Using PVS to Prove a Z Refinement: A Case Study

David W.J. Stringer–Calvert[1], Susan Stepney[2], Ian Wand[1] *

[1] Department of Computer Science, University of York, U.K.
[2] Logica UK Ltd., Cambridge, U.K.
{davesc,icw}@minster.york.ac.uk, stepneys@logica.com

To be presented at FME 97, University of Graz, Austria.
September 1997

Abstract. The development of critical systems often places undue trust in the software tools used. This is especially true of compilers, which are a weak link between the source code produced and the object code which is executed. Stepney [23] advocates a method for the production of *trusted* compilers (i.e. those which are guaranteed to produce object code that is a correct refinement of the source code) by developing a proof of a small, but non trivial compiler by hand in the Z specification language. This approach is quick, but the type system of Z is too weak to ensure that partial functions are correctly applied.
Here, we present a re–working of that development using the PVS specification and verification system. We describe the problems involved in translating from the partial set theory of Z to the total, higher order logic of the PVS system and the strengths and weaknesses of this approach.

1 Introduction

Computer systems are increasingly being used in applications where their failure could lead to financial or environmental disaster, or even to loss of life. Such systems are called *Critical* and as such must be engineered to the highest quality, to anticipate potential faults and to reduce the possibility of errors in the system.

The major enemy in seeking assurance in a computer system is complexity. Complex systems are difficult to specify and to reason about, hence mistakes and omissions *will* be made during the design and implementation process. One characteristic of a good design process is that it should provide for validation and verification of the 'product' at every stage, so as to catch introduced errors

* Logica UK Ltd carried out the original work on the compiler method for RSRE (now DRA Malvern), and is continuing the development for AWE plc. D.W.J. Stringer–Calvert is funded by a CASE studentship from the Engineering and Physical Sciences Research Council and Siemens Plessey Systems.

earlier (allowing solutions to be implemented more cheaply) and to provide a trace of the development, which may be used in a safety argument.

At the highest level of rigour, we would use a refinement process to provide a fully formal development route from the requirements specification (expressed in some suitable formal notation), via formal refinement rules (which have been proven to preserve meaning), to code in a high level language, say Ada.

Having now, at great expense, formally refined our requirements into a concrete implementation in Ada, we have a dilemma. As noted, Ada is a *high level* language, indicating that it contains constructs intended to make the programming task easier and less error prone. However, nothing comes for free and the payment for this is made by the need to use a compiler or interpreter (a large and complex piece of software) in order for the program we have written to be executed on a real computer. Thus, the compiler and associated tools (including an assembler, linker and libraries) are in a position of great trust in the building of our software system.

Formal methods are often seen as the solution to the problem of assurance in critical systems, but their application is usually limited to the most critical parts of a system, as they are mostly seen as costly and difficult to use. Thus a cost/rigour tradeoff has developed, with 'formalised' methods (such as Z [22] with hand proofs) which provide less than full verification, becoming the popular choice in industry.

Recent developments in full strength *mechanised* verification have lent more credibility to the possible use of full formal verification on larger numbers of critical systems developments. Systems such as the Prototype Verification System (PVS) [16,19] provide the necessary automation of much of the tedious detail of fully formal proofs, and also allow for the construction of proof strategies specific to the problem domain where necessary.

This paper presents the initial stages of our work on compiler correctness, moving from a specification in Z, with hand proofs, through to a rigorous treatment with PVS. This treatment has revealed several errors in the original hand development, and forms the basis for further investigation into the extent to which mechanisation can be applied to automate proofs in this problem domain.

2 The Starting Point

In this paper we build on the work of Stepney [23] who has developed a *demonstrably correct* compiler for a simple (but non-trivial) language Tosca, targeted at an imaginary assembler Aida (similar to that used on the Viper [5] processor). Tosca contains the usual features of a high level language — sequencing, assignment, choice, while loops, and a simple model of input/output based on streams of integers. The target language (Aida) is a simple assembly language with load/store, input/output, and arithmetic/logical operations.

Both languages are defined denotationally (using the Z specification language [22]), and the source language is given an operational semantics in terms of templates of target instructions, thereby defining a compiler. Correctness proofs are discharged by hand, using the principle of structural induction over source language constructs.

The work described here is a re-engineering of those hand proofs within the framework of the PVS specification and verification system. The specification language of PVS is based on classical, typed, higher order logic. It contains constructs intended to ease the natural development of specifications such as parameterised modules, records, subtypes, and abstract datatypes. Unlike Z, the PVS logic does not admit partial functions, although this can be modelled using subtypes and dependent types.

