Theorem Prover Supported Logics for Small Imperative Languages

I.S.W.B. Prasetya, A. Azurat, T.E.J. Vos
A. van Leeuwen, H. Suhartanto

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-046

www.cs.uu.nl

Theorem Prover Supported Logics for Small Imperative Languages

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

November 15, 2005

UU: Department 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`, `arthurvl@cs.uu.nl`

Abstract

This report describes a simple imperative programming language $\mathcal{L}_0$ and its logic, which are integrated to the theorem prover HOL through syntactical embedding. The approach allows HOL's own type checker to be reused and its concrete syntax to be, to some degree, customized. The logic of $\mathcal{L}_0$ is Hoare-based and is completely syntax driven. Depending on how we limit its assertion language it can be made first order. A possible application of $\mathcal{L}_0$ is to use it as a core in rapid prototyping of small imperative languages with integrated verification support. This report gives two examples of new languages built on $\mathcal{L}_0$: a language to write a suite of $\mathcal{L}_0$-unit tests and a language for scripting database transactions.

Contents

1 Introduction

Theorem provers like Isabelle, COQ, and HOL have very expressive base logics so that we can embed a wide range of other logics in them, including programming logics. There have been many examples of the latter, ranging from logics for simple languages, e.g. UNITY [7], to that of real languages, e.g. ML [1], C [5], and Java [2]. Most embeddings (of programming logics), including the examples just mentioned, are semantical embedding. If L is the object language, a semantical embedding maintains representations of L syntax, up to some level of detail, and its semantic. Distinction between *shallow* and *deep* embedding is often made, depending on how detailed the syntax is represented. We can also embed L syntactically, which means that we only embed L's syntax, though usually in (much) higher detail than in a semantical embedding, for example up to its concrete syntax. A semantical embedding allows meta properties about L (e.g. the soundness of its logic) to be proven. However, it does not usually provide a concrete syntax. So, for actual use a full language-front-end is still needed, e.g. as the LOOP front-end compiler [12] is used to drive its in HOL semantically embedded Java logic back-end [2]. Meta properties cannot be proven in a syntactical embedding, but the approach allows a language front end, including the type checker, to be built with minimun effort, provided L's syntax and type system are still close to what the host theorem prover can express. Isabelle provides a dedicated infrastructure to do syntactical embedding. Examples of syntactically embedded logics (in Isabelle) are FOL, HOL, and ZF. Syntactical embedding of programming logics is rare.

We will discuss here a simple imperative programming language $\mathcal{L}_0$ featuring: basic statements, block, program call, return value, value and reference based parameter passing, and `old` keyword to refer to a variable's initial state[1]. The logic of $\mathcal{L}_0$ is Hoare based and is completely syntax driven. $\mathcal{L}_0$ and its logic are integrated to the theorem prover HOL through syntactical embedding. $\mathcal{L}_0$ itself is not a directly implementable language. Through restriction we can turn it to a concrete language which can be used stand alone. Our $\mathcal{L}_0$ library includes a minimalistic example of such a concrete instance, called $\mathcal{L}_0^{min}$, which comes with an ML translator for execution (and few other utilities, e.g. a test generator). We can quite easily customize $\mathcal{L}_0$ in various other ways. We can for example introduce syntactic sugar, and use HOL built-in rewriting functions to desugar. $\mathcal{L}_0$ logic is specified abstractly as a folded *algebra*. It is a well known technique in functional programming to abstractly specify a recursion over a data type. An algebra is a highly compositional structure, whose components can easily be adapted. In this way, we can easily adapt $\mathcal{L}_0$ logic. $\mathcal{L}_0$ can thus be used as a core when developing other small imperative languages, in particular when an integrated verification support is desired. We will give two examples of new languages obtained by customizing $\mathcal{L}_0$: TEST, a language to write a suite of unit tests for $\mathcal{L}_0$ programs, and *Lingu*, a language for scripting database transactions. In both cases we can reuse much of what we have built for $\mathcal{L}_0$. We have chosen HOL as our embedding host. Isabelle is probably a better choice; we did it in HOL mainly because our familiarity with it. HOL shares some of Isabelle's key features: it is based on typed λ-calculus which is used as the embedding medium, and it allows customization of its concrete syntax. These are sufficient to build our current prototypes. The work here can also be particularly interesting for HOL community. $\mathcal{L}_0$, $\mathcal{L}_0^{min}$, TEST, and Lingu can be downloaded from:

`http://www.cs.uu.nl/~wishnu/research/projects/xMECH`

Report Structure

This report is organized as follows. Section 2 gives small examples of semantical and syntactical embeddings in HOL and explains the distinction between them. Section 4 introduces $\mathcal{L}_0$, explains how it is syntactically embedded in HOL, and describes the logic. Section 5 discusses the implementation of this logic in the meta language of HOL. Section 7 describes TEST. Section 8 describes Lingu . Finally, 9 gives some closing remarks.

[1] Examples of features which are *not* in $\mathcal{L}_0$: recursion, exception, expressions with side effect, object (as in Java), and compilation module (as in ML or Haskell).

2 Semantical and Syntactical Embedding

To better illustrate the difference between syntactical and semantical embedding we will give examples. Consider the following simplistic statement language L —the typing rules are not shown; they are quite standard.

$$\begin{array}{lll} \textit{Stmt} & :: & \texttt{skip} \\ & & \mid \textit{Variable} := \textit{Expr} \\ & & \mid \{\ \textit{Statement};\ \ldots\ ;\ \textit{Statement}\ \} \\ \\ \textit{Expr} & :: & \textit{Variable} \mid \textit{Bool-constant} \mid \textit{Integer-constant} \end{array}$$

2.1 Semantical Embedding

We can represent L statements in HOL with (higher order) functions from state to state, and states by functions from variable-names (e.g. represented by strings) to values. In the same line we can represent expressions and assertions:

$$\begin{array}{lll} \texttt{state} & = & \texttt{string} \rightarrow \texttt{val} \\ \texttt{stmt} & = & \texttt{state} \rightarrow \texttt{state} \\ \texttt{expr} & = & \texttt{state} \rightarrow \texttt{value} \\ \texttt{assertion} & = & \texttt{state} \rightarrow \texttt{bool} \end{array}$$

The type `value` represents values which our program variables may take. For L, it has to be rich enough to represent Boolean values and integers. This HOL data-type representation will do:

```
hol_datatype value  =  Bval of bool | Ival of int
```

Constants of appropriate types can be introduced to represent the three kinds of statements of L:

$$\begin{array}{lll} \texttt{skip} & : & \texttt{stmt} \\ \texttt{asg} & : & \texttt{string} \rightarrow \texttt{expr} \rightarrow \texttt{stmt} \\ \texttt{seq} & : & \texttt{stmt} \rightarrow \texttt{stmt} \rightarrow \texttt{stmt} \end{array}$$

This is sufficient to embed the language, up to its abstract syntax. An (abstract) semantics can be added simply by defining those constants, e.g.:

$$\begin{array}{lll} \mathsf{skip}\ \mathsf{s} & = & \mathsf{s} \\ \mathsf{asg}\ \mathsf{v}\ \mathsf{e}\ \mathsf{s} & = & (\lambda\, \mathsf{x}.\ \mathsf{if}\ \mathsf{x}{=}\mathsf{v}\ \mathsf{then}\ \ \mathsf{e}\ \mathsf{s}\ \ \mathsf{else}\ \ \mathsf{s}\ \mathsf{x}) \\ \mathsf{seq}\ \mathsf{S}_1\ \mathsf{S}_2\ \mathsf{s} & = & \mathsf{S}_2\ (\mathsf{S}_1\ \mathsf{s}) \end{array}$$

A notion of Hoare-triple can be defined semantically, e.g.:

$$\texttt{HOA}\,(\mathsf{P},\mathsf{S},\mathsf{Q}) \ = \ (\forall \mathsf{s}.\ \mathsf{P}\ \mathsf{s} \Rightarrow \mathsf{Q}\ (\mathsf{S}\ \mathsf{s}))$$

Inference rules for the corresponding Hoare logic can be represented as HOL formulas; e.g. this rule for `skip`:

$$\frac{P \Rightarrow Q}{\{P\}\ \texttt{skip}\ \{Q\}} \tag{1}$$

is represented by the formula:

$$(\forall \mathsf{s}.\ \mathsf{P}\ \mathsf{s} \Rightarrow \mathsf{Q}\ \mathsf{s}) \ \Rightarrow\ \texttt{HOA}\,(\mathsf{P},\texttt{skip},\mathsf{Q})$$

Because we have a semantics, we can actually verify that L's inference rules, like the one above for `skip`, follow from L's semantics. By doing so we effectively verify the soundness of L's logic. This feature is the main advantage of semantical embedding. Once verified the rules can be turned to HOL theorems. Subsequent proofs built purely on HOL theorems are guaranteed to be safe.

