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

How to Construct Correct Executables

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

How to Construct Correct Executables

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

Verifix Goals

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 $\mathcal{C}_{\mathbf{TL}}^{\mathbf{SL}}$ between abstract source and target languages **SL** and **TL**, and compiling (specification) verification w.r.t. language semantics $[\![\cdot]\!]_{\mathbf{SL}}$, $[\![\cdot]\!]_{\mathbf{TL}}$ and an appropriate semantics relation $\sigma_{\mathbf{TL}}^{\mathbf{SL}}$.

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

③ Low level implementation of a corresponding compiler executable $m_{\mathbf{TL}}$ written in binary target machine language **TL**, and low level compiler implementation verification w.r.t. $[\![\,\pi_{\mathbf{SL}}\,]\!]_{\mathbf{SL}}$.

Three Steps Towards Trusted Realistic Compilation

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

② Implementation of a corresponding compiler program $\pi_{\mathbf{SL}}$ in high level implementation language **SL** (close to the specification language), and high level compiler implementation verification w.r.t. $\mathcal{C}_{\mathbf{TL}}^{\mathbf{SL}}$.
[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 $m_{\mathbf{TL}}$ written in binary target machine language **TL**, and low level compiler implementation verification w.r.t. $[\![\,\pi_{\mathbf{SL}}\,]\!]_{\mathbf{SL}}$.
virtually nothing, only demands [Chirica and Martin 1986], [Moore 1988]

Towards Trusted Realistic Compilation

① Specification of a compiling relation $\mathcal{C}^{\mathsf{SL}}_{\mathsf{TL}}$ between abstract source and target languages **SL** and **TL**, and compiling (specification) verification w.r.t. language semantics $[\![\,\cdot\,]\!]_{\mathsf{SL}}$, $[\![\,\cdot\,]\!]_{\mathsf{TL}}$ and an appropriate semantics relation $\sigma^{\mathsf{SL}}_{\mathsf{TL}}$.

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. $\mathcal{C}^{\mathsf{SL}}_{\mathsf{TL}}$.

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. $[\![\,\pi_{\mathsf{SL}}\,]\!]_{\mathsf{SL}}$.

Towards Trusted Realistic Compilation - Reality

① Specification of a compiling relation $\mathcal{C}_{\mathbf{TL}}^{\mathbf{SL}}$ between abstract source and target languages **SL** and **TL**, and compiling (specification) verification w.r.t. language semantics $[\![\,\cdot\,]\!]_{\mathbf{SL}}$, $[\![\,\cdot\,]\!]_{\mathbf{TL}}$ and an appropriate semantics relation $\sigma_{\mathbf{TL}}^{\mathbf{SL}}$.

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

Towards Trusted Realistic Compilation - Reality

① Specification of a compiling relation $\mathcal{C}^{\mathbf{SL}}_{\mathbf{TL}}$ between abstract source and target languages **SL** and **TL**, and compiling (specification) verification w.r.t. language semantics $[\![\cdot]\!]_{\mathbf{SL}}$, $[\![\cdot]\!]_{\mathbf{TL}}$ and an appropriate semantics relation $\sigma^{\mathbf{SL}}_{\mathbf{TL}}$.

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

③′ Strong Compiler Bootstrap Test: Compile $\pi_{\mathbf{SL}}$ to $m_{\mathbf{TL}}$ by a twofold bootstrapping, using an unverified **SL**-compiler $\overline{m}$. Apply $m_{\mathbf{TL}}$ to $\pi_{\mathbf{SL}}$ and test if $m_{\mathbf{TL}}$ reproduces itself.

DEMO

Reasonable Semantical Relations

Semantical relations $\sigma^{\mathsf{SL}}_{\mathsf{TL}} : \mathrm{Sem}_{\mathsf{SL}} \rightharpoonup \mathrm{Sem}_{\mathsf{TL}}$ 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, ...