The PVS proof checker is interactive, using a sequent style presentation. It provides powerful basic commands, and a mechanism for building re-usable strategies based on these. The power of the system comes from the use of decision procedures to automate efficiently the decidable aspects of the logic, and their close integration with rewriting.

As the type system of the PVS logic is so rich and expressive, type checking is undecidable. To overcome this, the type checker emits type correctness conditions (TCCs) to which the full weight of the theorem prover can be applied. Most of these TCCs are discharged automatically by standard strategies. TCCs are also emitted during proof as extra subgoals where required to ensure type correctness, for example when instantiating a lemma.

PVS was chosen over, say, a Z-based prover such as Z/EVES [20] because we anticipated that the automation it provided would make the verification task easier. In addition, we expected that the PVS type system would make the development less error prone than a hand development in Z, as PVS has both domain checking (i.e. ensuring that functions are applied only to arguments within their domain) and conservative definitions.

The remainder of this paper provides a short outline of the original Z development, followed by a discussion of some of the problems encountered in translating this into the PVS system. A brief discussion of related work is included, together with our future plans for research in this area.

3 Z Development

In this section we give an overview of the original Z development of the compiler, with an explanation of why the various design decisions were made. More detail is given elsewhere [23].

The development of a compiler by this method has three components:

1. **Specification**: The denotational semantics of both the high level source language (Tosca), and the low level assembly language (Aida), are specified in Z, as is the operational semantics of the high level language in the form of a set of templates of low level language instructions.

2. **Implementation**: The Z specification of the Tosca semantics are translated into Prolog, where they are executable. Executing the denotational semantics gives an interpreter; executing the operational semantics gives a compiler.
3. **Proof**: The operational semantics are proved to be equivalent to the denotational semantics: the compiler transformation is *meaning preserving*, and hence the compiler is correct.

For example, the meaning of Tosca's assignment statement `x := e` is a state change: the state σ is updated so that x maps to the value of the expression `e`. In Z this can be written as[1]:

$$\mathcal{M}[\![\mathit{assign}(\mathtt{x}, \mathtt{e})]\!]\sigma = \sigma \oplus \{x \mapsto \mathcal{M}[\![\mathtt{e}]\!]\sigma\}$$

In Prolog this becomes

```
meaning(assign(Name,Expr),Pre,Post):-
    meaning(Expr,Pre,Value),
    update(Name,Value,Pre,Post).
```

Similarly, the meaning of Aida's store instruction (store the contents of the accumulator at the location l) updates its store[2]. In Z:

$$\mathcal{A}[\![\mathit{store}\ \mathtt{l}]\!]\rho_\iota\theta\sigma_\iota = \theta(\sigma_\iota \oplus \{l \mapsto \sigma_\iota A\})$$

Here, σ_ι represents the Aida store, ρ_ι the environment mapping from program labels to continuations, θ the current continuation, and the term $\sigma_\iota A$ the value contained in the accumulator.

The operational semantics define the sequence of Aida instructions corresponding to the translation of each Tosca instruction. The assignment statement is translated into a sequence of instructions to evaluate the expression, followed by an instruction to store that value at the appropriate location. In Z:

$$\mathcal{O}\langle\!| \ \mathit{assign}(\mathtt{x}, \mathtt{e}) \ |\!\rangle = \mathcal{O}\langle\!| \ \mathtt{e} \ |\!\rangle \frown \langle \mathit{store}\ x \rangle$$

In Prolog this becomes

```
compile(assign(Name,Expr),[InstrList,store(Name)]):-
    compile(Expr,InstrList).
```

[1] The square brackets $[\![\ ]\!]$ are the conventional denotational semantics brackets, *not* Z's bag brackets.

[2] The semantics of Aida is more complicated than Tosca's, and needs to use 'continuation' arguments, because the assembly language allows arbitrary jumps. See [23] for more explanation.

