A Mechanically Verified Compiling Specification for a Lisp Compiler*

Axel Dold and Vincent Vialard

Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
Fax: +49/(0)731/50-24119
{dold|vialard}@informatik.uni-ulm.de

Abstract. We report on an ongoing effort in mechanically proving correct a compiling specification for a bootstrap compiler from ComLisp (a subset of ANSI Common Lisp sufficiently expressive to serve as a compiler implementation language) to binary Transputer code using the PVS system. The compilation is carried out in four steps through a series of intermediate languages. This paper focuses on the first phase, namely, the compilation of ComLisp to the stack-intermediate language SIL, where parameter passing is implemented by a stack technique. The context of this work is the joint research effort *Verifix* aiming at developing methods for the construction of correct compilers for realistic programming languages.

1 Introduction

The use of computer based systems for safety-critical applications requires high dependability of the software components. In particular, it justifies and demands the verification of programs typically written in high-level programming languages. Correct program execution, however, crucially depends on the correctness of the binary machine code executable, and therefore, on the correctness of system software, especially compilers. As already noted in 1986 by Chirica and Martin [3], full compiler correctness comprises both the correctness of the compiling specification (with respect to the semantics of the languages involved) as well as the correct implementation of the specification.

Verifix [6, 9] is a joint German research effort of groups at the universities Karlsruhe, Kiel, and Ulm. The project aims at developing innovative methods for constructing provably correct compilers which generate efficient code for realistic, practically relevant programming languages. These realistic compilers are to be constructed using approved development techniques. In particular, even standard unverified compiler generation tools (such as Lex or Yacc) may be used, the correctness of the generated code being verified at compile time using verified

* This research has been funded by the Deutsche Forschungsgemeinschaft (DFG) under project "*Verifix*".

program checkers [7]. *Verifix* assumes hardware to behave correctly as described in the instruction manuals.

In order not to have to write the verified parts of the compiler and checkers directly in machine code, a fully verified and correctly implemented *initial compiler* is required, for which efficiency of the produced code is not a priority. The initial correct compiler to be constructed in this project transforms ComLisp programs into binary Transputer code. ComLisp is an imperative proper subset of ANSI-Common Lisp and serves both as a source and implementation language for the compiler. The construction process of the initial compiler consists of the following steps:

– define syntax and semantics of appropriate intermediate languages.
– define the compiling specification, a relation between source and target language programs and prove (with respect to the language semantics) its correctness according to a suitable correctness criterion.
– construct a correct compiler implementation in the source language itself (a transformational constructive approach is applied which builds a correct implementation from the specification by stepwise applying correctness-preserving development steps [5]).
– use an existing (unverified) implementation of the source language (here: some arbitrary Common Lisp compiler) to execute the program. Apply the program to itself and bootstrap a compiler executable. Check syntactically, that the executable code has been generated according to the compiling specification. For this last step, a realistic technique for low level compiler verification has been developed which is based on rigorous a posteriori syntactic code inspection [8,11]. This closes the gap between high-level implementation and executable code.

The size and complexity of the verification task in constructing a correct compiler is immense. In order to manage it, suitable mechanized support for both specification and verification is necessary. We have chosen the PVS specification and verification system [16] to support the verification of the compiling specification and the construction process of a compiler implementation in the source language.

In this paper, we focus on the mechanical verification of the compiling specification for the ComLisp compiler. In particular, we describe the formalization and verification process of the first compilation phase from ComLisp to the stack-based intermediate language SIL, the first of a series of intermediate languages used to compile ComLisp programs into binary Transputer machine code:

$$\text{ComLisp} \rightarrow \text{SIL} \rightarrow \text{C}^{\text{int}} \rightarrow \text{TASM} \rightarrow \text{TC}$$

First, ComLisp is translated into a stack intermediate language (SIL), where parameter passing is implemented by a stack technique. Expressions are transformed from a prefix notation into a postfix notation according to the stack principle. SIL is then compiled into C^{int} where the ComLisp data structures (s-expressions) and operators are implemented in linear integer memory using a run-time stack and a heap. These two steps are machine independent. In the next step, control structures of $\mathrm{C}^{\mathrm{int}}$ are implemented by linear assembler code with jumps, and finally, abstract assembler code is transformed into binary Transputer code.

