DOI 10.1007/s00165-006-0013-4
BCS © 2006
Formal Aspects of Computing (2007) 19: 191–203

Formal Aspects of Computing

Code-carrying theories

Bart Jacobs, Sjaak Smetsers and Ronny Wichers Schreur

Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. This paper is both a position paper on a particular approach in program correctness, and also a contribution to this area. The approach entails the generation of programs (code) from the executable content of logical theories. This capability already exists within the main theorem provers like Coq, Isabelle and ACL2 and PVS. Here we will focus on issues portraying the use of this methodology, rather than the underlying theory. We illustrate the power of the approach within PVS via two case studies (on unification and compression) that lead to actual running code. We also demonstrate its flexibility by extending the program generation capabilities. This paper fits in a line of ongoing integration of programming and proving.

Keywords: PVS; Functional languages; Proof assistants; Code generation; Unification; Compression

1. Introduction

Program correctness has always been an important topic in computer science. Various techniques have been developed in the past. They all involve a suitable addition of logical statements to executable code. Scalability is one of the main concerns that hamper real impact and progress. The underlying issues dealing with language semantics are non-trivial for realistic languages (such as Java, C or C++). Static techniques (like in [Cop05, HoP04]) can be effective, but only cover logical statements of limited complexity. A more complex logic requires user interaction.

We briefly review two approaches in more detail. In traditional program verification, one uses Hoare logic, possibly in combination with a weakest precondition calculus. Proof construction and verification are intended to be performed during code development. A modern incarnation of this approach is the Java Modeling Language (JML), see [BCC05]. Experience in this area (see for instance [JaP04]) indicates that the expressive power of the logical language (such as JML) quickly becomes a bottleneck. Verifications often require many auxiliary logical definitions and results. Hoare logic was not designed to handle them.

Proof-carrying code (PCC) is a more recent technique, see [Nec97]. The crucial point is to separate proof construction and proof checking—exploiting the fact that construction is generally much harder than checking. Within PCC, proof construction is still completed during code development. The code can then be shipped with the proof, so that any user can check the proof before execution. This approach is intended specifically for mobile code. It requires that checking is relatively easy and thus only works for simple properties (like resource-usage bounds and memory safety).

Here we are interested in the correctness of relatively complex programs. A typical example is compression and decompression, as studied in Sect. 4. In this case, the previously mentioned approaches (traditional Hoare logic and PCC) do not work: the logical complexity required for verification lies beyond these methods. An alternative approach that we wish to explore turns things around. Instead of adding logical statements to existing programs, it starts from an expressive logical language—such as available in modern theorem provers—and formulates both the program and its correctness within the logical setting. Typically one can define algebraic data types (sometimes

Correspondence and offprint requests to: Sjaak Smetsers, E-mail: s.smetsers@cs.ru.nl

even coalgebraic ones) and define recursive functions on them. One then generates a conventional program from the (executable part of the) logical theories. We like to call this the "code-carrying theories" (CCT) approach, as a linguistic variation on "proof-carrying code" (PCC). It is not the same as program extraction from constructive proofs (like in the theorem prover Coq), as will be discussed below. The approach has been around in one form or another for quite some time, see for instance [KMM00, BeN00, PaW93, Mun03], and is often presented as a form of rapid prototyping. The main aim of this paper is to put it more firmly on the agenda, by illustrating its usability potential, especially in the context of the theorem prover PVS. We shall largely ignore the underlying theory, and take the translation itself for granted.

The main advantage of CCT is that it uses one single powerful framework for the development of both the code, and of all the logical machinery (auxiliary definitions and lemmas) needed to prove its correctness. In order to prevent confusion we shall refer to the representation of executable code at this logical level as "abstract code". It is transformed into "conventional code". The transformation is needed because theorem provers do not provide an efficient environment for execution. As stated above, for non-trivial programs the required logical machinery is extensive and goes beyond what other approaches offer.

The conventional program generated is typically a functional one, in a language such as Lisp, ML, Ocaml or Clean. In principle, there is no objection to generating imperative programs, though it requires more work. We shall not focus on that aspect.

We shall have a closer look at two specific incarnations, in Coq and in PVS—but there is also comparable work for Isabelle [BeN00] and ACL2 [KMM00].

1.1. Program development in Coq

Since the early work of Paulin [Pau89], program development and extraction has been an important topic in the line of work surrounding the theorem prover Coq. This has already yielded some impressive results, like the *Compcert* project [Ler06] in which a formally certified C-compiler has been developed. The basic idea is to use the propositions-as-types paradigm to view proofs as terms that contain computational content, which can be extracted. However, what is called "program extraction" in the Coq setting can be done in basically two manners (see also [BeC04]):

1. The **direct** approach, in which an abstract functional program is written directly in the logical language of Coq (much like in ML). Properties about such abstract programs can then be formulated in separate statements and proven. Independently, the abstract program can then be translated into a conventional one, in a relatively straightforward manner.
2. The **propositions-as-types** approach, in which the rich type-based logical language of Coq is used for specification, for instance in order to construct a proof/term inhabiting the proposition/type:

$$\{i{:}\,\mathbb{Z} \mid i \geqslant 0\} \longrightarrow \{p{:}\,\mathbb{Z} \mid p \text{ is } i\text{th prime}\}.$$

   Such a proof/term implicitly contains a program that yields a prime number for each non-negative integer. Extracting such programs from proofs is more advanced (and profoundly studied) than the previously mentioned direct approach. It also has more problematic issues, especially with respect to efficiency (see for instance [CrS03]).

