of the resulting pre-condition. For L^- we need no wrapping. So, $m\ e$ is just e; `return` is just the identity function; and the monad composition operator `>>=` becomes the plain function composition. The resulting concrete rules are given below:

Definition VI.1 : Concrete Rules of $_{\texttt{std}}\texttt{logic}$

$$\begin{array}{lcl} \texttt{ruleAsg}^-\ x\ e\ q & = & \texttt{subst}\ (x, e)\ q \\ \texttt{ruleSeq}^-\ t_1\ t_2\ q & = & t_1\ (t_2\ q) \end{array}$$

An evaluator semantics for L^- is given Figure 3: for a statement S, $\mathcal{E}\ S$ is a function that maps an initial state σ to S's final state when executed in σ. For expressions e, $\mathcal{E}\ e\ \sigma$ returns the value of the expression e on the state σ. The function `update` in the semantics of assignment changes a state. Rather than giving it a concrete definition we abstractly characterize it in terms of predicate substitution as follow:

Definition VI.2 : `update`

$$\texttt{update}\ (x, e)\ \sigma \vdash q\ =\ \sigma \vdash \texttt{subst}\ (x, e)\ q$$

Notice that this definition is independent of structure of σ, q, and how $\vdash$ interprets q on σ.

For the soundness criterion we take a simplified $C_{\texttt{std}}$ from Section V:

Definition VI.3 : Standard Soundness

$$C_{\texttt{std}}\ S\ t\ =\ (\forall q, \sigma ::\ \sigma \vdash t\ q\ \Rightarrow\ \mathcal{E}\ S\ \sigma \vdash q)$$

The logic $_{\texttt{std}}\texttt{logic}$ is *sound* with respect to the above criterion. To prepare for the incremental proof later, we will prove this soundness result inductively. For the assignment, it comes down to proving this:

Lemma VI.4 : ASG-CASE L^-

$$(\forall q, \sigma :: \ \sigma \vdash \texttt{ruleAsg}^-\ x\ e\ q \ \Rightarrow\ \mathcal{E}_{\texttt{asg}}\ x\ e\ \sigma \vdash q)$$

Proof: unfolding the definitions of $\texttt{ruleAsg}^-$ and $\mathcal{E}_{\texttt{asg}}$, it comes down to proving that $\sigma \vdash \texttt{subst}\ (x, e)\ q$ implies $\texttt{update}\ (x, e)\ \sigma\ \vdash q$. This follows from the definition of `update`.

For sequential composition it comes down to proving:

Lemma VI.5 : SEQ-CASE L^-

$$\begin{array}{l}(\forall q, \sigma :: \sigma \vdash t_1\ q \Rightarrow r_1\ \sigma \vdash q) \wedge \\ (\forall q, \sigma :: \sigma \vdash t_2\ q \Rightarrow r_2\ \sigma \vdash q) \\ \Rightarrow \\ (\forall q, \sigma :: \sigma \vdash \texttt{ruleSeq}^-\ t_1\ t_2\ q \ \Rightarrow\ \mathcal{E}_{\texttt{seq}}\ r_1\ r_2\ \sigma \vdash q)\end{array}$$

Proof:

$$\begin{array}{ll} & \mathcal{E}_{\texttt{seq}}\ r_1\ r_2\ \sigma \vdash q \\ = & \quad \{\ \text{def. } \mathcal{E}_{\texttt{seq}}\ \} \\ & r_2\ (r_1\ \sigma) \vdash q \\ \Leftarrow & \quad \{\ \text{the second inductive assumption}\ \} \\ & r_1\ \sigma \vdash t_2\ q \\ \Leftarrow & \quad \{\ \text{the first inductive assumption}\ \} \\ & \sigma \vdash t_1\ (t_2\ q) \\ = & \quad \{\ \text{def. } \texttt{ruleSeq}^-\ \} \\ & \sigma \vdash \texttt{ruleSeq}^-\ t_1\ t_2\ q \end{array}$$

Now we can conclude the soundness result:

Theorem VI.6 : SOUNDNESS OF $_{\texttt{std}}\texttt{logic}$

$_{\texttt{std}}\texttt{logic}$ is sound with respect to $C_{\texttt{std}}$

Proof: by an inductive proof. Lemma VI.4 proves the assignment case and Lemma VI.5 proves the sequential composition case.

