Theorem Prover Supported Logics for Imperative Programs

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

jul. 2004

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

---

Abstract

This report describes a simple imperative programming language $\mathcal{L}_0$ and its logic. Through syntactical embedding in the theorem prover HOL we were able to quickly implement a verification support for $\mathcal{L}_0$. The approach gives access to HOL built-in utilities, like tactics and rewrite functions, and allows much reuse of HOL built-in parser and type checker. However, unlike semantical embedding, meta theorems cannot be proven. The logic of $\mathcal{L}_0$ is Hoare-based and is completely syntax driven: it reduces a given specification to a set of verification conditions, which are plain HOL formulas —thus can be verified in HOL in the standard way. $\mathcal{L}_0$ features only basic imperative constructs (including block and program call). Although it can be used stand alone, its intended use is to serve as a stable core from which other languages can be built, e.g. via customization and extension. This report gives two examples of new languages built on $\mathcal{L}_0$: TEST, a language to write a suite of unit tests for $\mathcal{L}_0$ programs, and Lingu, a language for scripting database transactions.

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, e.g. logics for C [4], Java [2], ML [1], and UNITY [6]. All these examples, and most other embeddings of programming logics, are what we call *semantical embeddings*. Such an embedding maintains representations of both the abstract syntax and semantic of an object language L in the theorem prover's logic. Distinction between *shallow* and *deep* embedding is often made, depending on how detailed the syntax is represented. Isabelle is also suitable to do *syntactical embedding*. This is a different kind of embedding; its goal is not to have a representation of L's semantic in the theorem prover, but rather: to embed as much as possible of L's concrete syntax.

The drawback of syntactical embedding is clear. Without the semantic we will not be able to verify meta-theorems about L (e.g. the soundness of its logic). However, syntactical embedding gives us, with minimum effort, a parser and type checker for the object language, which are obtained mostly by reusing the theorem prover's own parser and type checker. This cannot be done in a semantical embedding; a full language-front-end will have to be built to complement it, e.g. as the LOOP front-end compiler [8] is used to drive its back-end Java logic, which is semantically embedded in HOL [2]. Semantical embedding is best suited for studying meta properties of a logic. To actually implement logics, in particular of programming languages, whose syntax are often complicated, syntactical embedding would be preferred. If a semantical embedding already exists, a syntactical embedding can serve as its language front-end.

We will discuss here a syntactical embedding of a Hoare-styled logic for an imperative programming language, which we simply call $\mathcal{L}_0$, in HOL. It is a minimalistic language, featuring only basic constructs: basic statements, block, program call, return value, pass-by-value and pass-by-reference parameters, and `old` keyword to refer to a variable's initial state. It has no notion of object nor compilation module. It also leaves the syntax of expressions and assertions underspecified. After sufficient concretization we can turn $\mathcal{L}_0$ to a concrete language which can be used stand alone. Our $\mathcal{L}_0$ library includes an example of such a concrete instance of $\mathcal{L}_0$, called $\mathcal{L}_0^{min}$, which comes with an ML translator for execution (and few other utilities, e.g. a test generator). However the intended use of $\mathcal{L}_0$ is to serve as a stable core from which other languages can be built, e.g. via customization and extension. This report also gives two examples of new languages built on $\mathcal{L}_0$: 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 $\mathcal{L}_0$. The new parts can be quite easily implemented by reusing utilities already provided by HOL.

Isabelle is commonly used to do syntactical embedding; it provides a framework and tools especially made for this purpose. We use HOL however, primarily because we are more familiar with it. HOL is not especially made for syntactical embedding and is more often used for semantical embedding. However, underneath HOL offers the same typed λ-calculus as the embedding medium. Later versions of HOL also allow syntax customization, which is very useful in syntactical embedding. It makes HOL at least worth considering for people already investing substantial effort in HOL.

A prototype of $\mathcal{L}_0$, $\mathcal{L}_0^{min}$, TEST, and Lingu can be downloaded from:

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

2 Two Kinds of 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.

Stmt :: `skip`
| *Variable* := *Expr*
| { *Statement*; ... ; *Statement* }

Expr :: *Variable* | *Bool-constant* | *Integer-constant*

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:

```
state      =  string → val
stmt       =  state → state
expr       =  state → value
assertion  =  state → bool
```

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 | Bint of int
```

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

```
skip  :  stmt
asg   :  string → expr → stmt
seq   :  stmt → stmt → stmt
```

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

```
skip s       =  s
asg v e s    =  (λ x. if x=v then  e s  else  s x)
seq S1 S2 s  =  S2 (S1 s)
```

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