What we have labeled the "code-carrying theories" approach, belongs to the set of direct approaches. It is simple and straightforward. For further exploration we shall use PVS rather than Coq. The main reason is that the constructive nature of Coq (with explicit proof objects) introduces unnecessary complications that only distract from the essentials and make things harder than needed.

1.2. Program development in PVS

Version 2.3 of PVS introduced a relatively unknown feature called "ground evaluation". It allows execution of functional definitions in PVS, after having been translated to Common Lisp. Munoz [Mun03] improved the usability of this feature by combining it with PVS' semantic attachments in his system PVSio. It is a library extension that enriches PVS with standard programming functionalities, such as string and file manipulation, printing, floating point arithmetic, etc. PVSio extends the programming capabilities of PVS: via such semantic attachments, various features of specific programming languages can be incorporated in the PVS language, to

ensure a more complete generation of programs from logical theories. Hence this approach, within PVS, offers a great potential for actual use.

1.3. Programming and proving

One can expect that the willingness to prove correctness of programs increases proportionally to the level of abstraction in which they have been written. In particular, *declarative* (functional or logic) programming languages command programmers to use a programming style that results in programs which are much closer to a formal specification than for instance *imperative* programming. Besides, most theorem provers support the specification of functions in a declarative style. Hence, the step from declarative programming to theorem proving is relatively small. As there is a remarkable, but often unnoticed resemblance between proving and programming (both activities require a similar mixture of creativity, insight, and drive), the integration of theory and program development will stimulate, on the one hand, programmers to specify "executable theories", and on the other hand, theoreticians to write "correct programs".

1.4. Outline

The rest of this paper is organized as follows. We initiate by illustrating the actual use of the CCT approach for two classical algorithms, namely unification (for first order theories) in Sect. 3 and compression in Sect. 4. These two algorithms illustrate two different ways in which to use CCT. Unification is just one small aspect of algebraic specifications. It requires standard theory development, including signatures, algebras, term models (as initial algebras), etc. Hence, unification appears in an approach "from theories to programs". We shall present (de)compression in Sect. 4 as an illustration in the other direction: an algorithm that requires a non-trivial amount of ad hoc theory for its correctness. These different directions illustrate the point that CCT may partially be based on a standard library of theories and algorithms, while partly on ad hoc theories developed for a specific purpose. The first point may constitute a community effort, leading to open source repositories of code-carrying theories, supported by translations to different back-end programming languages. On the side we would like to remark that the actual verification of the (de)compression algorithms is highly non-trivial, and may be seen as a contribution itself. As far as we are aware, a formalized proof has not been given before. Section 5 describes the conventional programs that arise from theory developments in PVS. Unification is translated to a program in the functional language Clean [PlE01], and (de)compression is translated to Lisp. The resulting conventional programs can be run, for instance from the command line, or as a constituent of a larger development. The translation to Clean is ongoing work, which will be described in greater detail elsewhere. Finally, Sect. 6 wraps up with conclusions, contributions, experiences, and directions for further work.

2. PVS

As a short introduction to PVS, we will briefly recall the basics (see also [OSR01a]).

The specification language of PVS is based on classical, typed higher-order logic. Both the use of basic types, like integers, booleans and reals, and compound types (built with type constructors, such as records, tuples, and function types) are permitted in PVS. Moreover, recursive data types can be introduced via algebraic type definitions.

As an example of a user-defined data type, consider the following parameterized definition of a binary tree:

```
BinTree[V : TYPE] : DATATYPE
BEGIN
  leaf   : leaf?
  node(el:V, left, right: BinTree) : node?
END BinTree
```

The data type has two *constructors*, `leaf` and `node`, with which trees can be built. In addition, two *recognizers* `leaf?` and `node?` are defined (observe that in PVS question marks are permitted in identifiers), which can be used as predicates for testing whether a tree object starts with the corresponding constructors. The field names `el`, `left` and `right` can be used as *accessors* to extract these components from a node. However, often for extraction it is

more convenient to use the built-in pattern matching, via CASES expressions. Consider, for example, the following function tree2List that collects all elements of a tree in a list.

```
tree2List(t:BinTree) : RECURSIVE list[V] =
    CASES t OF
        leaf: null,
        node(e,l,r): append(tree2List(l),cons(e,tree2List(r)))
    ENDCASES
MEASURE size(t)
```

The MEASURE specification is mandatory when defining a recursive function, such as tree2List shown above. For, in PVS it is required that all functions are total. This measure is used to show that the function terminates. This is done by generating a proof obligation indicating that the measure strictly decreases at each recursive call. Obviously, the size of the tree fulfills this property. But how can we give an appropriate measure for the function size itself? A solution is that for every data type PVS generates a number of functions and axioms, which can be used freely in any theory that imports the data type. Among them, one is the ordering $\ll$ on trees indicating whether the first argument tree is a subtree of second one. In a measure specification, $\ll$ can be used as follows:

```
size(t:BinTree) : RECURSIVE int =
    CASES t OF
        leaf: 0,
        node(e,l,r): size(l) + size (r) + 1
    ENDCASES
MEASURE t BY <<
```

Predicate subtypes and dependent types can be used to introduce type constraints. For instance, sorted list can be defined as follows:

```
sortList : TYPE = { l : list[V] | ∀ (i,j:below(length(l))): i<j ⇒ nth(i) ⩽ nth(j) }
```

Here, below(n) is an example of a dependent type: the set of all natural numbers smaller than n. Because of this dependency, the list elements indexed by i and j (zero based) are guaranteed to exist. As explained below, operators can be overloaded in PVS. The type checker determines that $\leqslant$ is a binary relation over V.