Now we extend L^- by adding `raise` and `try` − `catch` constructs. Let us call the new language L_e^-. As with L^- the semantics will be given in terms of an evaluator called $\mathcal{X}$. This evaluator operates on a slightly extended state structure; it is of the form (σ, exc) where σ is an ordinary state, i.e. as used by the evaluator $\mathcal{E}$, and exc is a flag which is set to `true` to indicate an exceptional state, and `false` to inducate a normal state. On the new state structures we also define the following operations:

1. to bring a state back to a normal state: $\texttt{N}\ (\sigma, exc) = (\sigma, \texttt{false})$.

$\mathcal{X}\ S\ =\ (\!|(\mathcal{X}_{\texttt{asg}}, \mathcal{X}_{\texttt{seq}}, \mathcal{X}_{\texttt{raise}}, \mathcal{X}_{\texttt{try}})|)\ S$ where:

$$\begin{array}{lcl} \mathcal{X}_{\texttt{asg}}\ x\ e\ \rho & = & \texttt{if norm}\ \rho \\ & & \texttt{then}\ (\mathcal{E}_{\texttt{asg}}\ x\ e\ (\texttt{fst}\ \rho), \texttt{false}) \\ & & \texttt{else}\ \rho \\ \mathcal{X}_{\texttt{seq}}\ r_1\ r_2\ \rho & = & \mathcal{E}_{\texttt{seq}}\ r_2\ r_1\ \rho \\ \mathcal{X}_{\texttt{raise}}\ (\sigma, exc) & = & (\sigma, \texttt{true}) \\ \mathcal{X}_{\texttt{try}}\ r_{\texttt{try}}\ r_{\texttt{catch}}\ \rho & = & \texttt{if norm}\ r_{\texttt{try}}\ \rho \\ & & \texttt{then}\ r_{\texttt{try}}\ \rho \\ & & \texttt{else}\ r_{\texttt{catch}}\ (\texttt{N}\ (r_{\texttt{try}}\ \rho)) \end{array}$$

Figure 4: The evaluator semantics for L_e^-.

2. to check if a state is a normal state: $\texttt{norm}\ (\sigma, exc) = \neg exc$

We also need these functions to obtain the components of a pair: $\texttt{fst}\ (\sigma, exc) = \sigma$ and $\texttt{snd}\ (\sigma, exc) = exc$. The complete $\mathcal{X}$ semantics is given in Figure 4.

We use a pair of predicates to specify sets of extended states. Let $\rho = (\sigma, exc)$ be an extended state. Overloading the symbol $\vdash$, we write $\rho \vdash (q, z)$ to mean ρ satisfies (q, z), defined as:

Definition VI.7 : $\vdash$ ON EXTENDED STATE

$$\begin{array}{lcl} (\sigma, \texttt{true}) \vdash (q, z) & = & \sigma \vdash q \\ (\sigma, \texttt{false}) \vdash (q, z) & = & \sigma \vdash z \end{array}$$

As the logic for L_e^- we take L^-'s logic and extend it with the `raise` and `try` rules defined previously in Definitions IV.4 and IV.5 in Section IV:

$$_{\texttt{e}}\texttt{logic} \ =\ _{\texttt{E}}\texttt{ruleRaise}\ \mid\ _{\texttt{E}}\texttt{ruleTry}\ \mid\ _{\texttt{std}}\texttt{logic}$$

As before with $_{\texttt{std}}\texttt{logic}$, this only provides an *abstract* definition. The concrete rules of $_{\texttt{e}}\texttt{logic}$ depend on the concrete monad m we choose to use. As explained in Section IV the predicate transformers should now work on pairs like (q, r) as the post-condition, where q specifies the post-confition if a program terminates in a normal state, and r is the post-condition if the program terminates in an exceptional state. The idea in Section IV is to thread r via the monad m. To do so, we can concretely use $e' \rightarrow (e', e)$ as the monad $m\ e$ where e' represents the threaded exceptional post-condition. The corresponding monad operations are:

$$\begin{array}{lcl} \texttt{return}\ p & = & (\lambda z.\ (p, z)) \\ f\ \texttt{>>=}\ g & = & \texttt{uncurry}\ g \circ f \end{array}$$

where $(\texttt{uncurry}\ g)\ (x, y) = g\ x\ y$. Furthermore $_{\texttt{E}}\texttt{ruleRaise}$ and $_{\texttt{E}}\texttt{ruleTry}$ also require m to be an instance of the class $\texttt{Monad}^{\texttt{R}}$ in Definition IV.3; the concrete definition of the class' operations must thus

be specified. These are `getPostE` and `setPostE` used to extract and set the threaded exceptional post-condition. Their definition:

$$\begin{array}{lcl} \texttt{getPostE} & = & (\lambda z.\ (z,z)) \\ \texttt{setPostE}\ z & = & (\lambda z'.\ (z,())) \end{array}$$