To show that the compiler is correct, we need to show that the translation into Aida preserves the semantics of the Tosca instruction. We do so by structural induction over Tosca's abstract syntax, proving that for each construct, the Aida meaning of the translation template is the same as the Tosca meaning of the original instruction. For the assignment example we need to prove that:

$$\vdash \mathcal{M}[\![assign(\mathtt{x}, \mathtt{e})]\!] = \mathcal{A}[\![\mathcal{O} \langle\!| \ assign(\mathtt{x}, \mathtt{e}) \ |\!\rangle]\!]$$

As we can see from above, the specifications serve (at least) three conflicting purposes in this approach:

1. **Language definition**: the denotational semantics act as the language definition, and so the specification style should be as clear and abstract as possible, to make the definition comprehensible to human readers [1].
2. **Implementation**: the various semantics are translated into an executable language, and so should be written in a concrete, algorithmic, style that facilitates this translation. The declarative language Prolog, rather than some imperative language, is chosen as the target language in order to minimise this implementation step.
3. **Proof**: the various semantics need to be manipulated mathematically, in order to perform the correctness proofs, and so should be written as abstractly as possible.

In the original work the only tool available was *f*UZZ, a Z type checker [21]: no proof tools were used, and hence 'suitable for tools' was not a design criterion. In particular, Z's partial functions were exploited to structure the specifications, and to provide Tosca with various static checking semantics.

4 Translation to PVS

The first attempt to translate the Z specifications into PVS took a direct *naive* approach, where the specifications were relatively quickly translated into the logic of the PVS system with little modification. With this approach the main branch of the correctness theorems followed closely the original hand proofs, except for a few cases where unproven assumptions had been made, where the translation had been incorrect and also where the detail of a series of complex reasoning steps was omitted and presented as one step.

We therefore set about re-working the specification to allow the correctness theorems to be proved. This work involved the tightening of the specification, thereby making assumptions explicit, as will be shown in the following sections.

4.1 PVS Types

Whilst augmenting the functions in the specification, we found it very useful to also augment the types of those functions. As type information is available to the ground prover in PVS, this approach minimises the number of type correctness conditions generated as side effects of proof steps. For example, the following function shows the use of a dependently typed argument, and a predicate subtype for the range:

```
OE(epsilon : Expr)(rho_o : Inj_Env)
  (SP : {l : Locn | l >= top(rho_o)}) :
  RECURSIVE {l : list[Instr] | cons?(l) AND sequential?(l)} =

  ...
```

Here, the function `OE` returns a list of `Instr`, which satisfies the predicates `cons?` (i.e. it is a non-empty list), and `sequential?`, a predicate which we have defined to mean the list contains no `jump` or `goto` instructions.

In the remainder of this section we discuss some of the more interesting aspects of this conversion from Z to PVS, highlighting some of the original assumptions and describing the approaches taken to overcome them. Fragments of PVS specifications appear as boxed figures.

4.2 Total functions

Partial functions are a natural method for modelling many situations, and heavy use is made of them in Z specifications. It is difficult however to provide support for mechanical reasoning using partial functions, and as such PVS does not support their use. Therefore, the most challenging part of translating a Z specification into the logic of the PVS system is removing the use of such functions.

There are three basic methods for making a partial function into a total one:

1. Cause the function to return a specific, fictitious value (a *bottom*) for undefined cases.
2. Cause the function to return an arbitrary value of the correct type for undefined cases.
3. Constrain the type of the function arguments so that it is a total function over a restricted domain.

The second option can be achieved by the use of Hilbert's epsilon operator [13], which, given a predicate, returns a value which satisfies that predicate (if possible) and if it is not satisfiable returns an arbitrary value of the appropriate type. Epsilon is a useful specification tool, as it allows for a clever abstraction from detail, but for our purposes here, options one and three seem more appropriate[3].

[3] See section 2.4.2 of [19] for more discussion on the use of the epsilon operator in PVS

In our specification of the compiler problem in PVS, we have experimented with the use of methods one and three from the above list. We believe there is a tradeoff in their use between the readability of the specification, and ease of proof. Whereas both of these methods make explicit the inherent partial nature of a function, the use of a bottom element (method one) tends to clutter the body of the function, and the use of complex types (method three) tends to make the function 'head' quite opaque and results in extra type correctness conditions to ensure applications of the function are within its domain.

After experimenting with the use of both approaches (as will be seen later in this paper), we would now advocate the use of complex types to approach this problem. If we use a bottom element, the tool can give us no assistance in noting our mistakes and inconsistencies, but the correctness conditions generated by the use of complex types are extremely valuable for locating errors and omissions.

Such support leads us to believe that PVS would be a useful tool for language designers starting from scratch, using the PVS type system to explore and define various static semantics for the language after noting where and which obligations were generated.

4.3 Store

The most significant change to the specification involved the memory maps at both the Tosca and Aida levels. Pictorially, they are as shown in figure 1.