PVS specifications are organized into parameterized theories that may contain declarations of functions, axioms, theorems, etc. The PVS language provides the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers. Names may be overloaded, including those of the built-in operators such as $<$ and $+$. Unlike most functional languages, the type system of PVS is not polymorphic. However, by using parameterized theories one can simulate polymorphism to some extent. As an example, consider the following theory.

```
FOLD[X,Y : TYPE] : THEORY
    BEGIN
    ifoldr(n:nat)(gen_fun:[below(n) → X], zero:Y, comb:[X,Y → Y]) : RECURSIVE Y =
       IF n = 0
       THEN zero
       ELSE comb(gen_fun(0), ifoldr(n-1)(λ(i:below(n-1)): gen_fun(i+1), zero, comb))
       ENDIF
    MEASURE n

    ifoldr_all : LEMMA
        ∀(n:nat,gen_fun:[below(n) → X], zero:Y, comb:[X,Y → Y]):
            ∀ (p:pred[X],q:pred[Y]):
                (∀ (x:X,y:Y): p(x) ∧ q(y) ⇔ q(comb(x,y))) ⇒
                    (((∀ (i:below(n)) : p(gen_fun(i))) ∧ q(zero))
                            ⇔
                       q(ifoldr(n)(gen_fun,zero,comb)))
    ifoldr_some : LEMMA
        ∀(n:nat,gen_fun:[below(n) → X], zero:Y, comb:[X,Y → Y]):
            ∀ (p:pred[X],q:pred[Y]):
```

```
    (∀ (x:X,y:Y): p(x) ∨ q(y) ⇔ q(comb(x,y))) ⇒
        (((∃ (i:below(n)) : p(gen_fun(i))) ∨ q(zero))
            ⇔
        q(ifoldr(n)(gen_fun,zero,comb)))
END FOLD
```

In essence, the operation `ifoldr` is the same as the standard polymorphic *foldr* operation applied to the list `gen_fun(0)`, `gen_fun(1)`, ..., `gen_fun(n-1)`. Specified in a functional language, this operation would get type:

```
ifoldr : : Int → (Int → a) → b → (a → b → b) → b
```

The polymorphic type variables are "implemented" as the theory parameters `X` and `Y`. Unfortunately, the use of `ifoldr` in another theory will require an import of the `FOLD` theory in which the actual types are explicitly specified, although these types can be derived easily from the context in which the `ifoldr` occurs. Opposing the inconvenience is the elegant possibility to provide the function definition with some basic properties, such as `ifoldr_all` and `ifoldr_some`.

The present paper focusses on specifications, not on proofs. Hence we will not go into the details of the PVS-proof checker; see for example [OSR01b].

3. From theories to programs

Typing is an application area in which theory and practice are closely related. A formal treatment of typing usually consists of a (syntax-directed) derivation system, and a type-inference algorithm on which actual implementations of such a system are based. Correctness proofs, connecting type derivation and type inference, are typically done manually. The same is true for additional properties, such as *subject reduction* relating typing to an operational semantics. In this section we will show that our proposed method enables the development of both theory and implementation in a single framework. This has the advantage that not only the correctness proofs themselves but also the generated type-inference algorithm are correct, contrary to the "manual approach". Indeed, close inspection of type theoretical papers often reveals many relatively small errors. However, our main goal is not to track down these kinds of mistakes. Instead, this section is meant as an illustration of our point of view, that program correctness and program development can go hand in hand.

Working out a fully fledged typing system is beyond the scope of this paper. Here, we restrict ourselves to the core of most polymorphic type-inference algorithms, *Robinson Unification* [Rob65].

We initiate by introducing some basic notions. In order to represent types we introduce a data type for *first order terms*. These are composed of variables and application.

```
Terms[V : TYPE,          %for variables
      F : TYPE,          %for function symbols
      ar : [F → nat]     %for arity of function symbols
     ] : DATATYPE
BEGIN
  vrbl  (vrblindex:V) : vrbl?
  funapp(fun:F, args:[below(ar(fun)) → Terms]) : funapp?
END Terms
```

As usual, each *symbol* in `F` is equipped with an arity that indicates the number of arguments which an application of that symbol is supposed to have. Dependent types offer an elegant way to express this requirement. In fact, this theory is so general that what we call terms, may just as well be called types.

A substitution, and the related domain and range sets *dom?* and *ran?* are defined as follows.

```
Substitution : TYPE = [V → Terms]

dom?(s:Substitution): PRED[V] = {v:V | ¬s(v) = vrbl(v)}
ran?(s:Substitution): PRED[V] = {v:V | ∃(w:V) : dom?(s)(w) ∧ fvs(s(w))(v)}
```

Here, `fvs` is a (recursive) predicate that returns the free variables of a given term.