The first problem with semantical embedding is that we have no concrete syntax. For example, the statement $\{\mathtt{x} := \mathtt{0};\ \mathtt{b} := \mathtt{F}\}$ is represented by this hard to read formula:

```
seq (asg "x" (Ival 0)) (asg "b" (Bval F))
```

Secondly we also lose type checking. We would expect HOL built-in type checker to do that. Unfortunately, statements like this one:

```
seq (asg "x" (Ival 0)) (asg "x" (Bval F))
```

will be accepted by HOL. It represents {`x` := `0`; `x` := `F`}, which would be type incorrect in L.

2.2 Syntactical Embedding

In the syntactical embedding we are not concerned with the embedding of the semantics of L. It is sufficient to know that a statement is an object which is different than, for example, Boolean values and integers. To enforce this, we represent statements with a new type, say `Stmt`. To represent skip and sequence (of statements), we can introduce these constants:

$$\begin{array}{lcl} \texttt{skip} & : & \texttt{Stmt} \\ \texttt{seq} & : & \texttt{Stmt} \rightarrow \texttt{Stmt} \rightarrow \texttt{Stmt} \end{array}$$

There is no need to give concrete definition for these constants as in the semantical embedding.

The syntax of L uses the infix symbol `:=` to denote assignment. HOL already uses this symbol for something else, and unfortunately its syntax customization does not allow us to overload the symbol. We will use a different infix symbol, and overload it to allow both boolean and integer assignments:

$$\begin{array}{lcl} /{:=} & : & \texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{Stmt} \\ /{:=} & : & \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{Stmt} \end{array}$$

The concrete syntax for sequences can be obtained via HOL syntax customization of its list-like syntax. The customization code is shown below. We choose to use different delimeter symbols:

```
add_listform {separator = ";",
              leftdelim = "/{",
              rightdelim = "/}",
              cons = "seq", nilstr = "skip"};
```

So now we can write /{ `x`/:=`0`; `b`/:=`F` /}, which will be parsed by HOL to:

```
seq (x/:=0) (seq (b/:=F) skip)
```

and typed as a value of type `Stmt`. Note that the statement /{ `x`/:=`0`; `x`/:=`F` /} is now rejected by HOL type system (because it requires the HOL variable `x` to have both the type `int` and `bool`). Inference rules such as (1) are coded at the meta level. In HOL we have ML serving as its meta programming language. The implementation of (1) in ML could look like:

```
fun skip_rule spec =
    let val (P,Q) = match_skip spec
    in
    mk_imp(P,Q)
    end
```

where `match_skip` is a function that checks if an input term has the form of a Hoare triple over a `skip`, and if so it returns its pair of pre- and post-conditions. Notice that unlike the counterpart of this rule in semantical embedding, the rule above is not a HOL formula. It is a plain ML code; HOL cannot say anything about its soundness.

3 The Language $\mathcal{L}_0$

$\mathcal{L}_0$ is a simple imperative programming language. Below is a simple example of an $\mathcal{L}_0$ program called `swap`:

```
swap (REF x,REF y)
        =
        pre  T
        post (x = old y) /\ (y = old x)
        do
        let tmp = 0
        in /{ tmp /:= x   ;
              x   /:= y   ;
              y   /:= tmp   /}

        return void
```

A program can take parameters. The keyword `REF` before a formal parameter means that the parameter will be passed as a reference; without the keyword it will be passed as a value. The `pre` and `post` sections specify pre- and post-conditions. In the post-condition of a program P we can use an assertion of the form `old` y where y is a (formal) parameter of P. This refers to the initial value of y; that is the value of y when it is passed to P during the call to P. So, the specification above says that when the program ends, the value of `x` and `y` will be swapped.

The syntax of various kind of statements (or 'instructions', as they are called in $\mathcal{L}_0$) is listed below. The complete syntax is the Appendix.

1. `skip`
2. An instruction to print to the screen: `print (`*Expr*`)`. Only an expression of type `int` or `string` can be an argument of `print`.
3. Assignment: `Expr /:= Expr`
4. Sequence of instructions: `/{` *Instr* `;` ... `;` *Instr* `/}`
5. Program call. In the first variant the return value is ignored; in the second the return value is assigned to the target (variable) at the left-hand side:

   `/@` *ProgName* `(` *ActualParams* `)`

   Expr `/@=` *ProgName* `(` *ActualParams* `)`
6. Conditional: `if (`*Expr*`) then /{` *Instr* `/} else /{` *Instr* `/}`
7. Introducing initialized local variables:

   `let`
   Var `=` *Expr* `and`
   ...
   Var `=` *Expr*
   `in /{` *Instr* `/}`
8. Loop: `while (`*Expr*`) wdo /{` *Instr* `/}`
9. `assert (`*Assertion*`)`. This instruction is used to add an assertion to the code. This is used for verification purpose only. During the verification it will be seen as a specification that the asserted predicate must hold at that point. During the execution it will be ignored[2]

[2]Optionally, one can opt to check the asserted predicate at the run time, though this will affect the performance and may require expensive roll back. Here, we simply assume that asserted predicates in a program are first verified before the program is used.

The syntax of program declaration is shown below:

$$\mathit{ProgDecl} \rightarrow \mathit{ProgName}\ \texttt{(}\ \mathit{FormalParams}\ \texttt{)}\ \texttt{=}\ \mathit{ProgDeclRHS}$$

$$\begin{array}{ll} \mathit{ProgDeclRHS} \rightarrow & \texttt{pre}\ (\mathit{Assertion}) \\ & \texttt{post}\ (\mathit{AssertionRet}) \\ & \texttt{do}\ /\{\ \mathit{Instr}\ /\}\ \texttt{return}\ (\mathit{Expr}) \end{array}$$

The `pre` and `post` sections are obligatory. A `return` instruction can only appear as the last instruction in a program, and is obligatory. The assertion in the `post` may use the keyword `ret`, which refers to the value returned by the program. Recursion is not supported.

In principle, $\mathcal{L}_0$ distinguishes *assertions* from *expressions*. An expression is intended to be executable whereas an assertion is part of a program's specification and as such does not have to be executable. Assertions are used in the pre- and post-conditions and in the `assert` statement. Expressions are used for example in the assignment and as the guard of a loop. In the actual syntax of $\mathcal{L}_0$ the distinction between the two is however small, but this is more because in this respect $\mathcal{L}_0$'s syntax is intentionally left under specified. One of the distinction is that in an assertion we can write `old` v where v is a program variable[3]. This refers to the 'old' value of v; more precisely, the value of v when it is initialized in the innermost block that encloses the assertion. Passing parameters during a program call counts as a block. So, if v is a parameter of a program P, then `old` v in the body of P (if it is not overshadowed by a local v) as well as in the pre- and post-conditions of P refer to the value of v when it is passed to P during the call to P. Such a notation is really convenient, though its treatment in the logic is quite complicated (see Subsection 5). This probably applies to all 'add-on notations' that try to enhance the 'temporal' expresiveness of Hoare logic.

An $\mathcal{L}_0$ program is *not* allowed to access any global variable, except if it is passed as a `REF` parameter to it. This can be enforced by checking that a program declaration contains no free variable.

The above constraint also means that in pre- and post-conditions of a program P we can only refer to the parameters of P. Note that if v is a pass-by-value parameter of P, occurences of v in the `post` section of P refers to its initial value (hence, equal to `old` v), and not to v's final value[4].

$\mathcal{L}_0$ leaves the syntax of expressions, assertions, and the left hand side of assignment under specified: any well-formed HOL term is allowed at those positions. This leaves some room for customization. However, this makes $\mathcal{L}_0$ not directly implementable; e.g. assignments like:

$$\texttt{b}\ /{:=}\ (\exists \texttt{f}'.\ (\forall \texttt{x}.\ \texttt{f}'(\texttt{sin}\ \texttt{x}) = \texttt{x}))$$

would be rather hard to implement. For implementation, an instance of $\mathcal{L}_0$ whose syntax is sufficiently narrowed is needed.