```
HOA (P,S,Q)  =  (∀s. P s ⇒ Q (S s))
```

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

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

is represented by the formula:

```
(∀s. P s ⇒ Q s)  ⇒  HOA (P, skip, Q)
```

Because we have a semantic, we can actually verify that L's inference rules, like the one above for `skip`, follow from L's semantic. 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 {`x` := `0`; `b` := `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 semantic 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:

```
skip  :  Stmt
seq   :  Stmt → Stmt → Stmt
```

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:

```
/:=   :  bool → bool → Stmt
/:=   :  int → int → Stmt
```

The representation so far does not however gives us the concrete syntax for sequences. HOL syntax customization allows us to define our own list-like syntax; this is what we use to embed the concrete syntax of L's sequence. However, the customization is limited. It won't allow us to get the exact syntax of L sequences, though we can get something that is quite close to it:

```
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`. The statement /{ x/:=0; x/:=F /} will however be rejected by HOL, because it requires `x` to have both the type `int` and `bool`.

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. An assertion of the form `old y` refers to

the initial value of `y`; if `y` is a parameter of a program, it refers to the value of `y` when it is passed to the program. So, the specification of `swap` 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[1].

The syntax of program declaration is shown below:

ProgDecl → *ProgName* `(` *FormalParams* `) =` *ProgDeclRHS*

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

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.

By intention $\mathcal{L}_0$ 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

[1] Optionally, one can opt to actually check the asserted predicate at the run time, though this will be at the cost of performance. Our position here is that assertions are checked during the verification or testing, but not at the run time.

$\mathcal{L}_0$'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[2]. 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.

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[3].

$\mathcal{L}_0$ 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:

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

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

$$:= \ :\ 'a \to 'a \to \texttt{Stmt}$$

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:

```
wloop : bool → INSTR → 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))};
```

[2]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.

[3]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`.

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

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

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:

```
print  :  int → INSTR
print  :  string → INSTR
```

The representation of `assert` is straight forward:

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

4.1 Program Call

Program call is a bit more complicated. A program call like $\mathtt{x}\ /@ = \mathtt{P}(0, 1)$ 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:

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

So, $\mathtt{P}(0, 1)$ 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:

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

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:

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

HOL will then accept calls like $\mathtt{swap}(\mathtt{REF}\ \mathtt{e}_1,\ \mathtt{REF}\ \mathtt{e}_2)$, but not $\mathtt{swap}(0, 1)$ because `0` and `1` are not of type `REF`. We may want to restrict the syntax of actual parameters, for example, so that $\mathtt{e}_1$ and $\mathtt{e}_2$ 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. Let $p =$ `reduce` $S\ q$. The function works in such a way so that the validity of all predicates in V obtained after executing `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 $p =$ `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.

In the sequel, let $q[e/v]$ denote the expression obtained by replacing all free occurences of v in q with e. 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.

The predicates in V are also called *verification conditions* and the function `reduce` is also called *verification condition generator*. The algorithm is described below:

1. Print:

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

2. To rule for assignment would normally look like this:

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

   However, this ignores the fact that q may contain expressions `old` x. The x in the latter should not be substituted. Recall that `old` x does not refer to the current value of x, but rather to its initial value; hence the assignment should not influence the semantic of `old` x. The following rule does the trick:

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

   where $@x$ is a fresh variable.

3. Sequences:

   $$\texttt{reduce}\ (i_1; i_2)\ q \quad = \quad \texttt{reduce}\ i_1\ (\texttt{reduce}\ i_2\ q)$$

4. The rule for conditional is standard:

   $$\texttt{reduce}\ (\texttt{if}\ g\ \texttt{then}\ i_1\ \texttt{else}\ i_2)\ q \quad = \quad \texttt{if}\ g\ \texttt{then reduce}\ i_1\ q\ \texttt{else reduce}\ i_2\ q$$

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 our logic will simply assume `T` as the invariant. The rule:

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

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:

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

where $@v$ is a fresh variable. 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 expressions of `old` v which may occur in p and V as the result of the reduction on S. It refers to the initial value of v, which is e. So, they should be substituted with e. Consider now this rule:

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

This may seem to solve the problem discussed above. However, the subtitution on all predicates in V is incorrect. It should only be applied to the new predicates 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. The following one will cure this:

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

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:

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

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

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

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.

The following rule does it:

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

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.

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

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.

The `reduce` function (over instructions) can be implemented in the algebraic-style ala [3], 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. As 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: [7].

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. An example 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 the evaluation of e should be implementable. So, we restrict the syntax of e: it should be an ordinary *Expr* rather than *Assertion* (assuming that *Expr* is implementable).