Substitutions are lifted to types in the usual way, which also enables the definition of substitution composition. We make use of the `reduce` function for the `Terms` data type. This (fold-like) function is internally generated by PVS and can often be used as a substitute for explicit recursion.

```
subst(t:Terms, s:Substitution) : Terms
    = reduce(s,λ(f:F,args:[below(ar(f))→Terms]):funapp(f,args))(t)

o(s2, s1:Substitution) : Substitution = λ(v:V) : subst(s1(v), s2)
```

A substitution s is *idempotent* if $s \circ s = s$. Proving the following two properties is not difficult.

```
idempotent?(s:Substitution) : bool = ((s o s) = s)

idempotent_variables : LEMMA idempotent?(s) ⇔ dom?(s) ∩ ran?(s) = ∅

⩽(s1, s2:Substitution) : bool = ∃ (s3:Substitution) : s2 = (s3 o s1)

substitution_preorder : LEMMA  preorder?[Substitution](⩽)
```

Note that `preorder?` is a predefined PVS predicate. Similarly, we can introduce relatively standard notions such as *instance* and *most general unifier*, thus providing a framework for developing the theory of types.

3.1. Unification

The basis for almost any type-inference system, and also for logical programming, is *unification* or *resolution*: the process of identifying two terms through systematic replacement of variables by other expressions. An actual implementation of this algorithm was first described by Robinson [Rob65]. We have used a slightly different version corresponding to the inference system as presented in [BaS01]. The difference with the original algorithm is that recursion is replaced by iteration. As usual, recursion removal requires the explicit use of a stack which, in this particular case, is implemented as a list of type equalities. Moreover, while solving these equalities the algorithm maintains an accumulator which keeps track of all substitutions that have been performed. To deal with the possibility that unification can fail, the result of the algorithm is not merely a substitution, but rather a *lifted* substitution in which the $\perp$-case indicates failure.

Our claim is that the algorithm gives the *most general unifier*, provided that the initial terms are unifiable. Otherwise, it returns $\perp$. This calls for the following definitions.

```
EqStack : TYPE = list[[Terms, Terms]]

unify(eqs: EqStack,s:Substitution) : RECURSIVE lift[Substitution] =
  CASES eqs OF
    null        : up(s), %lifted substitution
    cons(eq, l) :
        LET  (t1,t2) = eq IN
        IF    t1 = t2
        THEN  unify(l, s)
        ELSIF vrbl?(t1)
        THEN  IF   fvs(t2)(vrblindex(t1))
              THEN bottom
              ELSE LET t_for_v = unit WITH [(vrblindex(t1)):=t2] IN
                        unify(subst(l, t_for_v), t_for_v o s)
              ENDIF
        ELSIF vrbl?(t2)
        THEN  IF   fvs(t1)(vrblindex(t2))
              THEN bottom
              ELSE LET t_for_v = unit WITH [(vrblindex(t2)):=t1] IN
                        unify(subst(l, t_for_v), t_for_v o s)
              ENDIF
        ELSIF fun(t1) = fun(t2)
        THEN  unify((zipFuns(ar(fun(t1)))(args(t1),args(t2))(l)),s)
```

```
        ELSE  bottom
        ENDIF
  ENDCASES
MEASURE stack_measure(eqs)
```

Here, the function `zipFuns` takes two functions as arguments, and pushes the terms given by those two functions pairwise onto the stack. `unit` denotes the identity substitution. These functions are defined by:

```
zipFuns(n:nat)(f1, f2:[below(n) → Terms])(st:EqStack) : EqStack =
  ifoldr(n)(λ(i:below(n)): (f1(i),f2(i)), st, cons)

unit : Substitution = λ(v:V) : vrbl(v)
```

As explained in Sect. 2, a recursive function definition in PVS should always contain a MEASURE specification, which guarantees that the function itself will terminate. The measure we have used in `unify` is based on the observation that each iteration will decrease either the number of distinct variables, say n_V, or the size of the types, say s_T, appearing on the stack. This leads to the following definition in terms of the lexicographic ordering on $\langle n_V, s_T \rangle$, using `lex2` (an ordering on pairs based on `ordinals`) from the PVS prelude.

```
stackeq_measure(se:EqStack) : ordinal = lex2(card(fvs(se)), size(se))
```

Proving that this measure is correct (using ordinal induction) is straightforward. Before stating the main theorem about `unify` we introduce the notion of *unifier* and two auxiliary lemmas from which correctness follows almost directly.

```
unifier?(s:Substitution)(t1,t2:Terms)     : bool = subst(t1,s) = subst(t2,s)
unifier?(s;Substitution)(eqs:EqStack)     : bool = every(unifier?(s))(eqs)
unifier?(s:Substitution)(s1:Substitution): bool = (s = s o s1)

unifier_sound : LEMMA
  ∀ (eqs:EqStack, s1:(idempotent?), s2:Substitution):
      disjoint?(dom?(s1),fvs(eqs)) ∧ unify(eqs, s1) = up(s2) ⇒
          (unifier?(s2)(eqs) ∧ unifier?(s2)(s1))

unifier_complete : LEMMA
  ∀ (eqs:EqStack, s1, s2:Substitution):
      unifier?(s2)(eqs) ∧ unifier?(s2)(s1) ⇒
          LET ls = unify(eqs, s1) IN up?(ls) ∧ down(ls) ⩽ s2
```

The proofs of both lemmas are again done by ordinal induction. The side condition `disjoint?(dom?(s1),fvs(eqs))` used in `unify_sound` appears to be necessary for applying the induction hypothesis in the case that a variable is unified with a term.