4 Embedding $\mathcal{L}_0$

We introduce a new type `INSTR` to represent instructions. Skip and sequence are represented as in Subsection 2.2, except that we call the type `INSTR` here instead of `Stmt`. The representation of

[3]In JML assertion like `old` e where e is an arbitrary assertion is allowed. We won't allow this because its meaning is sometimes dubious. For example, what is the meaning of the assertion `old` $(\texttt{x}+\texttt{y})$ in:

```
let x=0 in assert (old (x+y))
```

Does this refer to the meaning of $\texttt{x}+\texttt{y}$ when `x` is initialized, or do we mean that only `x` should be interpreted in this state? Of course, we can define a meaning, but at the moment we decide to simply disallow it.

[4]The motivation is that the `pre` and `post` sections, in addition to specifying a notion of correctness for P, are treated by the $\mathcal{L}_0$ logic as the abstraction of P when handling a program call. For the caller Q, the final value of v is irrelevant, because it cannot see it. So there is no point in specifying it in `post`. Q still knows the 'initial' value v though (that is, the value of v when Q passed it to P), which is our chosen interpretation of v in `post`.

assignment also remains the same, except that we generalize its type to accomodate assigments of values of arbitrary types:

$$/{:=}\ :\ \texttt{'a} \rightarrow \texttt{'a} \rightarrow \texttt{INSTR}$$

Notice that the typing forces that the assigned expression to be of the same type as the assigment target.

There is no need to introduce a new syntax for `if-then-else` and the `let` construct; HOL already has them. The following constant is added to abstractly represent a `while` loop:

$$\texttt{wloop}\ :\ \texttt{bool} \rightarrow \texttt{INSTR} \rightarrow \texttt{INSTR}$$

The concrete syntax for `while` is introduced via HOL syntax customization, e.g.:

```
add_rule{term_name = "wloop", fixity = TruePrefix 19,
                 pp_elements = [PPBlock([TOK "while", BreakSpace(1,0),
                                         TM, BreakSpace(1,0),
                                         TOK "wdo",
                                         BreakSpace(1,0) ],
                                        (PP.CONSISTENT,3)),
                               BeginFinalBlock(PP.CONSISTENT,0) ],
                paren_style = OnlyIfNecessary,
                block_style = (AroundEachPhrase,(PP.CONSISTENT,0))};
```

This allows us to write for example $\texttt{while}\ (0 < \texttt{i})\ \texttt{wdo}\ /\{\ \texttt{i}\ /{:=}\ \texttt{i} - 1\ /\}$, which will be parsed by HOL to:

$$\texttt{wloop}(0 < \texttt{i})(\texttt{seq}\ (\texttt{i}\ /{:=}\ \texttt{i} - 1)\ \texttt{skip})$$

Notice also that the type of `wloop` forces the type of the loop guard to be `bool`.

The `print` instruction can be represented by a constant of function type, which is overloaded so that it can take either an int or a string as an argument:

$$\begin{array}{lcl} \texttt{print} & : & \texttt{int} \rightarrow \texttt{INSTR} \\ \texttt{print} & : & \texttt{string} \rightarrow \texttt{INSTR} \end{array}$$

The representation of `assert` is straight forward:

$$\texttt{assert}\ :\ \texttt{bool} \rightarrow \texttt{INSTR}$$

4.1 Program Call

Program call is a bit more complicated. A program call like $\texttt{x}\ /@= \texttt{P}(0,1)$ requires that the type of `x` matches the return type of `P`. To coerce HOL type checker to check this we make `P` to have the type:

$$(\texttt{int\#int}) \rightarrow \tau\ \texttt{PROG}$$

So, $\texttt{P}(0,1)$ would have the type τ `PROG`. This type can be thought to represent programs that return values of type τ. We now can introduce the constant $/@=$ (denoting program call), having this type:

$$/@=\ :\ \texttt{'a} \rightarrow \texttt{'a}\ \texttt{PROG} \rightarrow \texttt{INSTR}$$

Notice that this forces the type of the assigment target to be matched with the program's return type.

$\mathcal{L}_0$ allows parameters to be passed either by value or by reference. We introduce a new HOL data-type to represent reference (pointer):

```
hol_datatype REF  =  REF of 'a
```

So, a HOL term of type, for example, `int REF` represents an $\mathcal{L}_0$ pointer pointing to an $\mathcal{L}_0$ value of type `int`. So, the program `swap` (at the beginning of this Section) would have the HOL type:

$$(\texttt{int REF} \# \texttt{int REF}) \;\rightarrow\; \texttt{void PROG}$$

HOL will then accept calls like $\texttt{swap}(\texttt{REF}\ \texttt{e}_1,\ \texttt{REF}\ \texttt{e}_2)$, but not $\texttt{swap}(0,1)$ because `0` and `1` are not of type `REF`. We may want to restrict the syntax of actual parameters, for example, so that $\texttt{e}_1$ and $\texttt{e}_2$ here should be variables. Unfortunately, there is no way we can incorporate such a restriction in HOL now. So if it is desired, it has to be implemented as a separate syntax check (which is called after HOL's own syntax and type checks).

5 $\mathcal{L}_0$ Logic

$\mathcal{L}_0$ has a Hoare-styled, partial correctness based logic. It is quite standard; though the reader may find it interesting to look at how we deal with program call and `old` back-reference. We will deviate from the standard presentation of Hoare logic. We will express the logic in terms of a reduction function.

Given an instruction S and a post-condition q the function `reduce` returns another predicate p —note that with respect to $\mathcal{L}_0$, `reduce` is a meta function. There is a global list of predicates V on which `reduce` operates by adding new predicates to it, or modifying existing ones. The predicates in V are also called *verification conditions* and the function `reduce` is also called *verification condition generator*.

Let $p = \texttt{reduce}\ S\ q$. The function works in such a way so that the validity of all predicates in V obtained after executing $\texttt{reduce}\ S\ q$ implies the (partial) correctness of $\{p\}\ S\ \{q\}$. Consequently, when given a specification $\{p_0\}\ S\ \{q\}$, it is sufficient to: calculate $p = \texttt{reduce}\ S\ q$; prove the validity of all predicates in the final V; and prove $p_0 \Rightarrow p$. In our embedding, all these predicates would be plain HOL formulas; they can be proven in the standard way in HOL.

As we will see, the logic is not trivial. The major source of complication is the `old` construct. For example, this is no longer valid in $\mathcal{L}_0$:

$$\{q[0/v]\}\ v := 0\ \{q\}$$

Since $\texttt{old}\ v$ in q refers to initial value of v, its meaning is not affected by the assignment above, and hence should not be replaced by 0 in the pre-condition. A more sophisticated subsitution is now needed to handle assignment.

5.1 Substitution

In the sequel, $q[e/v]$ denotes the expression obtained by subsituting all free occurences of v in q with e; $q[e_1/v_1, e_2/v_2]$ denotes simultaneous substitution. If V is a list of expressions, $V[e/v]$ denote the list obtained by applying the substitution $[e/v]$ on every expression in V. These are quite standard. Additionally we introduce the following 'macros':

1. $q[e/v\ \texttt{except old}\ v]$ means that we replace every (free) v in q, which is not in the form $\texttt{old}\ v$, with e. For example:

   $$(v > \texttt{old}\ v)[0/v\ \texttt{except old}\ v] \quad = \quad 0 > \texttt{old}\ v$$

   `except` substitution can be implemented in terms of ordinary substitution:

   $$q[e/v\ \texttt{except old}\ v] \quad = \quad q[@z/\texttt{old}\ v][e/v][\texttt{old}\ v/@z]$$

   where $@z$ is a fresh variable. The rule for assignment can now be conveniently captured by:

   $$\{q[e/v\ \texttt{except old}\ v]\}\ \ v := e\ \ \{q\}$$

2. $q[e/\texttt{old}\ v\ \texttt{orelse}\ v]$ means that we *simultaneously* replace every `old` v and every v in q, the latter should not be in the form `old` v, with e. For example:

$$(v = \texttt{old}\ v)[v+1/\texttt{old}\ v\ \texttt{orelse}\ v] \quad = \quad v+1 = v+1$$

   It can be implemented in terms of ordinary substitution:

$$q[e/\texttt{old}\ v\ \texttt{orelse}\ v] \quad = \quad q[@z/\texttt{old}\ v][e/v, e/@z]$$

   where $@z$ is a fresh variable.

5.2 `reduce`

The algorithm of `reduce` is described below. Variables whose name start with @ as in $@x$ and $@y$ are assumed to be freshly introduced variables.