Given the above monad and instance of $\texttt{Monad}^{\texttt{R}}$, it can be shown that the concrete rules of ${}_{\texttt{e}}\texttt{logic}$ are:

Definition VI.8 : Concrete Rules of ${}_{\texttt{e}}\texttt{logic}$

$$\begin{array}{lcl} {}_{\texttt{E}}\texttt{ruleRaise}_e\ q\ z & = & (z,z) \\ \texttt{ruleTry}_e\ t_{\texttt{try}}\ t_{\texttt{catch}}\ q\ z & = & t_{\texttt{try}}\ q\ (\texttt{snd}\ (t_{\texttt{catch}}\ q\ z)) \\ \texttt{ruleAsg}_e\ x\ e\ q\ z & = & (\texttt{subst}\ (x,e)\ q,\ z) \\ \texttt{ruleSeq}_e\ t_1\ t_2\ q\ z & = & \texttt{uncurry}\ t_1\ (t_2\ q\ z) \end{array}$$

An important step towards an incremental soundness proof for ${}_{\texttt{e}}\texttt{logic}$ is to express the new incarnations of the old rules (`ruleAsg` and `ruleSeq`) in terms of their old incarnations. This is given by the theorem below, which is quite easy to prove:

Theorem VI.9 : Old and New Incarnations

$$\begin{array}{lcl} \texttt{ruleAsg}_e\ x\ e\ q\ z & = & (\texttt{ruleAsg}^-\ x\ e\ q,\ z) \\ \texttt{ruleSeq}_e\ t_1\ t_2\ q\ z & = & \texttt{ruleSeq}^-\ u_1\ u_2\ (q,z) \end{array}$$

where $u_1 = \texttt{uncurry}\ t_1$ and $u_2 = \texttt{uncurry}\ t_2$.

Let us first argue that the extension from ${}_{\texttt{std}}\texttt{logic}$ to ${}_{\texttt{e}}\texttt{logic}$ is safe in the sense that it preserves the old notion of soudness. Theorem V.2 will not let us directly compare the two logics, since they operate on different types of predicate transformers. We can however 'downcast' ${}_{\texttt{e}}\texttt{logic}$ to make it comparable to ${}_{\texttt{std}}\texttt{logic}$:

$$L_e\ S\ q = p\ \ \texttt{where}\ (p,r) = (\!|{}_{\texttt{e}}\texttt{logic}|\!)\ S\ (q,\texttt{true})$$

L_e has the same type of transformers as ${}_{\texttt{std}}\texttt{logic}$. Next, we define a $\leq$ ordering on these transformers:

$$t_1 \leq t_2\ =\ (\forall q :: t_2\ q \Rightarrow t_1\ q)$$

Next, we have to show that the old soundness criterion $C_{\texttt{std}}$ (Definition VI.3) is monotonic with respect to the above ordering, and that for all $S \in L^-$, $(\!|{}_{\texttt{std}}\texttt{logic}|\!)\ S \leq L_e\ S$. We will not show the proofs. In Theorem VI.6 we have shown that ${}_{\texttt{std}}\texttt{logic}$ is sound with respect to $C_{\texttt{std}}$. Then by Theorem V.2 it follows that:

Theorem VI.10 : Soundness Preservation of L_e

(L^-, L_e) is *sound* with respect to $C_{\texttt{std}}$

Next, let us prove that ${}_{\texttt{e}}\texttt{logic}$ is sound with respect to *its own* soundness criterion. It is the same as $C_{\texttt{std}}$, except now we are using the evaluator $\mathcal{X}$ which is more powerful and operates on an extended state structure:

Definition VI.11 : Extended Soundness Criterion

$$C_e\ S\ t\ =\ (\forall q, z, \rho :: \rho \vdash t\ q\ z \Rightarrow \mathcal{X}\ S\ \rho \vdash (q,z))$$

We will again prove the soundness by induction. Each of the following lemmas deals with each case. The first one below is the `raise` case:

Lemma VI.12 : Raise-case L_e^-

$$(\forall q, z, \rho :: \rho \vdash {}_{\texttt{E}}\texttt{ruleRaise}_e\ q\ z\ \Rightarrow\ \mathcal{X}_{\texttt{raise}}\ \rho \vdash (q,z)$$

Proof: it comes down to proving $\rho \vdash (z,z)$ implies $(\texttt{fst}\ \rho, \texttt{true}) \vdash z$. This is quite trivial, given Definition VI.7 of $\vdash$.