The proofs are quite extensive. A mixed top–down/bottom–up approach involving several more or less ad hoc lemmas to structure the overall proof appears to be much more convenient than performing the proofs in a single session. All in all, the proofs are rather straightforward. Interested readers are referred to the PVS files[1].

This brings us to the main property of `unify`:

```
unify_correct : LEMMA
    ∀ (eqs:EqStack, s:Substitution):
        (unify(eqs,unit) = up(s) ⇒ unifier?(s)(eqs))
                        ∧
        (unifier?(s)(eqs) ⇒ LET ls = unify(eqs,unit) IN up?(ls) ∧ down(ls) ⩽ s)
```

As said before, unification can be used to construct a complete type-inference algorithm. The usual procedure is to introduce an appropriate programming language accompanied with a type-derivation system. The latter can directly be formalized in PVS as an inductive predicate. Again, a correctness property that connects the algorithm to the derivation system can be formulated easily. For example, this can be done with Milner's type-inference

[1] All the proofs presented in this paper are stored in a zip archive that can be obtained via `http://www.cs.ru.nl/~sjakie/files/cct06.zip`.

algorithm $\mathcal{W}$ using Isabelle/HOL, as in [NaN99]. In essence, the proof itself is not more difficult than if it was to be done manually. The advantage is that of course in the end, the complete proof is fully (machine) checked.

4. From programs to theories

This section focuses on a version (from [SaR92]) of the classical Lempel–Ziv–Welch (LZW) compression algorithms [ZiL77, Wel84]. The aim is to prove the obvious requirement that decompression after compression yields the identity. At the same time the formalisation (in PVS) should yield an executable program. The verification challenge turns out to be far from trivial. It requires a substantial amount of logical machinery, of which a sketch will be presented below. Thus it forms an appropriate example where the CCT approach of investigating abstract code within a logical setting is most appropriate. Unlike the unification example from the previous section, the logical theory surrounding (de)compression is not developed in advance, independently of the algorithm. On the contrary, it is the algorithm that drives the ad hoc theory development.

4.1. The abstract code

During compression and decompression, suitable trees and tables are created, which are discarded at the end. The main verification task is to establish a suitable relationship between these trees and tables, from which correctness follows.

First, the following constants and types are fixed. Actually, our formalization uses parameters instead of constants, but we shall present the standard values here, in order to be more concrete.

```
codesize   = 4096          code      = {0,1, .., codesize-1}
symbolsize = 256           symbol    = {0,1, .., symbolsize-1}
counter = lift[code]       coderange = {symbolsize, .., codesize-1}
```

A symbol string or code string is then a list of symbols or codes, respectively. The idea of compression is to turn a symbol string into a code string by cleverly associating codes with symbol sub-strings. These associations are built up as the input symbol string is scanned, so we need a counter holding the next free code value. The maximum number of associations is `codesize`. Since the counter may reach this maximum value, we like to model it via a lift, so that the `bottom` value means overflow, and the admissible counter values appear with tag `up`.

Compression makes use of ternary trees, where each node may contain a pair of elements from `symbol` and `coderange`. In PVS this is defined as a datatype `LZWTree`, which is introduced with two constructors `leaf` and `node` in:

```
LZWtree : DATATYPE
BEGIN
  leaf : leaf?
  node(ch : symbol, co : coderange,
       sm : LZWtree, eq : LZWtree, gr : LZWtree) : node?
END LZWtree
```

In principle one can use a single tree to store all symbols and codes. But it turns out to be more efficient to use an array of trees, one for each symbol. The tree at position s: `symbol` contains the codes for all encountered substrings starting with symbol s. Thus we use:

```
LZWtreetable : TYPE = ARRAY [symbol → LZWtree]
inittable    : LZWtreetable = λ(s:symbol) : leaf
```

We shall now present the compression algorithm `lzw` in three steps, starting from the highest level:

```
lzw(i : list[symbol]) : list[code] = encode(up(symbolsize), i, inittable)
```