1. Print:

$$\texttt{reduce}\ (\texttt{print}\ e)\ q \quad = \quad \texttt{return}\ q$$

2. Assignment:

$$\texttt{reduce}\ (x\ /\!:= e)\ q \quad = \quad \texttt{return}\ q[e/x\ \texttt{except}\ \texttt{old}\ x]$$

3. Sequences:

$$\texttt{reduce}\ (i_1; i_2)\ q \quad = \quad \{\ p_2 := \texttt{reduce}\ i_2\ q\ ;\ p_1 := \texttt{reduce}\ i_1\ p_2;\ \texttt{return}\ p_1\ \}$$

4. The rule for conditional is standard:

$$\begin{array}{lll} \texttt{reduce}\ (\texttt{if}\ g\ \texttt{then}\ i_1\ \texttt{else}\ i_2)\ q & = & \{\ \ p_1 := \texttt{reduce}\ i_1\ q\ ; \\ & & \quad p_2 := \texttt{reduce}\ i_2\ q\ ; \\ & & \quad \texttt{return}\ ((g \Rightarrow p_1)\ \wedge\ (\neg g \Rightarrow p_2))\ \} \end{array}$$

5. The rule for loop requires an invariant. Automatically constructing invariant is in general undecidable, though for special cases this can be done. We are not going to concern ourselves with this issue. The programmer may specify an invariant using an `assert` instruction, which must be the first instruction in the loop's body. If it is not specified, then an `assert T` is inserted. The rule:

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{reduce}\ (\texttt{while}\ g\ \texttt{wdo}\ /\{\ \texttt{assert}\ inv;\ i\ /\})\ q} \\ = & \\ \{\ \ p := \texttt{reduce}\ i\ inv & ; \\ \quad V := [inv \wedge g \Rightarrow p,\ inv \wedge \neg g \Rightarrow q] +\!\!+ V & ; \\ \quad \texttt{return}\ inv & \} \end{array}$$

6. The rule for `assert` is simple:

$$\texttt{reduce}\ (\texttt{assert}\ p)\ q \quad = \quad \texttt{return}\ (p \wedge q)$$

7. Abstractly, the rule for local declaration looks like this:

$$\texttt{reduce}\ (\texttt{let}\ v = e\ \texttt{in}\ i)\ q \quad = \quad (\texttt{reduce}\ i\ q[@v/v])[e/v][v/@v]$$

The substitution $[e/v]$ reflects the initialization of v to e. Note that occurences of v in q refers to a different v than the v locally declared by the `let`. The substitution $q[@v/v]$ prevents the 'global' v from being substituted by the reduction as it encounters assignments to v inside S. After the body is reduced, the temporary name $@v$ can be restored to v, which is what the substitution $[v/@v]$ does.

The above does not however take into account assertions like `old` v which may occur in the pre-condition and verification conditions that result from the reduction on i. For example cosider:

$$\texttt{let}\ v = 0\ \texttt{in}\ i$$

Suppose reducing the body i produces `old` $v \geq 0$ as a verification condition. This cannot be proven, unless we use the knowledge about `old` v. We know it is initialized to 0, so we can actually further reduce the verification condition to $0 \geq 0$, which is a tautology.

Consider another example:

$$\texttt{let}\ v = x\ \texttt{in}\ i$$

Suppose reducing the body i produces `old` $v = x$ as a verification condition. This is not and should not be provable. However, if we blindly replaces `old` v with the initialization expression as we did above we would transform the verification condition to $x = x$, which is a tautology. This transformation is not sound. The right thing to do, in general case:

$$\texttt{let}\ v = e\ \texttt{in}\ i$$

is to replace `old` v (in the verification conditions coming from i) with a version of e in which we have replaced all free variables (of e) with fresh names. We will call this:

$$\texttt{freshed}\ e$$

This scheme will work for both examples above. To capture it in a rule:

$$\begin{array}{lcll}\texttt{reduce}\ (\texttt{let}\ v = e\ \texttt{in}\ i)\ q & = & \{ & p := \texttt{reduce}\ i\ (q[@v/v]) \quad ; \\ & & & V := V[\texttt{freshed}\ e/\texttt{old}\ v] \quad ; \\ & & & \texttt{return}\ p[e/\texttt{old}\ v\ \texttt{orelse}\ v][v/@v] \quad \} \end{array}$$

One thing is still incorrect: we should not apply the substitution [`freshed` e/`old` v] on all verification conditions in V, but only to those added by `reduce` i $(q[@v/v])$. Free occurences of v in other predicates in V refer to another v and should not be substituted. This final rule cures the problem:

$$\begin{array}{lcll}\texttt{reduce}\ (\texttt{let}\ v = e\ \texttt{in}\ i)\ q & = & \{ & V := V[@v/v] \quad ; \\ & & & p := \texttt{reduce}\ i\ (q[@v/v]) \quad ; \\ & & & V := V[\texttt{freshed}\ e/\texttt{old}\ v][v/@v] \quad ; \\ & & & \texttt{return}\ p[e/\texttt{old}\ v\ \texttt{orelse}\ v][v/@v] \quad \} \end{array}$$

Introduction of multiple local variables are handled analogously.

8. $\mathcal{L}_0$ logic treats a called program P as a black-box. It means that the logic assumes that the source code of P is not available, though its specification is.

   Cosider a program P with the following header and specification:

   $$\begin{array}{lll} P(\texttt{REF}\ r,\ v) & = & \texttt{pre}\ p \\ & & \texttt{post}\ q' \\ & & \ldots \end{array}$$

   The black-box based reduction for calls to P looks abstractly like this:

   $$\begin{array}{lll} \texttt{reduce}\ (x/@ = P(s,e))\ q' & = & \{\ \ V := [q \Rightarrow q'[\texttt{ret}/x]] +\!\!+ V\ ; \\ & & \quad\ \texttt{return}\ p[s/r, e/v]\ \} \end{array}$$

   However this ignores these details:

   (a) If q' set a constraint on some variable z which does not occur in q, we will not be able to prove $q \Rightarrow q'[\texttt{ret}/x]$. To get around this, we will express the condition as part of the calculated pre-condition instead. In this way subsequent information about x produced by the 'ancestral' calls to `reduce` will also be added to the implication. However note that s in q' refers to the state of s after the call. If it is shifted as it to the pre-condition side its meaning changes, namely the state of s before the call. This is incorrect. So to prevent this, s in q' will be renamed with a fresh variable. Similar thing has to be done to s in q.

   (b) Occurrences of v in q actually refers to v's value when it is passed to P. Similarly, `old` r and `old` v also refer to the the values of these variable when they are passed.

   (c) `old` s occuring in q' *does not* refer to the value of s when it is passed to P. Instead it refers to the value of s when it is initialized in the scope that directly encloses the call to P. It should be left unchanged.

   (d) `old` x in q' should not be replaced with `ret`.

   The following rule does it:

   $$\begin{array}{l} \texttt{reduce}\ (x/@ = P(s,e))\ q' \\ = \\ \texttt{return}\ (\ \ p[@s/\texttt{old}\ r\ \texttt{orelse}\ r][e/\texttt{old}\ v\ \texttt{orelse}\ v][s/@s] \\ \qquad\quad\ \ \wedge \\ \qquad\quad\ \ (\ \ q[@s/r][e/\texttt{old}\ v\ \texttt{orelse}\ v][s/\texttt{old}\ @s,] \\ \qquad\qquad\ \Rightarrow \\ \qquad\qquad\ q'[\texttt{ret}/x\ \texttt{except}\ \texttt{old}\ x][@s/s\ \texttt{except}\ \texttt{old}\ s]\ ) \end{array}$$

   The case for programs with more parameters, or calls using /@ (instead of /@ =), can be handled analogously.

We still have to give the reduction algorithm at the program level. This is given below. The function is also called `reduce`. It takes a program declaration, and reduces it to a list of verification conditions. The function itself returns nothing. The resulting verification conditions are stored in V.

$$\begin{array}{l} \texttt{reduce}\ (P(\texttt{REF}\ r,\ v)\ =\ \texttt{pre}\ p\ \texttt{post}\ q\ \texttt{do}\ i\ \texttt{return}\ e) \\ = \\ \{\ \ w := \texttt{reduce}\ (i;\ \texttt{ret}/{:=}e)\ (q[\texttt{old}\ v/v])\ \ \ ; \\ \quad\ V := [p \Rightarrow w] +\!\!+ V \qquad\qquad\qquad\qquad \} \end{array}$$

6 Implementing the Logic

Implementing $\mathcal{L}_0$ logic amounts to implementing the `reduce` functions. They have been described in sufficient detail so that their implementation is in principle straight forward. $\mathcal{L}_0$ instructions, programs, and predicates are all, in our embedding, HOL terms. So, the `reduce` functions are just functions that operates on HOL terms. They can be implemented in HOL's meta-language (moscow-ML). The variable V can be implemented by an assignable ML variable of the type list of (HOL) terms.