Why Non-Determinism - An Example

```
procedure p ();
  begin int x; x := 42 end;

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

begin p(); q() end.
```

Refinement vs. Preservation of Partial Correctness

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

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

$$\begin{tikzcd}
[\![\,\pi\,]\!]_{\mathsf{SL}} \in \mathsf{Sem}_{\mathsf{SL}} \arrow[d, "\sigma^{\mathsf{SL}}_{\mathsf{TL}}"'] & : & {}_iD^{\Omega}_{\mathsf{SL}} \arrow[r, "{[\![\,\pi\,]\!]_{\mathsf{SL}}}"] \arrow[d, "{}_i\rho^{\mathsf{SL}}_{\mathsf{TL}}"'] \arrow[dr, phantom, "\sqsupseteq_A"] & {}_oD^{\Omega}_{\mathsf{SL}} \arrow[d, "{}_o\rho^{\mathsf{SL}}_{\mathsf{TL}}"] \\
[\![\,m\,]\!]_{\mathsf{TL}} \in \mathsf{Sem}_{\mathsf{TL}} & : & {}_iD^{\Omega}_{\mathsf{TL}} \arrow[r, "{[\![\,m\,]\!]_{\mathsf{TL}}}"'] & {}_oD^{\Omega}_{\mathsf{TL}}
\end{tikzcd}$$

Definition: m *correctly implements* π *relative to* A, iff for any $d \in {}_iD^{\Omega}_{\mathbf{SL}}$ with $(\,[\![\,\pi\,]\!]_{\mathsf{SL}}\,;\,{}_o\rho^{\mathsf{SL}}_{\mathsf{TL}})\,(d) \cap U = \emptyset$ we have

$$(\,{}_i\rho^{\mathsf{SL}}_{\mathsf{TL}}\,;\,[\![\,m\,]\!]_{\mathsf{TL}}\,)\,(d) \subseteq (\,[\![\,\pi\,]\!]_{\mathsf{SL}}\,;\,{}_o\rho^{\mathsf{SL}}_{\mathsf{TL}})\,(d) \cup A$$

[Goerigk/Langmaack 2000], [Müller-Olm/Wolf 1999]

L-Simulation - Preservation of Partial Correctness

Choose $\Omega =_{\text{def}} \{\bot\}$ and $A =_{\text{def}} \{\bot\}$ $[\Longrightarrow U = \Omega \setminus A = \emptyset\,]$.

$$\begin{tikzcd}
\llbracket \pi \rrbracket_{\mathsf{SL}} \in \mathsf{Sem}_{\mathsf{SL}} \arrow[d, "\sigma^{\mathsf{SL}}_{\mathsf{TL}}"'] & : & {}_iD^{\{\bot\}}_{\mathsf{SL}} \arrow[r, "\llbracket \pi \rrbracket_{\mathsf{SL}}"] \arrow[d, "{}_i\rho^{\mathsf{SL}}_{\mathsf{TL}}"'] \arrow[dr, phantom, "\subseteq"] & {}_oD^{\{\bot\}}_{\mathsf{SL}} \arrow[d, "{}_o\rho^{\mathsf{SL}}_{\mathsf{TL}}"] \\
\llbracket m \rrbracket_{\mathsf{TL}} \in \mathsf{Sem}_{\mathsf{TL}} & : & {}_iD^{\{\bot\}}_{\mathsf{TL}} \arrow[r, "\llbracket m \rrbracket_{\mathsf{TL}}"'] & {}_oD^{\{\bot\}}_{\mathsf{TL}}
\end{tikzcd}$$

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

$$(\,{}_i\rho^{\mathsf{SL}}_{\mathsf{TL}}\,;\,\llbracket m \rrbracket_{\mathsf{TL}}\,) \subseteq (\,\llbracket \pi \rrbracket_{\mathsf{SL}}\,;\,{}_o\rho^{\mathsf{SL}}_{\mathsf{TL}}\,)$$

[Goerigk et al. 1996], [Müller-Olm 1996]

Syntax:

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

A Sample Program - Factorial:

```
(((defun fac (n) (if (= n 0) 1 (* n (fac (1- n))))))
 (n)
 (fac n))
```

Operational Semantics (interpreter function):