All variables in Tosca have global scope, so they are allocated storage statically, at compile time. Aida has static storage of the equivalent Tosca variables. It also has a stack, which grows from above the highest allocated memory location, to store temporary variables needed during expression evaluation.

The Z specification defines a function `restrict`, which performs a range restriction on the store, to yield the locations that are currently in use:

$$\mathit{restrict} : \mathit{Env}_O \times \mathit{Locn} \times \mathit{State}_I \rightarrow \mathit{State}_I$$

$$\begin{aligned}&\forall \rho_o : \mathit{Env}_O;\ \lambda : \mathit{Locn};\ \varsigma_\iota : \mathit{Store}_I;\ \mathit{in} : \mathit{Input};\ \mathit{out} : \mathit{Output} \bullet \\ &\quad \mathit{restrict}(\rho_o, \lambda, (\varsigma_\iota, \mathit{in}, \mathit{out})) = \\ &\qquad ((\{\mathrm{A}\} \cup \operatorname{ran} \rho_o \cup (\mathit{top} \mathinner{..} \lambda - 1)) \lhd \varsigma_\iota, \mathit{in}, \mathit{out})\end{aligned}$$

Our new treatment introduces a stack pointer, which in conjunction with the use of a dependent type removes the need for the restrict function. The basic type is `Inj_Env`:

```
Dead_Locn : Locn = 0

Inj_Env   : TYPE =
  [# top : Locn,
     map : {f : [Name -> {l : Locn | l < top}] |
       FORALL (xi1, xi2 : Name) :
         ((f(xi1) = f(xi2) AND NOT f(xi1) = Dead_Locn)
            IMPLIES xi1 = xi2)}
   #]
```

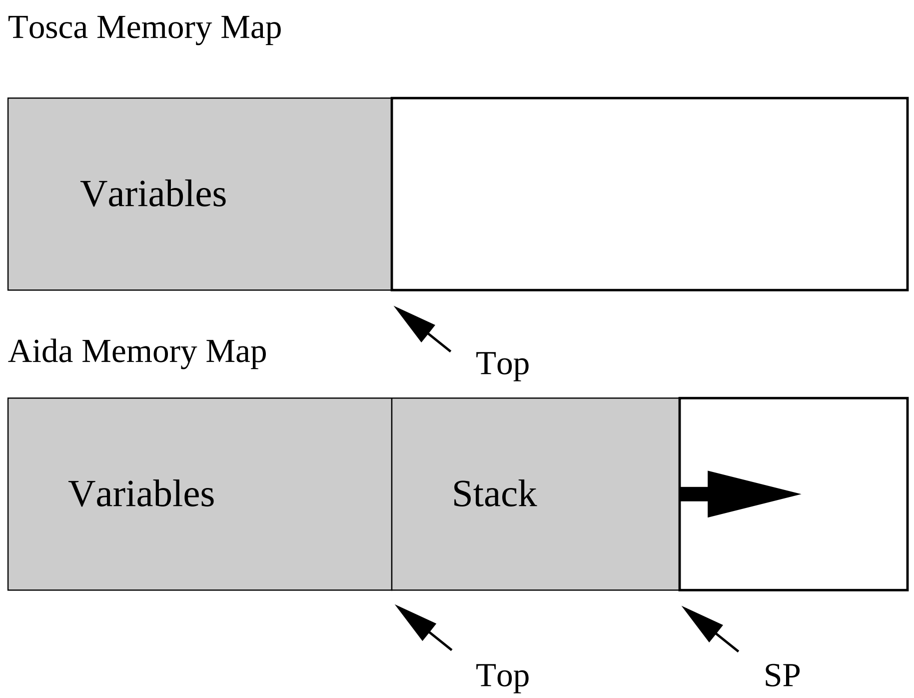

Fig. 1. Tosca and Aida Memory Maps

`Inj_Env` provides the function `map`, which maps variable names to locations. The range of this function is locations dependent on the first parameter in the record, `top`, but its domain includes *all* variable names — unused variables map to an unused location `Dead_Locn`. The quantified expression ensures that the function is injective (modulo the use of `Dead_Locn`).

The `Inj_Env` type is combined in the Aida semantics into a type `Env_Store_I`, which combines the concepts of environment and store into one dependent record type:

```
Env_Store_I : TYPE =
  [# Env : Inj_Env,
     A   : Value,
     SP  : {l : Locn | l >= top(Env)},
     Mem : [{l : Locn | l < SP} -> Value]
   #]
```

'`A`' represents the accumulator, containing a single value. '`SP`' is a stack pointer, which is constrained to be a location greater than or equal to the highest location allocated to a Tosca variable. Thus, the memory (`Mem`) is a mapping from locations to values, where it is only defined for locations below the stack pointer.