Implementing the substitutions requires a lot more work, if we have to do it from scratch. Luckily we can just borrow subsitution utilities already provided by HOL. By writing few additional combinators we can then code the substitutions almost as abstract as in the algorithm in the previous section.

It is actually more convenient, in particular for implementing `reduce` on instructions, to work on a separate data-type that reflects the structure of $\mathcal{L}_0$ instructions more explicitly rather than working directly on HOL terms, because the latter is more low level.

For simplicity, let us consider only three sort of instructions: skip, assignment, and sequence. The following ML data-type can be used to represent them:

```
datatype INSTR = SKIP
               | ASG of term * term
               | SEQ of INSTR list
```

where `term` is the ML-type of HOL terms. A function can be written to build an `INSTR`-representation out of a HOL term representing a $\mathcal{L}_0$ instruction, for example as shown below.

```
fun destAsg t =
    let val match = (map #residue o fst o match_term (Term ‘xxxx /:= yyyy‘)) t
    in
    SOME (el 1 match, el 2 match) handle _ => NONE
    end

fun buildINSTRtree t =
    (* SEQ *)
    case destSeq t of
         SOME z => SEQ (map  buildINSTRtree z)
       | NONE   =>
    (* ASG *)
    case destAsg t of
         SOME (v,e) => ASG (v,e)
       | NONE       =>
    (* SKIP *)
    case destSkip t of
         SOME _ => SKIP
       | NONE   => (* if nothing matches then raise error *)
```

The function `buildINSTRtree` recurses over the tree of the input term `t`. At eact point it checks if the top term of `t` matches either the pattern of a sequence of statements, an assignment, or a skip. The matching is done by the `dest-` functions. If one of these patterns is encountered, the corresponding `dest` function will deconstruct the term, and subsequently `buildINSTRtree` uses the resulting fragments to construct the corresponding `INSTR` structure. For pattern matching the `dest-` functions can use HOL built-in pattern matcher (`match_term`).

The `reduce` function (over instructions) can be implemented in the algebraic-style ala [4], namely as an so-called algebra folded over `INSTR`-trees (if we view values of `INSTR` as trees). The corresponding fold function (for `INSTR`) is:

```
fun foldINSTR (fskip,fasg,fseq) i =
    let
```

```
        fun fold SKIP                 = fskip
          | fold (ASG (lhs,rhs))      = fasg lhs rhs
          | fold (SEQ instructions)   = fseq (map fold instructions)
        in
        fold i
        end
```

The tuple (fskip, fasg, fseq) is called an INSTR-algebra. The function foldINSTR performs a recursion over an INSTR-tree. An algebra is passed to it which controls how values are combined during the recursion. Now, reduce can be implemented by folding the algebra below; V is here not needed because none of the instructions in this simplified setup produce verification conditions:

```
    val simplified_L0Logic_alg =
        let
        val fresh       = mk_fresh x
        fun fskip q     = q
        fun fasg  x e q = q<-(fresh/old x)<-(e/x)<-(fresh/old x)
        fun fseq  fs q  = foldr (op o) id fs q
        in
        (fskip,fasg,fseq)
        end
```

The advantage of this style of implementation is that variations of the existing logics can be easily and compositionally constructed by applying some alteration on the algebra that specifies the existing logics. For example if we want to have a variant of the above logic where assignments are treated differently, we can do it like this:

```
    val new_L0Logic_alg =
        let
        val (fskip,_,fseq) = simplified_L0Logic_alg
        val new_fasg       = ...
        in
        (fskip,new_fasg,fseq)
        end
```

Had we implemented simplified_L0Logic directly as a recursive function, then altering its recursive behavior will be impossible. See also our paper about compositional development of VCG: [8].

Other syntax-driven functions on $\mathcal{L}_0$ can also be built as folded algebras, for example a function for translating $\mathcal{L}_0$ to some an executable language, e.g. ML.

Current implemenation of $\mathcal{L}_0$ is prototype version 1 featuring: syntactical embedding in HOL, the logic, a translator to ML to produce executables, a unit test specification language , a test generator, and a test verifier (for the last three, see Section 7 for details).

7 TEST

TEST is a simple language to write a suite of unit tests on an $\mathcal{L}_0$ program. A *test suite* is either a test instruction or a list of other test suites. A *test instruction* is just an $\mathcal{L}_0$ instruction which includes one or multiple calls to the tested program. Such a call can be additionally marked with the keyword whitebox —we will explain its meaning later. We will assume that we have a way to execute $\mathcal{L}_0$ programs, so that we can also execute a TEST suite. An example of a suite is given below:

```
    SUITE /:: main.
```

```
/{ TEST /:: one. let x=0 and y=1
                 in
                 /{ whitebox (/@ swap(REF x, REF y))
                  ; assert((x=1) /\ (y=0))
                 /}

 ; TEST /:: two. let x0=x and y0=y
                 in
                 /{ whitebox (/@ swap(REF x, REF y))
                  ; whitebox (/@ swap(REF x, REF y))
                  ; assert((x=x0) /\ (y=y0))
                 /}
/}
```

The suite is called `main`; it consists of two test-instructions (which begins with the keyword `TEST`). The names after the symbol `/::` are just labels associated to the corresponding test instruction or suite. The second test instruction, for example, calls `swap` twice, and asserts that the value of `x` and `y` should then be restored to their initial values.

The syntax of TEST is given in Appendix B. Like $\mathcal{L}_0$, it has been syntactically embedded in HOL. We will not show how it is done, but it follows the same line as the embedding of $\mathcal{L}_0$.

The executional semantic of $\mathcal{L}_0$ ignores `assert` instructions. In TEST `assert` has a different semantic: `assert` e checks e; if it is true then nothing happens, else an exception is thrown. So, in TEST `assert` is used to actually test conditions. This does imply that e should be a computable predicate. Since *Assertion* is in general uncomputable, whereas *Expr* is, we restrict e (in TEST) to be an *Expr*.

Rather than executing a test suite, we can also verify it. A test suite is passed if none of the constituting test instructions throws an `assert`-violation exception. Under partial correctness, this is equivalent to showing that each test instuction t satisfies $\{\texttt{T}\}\ t\ \{\texttt{T}\}$ in the $\mathcal{L}_0$ logic. The program P we test is called somewhere in t. $\mathcal{L}_0$ logic will however treat P as a black box, which may not be what we want here because as we may actually have access to P's source code and want to use it. In other words, we want an option to treat P as a white box. This is possible in TEST by 'tagging' the call to P with the word `whitebox`, which will have the effect that the body of P will be expanded[5]. This flag is only meaningful for verification: during the execution it is simply ignored. The `whitebox` expansion can be implemented as a pre-processor, using HOL built-in rewrite functions to do it. After the expansion we get an ordinary $\mathcal{L}_0$ instruction, which can be reduced normally using the `reduce` function from Section 5.

8 Lingu

Database scripting is an interesting application. Databases usually have so-called *integrity constraints* which have to be maintained by every transaction. A common technique is to check the constraints on the run time, and to roll back when they are violated. However, this is at the expense of performance since run time check and roll back are expensive operations. The alternative is to do verification at the compile time. In [9] Qian describes a simple database scripting language with a Hoare-styled logic. For many practical purposes, it is sufficiently expressive. It is also first order, which is especially attractive for verification. We will describe here another first order database scripting language called Lingu (short hand of "little language") built by customizing $\mathcal{L}_0$. Its expresiveness is at the same level as [9]. Lingu shares much of $\mathcal{L}_0$ syntax and logic, and most Lingu specific constructs can be translated to $\mathcal{L}_0$ constructs. However, Lingu is neither a subset or superset of $\mathcal{L}_0$. For example, it forbids $\mathcal{L}_0$ while-loop, but it extends $\mathcal{L}_0$ with special syntax to specify queries and table manipulations. To handle some features (namely fresh

[5]Special care must be taken if P is recursive; this beyond our scope however, since $\mathcal{L}_0$ does not support recursion.

key and aggragate function) the logic of Lingu also deviates from $\mathcal{L}_0$. Here is an example of a simple Lingu script:

```
move6 (REF all, REF selected)
        =
        pre  T
        post (all union selected = (old all) union (old selected))
        do
        /{ selected /:= selected
                        union
                        select (map r. r) (only r. r.score>=6) all ;
           delete all (drop r. r.score>=6)
        /}
        return void
```

It defines a script called `move6` that moves all entries `r` in the table `all` such that `r.score` $\geq$ `6` to the table `selected`. The post-condition in the specification says that the union of the two tables is left unchanged, thus implying that there is no entry lost during the operation.