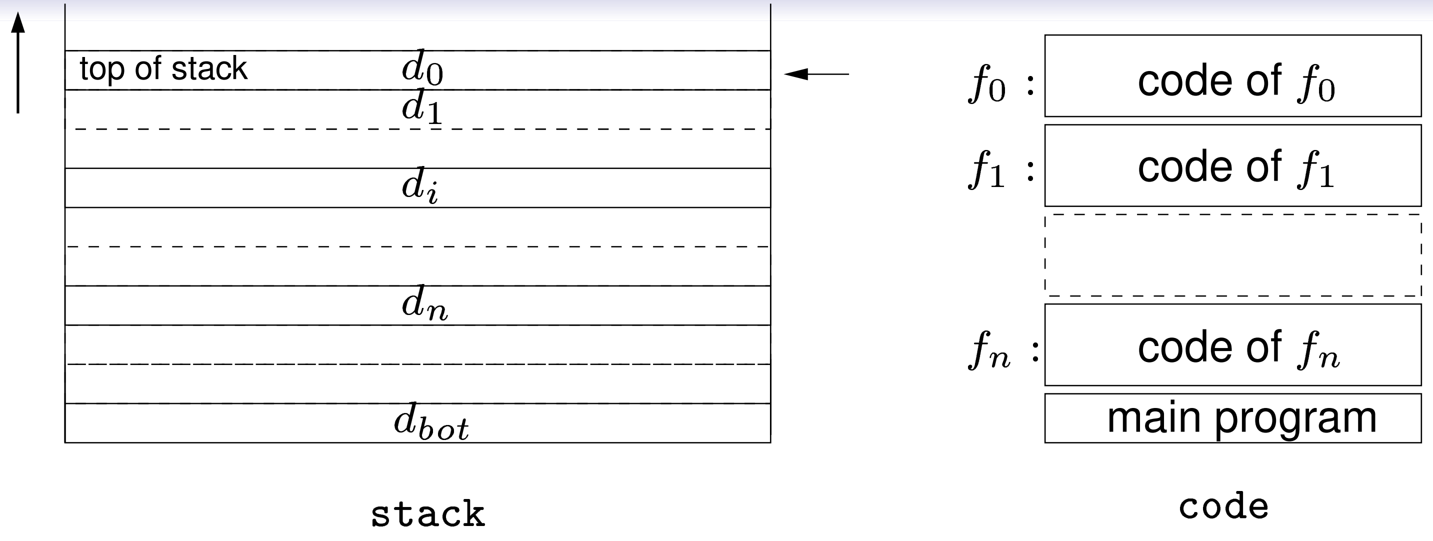
```
(defun evaluate (defs vars main inputs n) ...)
```

Semantics of forms (expressions):

(defun evl (form genv env n) ...)   returns ($[\![$form$]\!]$) or error

```
(defun evlist (forms genv env n) ...)
```

The Target Machine and Code



Machine Instructions

(PUSHC c)   (PUSHV i)   (POP n)   (IF m_1 m_2)   (OPR op)   (CALL f)

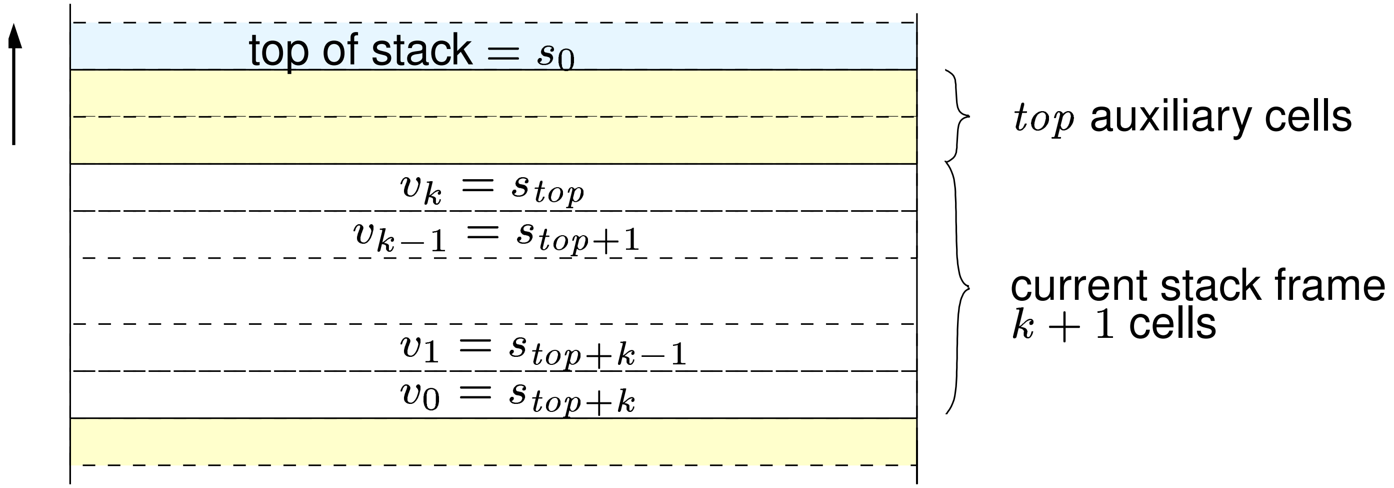
Operational Semantics (interpreter function):