Two extra instructions were added to the instruction set of the low level machine to increment (`spinc`) and decrement (`spdec`) the stack pointer, and these were inserted into the operational semantics at appropriate points in expression evaluation. The decrement operation makes use of the PVS function `restrict`[4], which takes a function defined over a type `X` and gives one which has a domain which is a subtype of `X`.

```
SP_INC(sigma_i : Env_Store_I) : Env_Store_I =
  (# Env := Env(sigma_i),
     A   := A(sigma_i),
     SP  := SP(sigma_i)+1,
     Mem := Mem(sigma_i) WITH [(SP(sigma_i)) := Unknown]
   #)

SP_DEC(sigma_i : {e : Env_Store_I |
                  SP(e) - 1 >= top(Env(e))}) : Env_Store_I =
  (# Env := Env(sigma_i),
     A   := A(sigma_i),
     SP  := SP(sigma_i) - 1,
     Mem := restrict(Mem(sigma_i))
   #)
```

This allows us to ensure that when the stack pointer is decremented, the domain of the function representing the memory shrinks appropriately.

Earlier we stated that all declarations in our high-level language are static. Now we have combined the environment and store in the same datatype, the state transition functions in the dynamic semantics could modify the environment, both by extension or by contraction. If a state transition removes mapping between a program variable and its location in memory, then subsequent transitions may fail.

Due to this, PVS emits TCCs during the proof, obliging us to prove that the environment remains static. For example, the last line of this TCC asks us to prove that the environment after executing $rho_i(phi)$ (a continuation) is the same as that in the initial state:

```
% Subtype TCC generated (line 109) for  rho_i(phi)(sigma_i)
  % Proved -- Complete
MI_TCC3: OBLIGATION
(FORALL (phi: Label, gamma: Instr,
        rho_i: Env_I, sigma_i: State_I, vartheta: Cont):
 NOT Halted?(sigma_i) AND NOT Step(sigma_i) = 0 AND gamma = goto(phi)
 IMPLIES Halted?(rho_i(phi)(sigma_i))
 OR Env(StoreOf_I(rho_i(phi)(sigma_i))) = Env(StoreOf_I(sigma_i)));
```

These TCCs are easily proved, but it is also straightforward to avoid their generation at proof time by constraining the type of state transition functions so

[4] This is not related to the Z function `restrict` defined earlier.

that they cannot modify the environment. The following definition is the type of arbitrary state transitions (continuations) in the low level language:

```
Cont : TYPE = [s1 : State_I -> s2 : State_I |
                         (Halted?(s1) IMPLIES Halted?(s2)) AND
                         (Env(StoreOf_I(s1)) = Env(StoreOf_I(s2))) AND
                         (Step(s1) = 0 IMPLIES s1 = s2)]
```

4.4 Halting

The original Z specification made heavy use of partial functions to structure the specification. For example, the dynamic meaning of Tosca's assignment statement (`x := e`) is partial: it says nothing about what state change occurs if, say,

- the lhs-name (`x`) is undeclared
- there are some uninitialised values in the rhs-expression `e`
- the lhs and rhs have different types

Making the dynamic meaning function total to include these checks would clutter the specification, and compromise the understandability of the language definition. So instead, the complete Tosca specification includes several other, static, meaning functions, such as type checking, and declaration-before-use checking, defined in a manner very similar to the dynamic meaning. If an assignment statement passes these checks in the context of some program, then it is guaranteed to be in the domain of the dynamic meaning function.

We have already noted that PVS does not admit partial functions, and as such the direct implementation of the Aida semantics generated many type correctness conditions highlighting the partial areas of the specification. Many of these cases were dealt with by augmenting the original semantics with sensible extensions, but there were several areas which dealt with 'exceptional' behaviour, for example arithmetic overflow/underflow. To deal with these cases, we added a `Halted?` flag to the low level state, which is set True when an error occurs. A similar concept is introduced at the high level, where a flag `Okay?` is introduced into the state. A high level state which is `NOT Okay?` is equivalent to any low level state which is `Halted?`.