For the execution of TEST the `whitebox` keyword has no meaning.

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 $\{\mathtt{T}\}\ t\ \{\mathtt{T}\}$ in the $\mathcal{L}_0$ logic. When unit testing we assume that we do have access to the source code of the tested programs. So, treating these programs as white boxes provides more information. Preceeding a program call with a `whitebox` keyword will tell the logic to expand the call to the appropriate instantiation of P's body[4]. This can be implemented as a pre-processor; we can use HOL built-in rewrite utilities 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

Lingu (short hand of "little language") is a variation of $\mathcal{L}_0$ intended for writing database transactions. A Lingu script has the same syntax as an $\mathcal{L}_0$ program, with a number of customizations: Lingu has its own syntax for expressions and assertions, it has a number of database specific instructions, and while-loop is not allowed in Lingu . Here is an example:

```
move6 (REF all, REF selected)
       =
       pre  T
       post (all union selected = (old all) union (old selected))
       do
       /{ let
          q = empty
          in
```

[4]If P is recursive, the white box expansion will only take place at the top level call. The recursive call is handled as a black box call. This implicitly requires that P has been proven correct with respect to its declared specification.

```
   /{ q /:= select (map r. r) (only r. r.score>=6) all ;
      delete all (drop r. r.score>=6) ;
      selected /:= q union selected
   /}
/}
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 table is left unchanged, thus implying that there is no entry lost during the operation.

A Lingu script as the one above can operate on tables. In Lingu a table is just a set of basic typed or 'flat record' entries[5]. 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.

In the database community 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.

SimplExpr → *Constant* | *Variable*
| *SimplExpr*.*FieldName* \\ field selection
| *UnOp SimplExpr*
| *SimplExpr BinOp SimplExpr*
| \\ record forming
`<|` *FieldName* `:=` *SimplExpr* , ... , *FieldName* `:=` *SimplExpr* `|>`

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. The following two lemmas can be prove 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.

[5]There are more sophisticated models of tables, e.g. [5]. At this point of research we stick to our simple model, keeping in mind that there are things like duplicate entries, aggregate column operations, sorted entries which are more difficult or even impossible to express with the current model.

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

Let $\text{FOL}_{\mathcal{S}}$ be the first order logic over the follwing set $\mathcal{T}$ of terms. $\mathcal{T}$ consists of all $\mathcal{S}$ expressions of which are either a set typed variable or a simple expression of a non-boolean type. The predicate symbols of $\text{FOL}_{\mathcal{S}}$ 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 $\text{FOL}_{\mathcal{S}}$ 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 $\text{FOL}_{\mathcal{S}}$ 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 $\text{FOL}_{\mathcal{S}}$ 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 $\text{FOL}_{\mathcal{S}}$ formula.

□

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 $\mathcal{S}$ expression of type set 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.

□

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

Proof: We only have to consider set expressions. By Lemma 8.4, a set expression of type set specifies a finite set. Operations $\cup$, $\in$, $\forall$, as well as comprehension, over finite sets are implementable.
□

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 queries a table, or construct a new table (in the sense that it is a set of entries). 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:

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

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:

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

2. Deletion:

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

3. Update:

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

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.

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‘--) ;
```

So now we can write in HOL:

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

which would have the semantic $\{0 \mid r \texttt{ IN } t \wedge r > 0\}$ 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. A simple HOL tactic can prove the verification conditions that we get after applying the logic of 4.1 on `move6`.

8.4 Verifying Key Constraints

Tables in databases often have keys. For example a *primary key* of a table t is an attribute K of t that uniquely identifies the elements of the table. This specifies a constraint, also called *key constraint*, for t, which we can express in Lingu :

$$\texttt{ALLof } t \ (\texttt{satisfy } r.\ \texttt{ALLof } t \ (\texttt{satisfy } r'.\ (r.K = r'.K) \Rightarrow (r = r'))) \tag{1}$$

Any script manipulating t should preserve this constraint.

One way to express key constraints is by specifying them explicitly in the post-condition of a script S. However this would clutter our specifications. So instead, we will use a pre-processor which appends an assertion C after each table assignment in S (remember that all database specific instructions in Lingu are basically table assignments —see the semantic in Subsection 8.2). The assertion C specifies all key constraints associated to the table targetted by the assignment. So, for example if t is a table with the attribute K as its only primary key, then an assignment $t /:= u$ will be transformed to:

$$t /:= u \ ; \ \texttt{assert } (C_{K,t})$$

where $C_{K,t}$ is the same formula as (1). Subsequently we simply use the same $\mathcal{L}_0$ logic to reduce the resulting program.