where `encode` is a recursive function that builds up the output code string via an auxiliary `insert` function:

```
encode(ctr : counter, i : list[symbol], tt : LZWtreetable) RECURSIVE list[code] =
  CASES i OF
    null     : null,
    cons(h,t): IF t = null THEN (: h :) %return singleton list
```

```
                ELSE LET %insert symbol h (as code) in tree tt(h)
                       (c,l,s) = insert(ctr, h, t, tt(h))
                     IN %put resulting code c at head, and continue
                        %encoding with rest of list and updated treetable.
                       cons(c, encode(inc(ctr), l, tt WITH [h:=s]))
                ENDIF
  ENDCASES
MEASURE length(i)

insert(ctr : counter, prev : code, i : (cons?[symbol]), t : LZWtree)
      : RECURSIVE [ code, list[symbol], LZWtree ] =
  CASES t OF
    leaf : IF up?(ctr) %maximum not yet reached
       THEN ( prev, i, node(car(i), down(ctr), leaf, leaf, leaf) )
       ELSE ( prev, i, leaf ) ENDIF,
    node(ch,co,sm,eq,gr) : IF car(i) < ch
      THEN  LET (c,l,s) = insert(ctr, prev, i, sm)
            IN  ( c, l, node(ch,co,s,eq,gr) )
      ELSIF car(i) > ch
      THEN  LET (c,l,s) = insert(ctr, prev, i, gr)
            IN ( c, l, node(ch,co,sm,eq,s) )
      ELSIF cdr(i) = null %end of input reached
      THEN  (co, null, t)
      ELSE  LET (c,l,s) = insert(ctr, co, cdr(i), eq)
            IN ( c, l, node(ch,co,sm,s,gr) )
      ENDIF
   ENDCASES
MEASURE ct BY <<
```

The MEASURE used in `insert` is the standard subterm order on trees, denoted by $<<$. The notation `(cons?[symbol])` in one of the argument types of `insert` is the type of non-empty symbol strings.

The decompression function `wzl` works the other way around, by reconstructing a symbol string from a code string. It also builds up data structures along the way, which are not trees this time but rather two tables, called `pTable` for prefix table and `lsTable` for last symbol table, consisting of mappings from codes to (lower) codes and to symbols, respectively:

```
pTable       : TYPE =  {a : [code → code] | ∀(c:code) : c ⩾ symbolsize ⇒ a(c) < c}
init_pTable  : pTable  = λ(c:code) : 0
lsTable      : TYPE = [code → symbol]
init_lsTable : lsTable = λ(c:code) : 0
```

Associated with these tables there are auxiliary functions for constructing symbol strings and for updating:

```
firstSymbol(co:code, pt:pTable) : RECURSIVE symbol =
  IF   co < symbolsize THEN co
  ELSE firstSymbol(pt(co), pt)
  ENDIF MEASURE co

code2string(co:code, pt:pTable, lst:lsTable) : RECURSIVE (cons?[symbol]) =
  IF   co < symbolsize THEN (: co :)
  ELSE append(code2string(pt(co), pt, lst), (: lst(co) :))
  ENDIF MEASURE co

updateTables(new,next:code, prev:below(next), pt:pTable, lst:lsTable) : [pTable, lsTable] =
  ( pt  WITH [(next) := prev],
    lst WITH [(next) := IF   new = next
                        THEN firstSymbol(prev, pt)
                        ELSE firstSymbol(new, pt) ENDIF] )
```

We are finally in a position to define the decompression algorithm:

```
wzl(i:list[code]): lift[list[symbol]] =
  CASES i OF
    null      : up(null),
    cons(h,t) : %head of the input must be a symbol that appears as
                %head of the output.
                IF h < symbolsize
                THEN CASES decode(up(symbolsize), h, t,
                                  init_pTable, init_lsTable) OF
                       bottom : bottom,
                       up(d)  : up(cons(h, d))
                     ENDCASES
                ELSE bottom
                ENDIF
  ENDCASES
```

where the recursive function decode does the real work:

```
decode(ctr:counter, prev :{c:code | c < ctr}, i : list[code], pt : pTable, lst:lsTable)
      : RECURSIVE lift[list[symbol]] =
  CASES i OF
    null      : up(null),
    cons(h,t) : IF ctr = bottom
      THEN  CASES decode(ctr, h, t, pt, lst) OF
              bottom : bottom,
              up(d)  : up(append(code2string(h, pt, lst), d))
            ENDCASES
      ELSIF h ⩽ down(ctr)
      THEN  LET (npt,nlst) = updateTables(h,down(ctr),prev,pt,lst)
            IN CASES decode(inc(ctr), h, t, npt, nlst) OF
                 bottom : bottom,
                 up(d)  : up(append(code2string(h, npt, nlst), d))
               ENDCASES
      ELSE bottom
      ENDIF
  ENDCASES MEASURE length(i)
```

This completes the description of the abstract compression and decompression code in PVS.

4.2. Verification sketch

For the verification we need to establish that the tree that is created during the compression of a symbol string i, is suitably related to the tables that arise during the decompression of the code string lzw(i). This match relation between trees and tables is formalized as follows.

```
tables_match_tree?(t:LZWtree, p:code, pt:pTable, lst:lsTable) : RECURSIVE bool =
  CASES t OF
    leaf                : TRUE,
    node(ch,co,sm,eq,gr) : pt(co) = p ∧ lst(co) = ch ∧
      tables_match_tree?(sm,p,pt,lst) ∧
      tables_match_tree?(eq,co,pt,lst) ∧
      tables_match_tree?(gr,p,pt,lst)
  ENDCASES MEASURE t BY ≪
```

