Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof

Wolfgang Goerigk

Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Olshausenstraße 40, D-24098 Kiel, Germany. wg@informatik.uni-kiel.de

Abstract. In this paper[1] we present an exercise in compiler source level verification. Actually, the source and target language and the compiler have already been used in our article for the ACL2 Case Studies book [Goe00a], where we prove that source level correctness is not at all sufficient to prove compiler executables correct. However, the proof is interesting for itself and the fact, that the compiler used in [Goe00a] is indeed proved correct, is essential for the message of that article. So we want to give a more detailed documentation of that proof. The main point is that we use ACL2 to formally and mechanically prove *preservation of partial program correctness*, which is a very practically motivated implementation correctness notion that allows for *trusted* machine program execution even if the source program is not proved to be totally correct. As far as we know, a mechanical proof of preservation of partial correctness has not yet been documented, at least not in ACL2.

1 Introduction

Compilers are sequential transformational programs. A compiler takes a (syntactical representation of) a *source program* $\pi \in \mathbf{SL}$ as input and sometimes it successfully terminates and returns a *target program* $m \in \mathbf{TL}$. If so, we hope that m has something to do with π.

We want m to be a *correct implementation* of π. But this needs further explanation and we will go into detail on the precise meaning of correct implementation later (section 2). A practical compiler, however, will fail in most cases. Actually, it will fail on nearly every source program (with the precise mathematical meaning of *nearly every*). There are usually infinitely many and arbitrarily large source programs, whereas the compiler will run on a finite machine with hard resource restrictions.

Thus, with trusted execution on a real machine in mind, we can not hope to be able to prove that a compiler (executable) will succeed on every proper source program. But we can prove that if it succeeds and returns a machine program m, then m is a correct implementation of π.

The intuition behind this kind of requirement is crucial to many other transformational programs as well, like proof checkers, model or equivalence checkers,

[1] The work reported here has been supported by the Deutsche Forschungsgemeinschaft (DFG) in the *Verifix* and *VerComp* projects on *Correct Compilers* and *Techniques for Compiler Implementation Verification*

batch runs that move our money from one to another account and so on. We accept failures, but we do not want to see incorrect non-erroneous results [Pol81].

Probably most of the readers will feel quite comfortable if we could give a guarantee that a program at most returns correct results. This is in fact what we often depend on. Not more. In our every day experience using programs we observe them to fail with a segmentation fault or bus error, for instance due to lack of memory, a programmer's error, a compiler bug, or a misuse of an optimizing compiler under wrong assumptions. Although very annoying, we all live with software errors, but we all hope that the application programmers, compiler constructors, operating system designers and even hardware engineers have been sensible enough to detect and signal any such runtime error. Undetected errors might have harmful consequences, in particular if they are intentional, in which case we would call them a *virus* or *Trojan Horse* [Goe00a].

As a compiler constructor, we can not relieve the application programmer entirely from the burden to prove his or her application correct. We can not guarantee correctness of implementations for incorrect source programs. Actually we have to construct the compiler without any knowledge about the intended meaning of source programs. On the other hand, a compiler can not preserve every property of the source program. We have to negotiate on a contract between user, language designer and implementor that involves the concrete description of the concrete language to be implemented, the concrete description of the target machine, definitions of both which are sufficiently mathematically exact so that user and implementor can agree on them without misunderstanding.

Once such a contract has been negotiated, however, all bets are off for properties of programs not mentioned in that contract.

> Thesis: We can *trust machine program execution* if we can guarantee partial correctness [Hoa69] for the machine program with sufficient mathematical rigor. In this context, to be *undefined* means not to return a non-erroneous result, *i.e.,* to abort with a runtime error or not to terminate at all.

Our intuitive compiler correctness requirement is to guarantee partial correctness of the generated machine program. But we did not yet say what kind of correctness property we expect for the source program in order to be able to give such a guarantee. This question actually opens up a deep and subtle discussion on source program properties which we want to preserve or need for instance for optimization (see for instance [MOW99]).

However, as long as we do not really depend on sophisticated optimizations, or on the guarantee that termination of the source program is preserved, partial source program correctness is sufficient as well. Hence, our intuitive requirement for a correct compiler program will be, that it *preserves partial program correctness* [GDG$^+$96, MO97, GH98b, Goe00a] (cf. section 2).

This paper is to formally and mechanically prove *preservation of partial correctness* for a compiler from a subset **SL** of ACL2 to the machine code **TL** of an abstract machine. We use M. Kaufmann's and J Moore's ACL2 logic and theorem prover [KM94] to formalize the **SL** semantics (section 3) and the abstract stack machine (section 4). And we mechanically prove a Lisp (**SL**) compiler (section 5) to *preserve partial correctness* (L-simulation [Eng97]) (section 6).

2 Compiler Correctness for Transformational Programs

Before we actually start proving source level correctness for a concrete compiler, let us first give a general definition of *correct implementation* and *preservation of partial correctness* as we understand these notions for (sequential) transformational programs, *i.e.,* for programming languages equipped with a semantics definition that maps programs to input/output relations (or, as in our case, to partial functions mapping input values to result values). This section is to define these notions. Correct implementation indeed preserves partial program correctness (Theorem 3 below).

Informally, we call a (machine) program m a *correct implementation* of a (source) program π, if every non-erroneous result of m is also a possible result of π. The machine may fail, but it will never return an unexpected non-erroneous result. We call this property *Preservation of partial program correctness* [GDG$^+$96, MO96]. In case of non-determinism, it additionally allows the target program to be more deterministic than the source program.

Consequently, we call a compiler, *e.g.,* our compiler `compile-program` from section 5, *correct* or say that it *preserves partial correctness*, if it at most generates correct implementations m of source programs π in the above sense.

Target program execution either returns the correct result, or signals an error, or it returns no result at all. Preservation of partial correctness is very close to the every day programmer's intuition. And it is sufficient to prove full compiler correctness [Goe00b].