4.5 Termination

The largest assumption made in the hand proof is that loops and hence programs terminate, therefore it is a 'partial-correctness' proof. The following definition of the dynamic semantics of loops gives a potentially infinite recursion:

$$\mathcal{M}_C[\![\texttt{loop}\ (\epsilon,\gamma)]\!]\rho\sigma = \\ \quad \texttt{if}\ \mathcal{M}_E[\![\epsilon]\!]\rho\sigma = bool_v\texttt{T} \\ \quad \texttt{then}\ \mathcal{M}_C[\![\texttt{loop}(\epsilon,\gamma)]\!]\rho(\mathcal{M}_C[\![\gamma]\!]\rho\sigma) \\ \quad \texttt{else}\ \sigma$$

The termination assumption is necessary in order to use proof by induction, and if the loop is non-terminating then this induction is not well founded. In [23, Appendix B], a non-recursive definition of the dynamic semantics of loops is given, using an infinite family of functions:

$$\mathcal{W} : \mathbb{N} \to (\mathit{CMD} \times \mathit{EXPR}) \to \mathit{Env} \nrightarrow \mathit{State} \nrightarrow \mathit{State}$$

$$\forall\, n : \mathbb{N};\ \gamma : \mathit{CMD};\ \epsilon : \mathit{EXPR};\ \rho : \mathit{Env};\ \sigma : \mathit{State} \bullet \\ \quad \mathcal{W}_0(\gamma,\epsilon)\rho = \varnothing[\mathit{State} \times \mathit{State}] \\ \quad \wedge\ \mathcal{W}_{n+1}(\gamma,\epsilon)\rho\sigma = \\ \qquad \texttt{if}\ \mathcal{M}_E[\![\epsilon]\!]\rho\sigma = bool_v\texttt{T}\ \texttt{then}\ \mathcal{W}_n(\gamma,\epsilon)\rho(\mathcal{M}_C[\![\gamma]\!]\rho\sigma)\ \texttt{else}\ \sigma$$

$$\mathcal{M}_C[\![\texttt{loop}(\epsilon,\gamma)]\!] = \bigcup\{n : \mathbb{N} \bullet \mathcal{W}_n(\gamma,\epsilon)\}$$

This definition does not easily translate into the logic of the PVS system, so we took a slightly different approach based on a loop counter.

The loop counter is made part of the state at the Tosca and Aida levels, and decrements by one every time a loop body is executed. Thus, we can use induction over the naturals to prove that $\forall\, n : n$ unfoldings of the loop are correctly translated. The dynamic semantics of loops at the Tosca level therefore becomes:

```
MC(gamma : Cmd)(sigma : State) : RECURSIVE State =
  CASES gamma OF
    ...
    loop(epsilon,gamma) :
      IF ME(epsilon)(sigma) = BoolV(TRUE) THEN
        MC(loop(epsilon,gamma))
          (MC(gamma)(sigma) WITH [(Step) := Step(MC(gamma)(sigma)) - 1])
        ELSIF ME(epsilon)(sigma) = BoolV(FALSE) THEN
          sigma
        ELSE
          sigma  WITH [(Okay?) := FALSE]
        ENDIF
     ...
  END CASES
```

This form of definition also allows the recursive function `MC` to be guaranteed to terminate. Non-terminating recursive functions are in essence partial functions, and as such are not permitted in PVS. A termination TCC is generated to ensure that, on each recursive call to a function that its *measure* decreases. A measure function is attached to every recursive function in PVS, and here we use:

```
MEASURE sizeof(gamma) + Step(sigma)
```

`sizeof` is a function we have defined which gives the 'size' of a Tosca command (which is represented as an abstract datatype). This decreases on the recursive call (for evaluating embedded commands) in every case except loop, where the 'embedded' command is not just the loop body, but the *loop itself*, which has the same 'size'. However, using the loop counter, the `Step` decreases on each execution of the loop, hence the overall measure also decreases.

4.6 Remaining Assumptions

There are several axioms in our specification, detailing things that cannot be directly derived from the functional description of the languages and compiler. Three of these relate to the partial nature of the Z specification:

1. Any two Tosca states that are `NOT Okay?` are equivalent.
2. Any two Aida states that are `Halted?` are equivalent.
3. Any Tosca state that is `NOT Okay?` is equivalent to any Aida state that is `Halted?`.

The other remaining assumptions concern the environment Env_I which maps from labels in an Aida program to the continuations which they represent. It is not possible to 'build' this environment as the compiler progresses through the source text, as many label references in `goto` and `jump` instructions are forward references. Also, to build the continuations to which they would refer would require knowledge of the entire target program text, which is not available until after compilation.