The code parameter p corresponds to the index of a tree in a tree table. Using the notion of a path in a tree we can give an alternative description of this match relation. This requires some auxiliary definitions. First we define types,

```
direction : TYPE = {L, M, R} %left, middle, right
treepath  : TYPE = list[direction]
```

and then the associated "look-up" functions:

```
path2node(tp:treepath, tr:LZWtree) : RECURSIVE lift[[symbol, coderange]] =
  CASES tr OF
    leaf                : bottom,
    node(ch,co,sm,eq,gr) : CASES tp OF
                             null      : up((ch,co)),
                             cons(h,t) : CASES h OF
                                           L : path2node(t,sm),
                                           M : path2node(t,eq),
                                           R : path2node(t,gr)
                                         ENDCASES
                           ENDCASES
  ENDCASES MEASURE tr BY <<

path2previous(tp:treepath, tr:LZWtree, p:code) : RECURSIVE code =
  CASES tr OF
    leaf                : p,
    node(ch,co,sm,eq,gr) : CASES tp OF
                             null      : p,
                             cons(h,t) : CASES h OF
                                           L : path2previous(t,sm,p),
                                           M : path2previous(t,eq,co),
                                           R : path2previous(t,gr,p)
                                         ENDCASES
                           ENDCASES
  ENDCASES MEASURE tr BY <<
```

We can then prove, for variables `t:LZWtree, p:code, pt:pTable, lst:lsTable`,

```
tables_match_tree?(t, p, pt, lst)
  ⇔
∀(tp:treepath) : up?(path2node(tp,t)) ⇒
  LET node = down(path2node(tp,t)) IN
    pt(proj_2(node))  = path2previous(tp,t,p) ∧ lst(proj_2(node)) = proj_1(node)
```

A basic result is that identical strings are contained in matching trees and tables. This forms the basis for the correctness result. What remains to be established is showing that the match relation is indeed a suitable invariant. Our method of achieving this is rather ad hoc. We define a new coding function `endecode` that combines both encoding and decoding on shared data. A crucial element of the proof is showing that the match relation holds for all iterations of this `endecode` function. This involves many intermediate results, as can seen in the PVS files. We conclude with the main result:

```
wzl_lzw : THEOREM
  ∀(i:list[symbol]) : wzl(lzw(i)) = up(i)
```

5. Running programs

To further test the CCT approach we implemented a prototype of a translator from PVS theories to Clean modules. This section briefly reports the set-up of this prototype.

The translator is not designed to handle the PVS language completely. Rather, we assume that the programmer uses a suitable subset of PVS that can be translated to Clean. The translator should generate readable Clean code. This increases the trust that the Clean code correctly implements the original PVS theory. Where possible the translator preserves typing information. While not all proof information can be represented in Clean, the types

do provide some safeguard that the functions are used correctly. This is especially important in the generated interfaces which form the boundary between those modules that have been proved correct, and the rest of the program.

5.1. Translation details

For the largest part, PVS and Clean are very similar and the translator's job is mostly pretty-printing each PVS language construct in the corresponding Clean syntax. Both languages are purely functional languages with record types, algebraic types, lists, tuples, and so on. The main differences lay in the type systems, the module systems, and the handling of arrays.

The PVS language contains predicate subtypes, which have no counterpart in Clean. The translator omits the subtype predicate. For example, for the definition of `sortList` in Sect. 2 the translator generates the following Clean type synonym definition.

```
: : SortList v :== List v
```

Clean's type checker cannot guarantee that values of type `SortList` are indeed sorted, but for the generated Clean code this property *does* hold, as this was already proved in PVS. For Clean code which was *not* generated from PVS and that uses generated Clean code the property may not be satisfied. In such a situation it is conceivable that the translator generates code to perform run-time checks on the subtype predicates. This has not been implemented in the current prototype.

PVS supports operator overloading in which the same function symbol can represent different functions, depending on the number and types of the function's arguments. In Clean, overloaded functions with same name have to be instances of the same class. These two approaches are not directly compatible. The prototype translator shuns this issue, and assumes that all functions with the same name in PVS form members of a common class in Clean. A more refined translator could disambiguate the function symbols if it is impossible to express them in Clean as instances of the same class, or allow the programmer to choose a suitable mapping.

The translator generates a separate Clean module for each PVS theory. In PVS, theories can be parameterized with types and values. The type parameters appear as polymorphic type variables in the generated declarations. The translator passes the value parameters as additional arguments to the respective generated functions.

The Clean programming language supports arrays with destructive updates. Clean's uniqueness type system ensures that these arrays are used single-threadedly and thus that these updates can be performed safely. It is difficult to exploit destructive arrays directly in the generated Clean code, without essentially doing the same uniqueness analysis in PVS. Instead, we might expect the programmer to write programs with array updates in a monadic style (e.g., see [JoW93]). This could also be the first step towards an entirely imperative version of the program that could be translated to Java or another imperative target language.

In PVS all functions are total, and as such the evaluation order of expressions does not influence the outcome. By default, the evaluation strategy of Clean is lazy, however the evaluation order can be made strict via strictness annotations. This is often more efficient than lazy evaluation. For this reason the current translator adds these strictness annotations where possible.

We have tested the implementation with both the unification and the LZW algorithm with positive results. However, at this stage we are not yet ready to present concrete performance figures.

6. Conclusions

In this paper, we have illustrated how programming and theorem proving can be integrated.