Correct Implementation

Let us give precise definitions. For that, let $(\mathbf{SL}, \mathcal{M}_{\mathbf{SL}}(\cdot))$ and $(\mathbf{TL}, \mathcal{N}_{\mathbf{TL}}(\cdot))$ be (transformational) source and target languages, equipped with input domain $D^{in}_{\mathbf{SL}}$ (resp. $D^{in}_{\mathbf{TL}}$), output domain $D^{out}_{\mathbf{SL}}$ (resp. $D^{out}_{\mathbf{TL}}$), and with semantics

$$\begin{aligned} \mathcal{M}_{\mathbf{SL}}(\cdot) : \mathbf{SL} &\longrightarrow (D^{in}_{\mathbf{SL}} \rightharpoonup D^{out}_{\mathbf{SL}}) \quad \text{and} \\ \mathcal{N}_{\mathbf{TL}}(\cdot) : \mathbf{TL} &\longrightarrow (D^{in}_{\mathbf{TL}} \rightharpoonup D^{out}_{\mathbf{TL}}). \end{aligned}$$

Thus, we assume the semantical functionals $\mathcal{M}_{\mathbf{SL}}$ and $\mathcal{N}_{\mathbf{TL}}$ to map programs to input/output *transformations*, *i.e.,* to relations between input and output domains, which in our case will be the s-expressions on source level, and stacks containing s-expressions on top on target level. We use $(A \rightharpoonup B)$ to denote the domain of *binary relations* between A and B, and as usual, ";" denotes relational composition, *i.e.,* if $r_1 \subseteq A \times B$ and $r_2 \subseteq B \times C$ then $r_1 \,;\, r_2 =_{\text{def}} \{ (a,c) \mid \exists\, b \in B \text{ s.t. } (\mathrm{a},\mathrm{b}) \in \mathrm{r}_1 \wedge (\mathrm{b},\mathrm{c}) \in \mathrm{r}_2 \}$

Let $\rho_{in} \in D^{in}_{\mathbf{SL}} \rightharpoonup D^{in}_{\mathbf{TL}}$ and $\rho_{out} \in D^{out}_{\mathbf{SL}} \rightharpoonup D^{out}_{\mathbf{TL}}$ be the corresponding data representation relations. In our case both $\rho_{in} \in D^{in}_{\mathbf{SL}} \rightharpoonup D^{in}_{\mathbf{TL}}$ and $\rho_{out} \in D^{out}_{\mathbf{SL}} \rightharpoonup D^{out}_{\mathbf{TL}}$ map s-expressions (or s-expression tuples) to stacks containing the s-expressions (or s-expression lists) on top (in reverse order).

Definition 1. (Correct Implementation or L-simulation, cf. Figure 1 below). Let π and m denote source and target programs, respectively. Then we say that m *correctly implements* (or *L-simulates*) π, if

$$\rho_{in} \,;\, \mathcal{N}_{\mathbf{TL}}(m) \;\subseteq\; \mathcal{M}_{\mathbf{SL}}(\pi) \,;\, \rho_{out}.$$

That is to say: If the target program m is defined on the representation of a regular source program input, then the source program π can return a corresponding result as well. In our deterministic ACL2 setting we will prove that π returns *the* corresponding result.

$$\begin{tikzcd}
D^{in}_{\mathbf{SL}} \supseteq P \arrow[r, "\mathcal{M}_{\mathbf{SL}}(\pi)\,|_P \;\subseteq Q"] \arrow[d, "\rho_{in}"'] \arrow[dr, phantom, "\circlearrowleft"] & Q\!\downarrow_2 \subseteq D^{out}_{\mathbf{SL}} \arrow[d, "\rho_{out}"] \\
D^{in}_{\mathbf{TL}} \supseteq P_\rho \arrow[r, "\mathcal{N}_{\mathbf{TL}}(m)\,|_{P_\rho} \;\subseteq\; Q_\rho"'] & Q_\rho\!\downarrow_2 \subseteq D^{out}_{\mathbf{TL}}
\end{tikzcd}$$

Fig. 1. Correct implementation or preservation of partial correctness. $Q\!\downarrow_2$ denotes the 2nd projection of Q. This diagram shows the definition of *correct implementation*, *i.e.*, $\rho_{in}\,;\mathcal{N}_{\mathbf{TL}}(m) \;\subseteq\; \mathcal{M}_{\mathbf{SL}}(\pi)\,;\rho_{out}$. Since $\mathcal{M}_{\mathbf{SL}}(\pi)\,|_P \;\subseteq Q$ exactly is partial correctness of π w.r.t. P and Q, it also shows the idea of the proof of "$\Longrightarrow$" in theorem 3 below

Compared to specification refinement in VDM [Jon90], or to former work on compiler verification using ACL2 resp. its predecessor Nqthm ([Moo88, Fla92, Moo96]), there is a subtle difference to our notion of correct compilation: In [Moo96] for instance J Moore proves, that every non-erroneous result of (the Piton machine on) π will also be computed by m (on the FM9001), that m is more defined than π. This allows for optimizations, but trusted execution of m requires total correctness of π. Definedness of the source program is preserved for the machine executable. But note: How could we ever guarantee, that for instance a compiler executable on a finite machine is well-defined for any (arbitrarily large) source program? For source languages with full recursion and/or dynamic data types we need to relax requirements and to allow the machine to fail.

However, strongly speaking the word "relax" above is misleading. The two notions are incompatible. Neither of them implies the other. If source and target program semantics are both defined, however, then there is no difference: both programs then are proved to return corresponding results.

But if the source program is undefined for a given argument, either due to a finite error or because it does not terminate, preservation of partial correctness guarantees, that the machine executable is undefined as well. So the crucial difference is that we allow for executing target programs with a well-placed trust in their results, even if the source program is not totally correct. As a drawback, however, "pure" preservation of partial correctness requires for instance complete runtime error checking.

Preservation of Partial Correctness

If $P \subseteq D^{in}_{\mathbf{SL}}$ (the precondition) and $Q \subseteq D^{in}_{\mathbf{SL}} \times D^{out}_{\mathbf{SL}}$ (the postcondition) are predicates, then P and Q induce predicates $P_{\rho_{in}}$ (P_ρ for short) and $Q_{\rho_{in},\rho_{out}}$

(Q_ρ for short) on the target language input and output domains by

$$\begin{aligned} P_\rho &=_{\text{def}} \rho_{in}(P) && \subseteq D^{in}_{\mathbf{TL}} \\ Q_\rho &=_{\text{def}} \rho_{in}^{-1} \,;\, Q \,;\, \rho_{out} && \subseteq D^{in}_{\mathbf{TL}} \times D^{out}_{\mathbf{TL}}. \end{aligned}$$

P_ρ is the image of P under ρ_{in}, and with $\rho = (\rho_{in}, \rho_{out})$ we can say that Q_ρ is the image of Q under ρ, the set $\{\, (i_2, o_2) \mid (i_1, i_2) \in \rho_{in} \wedge (i_1, o_1) \in Q \wedge (o_1, o_2) \in \rho_{out} \,\}$.