The `try` induction case is below:

Lemma VI.13 : Try-case L_e^-

$$\begin{array}{l} (\forall q, z, \rho :: \rho \vdash t_{\texttt{t}}\ q\ z \Rightarrow r_{\texttt{t}}\ \rho \vdash (q,z)) \wedge \\ (\forall q, z, \rho :: \rho \vdash t_{\texttt{c}}\ q\ z \Rightarrow r_{\texttt{c}}\ \rho \vdash (q,z)) \\ \Rightarrow \\ (\forall q, z, \rho ::\ \ \rho \vdash \texttt{ruleTry}_e\ t_{\texttt{t}}\ t_{\texttt{c}}\ q\ z \\ \qquad\qquad \Rightarrow \\ \qquad\qquad \mathcal{X}_{\texttt{try}}\ r_{\texttt{t}}\ r_{\texttt{c}}\ \rho \vdash (q,z)) \end{array}$$

Proof:
Assume first that $r_{\texttt{t}}\ \rho$ produces a normal state. We derive:

$$\begin{array}{ll} & \mathcal{X}_{\texttt{try}}\ r_{\texttt{t}}\ r_{\texttt{c}}\ \rho\ \vdash\ (q,z) \\ = & \quad \{\ \text{def. of } \mathcal{X}_{\texttt{try}}\ \} \\ & r_{\texttt{t}}\ \rho\ \vdash\ (q,z) \\ = & \quad \{\ r_{\texttt{t}}\ \rho \text{ produces a normal state}\ \} \\ & (\texttt{fst}\ r_{\texttt{t}}\ \rho,\ \texttt{true})\ \vdash\ (q,z) \\ = & \quad \{\ \text{Def. VI.7 of} \vdash\ \} \\ & (\texttt{fst}\ r_{\texttt{t}}\ \rho,\ \texttt{true})\ \vdash\ (q,\ \texttt{snd}\ (t_{\texttt{c}}\ (q,z))) \\ = & \quad \{\ r_{\texttt{t}}\ \rho \text{ produces a normal state}\ \} \\ & r_{\texttt{t}}\ \rho\ \vdash\ (q, \texttt{snd}\ (t_{\texttt{c}}\ (q,z))) \\ \Leftarrow & \quad \{\ \text{first inductive assumption}\ \} \\ & \rho\ \vdash\ t_{\texttt{t}}\ q\ (\texttt{snd}\ (t_{\texttt{c}}\ (q,z))) \\ = & \quad \{\ \text{def. of } \texttt{ruleTry}_e\ \} \\ & \rho\ \vdash\ \texttt{ruleTry}_e\ t_{\texttt{t}}\ t_{\texttt{c}}\ q\ z \end{array}$$

Now assume that $r_{\texttt{t}}\ \rho$ produces an exceptional state.

$$\begin{array}{ll} & \mathcal{X}_{\texttt{try}}\ r_{\texttt{t}}\ r_{\texttt{c}}\ \rho\ \vdash\ (q,z) \\ = & \quad \{\ \text{def. of } \mathcal{X}_{\texttt{try}}\ \} \\ & r_{\texttt{c}}\ (\texttt{N}\ (r_{\texttt{t}}\ \rho))\ \vdash\ (q,z) \\ \Leftarrow & \quad \{\ \text{second inductive assumption}\ \} \\ & \texttt{N}\ (r_{\texttt{t}}\ \rho)\ \vdash\ t_{\texttt{c}}\ (q,z) \\ = & \quad \{\ \text{def. of N}\ \} \\ & (\texttt{fst}\ r_{\texttt{t}}\ \rho,\ \texttt{false})\ \vdash\ t_{\texttt{c}}\ (q,z) \\ = & \quad \{\ \text{Def. VI.7 of} \vdash\ \} \\ & (\texttt{fst}\ r_{\texttt{t}}\ \rho,\ \texttt{false})\ \vdash\ (q,\ \texttt{snd}\ (t_{\texttt{c}}\ (q,z))) \end{array}$$