A *database* is a set of tables. A Lingu script such as the one above specifies an operation, also called *transaction*, on a subset of the database. In practice a database system often serves multiple applications, which may concurrently try to execute transactions. In this context note that Lingu assumes that with respect to each other, transactions are executed atomically. Compared to e.g. [6] Lingu uses a simplistic model of table: it is just a finite set of basic typed or 'flat record' entries. Recall that $\mathcal{L}_0$ allows arbitrary HOL term as expressions and assertions. This will now be limited. We will first discuss the language $\mathcal{S}$ of limited set expressions. An important aspect of $\mathcal{S}$ is that it is first order and implementable. Lingu expressions have a more complicated syntax, but they are translated to $\mathcal{S}$.

8.1 $\mathcal{S}$ Expression

Basic types are types such as `bool` and `int`. A *flat record* is a record whose fields are not of a set type. A *flat set* is a set whose elements are of basic types or flat records. As said, we use a set to model a table. More precisely, Lingu tables are non-hierarchical; hence they are all flat sets. The names of the fields of a record are also called *attributes.* The term *attributes of a table* is also used to mean the attributes of the elements of the table, assuming that they are records.

Allowed types for $\mathcal{S}$ expressions, and also for Lingu expressions, are basic types, the types of flat records, and the types of flat sets.

An $\mathcal{S}$ expression is either a simple expression or a set expression. The syntax of simple expression is below; *UnOp* and *BinOp* are respectively unary and binary numeric or boolean operators; *BinOp* can also be = comparing two records.

$$
\begin{array}{lll}
\textit{SimplExpr} & \rightarrow & \textit{Constant} \mid \textit{Variable} \\
 & | & \textit{SimplExpr}.\textit{FieldName} \qquad \backslash\backslash \text{ field selection} \\
 & | & \textit{UnOp SimplExpr} \\
 & | & \textit{SimplExpr BinOp SimplExpr} \\
 & | & \backslash\backslash \text{ record forming} \\
 & & \texttt{<|}\ \textit{FieldName}\ \texttt{:=}\ \textit{SimplExpr}\ , \ldots ,\ \textit{FieldName}\ \texttt{:=}\ \textit{SimplExpr}\ \texttt{|>}
\end{array}
$$

Allowed set expressions in $\mathcal{S}$ are listed below; e is a simple expression, p, t, u are $\mathcal{S}$ expressions, p is a predicate (it is of type `bool`):

1. $\emptyset$ (empty set), $\{e\}$ (a singleton set), $t \cup u$ (set union)
2. Set comprehension: $\{e(r) \mid r \in t \wedge p(r)\}$
3. Set predicates: $e \in t$ and $(\forall r.\ r \in t \Rightarrow p(r))$

Expressions of the form $(\exists r.\ r \in t \wedge p(r))$ are not in the syntax, but are also allowed as shorthands for the negation of the corresponding $\forall$ expressions. Furthermore, all set typed variables in $\mathcal{S}$ are assumed to specify *finite* sets. The following two lemmas can be proven quite straight forwardly by induction over the structures of simple expressions and of $\mathcal{S}$:

Lemma 8.1 : Every simple expression is either of a basic type or a flat record.

Lemma 8.2 : Every $\mathcal{S}$ expression is either a flat record, a flat set, or has a basic type.

Notice that the second lemma implies that the syntax of $\mathcal{S}$ itself enforces the restriction that all tables in $\mathcal{S}$ should be flat sets.

Let $\mathrm{FOL}_{\mathcal{S}}$ be the first order logic over the follwing set $\mathcal{T}$ of terms. $\mathcal{T}$ consists of all $\mathcal{S}$ expressions which are either a set typed variable or a simple expression of a non-boolean type. The predicate symbols of $\mathrm{FOL}_{\mathcal{S}}$ are all numeric relations of $\mathcal{S}$ (such as $\leq$), $=$ over simple and flat record types, and $\in$. An $\mathcal{S}$ expression of type `bool` is also called an $\mathcal{S}$ predicate. An $\mathcal{S}$ predicate is called *first order* if it can be translated to an equivalent $\mathrm{FOL}_{\mathcal{S}}$ formula.

Theorem 8.3 : All $\mathcal{S}$ predicates are first order.

Proof: by induction. The non-trivial cases are:

- $(\forall r.\ r \in u \Rightarrow p(r))$ is first order if: (1) r ranges over the elements of a flat set, (2) $r \in u$ is first order, and (3) $p(r)$ is first order. The first follows from Lemma 8.2, which says that u must be a flat set; (2) and (3) follow from the inductive assumption.
- Case $e \in t$. We have a number of sub-cases, depending on the form of t:
    1. t is a set-typed variable: then both e and t are $\mathcal{T}$-terms and $e \in t$ is thus a $\mathrm{FOL}_{\mathcal{S}}$ formula.
    2. $e \in \emptyset$ is equivalent to false.
    3. $e \in \{e'\}$ is equivalent to $e = e'$. Furthermore, e and e' must be of a simple or flat record type. So, they are $\mathcal{T}$-terms, and hence $e = e'$ is a $\mathrm{FOL}_{\mathcal{S}}$ formula.
    4. $e \in u_1 \cup u_2$ can be translated to $e \in u_1 \ \wedge\ e \in u_2$; each conjunct is first order by the inductive assumption.
    5. $e \in \{e'(r) \mid r \in u \wedge p(r)\}$ can be translated to $\neg(\forall r.\ r \in u \ \Rightarrow\ p(r) \ \Rightarrow\ \neg(e = e'(r)))$, which is first order. This can be argued in almost the same way as the $\forall$-case before, plus the fact that e and $e'(r)$ must be $\mathcal{T}$-terms and therefore $e = e'(r)$ is a $\mathrm{FOL}_{\mathcal{S}}$ formula.

$\square$

It follows that general results on the first order logic apply to $\mathcal{S}$ predicates. For example, if the resulting first order logic translation is monadic or is in the Pressburger syntax (which admits a limited form of numeric expressions) then it is decidable. HOL suports a number of automated proof tools to handle first order formulas.

Lemma 8.4 : Every set typed $\mathcal{S}$ expression specifies a finite set.

Proof: This can be proven inductively. The non-trivial cases are:

- $\{e(r) \mid r \in t \wedge p(r)\}$ is finite because t, by the inductive assumption, is finite.
- $t \cup u$ is finite, because t and u are, by the inductive assumption, finite.

$\square$

Theorem 8.5 : If simple expressions are implementable, then so is $\mathcal{S}$.

Proof: We only have to consider set expressions. By Lemma 8.4, every set typed $\mathcal{S}$ expression specifies a finite set. Operations $\cup$, $\in$, $\forall$, as well as comprehension, over finite sets are implementable.
$\square$

8.2 Lingu Expressions and Database Specific Instructions

A Lingu expression is either a simple expression (with the same syntax as used in $\mathcal{S}$) or a *table expression*. A table expression is essentially a query to a table. Before evaluation, a table expression is first translated to an $\mathcal{S}$ expression. Allowed table expressions are listed below, along with their $\mathcal{S}$ semantic; e is here a simple expression, p, t, u are Lingu expressions, p is of type `bool`:

1. Empty, select, and union:

$$\begin{array}{lcl} \texttt{empty} & = & \emptyset \\ \texttt{select}\ t\ (\texttt{map}\ r.\ e(r))\ (\texttt{only}\ r.\ p(r)) & = & \{e(r) \mid r \in t \wedge p(r)\} \\ t\ \texttt{union}\ u & = & t \cup u \end{array}$$

2. Table predicates:

$$\begin{array}{llclcl} e\ \texttt{IN} & t & & = & e \in t \\ \texttt{ALLof} & t & (\texttt{satisfy}\ r.\ p(r)) & = & (\forall r.\ r \in t \Rightarrow p(r)) \\ \texttt{SOMEof} & t & (\texttt{satisfy}\ r.\ p(r)) & = & (\exists r.\ r \in t \wedge p(r)) \end{array}$$

Notice that the syntax allows for example `select` and `ALLof` expressions to be nested in each other.