Definition 2. (Preservation of partial correctness). We say that implementing π by m *preserves partial correctness*, if $\{\, P \,\}\, \pi \,\{\, Q \,\}$ implies $\{\, P_\rho \,\}\, m \,\{\, Q_\rho \,\}$ for any precondition P and postcondition Q.

An implementation step $\pi \mapsto m$ preserves partial correctness, if partial correctness of π with respect to P and Q implies partial correctness of m with respect to P_ρ and Q_ρ. Actually, correct implementation (L-simulation) and preservation of partial correctness are the same.

Theorem 3. *(Preservation of partial correctness is correct implementation). An implementation step $\pi \mapsto m$ preserves partial correctness, if and only if m is a correct implementation of π, i.e.,*

$$\begin{aligned} &\forall\, P, Q : (\, \{\, P \,\}\, \pi \,\{\, Q \,\} \implies \{\, P_\rho \,\}\, m \,\{\, Q_\rho \,\}\,) \\ &\qquad\qquad \iff \rho_{in} \,;\, \mathcal{N}_{\mathbf{TL}}\,(m) \;\subseteq\; \mathcal{M}_{\mathbf{SL}}\,(\pi) \,;\, \rho_{out} \end{aligned}$$

Hence, our notion of *correct implementation* exactly characterizes *preservation of partial correctness.* We do not want to prove this theorem here; a proof can be found in [Goe00b], and Figure 1 gives an idea at least of "$\Longrightarrow$", which is the more important direction. Instead we want to make all this concrete in ACL2 now, define source and target language, their operational semantics, a compiler, and prove that it preserves partial correctness (section 6).

3 The Source Language SL

SL is a small subset of ACL2 Lisp, with a few built in Lisp functions and a restricted syntax. It is similar to the language L3 [LS87] of first order mutually recursive functions, but with s-expressions as dynamic (or recursive) data type. It is exactly the source language as defined in [Goe00a]. A program is a list of function definitions, followed by a list of *input* variables and a *main* program expression which may use the input variables.

$$\begin{aligned} p &::= (\,(d_1 \;\ldots\; d_n)\; (x_1 \;\ldots\; x_k)\; e\,) \\ d &::= (\texttt{defun}\; f\; (x_1 \;\ldots\; x_n)\; e\,) \\ e &::= c \mid x \mid (\texttt{if}\;\; e_1\; e_2\; e_3) \mid (f\; e_1 \;\ldots\; e_n) \mid \\ &\qquad (op\;\; e_1 \;\ldots\; e_n) \end{aligned}$$

Expressions e (forms) are either constants c, variables x (symbols not equal to `nil` or `t`), conditional expressions, user defined function or operator calls. Functions and operators have a fixed number of arguments. Operators (op) are 23 of the Lisp standard operators as listed in the definition of function `evlop` below. Operators are restricted to have a fixed number of arguments. So for instance the well-known *factorial* program would read as:

```
(((defun fact (n)
   (if (equal n 0) 1 (* n (fact (1- n))))))
 (n)
 (fact n))
```

We assume that programs are *well-formed*, *i.e.,* that variables are bound, functions are defined, and any function or operator call has the correct number of argument expressions. We prove correct compilation only for well-formed programs. Otherwise the compiler (Section 5) would generate semantically incorrect code, for instance for an expression like `(cons 3)`. A definition of well-formedness is given in the appendix (see section A).

Environments

We use association lists to represent environments. Environment lookup will be a call to the function `assoc`. The function `bind` extends a given environment and associates every variable in the list `vars` of variables with the corresponding value from the list `args`.

```
(defun bind (vars args env)
  (if (endp vars) env
    (cons (cons (car vars) (car args))
          (bind (cdr vars) (cdr args) env))))
```

It subsequently adds pairs `(`x `.` v`)` to the association list `env`. If the number of elements in `vars` and `args` do not agree, some of the values are ignored or some of the variables will be bound to `nil`. Well-formedness of programs and expressions assures, however, that this will never happen. Variable environemnts are lists of the form

$$\rho \texttt{ = env = ((}x_1 \texttt{ . } v_1\texttt{) (}x_2 \texttt{ . } v_2\texttt{) ... (}x_2 \texttt{ . } v_2\texttt{))}$$

Since `assoc` will return the left-most (first) pair that it finds for a given variable, earlier bindings for variables with the same name are hidden. This is the standard and would also be correct if we had block nesting.

SL Semantics

We define an operational semantics for **SL** and also for the machine code **TL**. It is given by an interpreter function and hence operational (cf. [Sto77], chapter 13, page 338).

The semantics (`evaluate`) of an **SL** program is as follows: the top-level expression is evaluated after binding the input variables to some (given) inputs. Functions may not terminate, so the semantics of a program in general is a partial mapping from the inputs to the program result. The ACL2 formalization requires to add a *termination argument*, a natural number `n`, to define `evaluate` as a total ACL2 function making partiality explicit. The functions either return a list containing the value, or `'error`, if evaluation exhausts `n`.

The semantics of a form is defined by the mutually recursive interpreter functions `evl` and `evlist`. It depends on a function environment `genv` mapping function names to parameter lists and a body expression, a local environment `env` mapping free variables to values, and the termination argument `n` which decreases if and only if the body of a user defined function is interpreted. The function `evlop` evaluates operator calls.

```
(defun evlop (op args genv env n)
  (cond
   ((equal op 'CAR) (list (CAR (car args))))
   ...
   ((equal op 'CONS) (list (CONS (car args) (cadr args))))))
```

and so forth for the remaining unary and binary operators `CDR`, `CADR`, `CADDR`, `CADAR`, `CADDAR`, `CADDDR`, `1+`, `1-`, `LEN`, `SYMBOLP`, `CONSP`, `ATOM`, `EQUAL`, `APPEND`, `MEMBER`, `ASSOC`, `+`, `-`, `*`, `LIST1`, and `LIST2` with their standard ACL2 semantics.

```
(mutual-recursion
(defun evl (form genv env n)
  (cond
   ((zp n) 'error)
   ((equal form 'nil) (list nil))
   ((equal form 't) (list t))
   ((symbolp form) (list (cdr (assoc form env))))
   ((atom form) (list form))
   ((equal (car form) 'QUOTE) (list (cadr form)))
   ((equal (car form) 'IF)
    (let ((cond (evl (cadr form) genv env n)))
      (if (defined cond)
          (if (car cond)
              (evl (caddr form) genv env n)
            (evl (cadddr form) genv env n))
        'error)))
   (t (let ((args (evlist (cdr form) genv env n)))
        (if (defined args)
            (if (operatorp (car form))
                (evlop (car form) args genv env n)
              (evl (caddr (assoc (car form) genv))
                   genv
                   (bind (cadr (assoc (car form) genv)) args env)
                   (1- n)))
          'error)))))

(defun evlist (forms genv env n)
  (cond ((zp n) 'error)
        ((endp forms) nil)
        (t (let ((f (evl (car forms) genv env n))
                 (r (evlist (cdr forms) genv env n)))
             (if (and (defined f) (defined r))
                 (cons (car f) r)
               'error)))))
)
```

Note, that – for strictness reasons – the functions `evlop`, `evl`, and `evaluate` either return a (one element) list with their return value(s), or the symbol `error`, *i.e.,* the semantics (result value) of a form or program is *defined* if it is a true list, and *undefined*, if it is `error`.