This paper is organized as follows. The next section presents the formalization of the languages ComLisp and SIL, that is, their abstract syntax and semantics. Operational semantics in a structural operational style are provided for both languages by means of a set of inductive rules. Section 3 then focuses on the compilation process from ComLisp to SIL. Finally, Section 4 is concerned with the correctness of this compilation process.

2 Syntax and Semantics of the Languages

2.1 ComLisp

A ComLisp program consists of a list of global variables, a list of possibly mutual recursive function definitions, and a main form. ComLisp forms (expressions) include the *abort* form, s-expression constants, variables, assignments, sequential composition (*progn*), conditional, while loop, call of user defined functions, call of built-in unary (*uop*) and binary (*bop*) ComLisp operators, local let-blocks, *list∗* operator (constructing a s-expression list from its evaluated arguments), case-instruction, and instructions for reading from the input sequence and writing to the output. The ComLisp operators include the standard operators for lists (e.g. *length*), type predicates for the different kinds of s-expressions, and the standard arithmetic operations (e.g. $+, *, \mathit{floor}$). The only available datatype is the type of s-expressions which are binary trees built with constructor "cons", where the leaves are either integers, characters, strings, or symbols. The set of symbols includes T and NIL. The abstract syntax of ComLisp is given as follows:

$$\begin{array}{lll}
p & ::= & x_1, \ldots, x_k; f_1, \ldots, f_n; e \\
f & ::= & h(x_1, \ldots, x_m) \leftarrow e \\
e & ::= & \mathit{abort} \mid c \mid x \mid x := e \mid \mathit{progn}(e_1, \ldots, e_n) \mid \mathit{if}(e_1, e_2, e_3) \mid \mathit{while}(e_1, e_2) \mid \\
 & & \mathit{call}(h, e_1, \ldots, e_n) \mid \mathit{uop}(e) \mid \mathit{bop}(e_1, e_2) \mid \mathit{let}(x_1 = e_1, \ldots, x_n = e_n; e) \mid \\
 & & \mathit{list}{*}(e_1, \ldots, e_n) \mid \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n) \mid \\
 & & \mathit{read\_char} \mid \mathit{peek\_char} \mid \mathit{print\_char}(e)
\end{array}$$

The static semantics of ComLisp programs, function definitions, and forms is specified by means of several well-formedness predicates. A ComLisp form is *well-formed*—with respect to a local variable environment ζ (a list of formal parameters), a list of global variables γ, and a function environment Γ (a list of function definitions)—if the list of local and global variables are disjoint, all variables are declared (that is, occur either in ζ or γ) and each user-defined function is declared in Γ and called with the correct number of arguments (correct parameter passing). Formally, a relation $\mathit{wf}(e, \zeta, \gamma, \Gamma)$ is defined inductively on the structure of forms (omitted here). A function environment Γ is well-formed with respect to a list of global variables γ, if the function names in Γ are disjoint (no double declarations of functions), and each function body in Γ is well-formed with respect to its local parameter list, γ, and Γ. This is specified by predicate $\mathit{wf}_{\mathrm{proc}}(\Gamma, \gamma)$. Finally, the well-formedness relation for ComLisp programs is straightforward. Let $p = \gamma; \Gamma; e$. Then $\mathit{wf}_{\mathrm{program}}(p) ::= \mathit{wf}_{\mathrm{proc}}(\Gamma, \gamma) \ \wedge \ \mathit{wf}(e, [], \gamma, \Gamma)$.

For the intermediate languages occurring in the different compilation phases of the ComLisp to Transputer compiler, a uniform relational semantics description has been chosen. The (dynamic) semantics of ComLisp is defined in a structural operational way by a set of inductive rules for the different ComLisp forms. This kind of semantics is also referred to as *big-step semantics* or *evaluation semantics* in contrast to a transition semantics (small-step semantics) such as abstract state machines (ASM's). A ComLisp *state* is a triple consisting of an (infinite) input sequence (stream) of characters, an output list of characters, and the variable state which is a mapping from identifiers to values (s-expressions):

$$\mathit{state}_{\mathrm{CL}} ::= \mathit{sequence}[\mathit{char}] \times \mathit{char}^* \times (\mathit{Ident} \to \mathit{SExpr})$$