A Lingu predicate is a Lingu expression of type `bool`. Lingu has the following database specific instructions; whose meaning are defined in terms of $\mathcal{L}_0$ assignments over $\mathcal{S}$ expressions; e is here a simple expression, t, t_0 are Lingu expressions, p is a Lingu predicate:

1. Insertion:

$$\begin{array}{lcl} \texttt{ins}\ e\ t & = & t \mathrel{/{:}{=}} \{e\} \cup t \\ \texttt{insert}\ t_0\ t\ (\texttt{map}\ r.\ e(r))\ (\texttt{only}\ r.\ p(r)) & = & t \mathrel{/{:}{=}} \{e(r) \mid r \in t_0\ \wedge\ p(r)\}\ \cup\ t \end{array}$$

2. Deletion:

$$\begin{array}{lcl} \texttt{del}\ e\ t & = & t \mathrel{/{:}{=}} \{r \mid r \in t\ \wedge\ (r \neq e)\} \\ \texttt{delete}\ t\ (\texttt{drop}\ r.\ p(r)) & = & t \mathrel{/{:}{=}} \{r \mid r \in t\ \wedge\ \neg p(r)\} \end{array}$$

3. Update:

$$\begin{array}{l} \texttt{update}\ t\ (\texttt{map}\ r.\ e(r))\ (\texttt{only}\ r.\ p(r)) \\ = \\ t \mathrel{/{:}{=}} \{r \mid r \in t\ \wedge\ \neg p(r)\}\ \cup\ \{e(r) \mid r \in t\ \wedge\ p(r)\} \end{array}$$

Only a Lingu expression is allowed to be used as an assertion (plus the use of special keywords like `ret` and `old`).

Given the above semantics, a Lingu program and its pre/post specification can be translated into a plain $\mathcal{L}_0$ specification where the assertions and expressions are all $\mathcal{S}$ expressions. This can be reduced using $\mathcal{L}_0$ logic, resulting in verification conditions which are $\mathcal{S}$ predicates. The latter, by Theorem 8.3, can be further translated to first order logic formulas for verification. Furthermore, by Theorem 8.5 all Lingu expressions are also implementable. Also, since the language of assertions in Lingu is just the same as expressions, with the exception of the use of `old` and `ret` , checking them at the run time is implementable.

So far the logic of Lingu is the same as $\mathcal{L}_0$. However, later we will alter it slightly to handle keys and aggregate functions.

8.3 Implementation

Lingu extends $\mathcal{L}_0$ with its own syntax of expressions and a number of its own specific instructions. We can (syntactically) embed it in HOL simply by extending our $\mathcal{L}_0$ embedding. For example, to add the syntax and semantic of `select` we do:

```
Define ‘select t f p  =  {f r | r IN t /\ p r}‘ ;

new_binder_definition ("map_def",  --‘(map:('a->'b)->('a->'b)) body = body‘--) ;

new_binder_definition ("only_def", --‘(only:('a->bool)->('a->bool)) body = body‘--) ;
```

The first line defines the semantic of `select`, the other two introduce the syntax and semantic of the `map` and `only` quantifiers. Now we can write in HOL:

```
select  t  (map r. 0)  (only r. r>0)}
```

which would have the semantic $\{0 \mid \texttt{r IN t} \wedge \texttt{r} > 0\}$ in HOL.

The logic of Lingu works by first translating a Lingu specification to $\mathcal{S}$ (see previous Subsection), and then calling $\mathcal{L}_0$ logic. Implementing the translation is quite easy by using HOL built-in rewrite utilities.

The current implementation of Lingu is prototype 4.1. It can accepts, for example, the script `move6` at the beginning of Section 8 and reduce it to verification conditions; a simple HOL tactic can prove the latter. The next subsections discuss some further features which are not yet supported in the current prototype.

8.4 Integrity Constraint

A database often have a set of global invariants, also called *integrity constraints*, which have to be maintained by any transaction on it. Lingu can express various sorts of constraints, although of course they must be first order. For example a table t can be constrained to have its attribute K as a *primary key*, which means that the values of the K attribute in t uniquely identify the elements of t. This can be expressed by this Lingu predicate:

$$\texttt{ALLof}\ t\ (\texttt{satisfy}\ r.\ \texttt{ALLof}\ t\ (\texttt{satisfy}\ r'.\ (r.K = r'.K) \Rightarrow (r = r')))$$

Given a Lingu script P, to verify it against an integrity constraint c we can simply extend P's pre- and post-condition with c and verify the new specification.

8.5 Generating Fresh Keys

Primary key constraints are very common in database. Often, we need a way to generate a fresh key. There is a slight complication in how $\mathcal{L}_0$ handles this.

We introduce with a special instruction. If t is a table, the assigment k /:= `newkey` t assigns a value of type $'a$ `KEY` (for some compatible type $'a$) as a 'fresh' key to k. The key is fresh in the sense that it does not occur anywhere in t. The type $'a$ has to be an infinite type to guarantee that we can always generate a fresh key (so, `bool` is not allowed.). We will make a number of decisions to simplfy the formal treatmant of `newkey`:

1. `newkey` can only be used with the above syntax (it cannot be used in an arbitrary expression).
2. The generated key depends only on the state of the specified table. It means that, for example:

   $\texttt{k}_1$ /:= `newkey t` ; $\texttt{k}_2$; /:= `newkey t`

   will actually assign the same key to $\texttt{k}_1$ and $\texttt{k}_2$. If a different (fresh) key is desired for $\texttt{k}_2$, then $\texttt{k}_1$ has to be inserted first to `t`.

3. The logic treats the exact result of `newkey` as unspecified, except for the fact that it is fresh. This does mean that from the logic's view a `newkey` assignment behaves non-deterministically, so the original $\mathcal{L}_0$ rule for assignment cannot be used. We alter it as follows. Assume a table t with attributes $K_1, \ldots, K_n$ as primary keys:

$$\texttt{reduce}\ (k \mathrel{/{:}{=}} \texttt{newkey}\ t)\ q \quad = \quad \texttt{return}\ (\exists k.\ F_{k,t,K_1} \wedge \ldots \wedge F_{k,t,K_n}\ \Rightarrow\ q)$$

where $F_{k,t,K}$ is an $\mathcal{S}$ predicate stating that k does not occur in the K-column of t. So:

$$F_{k,t,K} \quad = \quad \neg(k \in \{r.K \mid r \in t\})$$

The predicate returned by the altered rule above is actually not an $\mathcal{S}$ expression, though it is still first order.

8.6 Folded Operation

If $\oplus$ is a binary operator, the fold of $\oplus$ over a sequence of values, for example x_1, x_2, x_3, is just $x_1 \oplus (x_2 \oplus x_3)$. For example, the sum function can be obtained by folding $+$. A folded operation is also called *aggregate*. In database we often want to fold over a column of a table. This is however more difficult in our set model. Because a set is unordered, only a commutative and associative operation can be folded. So, addition will work, but not substraction. We introduce the following syntax to denote fold:

$$\texttt{fold}\ \$\oplus\ t\ (\texttt{map}\ r.\ e(r))$$

The expression denotes the value obtained by folding $\oplus$ over the sequence $e(r_0), e(r_1), \ldots$, where the r_i's are elements of t —each element should only be taken once. If t is empty, the fold returns the identity element of $\oplus$. `fold` can be represented by a HOL function of type:

$$('b \to' b \to' b) \to' a\ \texttt{set} \to ('a \to' b) \to' b$$

The complete definition of `fold` requires a sequence (permutation) of elements of t to be chosen, over which we then fold the $\oplus$. Such a definition makes proving properties about folded operations in general hard, since we may have to quantify over the possible permutations over the elements of t. For example to show:

$$\texttt{fold}\ \$+\ t\ (\texttt{map}\ r.\ r) \ = \ \texttt{fold}\ \$+\ u\ (\texttt{map}\ r.\ r)$$

comes down to finding two permutations that will order elements of t and u respectively to the same sequence.