Also note, that mutually recursive functions have to be declared as such in ACL2, and our semantics is an ACL2 definition, not an **SL** program. The function `construct-genv` below constructs a true association list from the list `defs` of function definitions, mapping the function names to their extended bodies:

```
(defun construct-genv (defs)
  (if (consp defs)
      (cons (cons (cadar defs) (cddar defs)) ;; same as (cdar defs)
            (construct-genv (cdr defs)))
    nil))

(defun evaluate (defs vars main inputs n)
  (evl main (construct-genv defs) (bind vars inputs nil) n))
```

The function `evaluate` takes an **SL** program consisting of the declarations `defs`, the input variable list `vars`, the main expression `main`, and returns the value of `main` after binding `vars` to `inputs` and constructing (`construct-genv`) the function environment (`genv`).

The semantics functions `evl`, `evlist`, and `evaluate` are *strict* in `error` and *partial* due to the *termination argument* `n`. Of course, they are not partial in the sense of ACL2.

4 The Target Machine and Code TL

The target machine is an abstract stack machine. Its configuration consists of a `code` part and a separate state (or memory) `stack`, which is a data stack containing Lisp s-expressions. The `code` is never changed. The machine is exactly as defined in [Goe00a]. However, we repeat the definition here for this article to be self-contained.

The machine has six machine instructions. We can push a constant c onto the stack (`PUSHC` c), push the stack content (the variable) at a particular stack position (`PUSHV` i), pop the n stack elements below the top (`POP` n). There is a subroutine call (`CALL` f), that executes the code associated to a subroutine name within `code`, and the (`OPR` *op*) instruction applies an operator to the topmost (one or two) stack cell(s). Moreover, we have a structured `(IF` *then else*`)` instruction, that removes the top of stack and executes the instruction sequence *else* if the top has been `NIL`, *then* otherwise.

Machine programs (m) are sequences of (mutually recursive) subroutine declarations (d) together with a *main* instruction sequence which is to be executed on an initial `stack` after downloading the list of declarations into `code`.

m ::= `(`d_1 ... d_n `(`ins_1 ... ins_k`))`
d ::= `(defcode` f `(`ins_1 ... ins_k`))`

As an example, the stack code for the *factorial* program above would be the following stack machine program, which is exactly the program generated by the compiler (below):

```
((defcode fac
   ((PUSHV 0)
    (PUSHC 0)
    (OPR EQUAL)
    (IF ((PUSHC 1))
        ((PUSHV 0)
         (PUSHV 1)
         (OPR 1-)
         (CALL fac)
         (OPR *)))
    (POP 1)))
 ((PUSHV 0) (CALL fac) (POP 1))))
```

Machine Semantics

The function `opr` applies operators to the one or two topmost stack cells. For the `stack` we use a list that grows to the left, *i.e.,* we use `cons` to push an item onto the stack, and `nth` or `nthcdr` to read the contents or pop elements. The function `download` downloads the declarations into `code`, which is constructing a true association list from `dcls`. Again we add a *termination argument*, a natural number `n`, in order to force the machine to stop execution after at most `n` subroutine calls.

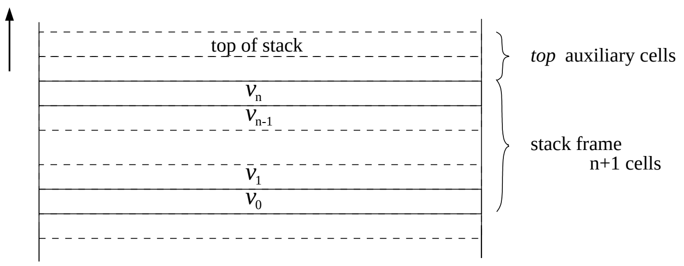
```
(defun opr (op code stack)
  (cond
   ((equal op 'CAR) (cons (MCAR (car stack)) (cdr stack)))
   ...
   ((equal op 'CONS)
    (cons (MCONS (cadr stack) (car stack)) (cddr stack)))))
```

and so forth for the remaining operators (cf. section 3). Note, that the machine operators `MCAR`, `MCONS` etc. are proved to be semantically equivalent to the corresponding ACL2 operators `CAR`, `CONS` etc., but for guard verification reasons they are explicitly defined as total functions (with *guard* `t`) respecting the ACL2 semantics. This is irrelevant for the proof, but necessary to efficiently execute machine programs, so for instance to bootstrap the compiler on the machine (cf. [Goe00a] for details, in particular the supporting ACL2 book `compiler.lisp` of the ACL2 distribution).

```
(mutual-recursion
(defun mstep (form code stack n)
  (cond
   ((or (zp n) (not (true-listp stack))) 'error)
   ((equal (car form) 'PUSHC) (cons (cadr form) stack))
   ((equal (car form) 'PUSHV) (cons (nth (cadr form) stack) stack))
   ((equal (car form) 'CALL)
    (msteps (cdr (assoc (cadr form) code)) code stack (1- n)))
   ((equal (car form) 'OPR) (opr (cadr form) code stack))
   ((equal (car form) 'IF)
    (if (car stack)
          (msteps (cadr form) code (cdr stack) n)
        (msteps (caddr form) code (cdr stack) n)))
     ((equal (car form) 'POP)
      (cons (car stack) (nthcdr (cadr form) (cdr stack))))))

(defun msteps (seq code stack n)
  (cond ((or (zp n) (not (true-listp stack))) 'error)
        ((endp seq) stack)
        (t (msteps (cdr seq) code (mstep (car seq) code stack n) n))))
)

(defun download (dcls)
  (if (consp dcls)
      (cons (cons (cadar dcls) (caddar dcls))
            (download (cdr dcls)))
    nil))

(defun execute (prog stack n)
  (let ((code (download (butlst prog))))
    (msteps (car (last prog)) code stack n)))
```

5 Compiling SL to TL

The compiler generates stack machine code according to the *stack principle*, *i.e.,* arguments are passed on the stack and a given expression e is compiled to a sequence of machine instructions pushing the value of e onto the stack. Functions or operators consume (pop) their arguments and push the result.



