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

Wolfgang Goerigk

Christian-Albrechts-Universität zu Kiel, Germany
wg@informatik.uni-kiel.de
http://www.informatik.uni-kiel.de/~wg/

Outline:

→ Background, Three Steps to Correct Realistic Compilation
→ Source Level Verification is not Sufficient
→ Correct Implementation, Preservation of Partial Correctness
→ Source and Target Language, the Compiler
→ The Correctness Proof in ACL2
→ Conclusions and Further Work
Generate correct executables from correct source programs

- manually
- using unverified compilers

- using verified compilers (trusted compiler executables)

Verifix DFG research group (Karlsruhe, Kiel, Ulm)
for realistic source languages and real target processors
Generate correct executables from correct source programs

- manually
- using unverified compilers
  - without verified compiling specification
    - manually semantically checked [state-of-the-art certification]
    - semantically checked by machine [Pnueli et al., Necula 1998, translation validation]
  - with verified compiling specification
    - manually syntactically checked [Goerigk, Hoffmann 1998]
    - syntactically checked by machine [Traverso et al., 1998]
- using verified compilers (trusted compiler executables)
  
  Verifix DFG research group (Karlsruhe, Kiel, Ulm)
  for realistic source languages and real target processors
Construct and correctly implement compilers and compiler generators

- for realistic imperative and object-oriented source languages
- for real target and host processors
- generating efficient code that compares to unverified compilers
- exploiting mechanical proof support, e.g., by PVS or ACL2
- industrially approved compiler architecture and construction techniques
- proof methodology supplements compiler construction, not vice versa

- exploit runtime result verification
  (a posteriori program or result checking) and
- an initial fully trusted compiler as sound bootstrapping basis
Three Steps Towards Trusted Realistic Compilation

1. **Specification** of a compiling relation $C_{TL}^{SL}$ between abstract source and target languages $SL$ and $TL$, and compiling (specification) verification w.r.t. language semantics $\llbracket \cdot \rrbracket_{SL}$, $\llbracket \cdot \rrbracket_{TL}$ and an appropriate semantics relation $\sigma_{TL}^{SL}$.

2. **Implementation** of a corresponding compiler program $\pi_{SL}$ in high level implementation language $SL$ (close to the specification language), and high level compiler implementation verification w.r.t. $C_{TL}^{SL}$.

3. **Low level implementation** of a corresponding compiler executable $m_{TL}$ written in binary target machine language $TL$, and low level compiler implementation verification w.r.t. $\llbracket \pi_{SL} \rrbracket_{SL}$. 
Three Steps Towards Trusted Realistic Compilation

① **Specification** of a compiling relation $\mathcal{C}_{\text{TL}}^{\text{SL}}$ between abstract source and target languages $\text{SL}$ and $\text{TL}$, and **compiling (specification) verification** w.r.t. language semantics $[\cdot]_{\text{SL}}$, $[\cdot]_{\text{TL}}$ and an appropriate semantics relation $\sigma_{\text{TL}}^{\text{SL}}$. theoretical comp. sc., progr. lang. theory, [McCarthy and Painter 1967], ...

② **Implementation** of a corresponding compiler program $\pi_{\text{SL}}$ in high level implementation language $\text{SL}$ (close to the specification language), and **high level compiler implementation verification** w.r.t. $\mathcal{C}_{\text{TL}}^{\text{SL}}$. [Polak 1981], [Moore 1988, 1996], [Curzon 1994, 1996] software eng., formal methods like VDM, RAISE, CIP, PROSPECTRA, Z, B, ...

③ **Low level implementation** of a corresponding compiler executable $m_{\text{TL}}$ written in binary target machine language $\text{TL}$, and **low level compiler implementation verification** w.r.t. $[\pi_{\text{SL}}]_{\text{SL}}$. virtually nothing, only demands [Chirica and Martin 1986], [Moore 1988]
Towards Trusted Realistic Compilation

1. **Specification** of a compiling relation $C_{SL}^{TL}$ between abstract source and target languages $SL$ and $TL$, and compiling (specification) verification w.r.t. language semantics $⟦·⟧_{SL}$, $⟦·⟧_{TL}$ and an appropriate semantics relation $σ_{SL}^{TL}$.