Hence, we have generated two axioms (for `loop` and `choice` instructions) which give a representation of how this environment maps into the final program text: This axiomatisation is sufficient for our purposes, but not ideal — it is far too easy to introduce inconsistencies into the specification with the use of axioms.

```
rho_i_loop : LEMMA
  FORALL (gamma : (loop?), rho_i : Env_I, phi : Label, n : nat,
          sigma_i : {s : State_I | NOT Halted?(s)}, sigma : State) :
    (FORALL (vartheta : Cont) :
       rho_i(1 + PROJ_1(OC(c(gamma : (loop?)))(Env(StoreOf_I(sigma_i)))
                         (SP(StoreOf_I(sigma_i)))(phi))) = vartheta
   AND rho_i(PROJ_1(OC(c(gamma : (loop?)))(Env(StoreOf_I(sigma_i)))
                      (SP(StoreOf_I(sigma_i)))(phi))) =
        MI_Star(PROJ_2(OC(gamma)(Env(StoreOf_I(sigma_i)))
                         (SP(StoreOf_I(sigma_i)))(phi)))
               (rho_i)(vartheta))
```

5 Correctness Theorems

The main branches of the correctness theorems follow the hand development quite closely. The exceptions are where our augmentations of the specification come into play, and where the hand proof makes light of some tricky details. The best examples of this are in the proof of the translation of `block` commands and entire programs, where Stepney assumes that these will follow directly (and simply) from earlier lemmas, which is not entirely true.

The effort required to perform the proof with PVS has been large (a person year, with the usual interruptions — considerably more that that to perform the proof by hand). It has been noted by the authors of PVS that this is one of the largest and more complex theorems passed through their system, and the theorems we are required to discharge here are of a very different nature from the theorems that have been specified in PVS previously. We have thus been stretching the limits of the type system, for example in the use of doubly-dependent types and abstract datatypes with subtypes.

We have succeeded in implementing strategies for discharging several of the correctness theorems in a near automatic manner. The next stage of this research is to see how robust these strategies are to changes in the specification, as a result of adding more high-level language statements.

6 Related Work

One of the earlier works in the area of compiler verification is that by Polak [18], who performed a partial correctness proof of a non-optimising compiler for a substantial subset of Pascal, using the Stanford Pascal Verifier [9]. However, it is reported by Young [25] that there are a large collection of unproven assumptions within his formal theory, and several inconsistencies in the axioms.

Computational Logic Inc. have performed verification of a compiler [14] for a simple language Micro-Gypsy using the Boyer-Moore (NQTHM) system [3] as part of their work on a trusted stack of system components [25]. The European ProCoS (Provably Correct Systems) project [2] also attempted use of NQTHM for their work on an Occam compiler, but with little success [24]. Recent work at Kiel [4] and Ulm [7] is using the PVS system with much greater success. A formalisation of denotational semantics is now available within the PVS framework [17].

Several works have been based on the HOL system [8] — verification of an assembler (Curzon) [6] and compilers for a small real-time language (Hale) [11] and a simple imperative language (Joyce) [12]. A notable rigorous by-hand work is that performed at Mitre and North Eastern University [10] for a compiler for Scheme.

7 Conclusions

We have seen how it is possible, with some work, to use PVS to prove a theorem cast in Z. This exercise highlighted some problems with the original proof, and also some problems with using a proof tool in a somewhat 'unnatural' manner.

Often, the *process* of performing a proof is more instructive than getting the yes/no answer out at the end. With a hand proof, that process can deepen the understanding of the original specification structure. It is important that this source of insight not be lost when using a tool. In this case other insights came from using PVS: the way of using it to explore the static semantics, for example.

PVS gives us greater confidence in the result of the compiler via several routes. With its expressive type system, and the requirement to discharge type correctness conditions, we have a method for noting incompleteness and weakness in the specification at a very early stage (which is therefore cheap to rectify). Discharging putative theorems about the specification gives us more confidence, noting hidden assumptions and blatant incorrectness in the specification, but we have noted this to be a very expensive activity, and will require more work on automation before this is a viable industrial proposition for proofs of compiler correctness.

We noted that the standard heuristics and built-in proof strategies of PVS were not well suited to this problem domain, largely due to the very specific ordering in which re-write rules must be fired. We are in the process of developing strategies which automate re-writing in this domain by specific ordering of applications, and controlled eagerness. Also, SRI are implementing more sensible methods for instantiation of universal strength quantifiers, possibly using unification or tableaux procedures, which should succeed in our proofs where the current heuristics fail.