In order to execute a function call $(f\ e_0\ \ldots\ e_n)$, we compute the argument forms $e_0\ \ldots\ e_n$ from left to right and push result by result onto the stack. After invoking f and using *top* auxiliary variables, we find the value v_i of the formal parameter x_i at position $(\mathit{top} + n - i)$. Using the current formal parameter list as a compile-time environment, we can find the variable positions and compute their *relative addresses.*

Let `get-stack-frame` be the function which for a given compile time environment *cenv* and a given *top* computes the stack frame for *cenv* with respect to *top*, that is (`get-stack-frame` $(x_0\ \ldots\ x_n)$ *top* *s*) = $(s_{top} \ldots s_{top+n})$. Let `rev` compute the reverse of a list, that is (`rev` $(s_{top} \ldots s_{top+n})$) = $(s_{top+n} \ldots s_{top})$. Then

$$\begin{aligned} \textit{env} &= \texttt{(bind } (x_0 \ldots x_n) \texttt{ (rev (get-stack-frame } (x_0 \ldots x_n) \ \textit{top} \ s\texttt{)))} \\ &= \texttt{((}x_0 \ \texttt{.} \ s_{top+n}\texttt{)} \ \ldots \ \texttt{(}x_n \ \texttt{.} \ s_{top}\texttt{))} \end{aligned}$$

is the environment which binds the values in the stack frame of f to the formal parameters of f. If we access the stack s at position $top + n - i$, that is at the relative address of x_i, we will get the value associated with x_i in *env*. If we push $s_{top+n-i}$ onto the stack s, then afterwards the stack contains the semantics of x_i in *env* on top. In a well-formed program variables are bound and hence any variable will be such an x_i:

Lemma 4. *(Variable access). If* $(x_0 \ \ldots \ x_n)$ *is a compile time environment,* `genv` *a function environment, top a natural number, and* $s \in \mathsf{SExpr}^*$ *a stack of s-expressions, and if env is the above association list representing the local environment* ρ*, then*

$$s_{top+n-i} \cdot s = \texttt{(car (evl } x_i \texttt{ genv } \textit{env} \texttt{ n))} \cdot s$$

whenever `n` *is sufficiently large* (≥ 1) *so that* `(evl` x_i `genv` *env* `n)` *is defined.*

The mechanical proof of this lemma needs some help. We talk about positions of values in the stack and relate them to positions of variables in the environment implicitly. We have to prove some lemmas in order to let the prover know about this correspondence as well. We do not want to go into too much technical detail.

More important is the fact, that we actually proved that the **TL** machine instruction `(PUSHV` $top + |x_i \ \ldots \ x_n| - 1$`)` will push the value associated with x_i in `(bind` $(x_0 \ldots \ x_n)$ `(rev (get-stack-frame` $(x_0 \ldots x_n)$ *top* s`)))` onto the stack, which is the semantics of x_i in this environment.

Lemma 5. *(Constants). For any stack* $s \in \mathsf{Sexpr}^*$ *and any constant* $c \in \mathsf{SExpr}$ *we have (for* `n` ≥ 1*)*

$$c \cdot s = \texttt{(car (evl } c \texttt{ genv } \textit{env} \texttt{ n))} \cdot s$$

This proves that the **TL** machine instruction `(PUSHC` c`)` pushes the semantics of c onto the stack.

Hence, the previous paragraphs characterize the conjecture that we are going to prove for any expression and its corresponding target code instruction sequence (compiled with respect to `cenv` and *top*): If the program and hence the expression e is well-formed (with respect to `genv` and the compile time environment `cenv`), if the machine succeeds in executing the code for the expression, applied to a stack s, and returns a new stack s', then the semantics of e in `genv` and `(bind cenv (rev (get-stack-frame cenv` *top* s`)))` is a defined value v as well and s' is $v \cdot s$. The machine pushes the value of e onto the stack.

We will prove this for the machine and the semantics applied to the same value of the termination argument `n`. If `n` is greater than 0, variable access is always defined for well-formed expressions, and the semantics of constants is as well. For `n = 0` both the machine and the semantics are undefined. Therefore the two previous lemmas prove the induction base case, provided that we compile variables and constants to the instructions mentioned above.

Compiling expressions and programs

Constants are pushed onto the stack using PUSHC. For a variable we push the content of the stack at its relative address using PUSHV. For a function or operator call we subsequently compile the argument forms, thereby incrementing the number *top* of used stack cells, and then generate a CALL or OPR. For a conditional, we compile the condition and then use the machine conditional containing the compiled code for the two alternatives.

For a function definition, we compile the body in a new environment, which is the formal parameter list, say of length n. The stack-frame will be on top initially, so top is zero. The final instruction (POP n) removes the arguments from the stack and leaves the result on top.

The function compile-program has three arguments corresponding to the three parts of an **SL** program, defs, vars, and main. It compiles the definitions in defs and appends the result to the compiled main expression. The final (POP (len vars)) removes the program inputs; the compiled program either returns a stack with the result on top, or an error.

The Compiler

```
(defun operatorp (name)
  (member name '(car cdr cadr caddr cadar caddar cadddr
                 1- 1+ len symbolp consp atom cons equal
                 append member assoc + - * list1 list2)))

(defun compile-forms (forms env top)
  (if (consp forms)
      (append (compile-form (car forms) env top)
              (compile-forms (cdr forms) env (1+ top)))
    nil))

(defun compile-form (form env top)
  (if (equal form 'nil) (list1 '(PUSHC NIL))
  (if (equal form 't) (list1 '(PUSHC T))
  (if (symbolp form)
      (list1 (list2 'PUSHV (+ top (1- (len (member form env))))))
  (if (atom form) (list1 (list2 'PUSHC form))
  (if (equal (car form) 'QUOTE) (list1 (list2 'PUSHC (cadr form)))
  (if (equal (car form) 'IF)
      (append (compile-form (cadr form) env top)
        (list1 (cons 'IF
                 (list2 (compile-form (caddr form) env top)
                        (compile-form (cadddr form) env top)))))
  (if (operatorp (car form))
      (append (compile-forms (cdr form) env top)
              (list1 (list2 'OPR (car form))))
      (append (compile-forms (cdr form) env top)
              (list1 (list2 'CALL (car form))))))))))))

(defun compile-def (def)
    (list1 (cons 'defcode
             (list2 (cadr def)
               (append (compile-form (cadddr def) (caddr def) 0)
                 (list1 (list2 'POP (len (caddr def)))))))))

  (defun compile-defs (defs)
    (if (consp defs)
        (append (compile-def (car defs))
                (compile-defs (cdr defs)))
      nil))

  (defun compile-program (defs vars main)
    (append (compile-defs defs)
            (list1 (append (compile-form main vars 0)
                           (list1 (list2 'POP (len vars)))))))
```