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

Stepwise Execution of Machine Instructions:

```
(defun mstep (instr code stack n) ... )
(defun msteps (instr-seq code stack n) ... )
```

Compiling Programs


top of stack $= s_0$

top auxiliary cells

$v_k = s_{top}$

$v_{k-1} = s_{top+1}$

$v_1 = s_{top+k-1}$

$v_0 = s_{top+k}$

current stack frame
$k + 1$ cells

`stack`

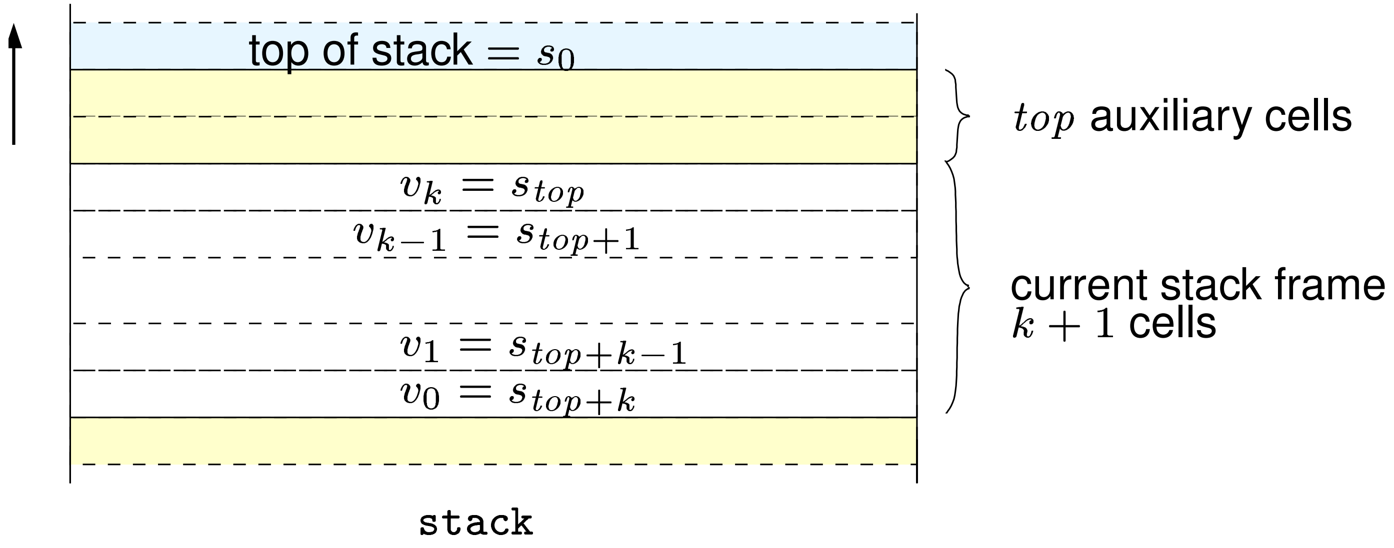
We compile expressions according to the stack principle:

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 `...` x_k`)` we find the value of x_i at position $top + |x_i \ \ldots \ x_k| - 1$ on the stack.