Alternatively, we can extend HOL theorem-base with a list of useful first order (but abstract/incomplete) properties of the fold of various operators and program the verification tatcics to also make use such a theorem-base. Example of such a property is the following, which assumes that $\oplus$ is $\leq$-monotonic:

$$\texttt{ALLof}\ t\ (\texttt{satisfy}\ r.\ e(r) \geq 0) \quad \Rightarrow \quad \texttt{fold}\ \$\oplus\ t\ (\texttt{map}\ r.\ e(r))\ \geq\ 0 \tag{2}$$

The propertyt gives a sufficient condition to prove that a fold returns a non-negative value. Another example is this set of general properties, which can be quite effective to simplify a fold expression as long as the target set expression is not a set comprehension:

$$\begin{array}{lll}
\texttt{fold}\ \$\oplus\ \emptyset\ (\texttt{map}\ r.\ e(r)) & = & \texttt{id}_{\oplus} \\[1ex]
\texttt{fold}\ \$\oplus\ \{r\}\ (\texttt{map}\ r.\ e(r)) & = & r \\[1ex]
(\texttt{ALLof}\ t\ (\texttt{satisfy}\ r.\ \neg\, r{\in}u)) & \Rightarrow & (\ \texttt{fold}\ \$\oplus\ (t \cup u)\ (\texttt{map}\ r.\ e(r)) \\
 & & = \\
 & & \texttt{fold}\ \$\oplus\ t\ (\texttt{map}\ r.\ e(r))\ \oplus\ \texttt{fold}\ \$\oplus\ t\ (\texttt{map}\ r.\ e(r)))
\end{array}$$

9 Closing Remarks

Syntactical embedding in a theorem prover offers an interesting alternative for rapid prototyping of small programming languages with an integrated theorem proving support. The main advatages are: the integration, a language front-end can be quickly obtained with few simple customizations, and high reuse of the theorem prover's own type checker. However, the approach is probably not suitable larger languages. Such a language may require a parser which is beyond the customization range of the host theorem prover and a type system which cannot be mapped naturally to the (often simple) type system of the theorem prover.

An alternative is to embed directly in the meta programming language of the theorem prover. The meta language is usually a functional language. Functional languages are known to be an excellent medium to implement Domain Specific Languages (languages for a specific sets of tasks, in contrast to general-purpose programming languages such as Java) or DSLs, by embedding the DSLs in the functional languages [3, 11]. However to also integrate a theorem prover as a verification support to a DSL embedded in the prover's meta language means that we would use the theorem prover directly on fractions of its own meta language; this requires some reflection ability at the meta language level. There is at the moment not much theorem provers with such a feature. HOL and Isabelle for example use Moscow-ML and Poly-ML respectively; both do not support reflection. Agda or Prufrock [13] may be (future) options. Both are implemented in Haskell which supports reflection e.g. via Template Haskell [10]. However, as software product neither is yet as mature as HOL and Isabelle (Prufrock is even still in development).

References

[1] D. Syme. Machine Assisted Reasoning About Standard ML Using HOL. Technical report, Australian National University, November 1992. ftp://ftp.cl.cam.ac.uk/hvg/papers/MLinHOL.thesis.ps.gz.

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

[3] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, 1999.

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

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

[6] Jan Paredaens, Paul De Bra, Marc Gyssens, and Dirk Van Gucht. *The Structure of the Relational Database Model.* Springer, 1989.

[7] Lawrence C. Paulson. Mechanizing UNITY in isabelle. *ACM Trans. Comput. Log*, 1(1):3–32, 2000.

[8] I.S.W.B Prasetya, A. Azurat, T.E.J. Vos, and A. van Leeuwen. Building verification condition generators by compositional extensions. In *Proceedings of 3rd IEEE International Conference on Software Engineering and Formal Methods.* IEEE Computer Society Press, 2005.

[9] Xiaolei Qian. An axiom system for database transactions. *Information Processing Letters*, 36(4):183–189, 15 November 1990.

[10] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[11] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In *Lecture Notes in Computer Science*, volume 1129, pages 184–207, 1996.

[12] Joachim van den Berg and Bart Jacobs. The LOOP compiler for java and JML. In *Proceeding of TACAS 2001*, pages 299–312, 2001.

[13] J. Ward, G. Kimmell, and P. Alexander. A general first-order theorem prover for Rosetta. `http://www.ittc.ku.edu/Projects/SLDG/files/prufrock-0.1.pdf`.

A Syntax of $\mathcal{L}_0$

```
ProgDecl      →  ProgName (FormalParam , ... ) = ProgDeclRHS

ProgDeclRHS   →   pre (Assertion)
                  post (AssertionRet)
                  do /{ Instr /} return (Expr)

FormalParam   →  REF? Var

Var           →  Identifier | ( Identifier : HOLType )

Assertion     →  HOLTerm   \\ ret is not allowed

AssertionRet  →  HOLTerm   \\ ret is allowed

Expr          →  HOLTerm   \\ ret and old are not allowed

Instr         →  skip
              |  Expr /:= Expr
              |  /{ Instr ; ... /}
              |  /@ ProgName ( ActualParam , ... )
              |  Expr /@= ProgName ( ActualParam , ... )
              |  if (Expr) then /{ Instr /} else /{ Instr /}
              |   let
                  Var = Expr and
                  ...
                  in /{ Instr /}
              |  while (Expr) wdo /{ Instr /}
              |  assert (Assertion)

ActualParam   →  REF Var    \\ pass-by-reference
              |  Expr    \\ pass-by-value
```

B Syntax of TEST

```
Test-Suite  →  TEST Label? Test-Instr
            —  SUITE Label? /{ Test-Suite ; ... ; Test-Suite /}

Label       →  /:: Identifier .
```

The syntax of *Test-Instr* is as *Instr*, except:

1. The following syntax is also allowed (white box call):

   `whitebox` (*ProgramCall*)

2. The syntax for `assert` is restricted to: `assert`(*Expr*)

C Syntax of Lingu

Lingu has the same syntax as $\mathcal{L}_0$, except the following. The syntax for assertions and expressions are the same:

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{SimpleExpr} \;\mid\; \mathit{TableExpr} \\
\\
\mathit{SimpleExpr} & \rightarrow & \mathit{Constant} \;\mid\; \mathit{Var} \\
 & \mid & \mathit{SimplExpr}\texttt{.}\,\mathit{FieldName} \\
 & \mid & \mathit{UnOp}\ \mathit{SimpleExpr} \\
 & \mid & \mathit{SimpleExpr}\ \mathit{BinOp}\ \mathit{SimpleExpr} \\
 & \mid & \texttt{<|}\ \mathit{FieldName}\ \texttt{:=}\ \mathit{SimplExpr}\ ,\ \ldots\ ,\ \mathit{FieldName}\ \texttt{:=}\ \mathit{SimplExpr}\ \texttt{|>} \\
\\
\mathit{TableExpr} & \rightarrow & \texttt{empty} \\
 & \mid & \texttt{select}\ \mathit{Expr}\ \texttt{(}\ \texttt{map}\ \mathit{Var}\ \texttt{.}\ \mathit{SimpleExpr}\ \texttt{)}\ \texttt{(}\ \texttt{only}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)} \\
 & \mid & \mathit{TableExpr}\ \texttt{union}\ \mathit{TableExpr} \\
 & \mid & \mathit{SimpleExpr}\ \texttt{IN}\ \mathit{Expr} \\
 & \mid & \texttt{ALLof}\ \mathit{Expr}\ \texttt{(}\ \texttt{satisfy}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)} \\
 & \mid & \texttt{SOMEof}\ \mathit{Expr}\ \texttt{(}\ \texttt{satisfy}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)}
\end{array}
$$

Lingu extends $\mathcal{L}_0$ instruction sets with the following:

$$
\begin{array}{lcl}
\mathit{Instr} & \rightarrow & \ldots \quad \backslash\backslash \text{ as in } \mathcal{L}_0 \\
 & \mid & \texttt{ins}\ \mathit{SimpleExpr}\ \mathit{Expr} \\
 & \mid & \texttt{insert}\ \mathit{Expr}\ \mathit{Expr}\ \texttt{(}\ \texttt{map}\ \mathit{Var}\ \texttt{.}\ \mathit{SimpleExpr}\ \texttt{)}\ \texttt{(}\ \texttt{only}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)} \\
 & \mid & \texttt{del}\ \mathit{SimpleExpr}\ \mathit{Expr} \\
 & \mid & \texttt{delete}\ \mathit{Expr}\ \texttt{(}\ \texttt{drop}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)} \\
 & \mid & \texttt{update}\ \mathit{Expr}\ \texttt{(}\ \texttt{map}\ \mathit{Var}\ \texttt{.}\ \mathit{SimpleExpr}\ \texttt{)}\ \texttt{(}\ \texttt{only}\ \mathit{Var}\ \texttt{.}\ \mathit{Expr}\ \texttt{)}
\end{array}
$$