The function `operatorp` identifies operators. The two mutually recursive functions `compile-form` and `compile-forms` compile expressions and expression lists, respectively. `Compile-forms` iterates `compile-form` over `forms`, thereby incrementing the number `top` of used stack cells. The function `compile-def` generates a **TL** subroutine, and `compile-defs` maps `compile-def` over `defs`. Finally, `compile-program` compiles the function definitions and appends them to the compiled main program expression, which additionally pops the input values off the stack.

6 Compiler Source Level Correctness

The previous three sections have been necessary to make this paper self-contained and to formalize everything we need in ACL2, *i.e.,* source language, target language and compiler. Everything was quite common. Note that we defined semantics to be *strict* in `error`. Let us now come back to the main topic of the paper and make the general setting of section 2 concrete in our situation, and formal in ACL2. We will prove that `compile-program` (cf. section 5) is a *correct*

$$\begin{tikzcd}
(s_1,\ldots,s_n) \in \mathsf{SExpr}^* \arrow[r, "(\texttt{evaluate}\ \pi\cdot\texttt{n})"] \arrow[d, "\rho_{tos}"'] \arrow[dr, phantom, "\circlearrowleft"] & \mathsf{SExpr} \ni s \arrow[d, "\rho_{tos}"] \\
(s_n,\ldots,s_1,\ldots) \in \mathsf{Sexpr}^* \arrow[r, "(\texttt{execute}\ m\cdot\texttt{n})"'] & \mathsf{Sexpr}^* \ni (s,\ldots)
\end{tikzcd}$$

Fig. 2. `compile-program` preserves partial correctness.

compiler from **SL** to **TL**. We will prove that it preserves partial correctness with respect to the **SL** semantics `evaluate` (cf. section 3) and to the **TL** semantics `execute` (cf. section 4). The crucial part is to prove correctness of expression compilation (section 7), but we want to start with the main result.

Theorem 6 below states that the diagram in (Figure 2) commutes in the sense of L-simulation, given that the well-formed source program π is compiled to the target program m.

The data representation relation ρ_{tos} maps s-expressions (or s-expression lists) to target machine `stacks` which contain the s-expression (list) on top (in reverse order). ρ_{tos} is a true relation, of course, but `evaluate` and `execute` are partial functions in our case.

Theorem 6.

```
(defthm compiler-correctness-for-programs
 (let ((new-stack (execute
                   (compile-program defs vars main)
                   (append (rev inputs) old-stack) n))
       (value (car (evaluate defs vars main inputs n))))
  (implies
   (and (wellformed-program defs vars main)
        (defined new-stack)
        (true-listp inputs)
        (equal (len vars) (len inputs)))
   (equal new-stack (cons value old-stack)))))
```

The correctness theorem for programs is, after some technical lemmas, a simple consequence of a similar theorem for correct expression compilation, applied to the main expression of the program. If the source program is well-formed, and if the target program (applied to an initial stack containing the correct number of `inputs` in reverse order on top) returns a non-erroneous result on top (is `defined`), then this result is `equal` to the semantics (`evaluate`) of the program applied to the inputs.

Note that this theorem is stronger and hence implies that `compile-program` *preserves partial program correctness*. It additionally proves, that `old-stack` remains unchanged, which is of course a necessary invariant. The proof is mechanically checked and part of the ACL2 distribution [Goe00a]. The crucial part of the proof, however, is to prove correctness of expression compilation.

7 Correctness Proof for Expression Compilation

The correctness proof for the translation from **SL** to **TL** crucially depends on correct translation of expressions. The correctness theorem for programs is a simple consequence, because the semantics of an **SL** program is defined by its `main` expression, and the semantics of a **TL** program executes its main instruction sequence, which is the compiled code for the main expression in a compiled program.

7.1 Compiling Correctness for Expressions

We will prove two theorems simultaneously by induction. The first is the correctness theorem for forms (or expressions), and the second is the theorem for form lists. Both of them are very similar. Informally, for well-formed expressions in well-formed programs we prove:

If the machine, executed on a compiled `form` (list), is defined on a `stack` for an `n`, then the following three conjectures hold:

1. The semantics of the `form` (list) – in the given function environment and with the free variables bound to their values in the current stack-frame – is defined for the same `n`,
2. the machine returns a new stack with the value(s) of the `form`(s) on top (in reverse order), and
3. the stack just below the result value(s) remaines unchanged.

or formally:

Theorem 7. *(Transformation correctness for expressions).*

```
(defthm compiler-correctness-form-forms
  (let ((value
          (evl form
               (construct-genv dcls)
               (bind cenv (rev (get-stack-frame cenv top stack)) env)
               n))
        (new-stack (msteps (compile-form form cenv top)
                           (download (compile-defs dcls)) stack n)))
  (implies
    (and (natp top)
         (wellformed-defs dcls (construct-genv dcls))
         (wellformed-form form (construct-genv dcls) cenv)
         (defined new-stack))
    (and (defined value)
         (equal new-stack (cons (car value) stack))))))
```

That is to say: If `top` is a natural number, if the function declarations of the program and the form are well-formed, and if the machine, applied to the compiled expression after downloading the compiled function definitions of the program is defined on a `stack` and for `n`, then the following two conjectures hold: (1) The source language semantics of `form` in the appropriate function environment and with the actual compile time environment bound to the reverse of the stack-frame with respect to `cenv` and `top` is defined for the same `n`, and (2) the machine returns a new stack which is as the old stack, but with the source code semantics of `form` on top.

The theorem for form lists reads similarly, however we assume the machine to be defined on the compiled code for the entire list of forms and prove (1) that `evlist` is defined on the list of forms, and (2) that the resulting machine stack will have the entire list of values on top in reverse order.