Compiling Expressions



top of stack $= s_0$

top auxiliary cells

$v_k = s_{top}$

$v_{k-1} = s_{top+1}$

$v_1 = s_{top+k-1}$

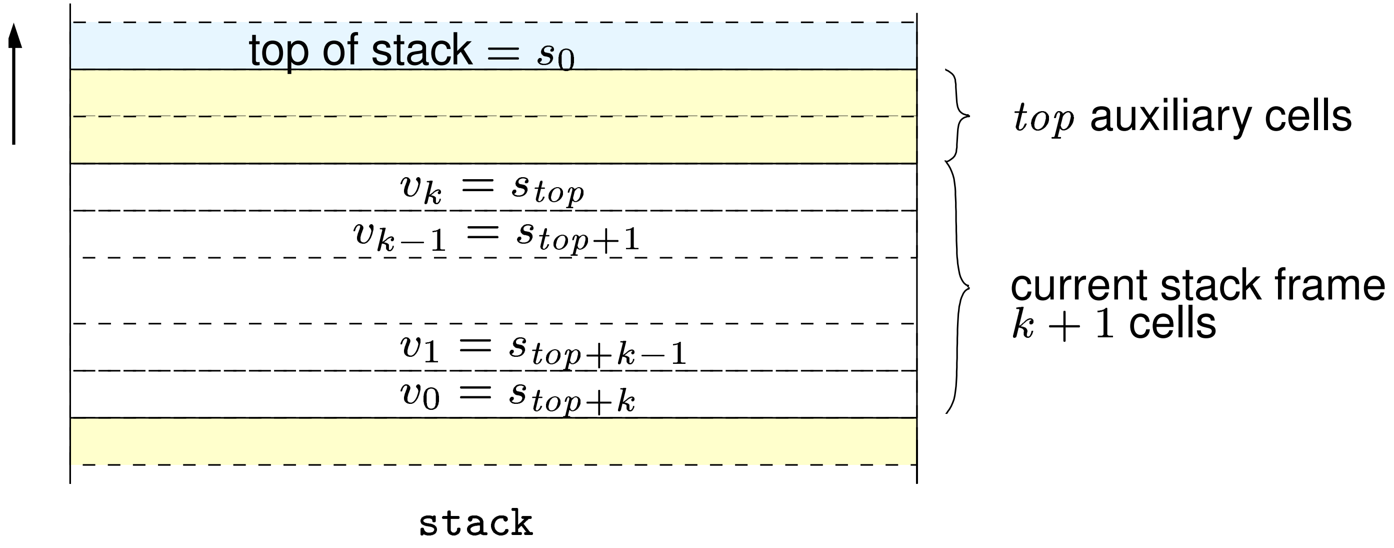
$v_0 = s_{top+k}$

current stack frame
$k+1$ cells

`stack`

$$\begin{array}{rcl}
\mathit{compile\text{–}form}\ (\mathit{form}, (x_0\ \ldots\ x_k), \mathit{top}) & = & \mathit{form}'_{top}\ = \\
c & \mapsto & ((\texttt{PUSHC c})) \\
x_i & \mapsto & ((\texttt{PUSHV}\ \mathit{top} + |x_i\ \ldots\ x_k| - 1)) \\
(\texttt{if}\ e_1\ e_2\ e_3) & \mapsto & e'_{1,top} \cdot (\texttt{IF}\ e'_{2,top}\ e'_{3,top}) \\
(f\ e_0\ \ldots\ e_n) & \mapsto & e'_{0,top} \cdot \ldots \cdot e'_{n,top+n} \cdot (\texttt{CALL}\ f) \\
(op\ e_0\ \ldots\ e_n) & \mapsto & e'_{0,top} \cdot \ldots \cdot e'_{n,top+n} \cdot (\texttt{OPR}\ op)
\end{array}$$

Compiling Expressions - Variable Access



$$\texttt{env} = (\texttt{bind}\ (x_0 \ldots x_k)\ (\texttt{rev}\ (\texttt{get-stack-frame}\ (x_0\ \ldots\ x_k)\ top\ s)))$$