To make the generation of the key constraints easier, we encode the information about key qualifiers in the type of the table. This way we can easily retrieve back the information via HOL type system. So, introduce a new HOL data type:

```
hol_datatype KEY  =  KEY of ’a
```

This type represent (primary) keys. If x is a plain value, e.g. an integer, `KEY` x turns it to a key. For every basic type we have in Lingu (e.g. `int`), we add `int KEY` as a new basic type.

If we now, for example, want to have a table t of personal records where the attribute `PersonId` is a primary key we would now declare t as a table over records of, for example, the type `Person` below:

```
hol_datatype   Person = <| PersonId : KEY int; Name : string |>
```

We also introduce a new form of assignment to generate new keys. 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.). To simplify the formalization of `newkey`, we make the choice of the generated key to depend only on the state of t. 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`.

As further simplification we leave the exact result of `newkey` unspecified (except for the fact that it is fresh). This does mean that usual rule to handle assignment cannot be used to handle the `newkey`. The new rule is given below. Assume a table t with attributes $K_1, \ldots, K_n$ as primary keys:

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

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

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

Note that with the addition of the data type `KEY`, $\mathcal{S}$ (the language underlying Lingu) is still first order (the underlying term language ($\mathcal{T}$) does change since we now also have terms of the form `KEY` e).

8.5 Aggregate Function

Expressing an aggregate function, like the sum over a column, is more difficult in our set model. Because a set is unordered, only a commutative and associative operation can be aggregated. So, addition will work, but not substraction. We will only discuss aggregare sum here. Let t be an $\mathcal{S}$ expression of a set type, and e be a simple expression. We introduce the following syntax:

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

to denote the sum of all $e(r)$ for all r taken from t. Each r should only be taken once. We now need the semantic of this syntax. We can give one, but it will be rather complicated. Instead, we will now choose to define the semantic axiomatically. It will be simple, though at the expense of completeness. We will restrict the sort of predicates we want to express about `sum` to be predicates of this form:

$$\texttt{sum}\ t\ (\texttt{map}\ r.\ d(r)) \quad = \quad \texttt{sum}\ u\ (\texttt{map}\ r.\ e(r)) \tag{2}$$

When we have this predicate in a verification condition, we can reduce it to the following predicate:

$$(\exists f.\ \ t \xleftrightarrow{f} u\ \ \wedge\ \ (\forall r.\ r \in t \Rightarrow (d(r) = e(f\ r)))) \tag{3}$$

where $t \xleftrightarrow{f} u$ means that f is a bijective function from t to u. The predicate means that every record r in t has exactly one counterpart $r' = f\ r$ in u, and moreover the value of $d(r)$ is the same as $e(r')$. Consequently, the specified sums above are equal. This is a sufficient condition, but as said not a necessary one.

The predicate in (3) cannot be express in $\mathcal{S}$, but we can still express it in HOL. It is also no longer first order due to the quantification over f.

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

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

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

[6] I. S. W. B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Inst. of Information and Comp. Science, Utrecht Univ., 1995. Download: `www.cs.uu.nl/library/docs/theses.html`.

[7] 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.

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

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:

Expr → *SimpleExpr* | *TableExpr*

SimpleExpr → *Constant* | *Var*
| *SimplExpr*`.`*FieldName*
| *UnOp SimpleExpr*
| *SimpleExpr BinOp SimpleExpr*
| `<|` *FieldName* `:=` *SimplExpr* , ... , *FieldName* `:=` *SimplExpr* `|>`

TableExpr → `empty`
| `select` *Expr* `(` `map` *Var* `.` *SimpleExpr* `)` `(` `only` *Var* `.` *Expr* `)`
| *TableExpr* `union` *TableExpr*
| *SimpleExpr* `IN` *Expr*
| `ALLof` *Expr* `(` `satisfy` *Var* `.` *Expr* `)`
| `SOMEof` *Expr* `(` `satisfy` *Var* `.` *Expr* `)`

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

Instr → ... \\ as in $\mathcal{L}_0$
| `ins` *SimpleExpr Expr*
| `insert` *Expr Expr* `(` `map` *Var* `.` *SimpleExpr* `)` `(` `only` *Var* `.` *Expr* `)`
| `del` *SimpleExpr Expr*
| `delete` *Expr* `(` `drop` *Var* `.` *Expr* `)`
| `update` *Expr* `(` `map` *Var* `.` *SimpleExpr* `)` `(` `only` *Var* `.` *Expr* `)`