The Proof

The proof is actually a combined computational and structural induction on the termination argument `n` and the structural depth of the expression. In ACL2 terminology this is a well-founded induction on the ordinals. However, since n is a natural number, we stay underneath ω^ω, and thus we can simply say that it is an induction on the (well-founded) lexicographical ordering of pairs (n . k), where n is the termination argument and k is the structural depth. That is to say: We prove the theorem for $n = 0$, where everything is undefined, and for $n > 0$, we use structural induction in any but the function-call case, where we additionally need the induction hypothesis with $n-1$ (computational induction) for the function body.

The induction is suggested by a large admissible ACL2 function which explicitly lists the entire set of induction hypothesises that we need for the proof to succeed. For instance for form lists we have to have the theorem for forms to hold for the first element of the list (the car), and for the rest list the theorem for form lists has to hold for top incremented by 1 and for the stack being the result of executing the machine on the code of the first element of the list.

The boolean flag distinguishes whether we want to have the induction hypothesis for the form theorem (flag = t) or for form lists (flag = nil). This is technical, although necessary because we prove the two theorems simultaneously.

Definition 8. (Combined computational and structural induction).

```
(defun compiler-induction (flag x cenv env top dcls stack n)
  (declare (xargs :measure (cons (1+ (acl2-count n)) (acl2-count x))))
  (if (or (zp n) (atom x)) (list x cenv env top dcls stack n)
    (if flag
        (if (base-form x) (list x cenv env top dcls stack n)
          (if (equal (car x) 'if)
              (list (compiler-induction
                      t (cadr x) cenv env top dcls stack n)
                     (compiler-induction
                      t (caddr x) cenv env top dcls
                      (cdr (msteps (compile-form (cadr x) cenv top)
                                   (download (compile-defs dcls))
                                   stack n))
                      n)
                     (compiler-induction
                      t
                      (cadddr x) cenv env top dcls
                      (cdr (msteps (compile-form (cadr x) cenv top)
                                   (download (compile-defs dcls))
                                   stack n))
                      n))
            (if (operatorp (car x))
                (compiler-induction
                   nil (cdr x) cenv env top dcls stack n)
              (list
                (compiler-induction nil
                 (cdr x) cenv env top dcls stack n)
                (compiler-induction
                 t
                 (get-body (car x) (construct-genv dcls))
```

```
                    (get-vars (car x) (construct-genv dcls))
                    (bind cenv (rev (get-stack-frame cenv top stack)) env)
                    0
                    dcls
                    (msteps (compile-forms (cdr x) cenv top)
                            (download (compile-defs dcls))
                            stack n)
                    (1- n))))))
         (list (compiler-induction  t (car x) cenv env top dcls stack n)
               (compiler-induction
                 nil (cdr x) cenv env (1+ top) dcls
                 (msteps (compile-form (car x) cenv top)
                         (download (compile-defs dcls))
                         stack n)
                 n)))))
```

The argument `x` stands for the form (`flag` = `t`) or form list (`flag` = `nil`), `cenv` and `top` determine the compile time environment, `dcls` contains the source program function declarations, and `env` and `stack` come from source and target program semantics, respectively.

As mentioned before, for function calls we need the induction hypothesises for the list of argument expressions (`flag` = `nil`, structural induction), and for the function body (`flag` = `t`) we assume it for `n - 1` (computational induction), which at the end makes the function `compiler-induction` admissible and hence the induction well-founded.

The induction base case

The induction base case for constants and variable access is essentially the formalization of Lemma 4 and Lemma 5 from section 5. Actually, for the termination argument `n` = 0 the theorem is trivial, because the premise does not hold. The machine is not defined for `n` = 0.

The induction step

For the induction step we need a lot of lemmas. Many of them are very technical. So for instance we have to prove that well-formedness of a form implies well-formedness of the sub-forms, which again requires to prove that syntactical correctness of a form implies syntactical correctness of the sub-forms.

But some of the lemmas are crucial, and we want to show a selection. First, we need, as usual, that the stepwise execution of machine instructions distributes over `append`:

Lemma 9. *(Stepwise execution distributes over instruction sequences).*

```
  (defthm msteps-distributes-over-append
    (equal (msteps (append m1 m2) code stack n)
           (msteps m2 code (msteps m1 code stack n) n)))
```

Then, we need to let the prover know that machine execution is strict:

Lemma 10. *(The machine is strict in* `error`*).*

```
(defthm machine-strictness
  (implies (defined (msteps m2 code (msteps m1 code stack n) n))
           (defined (msteps m1 code stack n))))
```

For the induction step for conditionals we need for instance that the machine code generated for the conditional works as expected. Recall that for a conditional expression the compiler first generates the code for the condition and then a machine conditional containing the code for the two alternatives. The following lemma says that this code actually executes the code for the first alternative after popping the value of the condition off the stack, if this value was not `nil`, and the second alternative otherwise.

Lemma 11. *(The machine conditional works correctly).*

```
(defthm code-for-if-works-correctly
  (implies
   (defined
     (msteps
      (append m1 (cons (list 'if m2 m3) m)) code stack n))
    (equal
     (msteps (cons (list 'if m2 m3) m) code (msteps m1 code stack n) n)
     (if (car (msteps m1 code stack n))
         (msteps (append m2 m) code (cdr (msteps m1 code stack n)) n)
       (msteps (append m3 m) code (cdr (msteps m1 code stack n)) n)))))
```

It is interesting, that we actually have to prove the definedness of the source code semantics and the correctness of the result of machine execution simultaneously as well. The reason is the conditional. The definedness of the conditional inductively depends on the value of the condition. It needs not be strict in both alternatives, and will not be for instance in recursive definitions. The conditional has actually been the challenging case to find this proof, not the function application as the reader might have expected. Function application is just captured by computational induction, whereas the conditional crucially influences the proof structure in the large.

8 Conclusions and Further Work

In this article we use ACL2 to formally and mechanically prove a compiler from a subset **SL** of ACL2 to abstract stack machine code **TL** correct in the sense of *preservation of partial correctness.* The proof is an interesting exercise in proving compiler source level correctness, and it accomplishes the message of [Goe00a], where we prove that source level compiler correctness is at the end not sufficient to guarantee the correctness of compiler executables, even after any additional precaution on source level. The proof is mechanically checked in ACL2, and it is part of the new ACL2 distribution.

In the context of the *Verifix* project on correct compilers [GDG$^+$96, GH98b, GZ99], this proof has been generalized to a mechanical PVS [ORS92] proof of preservation of partial correctness for a larger imperative source language

[DG00], using inductive relations to formalize partial functions. It is interesting, that – despite of some more or less technical changes (some of them are due to the PVS logic) and local adjustments – we can essentially reuse the proof idea and proof structure of the ACL2 proof presented here. In our opinion, this is a good news with respect to proof engineering.