$$= ((x_0\ .\ s_{top+k})\ \ldots\ (x_k\ .\ s_{top}))$$

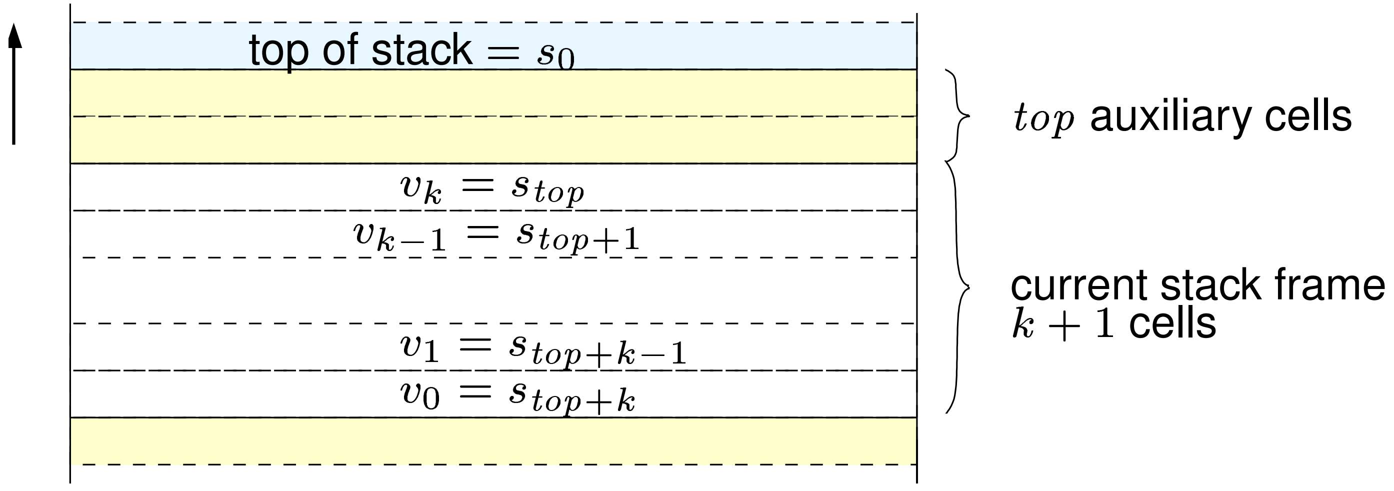
Lemma 1 (Variable access). For any $\texttt{n} \geq 1$, $(\texttt{evl}\ \ x_i\ \ \texttt{genv}\ \ \texttt{env}\ \ \texttt{n})$ is defined and

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

$$= (\texttt{mstep}\ \ (\texttt{PUSHV}\ \ top + |x_i\ \ldots\ x_k| - 1)\ \ \ldots\ \ s\ \ \texttt{n})$$

$$= (\texttt{msteps}\ \ (\texttt{compile-form}\ \ x_i\ \ (x_0\ \ldots\ x_k)\ \ top)\ \ \ldots\ \ s\ \ \texttt{n})$$

Compiling Expressions - Constants



`stack`

$$\begin{aligned} \texttt{env} &= (\texttt{bind}\ (x_0 \ldots x_k)\ (\texttt{rev}\ (\texttt{get–stack–frame}\ (x_0\ \ldots\ x_k)\ top\ s))) \\ &= ((x_0\ .\ s_{top+k})\ \ldots\ (x_k\ .\ s_{top})) \end{aligned}$$

Lemma 2 (Constants). For any $\texttt{n} \geq 1$, `(evl` c `genv env n)` is defined and

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

$$= \texttt{(mstep (PUSHC } c\texttt{) ... } s \texttt{ n)}$$

$$= \texttt{(msteps (compile-form } c\ (x_0 \ldots x_k)\ top\texttt{) ... } s \texttt{ n)}$$

Correct Compilation of Expression Lists - Informally

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.

Preservation of Partial Correctness for SL Programs

$$\begin{tikzcd}
(s_1 \; \ldots \; s_n) \arrow[d, dotted, no head] & \text{SExpr}^* \arrow[r, "(\texttt{evaluate}\; p \cdot \texttt{n})"] \arrow[d, "\rho_{tos}"'] \arrow[dr, phantom, "\subseteq"] & \text{SExpr} \ni \arrow[d, "\rho_{tos}"] & (s) \arrow[d, dotted, no head] \\
(s_n \; \ldots \; s_1 \; \ldots) & \text{Sexpr}^* \arrow[r, "(\texttt{execute}\; m \cdot \texttt{n})"'] & \text{Sexpr}^* \ni & (s \; \ldots)
\end{tikzcd}$$

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)))))
```

Correct Compilation of Expressions - Formally

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))))))
```

Correct Compilation of Expression Lists - Formally

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))))))
```

Combined Computational and Structural Induction

Induction on n and the structural depth of forms

```
(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) ...)
```

```
(defthm compiler-correctness-form-forms
  (if flag
      (theorem-1 x cenv env top dcls stack n)
    (theorem-2 x cenv env top dcls stack n))
  :hints (("Goal"
           :induct (compiler-induction flag x cenv env top dcls stack n)
           ...)))

(defthm compiler-correctness-for-expressions
  (theorem-1 x cenv env top dcls stack n)
  :hints (("Goal" :by
            (:instance compiler-correctness-form-forms (flag t)))))
```

Conclusions and Further Work

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)