2. **Implementation** of a corresponding compiler program $π_{SL}$ in high level implementation language $SL$ (close to the specification language), and high level compiler implementation verification w.r.t. $C_{SL}^{TL}$.

3. **Low level implementation** of a corresponding compiler executable $m_{TL}$ written in binary target machine language $TL$, and low level compiler implementation verification w.r.t. $⟦π_{SL}⟧_{SL}$. 
① **Specification** of a compiling relation $C^{SL}_{TL}$ between abstract source and target languages $SL$ and $TL$, and **compiling (specification) verification** w.r.t. language semantics $⟦·⟧_{SL}$, $⟦·⟧_{TL}$ and an appropriate semantics relation $σ^{SL}_{TL}$.

② **Implementation** of a corresponding compiler program $π_{SL}$ in high level implementation language $SL$ (close to the specification language), and **high level compiler implementation verification** w.r.t. $C^{SL}_{TL}$. 
① Specification of a compiling relation $C_{SL}^{TL}$ between abstract source and target languages $SL$ and $TL$, and compiling (specification) verification w.r.t. language semantics $[[\cdot]]_{SL}$, $[[\cdot]]_{TL}$ and an appropriate semantics relation $\sigma_{SL}^{TL}$.

② Implementation of a corresponding compiler program $\pi_{SL}$ in high level implementation language $SL$ (close to the specification language), and high level compiler implementation verification w.r.t. $C_{SL}^{TL}$.

③′ Strong Compiler Bootstrap Test: Compile $\pi_{SL}$ to $m_{TL}$ by a twofold bootstrapping, using an unverified $SL$-compiler $\overline{m}$. Apply $m_{TL}$ to $\pi_{SL}$ and test if $m_{TL}$ reproduces itself.
Semantical relations $\sigma_{TL}^{SL} : \text{Sem}_{SL} \rightarrow \text{Sem}_{TL}$ express notions of **correct implementation**. Here are some wishes:

- handle **non-determinism** of the source program semantics
- handle **resource limitations** of the target machine
- allow for **optimizations** that require well-definedness properties of the source program
- handle **(non-terminating) reactive** programs, e.g., preserve definedness properties of the source program
- allow for full recursion and dynamic data types, e.g. for **transformational** programs like compilers, ...
procedure p ();
    begin int x; x := 42 end;

procedure q ();
    begin int y; print (y) end;

begin p(); q() end.
Specification Refinement (intuitive):

The implementation should at least return every specified result, i.e., it should be at least as defined as the specification.

Preservation of Partial Correctness (intuitive):

The implementation should at most return specified results, i.e., we do not want to see any non-erroneous incorrect result.
Semantical Relations and Correct Implementation

\( \Omega : \) error outcomes, \( A \subseteq \Omega \) acceptable errors, \( U = \Omega \setminus A \) unacceptable (chaotic) errors

\[
\begin{align*}
\begin{array}{c}
\llbracket \pi \rrbracket_{SL} \in \text{Sem}_{SL} : \quad iD^\Omega_{SL} \\
\sigma^\Omega_{TL}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\llbracket m \rrbracket_{TL} \in \text{Sem}_{TL} : \quad iD^\Omega_{TL} \\
\end{array}
\end{align*}
\]

Definition: \( m \) correctly implements \( \pi \) relative to \( A \), iff for any \( d \in iD^\Omega_{SL} \) with

\[
( \llbracket \pi \rrbracket_{SL} ; \sigma^\Omega_{TL} ) (d) \cap U = \emptyset \]

we have

\[
( i\rho^\Omega_{TL} ; \llbracket m \rrbracket_{TL} ) (d) \subseteq ( \llbracket \pi \rrbracket_{SL} ; \sigma^\Omega_{TL} ) (d) \cup A
\]