We have to admit that our stack machine is quite abstract and far away from a real processor. In that sense, our paper only presents an exercise. Further work has to be done in order to mechanically verify subsequent compilation phases, so for instance (b) to a stack machine with a linear heap store (data refinement), (c) to linear assembly code, and (d) to real binary machine code of a concrete processor [GH98b].

However, the correctness of step (b) has already been proved manually, and the mechanical proof is nearly completed [Goe00b]. And step (c) and (d) are implemented [GH98b, GH98c, GH98a] and designed to preserve partial correctness. Thus, in addition to being a nice exercise, our proof can be seen as part of a medium to large scale proof effort to provide an initial fully verified compiler executable that preserves partial source program correctness [GH98b].

Acknowledgments

The work presented here is part of the German joint project *Verifix* on *Correct Compilers.* We would like to thank our *Verifix* colleagues and in particular Gerhard Goos, Friedrich von Henke, Hans Langmaack, and Wolf Zimmermann for many fruitful discussions. This paper is a supplement to our ACL2 Case Studies book contribution. The case study itself was originally worked out for a talk that the author gave to a class at the University of Texas. We felt a strong need to add the mechanical compiler correctness proof. But that would not have been possible without further ACL2 background and sometimes even back-up. For that, we would like to say special thanks to J Moore.

References

[DG00] Axel Dold and Wolfgang Goerigk. Proving Preservation of Partial Correctness in PVS. Technical report, University of Ulm, 2000. In preparation.

[Eng97] Kai Engelhardt. *Model-oriented Data Refinement.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1997.

[Fla92] A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation.* PhD thesis, University of Texas at Austin, 1992.

[GDG+96] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.

[GH98a] Wolfgang Goerigk and Ulrich Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU, October 1998.

[GH98b] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 122 – 136, 1998.

[GH98c] Wolfgang Goerigk and Ulrich Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998.

[Goe00a] Wolfgang Goerigk. Compiler Verification Revisited. In Matt Kaufmann, Pete Manolios, and J Strother Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[Goe00b] Wolfgang Goerigk. Trusted Program Execution. Habilitation thesis. Techn. Faculty, Christian-Albrechts-Universität zu Kiel, May 2000.

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201 – 230. Springer Verlag, 1999.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.

[KM94] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition).* John Wiley and Sons, New York, N.Y., 1987.

[MO96] Markus Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, October 1996.

[MO97] Markus Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[Moo88] J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.

[Moo96] J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Kluwer Academic Press, Dordrecht, The Netherlands, 1996.

[MOW99] Markus Müller-Olm and Andreas Wolf. On excusable and inexcusable failures: Towards an adequate notion of translation correctness. In *Proceedings of the 1999 International Symposium on Formal Methods FM'99*, Toulouse, France, 1999. To appear.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.

[Pol81] W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA., London, 1977.

A Well-formed Source Programs

We give the formal ACL2 definition of well-formedness for **SL** source programs. The syntax – we refer to functions `formp`, `form-listp`, and also `definitionp`, `definition-listp`, and `programp` below – is defined by a set of (mutually recursive) predicates on s-expressions and not shown here.

There are typical context conditions such as correct parameter passing that have to be guaranteed for source programs. Otherwise, the compiler will generate semantically incorrect code, for instance for the operator call `(cons 3)`, the code of which would consume an important previous stack entry. The language is not strongly typed, so it is just the number of parameters of operator calls and function calls (with respect to their formal parameter list) which have to conform.

Note that the definitions below are not part of the compiler. We do not check well-formedness, but use it as a pre-condition to the compiler correctness conjectures.

A.1 Well-formed forms, definitions, and programs

The definition of well-formed expressions (`wellformed-form`) refers to a global environment (`genv`), which is an association list mapping the defined function names to their formal parameter lists and bodies. The second argument (`env`) is a compile time environment, a list of bound variables (the list of input variables for the main form and the current formal parameter list for function bodies). A list of forms is well-formed, if every form is (`wellformed-forms`). The following two functions are mutually recursive.

```
(mutual-recursion
(defun wellformed-form (form genv env)
  (and
   (formp form)
   (if (equal form 'nil) t
   (if (equal form 't) t
   (if (symbolp form) (member form env)
   (if (atom form) t
   (if (equal (car form) 'QUOTE) t
   (if (equal (car form) 'IF)
       (wellformed-forms (cdr form) genv env)
   (if (operatorp (car form))
       (or
        (and
          (member (car form)
                  '(CAR CDR CADR CADDR CADAR CADDAR CADDDR 1+ 1-
                    LEN SYMBOLP CONSP ATOM LIST1))
          (consp (cdr form)) (wellformed-form (cadr form) genv env)
          (null (cddr form)))
        (and (member (car form) '(CONS EQUAL APPEND MEMBER ASSOC
                                  + - * LIST2))
          (consp (cdr form)) (wellformed-form (cadr form) genv env)
          (consp (cddr form)) (wellformed-form (caddr form) genv env)
          (null (cdddr form))))
       (and (assoc (car form) genv)
            (equal (len (cdr form))
                   (len (cadr (assoc (car form) genv))))
            (wellformed-forms (cdr form) genv env))))))))))

(defun wellformed-forms (flist genv env)
  (and (form-listp flist)  ;; should not be necessary
       (if (consp flist)
           (and (wellformed-form (car flist) genv env)
                (wellformed-forms (cdr flist) genv env))
         (null flist))))
)
```

A definition is well-formed, if it is syntactically correct, its body is well-formed w.r.t. the global environment and the formal parameter list. A list of definitions is well-formed, if every definition is:

```
(defun wellformed-def (def genv)
  (and (definitionp def)
       (wellformed-form (cadddr def) genv (caddr def))))

(defun wellformed-defs (defs genv)
  (and (definition-listp defs)
       (if (consp defs)
           (and (wellformed-def (car defs) genv)
                (wellformed-defs (cdr defs) genv))
         (null defs))))
```

A program is well-formed, if it is syntactically correct, if the declaration part is well-formed w.r.t `genv` and if the main form is well-formed w.r.t. `genv` and `vars`.

```
(defun wellformed-program (defs vars main)
  (and (definition-listp defs)
       (wellformed-defs defs (construct-genv defs))
       (symbol-listp vars)
       (wellformed-form main (construct-genv defs) vars)))
```