However, it should be remembered that there is more to a specification than just its proof opportunity: the other purposes of the specification must not be compromised in the pursuit of ease-of-proof, lest assurance be lost in other areas such as clarity, or translation to executable form.

7.1 Future Work

The work detailed here is ongoing, both at York and Logica. Since the original small language was specified in [23], further development of this approach has continued at Logica, on a compiler for AWE's high assurance ASP processor. The method has been used on a high level language that extends Tosca by adding more data types (bytes, unsigneds, arrays), functions and procedures, and separate compilation of modules, with a high-integrity linker [15].

At York, we are intending to extend the PVS treatment of the compiler, by augmenting the source language (and possibly target language) with new features including local scope, procedures and functions, an array datatype and separate compilation. Our plan is to manufacture proof strategies such that the augmentation of the source language requires as little manual intervention to re-run the correctness proofs as is possible.

8 Acknowledgments

Our thanks to John Rushby, Jeremy Jacob and the anonymous referees for their useful comments on drafts of this paper. For help with PVS, our thanks to Natarajan Shankar and Sam Owre of SRI International.

References

1. Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice.* BCS Practitioners Series. Prentice Hall International, 1994.
2. Jonathan Bowen, C.A.R. Hoare, Michael R. Hansen, Anders R. Ravn, Hans Rischel, Ernst-Rüdiger Olderog, Michael Schenke, Martin Fränzle, Markus Müller-Olm, Jifeng He, and Zheng Jianping. Provably correct systems - FTRTFT'94 tutorial. In *Proceedings of FTRTFT'94*, number 863 in Lecture Notes in Computer Science. Springer-Verlag, September 1994.
3. Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.
4. Karl-Heinz Buth. Automated code generator verification based on algebraic laws. ProCoS Project Document Kiel KHB 5/1, September 1995.
5. W.J. Cullyer. Implementing safety critical systems: The VIPER microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–25. Kluwer Academic Publishers, 1988.
6. Paul Curzon. A verified vista implementation final report. Technical Report 311, University of Cambridge Computer Laboratory, September 1993.
7. Axel Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formal verification of transformations for peephole optimizations. 1997. These proceedings.
8. Mike Gordon. A proof generating system for higher-order logic. Technical Report 103, University of Cambridge Computer Laboratory, January 1987.
9. Stanford Verification Group. Stanford Pascal verifier user manual. Technical Report 11, Stanford Verification Group, 1979.
10. Joshua D. Guttman, John D. Ramsdell, and Vipin Swarup. The VLISP verified scheme system. *LISP and Symbolic Computation*, 8:33–110, 1995.
11. R.W.S. Hale. Program compilation. In Jonathan Bowen, editor, *Towards Verified Systems*, chapter 6. Elsevier Science Publishers Series on Real-Time Safety Critical Systems, Amsterdam, 1993.
12. Jeffrey J. Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge Computer Laboratory, March 1989.
13. A.C. Leisenring. *Mathematical Logic and Hilbert's ϵ-symbol.* Gordon and Breach Science Publishers, New York, 1969.
14. J. Strother Moore. A mechanically verified language implementation. Technical Report 30, Computational Logic Inc., September 1988.
15. I. T. Nabney and S. Stepney. *High integrity separate compilation.* In preparation.
16. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
17. H. Pfeifer, A. Dold, F. W. v. Henke, and H. Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, Fakultät für Informatik, 1996.

18. Wolfgang Polak. *Compiler Specification and Verification.* Number 124 in Lecture Notes in Computer Science. Springer-Verlag, 1981.
19. J.M. Rushby and D.W.J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, Computer Science Laboratory, SRI International, August 1996.
20. Mark Saaltink. The Z/EVES system. In *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*, number 1212 in Lecture Notes in Computer Science, pages 72–85, Reading, UK, April 1997. Springer-Verlag.
21. J. M. Spivey. *The fUZZ manual.* Computing Science Consultancy, 2 Willow Close, Oxford, UK, 1988.
22. J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.
23. Susan Stepney. *High Integrity Compilation: A Case Study.* Prentice Hall International, 1993.
24. Deborah Weber-Wulff. Proven correct scanning. Procos internal report, August 1992.
25. William D. Young. A verified code generator for a subset of Gypsy. Technical Report 33, Computational Logic Inc., October 1988.