To be useful in program development, a proof assistant should provide a formalization of standard programming libraries. In general, a library is just a collection of functions, commonly characterized by type signatures. For theorem proving a library description should not merely contain an adequate specification of formal properties of each function (for which type signatures are probably insufficient) but also a collection of properties corresponding to combinations of these functions. For example, the PVS prelude contains several operations on lists, like `append`, `length`, and `reverse`, and properties of these operations themselves (like associativity of `append`, or invertibility of `reverse`), but also properties of combinations, such as the fact that the length of the result of appending two lists is the same the sum of both lengths. Clearly, the number of combined properties drastically increases with the number of operations.

To compete with standard program-developing tools in terms of effectiveness, there is a need for a large collection of basic theories, preferably organized in well-structured, easily accessible modules. This not only mandates a huge implementation effort, but also an appropriate infrastructure that enables both development and distribution of newly developed components. It goes without saying that standardization of existing theories is a must. This observation immediately raises the following questions.

- Which theories do we consider as fundamental in a programming context?
- For what kind of application areas do we believe that CCT will be successful?
- How can we set up an appropriate infrastructure for sharing libraries?
- Which source end destination language(s) should we support, and how do we handle language specific properties, like in PVS? For instance, the language Clean supports destructive updates, which, in particular, are very useful if the program contains arrays. However, the semantics of Clean requires that these arrays are used in a single threaded way. How can we enforce this in PVS?

These questions form the basis for further research.

References

[BaS01] Baader F, Snyder W (2001) Unification theory. In: Robinson A, Voronkov A, (eds) Handbook of automated reasoning, vol I, chap 8, Elsevier Amsterdam, pp 445–532

[BeN00] Berghofer S, Nipkow N (2000) Executing higher order logic. In: Callaghan P, Luo Z, McKinna J, (eds) Types for Proofs and Programs, number 2277 in Lect Notes Comp Sci, Springer, Berlin Heidelberg New York, pp 24–40

[BeC04] Bertot Y, Castéran P (2004) Interactive theorem proving and program development. Texts in Theor Comp Sci, Springer, Berlin Heidelberg New York

[BCC05] Burdy L, Cheon Y, Cok D, Ernst M, Kiniry J, Leavens G, Leino K, Poll E (2005) An overview of JML tools and applications. Int J Softw Tools Technol Transf 7(3):212–232

[Cop05] Copeland T (2005) PMD applied. Centennial Books, San Francisco

[CrS03] Cruz-Filipe L, Spitters B (2003) Program extraction from large proof developments. In: Basin D, Wolf B, (eds), Theorem proving in higher order logics, number 2758 in Lect Notes Comp Sci, Springer, Berlin Newyork 205–220

[HoP04] Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not., 39(12):92–106

[JaP04] Jacobs B, Poll E (2004) Java program verification at Nijmegen: developments and perspective. In: Futatsugi K, Mizoguchi F, Yonezaki N, (eds), Software Security — Theories and Systems, number 3233 in Lect Notes Comp Sci, pp 134–153. Springer, Berlin

[JoW93] Jones SLP, Wadler P (1993) Imperative functional programming. In: Conference record of the Twentieth Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, pp 71–84

[KMM00] Kaufmann M, Manolios P, Moore J (2000) Computer-aided reasoning: An Approach. Kluwer, Newyork

[Ler06] Leroy X (2006) Formal certification of a compiler back-end or: programming a compiler with a proof assistant. SIGPLAN Not., 41(1):42–54

[Mun03] Munoz C (2003) Rapid prototyping in PVS. Technical Report NIA Rep. No. 2003-03, NASA National Institute of Aerospace. `http://research.nianet.org/~munoz/PVSio/`

[NaN99] Naraschewski W, Nipkow T (1999) Type inference verified: algorithm W in Isabelle/HOL. J Automated Reasoning, 23(3–4):299–318

[Nec97] Necula GC (1997) Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on principles of programming languages (POPL '97), Paris, pp 106–119

[OSR01a] Owre S, Shankar N, Rushby J, Stringer-Calvert D (2001a) PVS language reference (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA

[OSR01b] Owre S, Shankar N, Rushby J, Stringer-Calvert D (2001b) PVS prover guide (version 2.4). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA

[Pau89] Paulin-Mohring C (1989) Extracting F_ω's programs from proofs in the Calculus of Constructions. In: Principles of programming Languages, ACM Press, pp 89–104 seattle

[PaW93] Paulin-Mohring C, Werner B (1993) Synthesis of ML programs in the system Coq. J Sym Comput, 5–6:607–640

[PlE01] Plasmeijer R, van Eekelen M (2001) Concurrent CLEAN Language Report (version 2.0). `http://www.cs.ru.nl/~clean/`.

[Rob65] Robinson J (1965) A machine-oriented logic based on the resolution principle. J. ACM, 12:23–41

[SaR92] Sanders P, Runciman C (1992) LZW text compression in Haskell. In: Launchbury J, Sansom P, (eds), Proceedings Glasgow Workshop on Functional Programming, Workshops in Computing, Ayr, Scotland. Springer, Berlin Heidelberg Newyork, pp 215–226

[Wel84] Welch TA (1984) A technique for high-performance data compression. IEEE Comput, 17(6):8–19

[ZiL77] Ziv J, Lempel A (1977) A universal algorithm for sequential data compression. IEEE Trans Inform Theory, 23(3):337–343

Received 22 March 2006
Revised 7 August 2006
Accepted 1 September 2006 by C. B. Jones
Published online 25 November 2006