ComLisp forms are expressions with side-effects, that is, they denote state transformers transforming states to pairs of result value and result state. The definition of the semantics of forms uses the following notation: $\Gamma \vdash s : e \to (v, q)$. It states that evaluating form e in state s and function environment Γ terminates and results in a value v and final state q. Given rules for each kind of form, the semantics is defined as the smallest relation $\to$ satisfying the set of rules. For example, the semantics of a function call is given by two rules. One for parameterless functions, and one for functions with parameters, where the parameters are sequentially evaluated, the resulting values being then bound to the parameters before evaluation of the body and unbound after returning the value:

$$\frac{\begin{array}{c}[f(x_1 \cdots x_n) \leftarrow \mathit{body}] \in \Gamma \ (n \geq 1) \\ \Gamma \vdash q_i : e_i \to (v_i, q_{i+1}) \ (1 \leq i \leq n) \\ \Gamma \vdash q_{n+1}[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] : \mathit{body} \to (v, r)\end{array}}{\Gamma \vdash q_1 : \mathit{call}(f, e_1, \ldots, e_n) \to (v, r[x_1 \leftarrow q_{n+1}(x_1), \ldots, x_n \leftarrow q_{n+1}(x_n)])}$$

The complete set of rules for ComLisp forms can be found in the appendix A.

The semantics of a ComLisp program is given by the input/output behavior of the program defined by a relation $P_{\mathrm{sem}_{\mathrm{CL}}}$ between input streams is and output lists ol. $P_{\mathrm{sem}_{\mathrm{CL}}}(p)(is, ol)$ holds if the evaluation of the main form e in an initial state, where the input stream is given by is, the output list is empty and all variables are initialized with NIL, terminates with a value v in some state q with output list ol. Formally:

$$P_{\mathrm{sem}_{\mathrm{CL}}}(p)(is, ol) ::= \exists v, q.\ (\Gamma \vdash (is, [], \lambda x.\mathit{NIL}) : e \to (v, q)) \ \wedge \ (q_{\mathrm{output}} = ol)$$

2.2 SIL

SIL, the stack intermediate language, is a language with parameterless procedures and s-expressions as available datatype. Programs operate on a runtime stack with frame-pointer relative addresses. A SIL program consists of a list of parameterless procedure declarations and a main statement. There are no variables, only memory locations and the machine has statements for copying values from the global to the local memory and vice versa. For example, $copy(i, j)$ copies the content at stack relative position i to relative position j, $gcopy(g, i)$ copies from the global memory at position g to the relative position i, and $itef(i, s_1, s_2)$ executes instruction s_2 if the content of stack relative position i is NIL, otherwise s_1 is executed.

$$\begin{array}{lcl} p & ::= & f_1, \ldots, f_n; s \\ f & ::= & h \leftarrow s \\ s & ::= & abort \mid copyc(c,i) \mid copy(i,j) \mid gcopy(g,i) \mid copyg(g,i) \mid \\ & & itef(i, s_1, s_2) \mid sq(s_1, \ldots, s_n) \mid fcall(h,i) \mid uop(i) \mid bop(i) \mid \\ & & while(i, s_1, s_2) \mid read\_char(i) \mid peek\_char(i) \mid print\_char(i) \mid list{*}(n,i) \end{array}$$

The static semantics is again specified by means of well-formedness predicates for SIL statements, SIL procedure declarations, and SIL programs (definitions omitted here). SIL statements denote state transformers, where a SIL state consists of the input stream, the output list, the global memory (a list of s-expressions), and the local memory (consisting of the frame pointer $base : Nat$ and the stack, a function from natural numbers to s-expressions).

$$state_{\mathrm{SIL}} ::= sequence[char] \times char^* \times SExpr^* \times Nat \times (Nat \to SExpr)$$

As for ComLisp, an evaluation semantics for SIL statements is defined as the smallest relation $\Gamma \vdash s : cmd \to q$ satisfying the set of rules given for the language constructs. The relation states that executing the statement cmd in state s and SIL procedure environment Γ (a list of procedure declarations) is defined, terminates, and results in a new state q. The rules for SIL statements are listed in the appendix B.

As for ComLisp, the semantics of a SIL program is its I/O behavior:

$$P_{\mathrm{sem_{SIL}}}(p)(is, ol) ::= \exists q.\ (\Gamma \vdash init : s \to q)\ \wedge\ (q_{\mathrm{output}} = ol)$$

where the initial state is defined by $init ::= (is, [], [NIL, \ldots, NIL], 0, \lambda n.NIL)$.