$=$ { $r_{\mathtt{t}}\ \rho$ produces an exceptional state }
$r_{\mathtt{t}}\ \rho\ \vdash\ (q, \mathtt{snd}\ (t_{\mathtt{c}}\ (q, z)))$
$\Leftarrow$ { first inductive assumption }
$\rho\ \vdash\ t_{\mathtt{t}}\ q\ (\mathtt{snd}\ (t_{\mathtt{c}}\ (q, z)))$
$=$ { def. of $\mathtt{ruleTry}_e$ }
$\rho\ \vdash\ \mathtt{ruleTry}_e\ t_{\mathtt{t}}\ t_{\mathtt{c}}\ q\ z$
□

Now we come to the interesting part, namely the assignment and the sequential composition cases. Concretely, $_{\mathtt{e}}\mathtt{logic}$ and $_{\mathtt{std}}\mathtt{logic}$ use different rules to handle these cases. However, we can expect a close relation between the two sets rules, because modulo the monads, they are the same rules! This is indeed so, as was given in Theorem VI.9. It is now possible to give an incremental proof for the new incarnation. Below is the assignment case:

Lemma VI.14 : ASG-CASE L_e^-

$$(\forall q, z, \rho ::\ \rho \vdash \mathtt{ruleAsg}_e\ x\ e\ q\ z\ \Rightarrow\ \mathcal{X}_{\mathtt{asg}}\ x\ e\ \rho \vdash (q, z))$$

Proof: if ρ is exceptional, then it is of this form: $(\sigma, \mathtt{true})$. Unfolding the definitons of $\mathtt{ruleAsg}_e$ and $\mathcal{X}_{\mathtt{asg}}$, it comes down to proving this:

$$(\sigma, \mathtt{true}) \vdash (\ldots, r)\ \Rightarrow\ (\sigma, \mathtt{true}) \vdash (q, r)$$

which is trivial. If ρ is normal, then it comes down to proving:

$$\mathtt{fst}\ \rho \vdash \mathtt{ruleAsg}^-\ q\ e\ \Rightarrow\ \mathcal{E}_{\mathtt{asg}}\ x\ e\ (\mathtt{fst}\ \rho) \vdash q$$

Now we can reuse the results from $_{\mathtt{std}}\mathtt{logic}$. The above then follows from Lemma VI.4.

The Seq-case, the last one, is below:

Lemma VI.15 : SEQ-CASE L_e^-

$(\forall q, z, \rho :: \rho \vdash t_1\ q\ z\ \Rightarrow\ r_1\ \rho \vdash (q, z)) \wedge$
$(\forall q, z, \rho :: \rho \vdash t_2\ q\ z\ \Rightarrow\ r_2\ \rho \vdash (q, z))$
$\Rightarrow$
$(\forall q, z, \rho :: \rho \vdash \mathtt{ruleSeq}_e\ t_1\ t_2\ q\ z\ \Rightarrow\ \mathcal{X}_{\mathtt{seq}}\ r_1\ r_2\ \rho \vdash (q, z))$

Proof: unfolding the definition of $\mathtt{ruleSeq}_e$ and $\mathcal{X}_{\mathtt{seq}}$, what we have to prove is:

$\rho \vdash \mathtt{ruleSeq}^-\ (\mathtt{uncurry}\ t_1)\ (\mathtt{uncurry}\ t_2)\ (q, z)$
$\Rightarrow$
$\mathcal{E}_{\mathtt{seq}}\ r_1\ r_2\ \rho \vdash (q, z)$

Now we reuse the results from $_{\mathtt{std}}\mathtt{logic}$, Lemma VI.5, which says that it suffices to show this:

$$\rho \vdash \mathtt{uncurry}\ t_1\ (q, z)\ \Rightarrow\ r_1\ \rho \vdash (q, z)$$

and something similar for t_2 and r_2. Notice that $\mathtt{uncurry}\ t_1\ (q, z) = t_1\ q\ z$. So the above is in fact the same as the first inductive assumption of the Lemma above.

Now we can conclude with the soundness result below; it is proven by induction over L_e^-; the cases follow from the four lemmas above.

Theorem VI.16 : SOUNDNESS OF $_{\mathtt{e}}\mathtt{logic}$ $_{\mathtt{std}}\mathtt{logic}$ is sound with respect to $C_{\mathtt{e}}$

VII 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 [14] 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 [15]. 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 [16]. 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 [17] and Liang and Hudak [2]. 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.

Our notion of modifier also seems to correspond to Cartwright and Felleisen's systematic changes of the `admin` function in extensible denotational semantics [18]. The use of algebras to implement a syntax directed logic is related to the attribute grammar approach [19]. 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 [21]). In-

tricate weavings of bottom-up and top-down computations are often found in the implementation of modern type checkers. This can be conveniently specified in an attribute grammar formalism, whereas using a monad this would be akward. 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.