[Goerigk/Langmaack 2000], [Müller-Olm/Wolf 1999]
Choose $\Omega = \text{def} \{ \perp \}$ and $A = \text{def} \{ \perp \} \implies U = \Omega \setminus A = \emptyset$.

\[
\begin{array}{c}
\left[ \pi \right]_{\text{SL}} \in \text{Sem}_{\text{SL}} : i D_{\text{SL}} \{ \perp \} \xrightarrow{\left[ \pi \right]_{\text{SL}}} o D_{\text{SL}} \{ \perp \}
\\
\sigma_{\text{TL}} \downarrow i \rho_{\text{TL}} \\
\left[ m \right]_{\text{TL}} \in \text{Sem}_{\text{TL}} : i D_{\text{TL}} \{ \perp \} \xrightarrow{\left[ m \right]_{\text{TL}}} o D_{\text{TL}} \{ \perp \}
\end{array}
\]

**Definition:** We say that $m$ *L-simulates* $\pi$ (or that the step $\pi \mapsto m$ preserves partial correctness) iff

\[
(i \rho_{\text{TL}} ; \left[ m \right]_{\text{TL}}) \subseteq (\left[ \pi \right]_{\text{SL}} ; o \rho_{\text{TL}})
\]

Syntax:

\[
p ::= ((d_1 \ldots d_n) (x_1 \ldots x_k) e)
\]
\[
d ::= (\text{defun } f (x_1 \ldots x_n) e)
\]
\[
e ::= c \mid x \mid (\text{if } e_1 e_2 e_3) \mid (f e_1 \ldots e_n) \mid (\text{op } e_1 \ldots e_n)
\]

A Sample Program - Factorial:

\[
(((\text{defun } \text{fac } (n) (\text{if } (= n 0) 1 (* n (\text{fac } (1- n))))))
\]

Operational Semantics (interpreter function):

\[
(\text{defun } \text{evaluate } (\text{defs } \text{vars } \text{main } \text{inputs } n) \ldots)
\]

Semantics of forms (expressions):

\[
(\text{defun } \text{evl } (\text{form } \text{genv } \text{env } n) \ldots) \text{ returns } ([\text{form}]) \text{ or } \text{error}
\]
\[
(\text{defun } \text{evlist } (\text{forms } \text{genv } \text{env } n) \ldots)
\]
Machine Instructions

(PUSHC c) (PUSHV i) (POP n) (IF m₁ m₂) (OPR op) (CALL f)

Operational Semantics (interpreter function):

(defun execute (prog stack n) ...)

Stepwise Execution of Machine Instructions:

(defun mstep (instr code stack n) ...)
(defun msteps (instr-seq code stack n) ...)
We compile expressions according to the stack principle:

The instruction sequence $m$ for the expression $e$ pushes the value $v$ of $e$ onto the stack. Operators and functions consume their arguments.

Variable Access

For any $x_i$ in $(x_0 \ldots x_k)$ we find the value of $x_i$ at position $top + |x_i \ldots x_k| - 1$ on the stack.
Compiling Expressions

\[
\text{top of stack} = s_0
\]

\[
v_k = s_{\text{top}}
v_{k-1} = s_{\text{top}+1}
v_1 = s_{\text{top}+k-1}
v_0 = s_{\text{top}+k}
\]