VIII Conclusion

We have described a technique to change/extend a logic to obtain derivative logics 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 confirm the technique's advantages; thus we believe that it is worth further investigation.

References

[1] G. Malcolm, "Data structures and program transformation," *Science of Computer Programming*, vol. 14, no. 2–3, pp. 255–280, Oct. 1990.

[2] S. Liang and P. Hudak, "Modular denotational semantics for compiler construction," in *ESOP'96, Proc.*, ser. LNCS, vol. 1058, 1996, pp. 219–234.

[3] M. Huisman, "Java program verification in Higher-order logic with PVS and Isabelle," Ph.D. dissertation, University of Nijmegen, The Netherlands, 2001.

[4] M. Norrish, "C formalised in HOL," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-453, 1998.

[5] R. J. Bird, *Introduction to Functional Programming using Haskell*, 2nd ed., ser. Prentice-Hall Series in Computer Science. London, UK: Prentice-Hall Europe, 1998.

[6] C. Hoare, "An axiomatic basis for computers programs," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576–583, 1969.

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

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

[9] P. V. Homeier and D. F. Martin, "Trustworthy tools for trustworthy programs: A verified verification condition generator," *LNCS*, vol. 859, pp. 269–284, 1994.

[10] J. Jeuring, "Theories for algorithm calculation," Ph.D. dissertation, Utrecht University, 1993.

[11] P. Wadler, "How to declare an imperative," *ACM Computing Surveys*, vol. 29, no. 3, pp. 240–263, Sept. 1997.

[12] C. Pierik and F. d. Boer, "A syntax-directed hoare logic for object-oriented programming concepts," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI*, 2003, pp. 64–78.

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

[14] B. Jacobs and E. Poll, "A monad for basic Java semantics," *Lecture Notes in Computer Science*, vol. 1816, pp. 150–164, 2000.

[15] B. Jacobs, J. van den Berg, H. Huisman, M. van Berkum, U. Hensel, and H. Tews, "Reasoning about Java classes: preliminary report," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 329–340, Oct. 1998.

[16] J. Hurd, "Formal verification of probabilistic algorithms," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-566, 2003.

[17] E. Moggi, "An abstract view of programming languages," Laboratory for Foundations of Computer Science, University of Edinburgh, Tech. Rep. ECS-LFCS-90-113, 1989.

[18] R. Cartwright and M. Felleisen, "Extensible denotational language specifications." in *Symposium on Theoretical Aspects of Computer Software*, 1994.

[19] J. Paakki, "Attribute grammar paradigms – A high-level methodology in language implementation," *ACM Computing Surveys*, vol. 27, no. 2, pp. 196–255, 1995.

[20] S. Swierstra, "Homepage of the AG system," 1999, `www.cs.uu.nl/groups/ST/Software/UU_AG`.

[21] "x-Mech home page," `www.cs.uu.nl/~wishnu/research/projects/xMECH`.

Ignatius Sri Wishnu Brata Prasetya was born in Jakarta, Indonesia, on 8-th December 1966. He received his Ph.D. in Computer Science from Utrecht University in 1995. Dr. Prasetya is a member of IEEE and now works as a lecturer and researcher at the Department of Information and Computing Sciences of Utrecht University, The Netherlands. His research areas are verification tools and theories of distributed programming.

Tanja E. J. Vos was born in Hilversum, The Netherlands on the 8th of October 1971. In 1995 she got a master in Computer Science and in 2000 she recieved a PhD degree on verification of distributed systems, both from the University in Utrecht. Dr. Vos now works as a researcher in ITI (Instituto Tecnológico de Informática), where she is the director of the R&D group SQuaC (Software Quality and Correctness). Moreover, she is a part-time teacher at the Technical University of Valencia in Spain. Dr. Vos participates in various research projects on software quality and testing funded by the Europe Commission, the Spanish government and by industry. Previously, she has worked as a researcher and professor at various universities like the Utrecht University (The Netherlands), Universidad Mayor de San Simón (Bolivia), University of Cambridge (UK), and the Mediterranean University of Science and Technology in Spain.

Arthur van Leeuwen received a master degree in Computer Science from the University of Nijmegen in 2002. He is currently a research programmer at the Department of Information and Computing Sciences of Utrecht University, The Netherlands. His main research interest is the field of Programming Languages.