\{ top auxiliary cells \}

\{ current stack frame \}

\{ k + 1 cells \}

\[\text{compile-form}\ (form, (x_0 \ldots x_k), top) = form'_{\text{top}} =\]

\[ c \mapsto ((\text{PUSHC } c)) \]

\[ x_i \mapsto ((\text{PUSHV } top + |x_i \ldots x_k| - 1)) \]

\[ (\text{if } e_1 e_2 e_3) \mapsto e'_{1,\text{top}} \cdot (\text{IF } e'_{2,\text{top}} e'_{3,\text{top}}) \]

\[ (f e_0 \ldots e_n) \mapsto e'_{0,\text{top}} \cdot \ldots \cdot e'_{n,\text{top}+n} \cdot (\text{CALL } f) \]

\[ (op e_0 \ldots e_n) \mapsto e'_{0,\text{top}} \cdot \ldots \cdot e'_{n,\text{top}+n} \cdot (\text{OPR } op) \]
Lemma 1 (Variable access). For any \( n \geq 1 \), \( (\text{evl } x_i \ \text{genv } \text{env } n) \) is defined and

\[
\begin{align*}
\text{env} & = (\text{bind } (x_0 \ldots x_k)(\text{rev } (\text{get-stack-frame } (x_0 \ldots x_k) \ \text{top } s))) \\
& = ((x_0 \cdot s_{\text{top}+k}) \ldots (x_k \cdot s_{\text{top}}))
\end{align*}
\]
Lemma 2 (Constants). For any \( n \geq 1 \), \( \text{evl } c \text{ genv env n} \) is defined and

\[ c \cdot s = \text{car (evl } c \text{ genv env n)} \cdot s = \text{mstep (PUSHC } c) \ldots s \ n \]

\[ = \text{msteps (compile-form } c (x_0 \ldots x_k) \text{ top) } \ldots s \ n \]
Theorems 1 and 2 (Compiler correctness for forms (form lists))

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).

3. The stack just below the result value(s) remains unchanged.
Theorem 3 (Compiler preserves partial correctness)

(defthm compiler-correctness-for-programs
  (let ((new-stack (execute (compile-program defs vars main)
                              (append (rev inputs) 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 stack))))
Theorem 1 (Compiler correctness for forms)

(defthm compiler-correctness-for-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))))))
Theorem 2 (Compiler correctness for form lists)

(defthm compiler-correctness-for-form-lists
  (let ((values
          (evlist forms
            (construct-genv dcls)
            (bind cenv (rev (get-stack-frame cenv top stack)) env)
            n))
        (new-stack (msteps (compile-forms forms cenv top)
                           (download (compile-defs dcls)) stack n)))
  (implies
   (and (natp top)
        (wellformed-defs dcls (construct-genv dcls))
        (wellformed-forms forms (construct-genv dcls) cenv)
        (defined new-stack))
        (and (defined values)
        (equal new-stack (append (rev values) stack)))))))
Induction on \( n \) and the **structural depth** of forms

```lisp
(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)
    ...
    ;; function call
    (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))))))
  ...)
```
Simultaneous Proof of Theorems 1 and 2

Prove Theorems 1 and 2 simultaneously:

(defmacro theorem-1 (form cenv env top dcls stack n) ...)
(defmacro theorem-2 (forms cenv env top dcls stack n) ...)

(defun prove-theorems (flag)
  (if flag
      (theoem-1 x cenv env top dcls stack n)
      (theoem-2 x cenv env top dcls stack n))
:hints ("Goal"
  :induct (compiler-induction flag x cenv env top dcls stack n)
  ...)"

(defun prove-theoem-2 (flag)
  (theoem-1 x cenv env top dcls stack n)
  :hints ("Goal" :by
  (:instance compiler-correctness-form-forms (flag t)))))
Conclusions

- We seriously and rigorously have to tackle target level implementation verification as well.
- Source level verification and testing or validation alone are not sufficient!
- As it stands, this fact is now mechanically proved in ACL2. [Goerigk 1999, 2000].
- There is a repeatable technique for constructing initial, fully verified compiler implementations from the scratch and for realistic systems implementation languages [Goerigk and Hoffmann 1998, Hoffmann 1998] → a major Goal of Verifix.
- The known gap between high level verification and software integration [Verifix, since 1994, BSI, 1996] can be closed.

Some Future Work

- Formalize further compilation phases, i.e., data refinement, code linearization, machine code generation.
- Prove full compiler correctness formally and mechanically in ACL2 (including target level implementation correctness).