2.3 PVS Formalization of the Languages

Abstract syntax, static and dynamic semantics of the languages have to be formalized in the PVS specification language. The language is based on classical higher-order logic with a rich type system including dependent types. In addition, the PVS system provides an interactive proof checker that has a reasonable amount of theorem proving capabilities. A strategy language enables to combine atomic inference steps into more powerful proof strategies allowing to define reusable proof methods.

1. Abstract Syntax: the PVS abstract data type (ADT) construct is used. ComLisp forms, for example, are defined by an ADT, where for each kind of form there exists a corresponding constructor. For ADT definitions in PVS, a large theory is automatically generated including induction and reduction schemes for the ADT, termination measures, and a set of axioms stating that the data type denotes the initial algebra defined by the constructors. Note that the formalizations make heavily use of library specifications. However, a lot of new types, functions, and predicates must be added for the specifications, as well as lemmas for their useful properties (which have to be proved).
2. Static Semantics: the well-formedness predicates must be formalized. Since each function must be total in PVS, a termination measure must be provided for the recursive definitions. We have specified the structural size of a ComLisp form using the reduction scheme from the ADT theory.
3. Dynamic Semantics: the rules must be represented in PVS. A set of structural rules is represented as an inductive PVS relation which combines all the rules in one single definition $E(\Gamma)(s, e, v, q, N)$ which denotes $\Gamma \vdash s : e \rightarrow (v, q)$. Free logical variables in the rules are existentially quantified in the corresponding PVS relation. In general, properties about inductive relations can be proved by rule induction. Here, the definition of relation E has an additional counter parameter N to formulate an induction principle needed for the proof for the selected notion of correctness (see Sect. 4). N is decreased when entering the body of a function or while loop, since in this case the forms in the antecedents of the corresponding rules are not structurally smaller, and left unchanged otherwise.

3 Compiling ComLisp to SIL

The compilation from ComLisp to SIL generates code according to the stack principle and translates parameter passing to statements which access the data stack. For a given expression e, a sequence of SIL instructions is generated that computes its value and stores it at the top of the stack (relative position k in the current frame). The parameters $x_1, \ldots, x_n$ of a function are stored at the bottom of the current frame (at relative positions $0, \ldots, n-1$) (see Fig. 1). A SIL function call $\mathit{fcall}(h, i)$ increases the frame pointer base by i which is reset to its old value after the call and local variables introduced by let are represented within the current frame. For each syntactical ComLisp category, a compiling function is specified.

– $\mathcal{C}_{\text{form}}(e, \gamma, \rho, k)$ is defined inductively on e. It takes a form e, a global environment γ (a list of identifiers), a compile time environment ρ (an association list which associates relative positions in the current stack frame with local variables), and a natural number k (denoting the current top of stack) and produces a SIL statement. Its definition can be found in the appendix C.
– A function definition is compiled by compiling the body in a new environment (where the formal parameters are associated with relative positions $0, \ldots, n-1$) with the top of stack set at position n. Finally, the current stack frame has

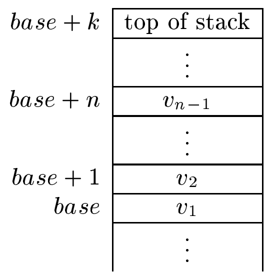

Fig. 1. Parameter passing on the stack

to be removed, leaving only the result on top (achieved by a copy instruction from position n to 0).

$$\mathcal{C}_{\text{def}}(h(x_1,\ldots,x_n) \leftarrow e)(\gamma) ::= h \leftarrow sq(\mathcal{C}_{\text{form}}(e,\gamma,[x_i \leftarrow (i{-}1)],n), copy(n,0))$$

– A function environment Γ is compiled by compiling each function definition in Γ:
$$\mathcal{C}_{\text{defs}}([f_1,\ldots,f_n])(\gamma) ::= [\mathcal{C}_{\text{def}}(f_1)(\gamma),\ldots,\mathcal{C}_{\text{def}}(f_n)(\gamma)]$$
– A program $p = \gamma;\Gamma;e$ is compiled by compiling its function environment and main form:
$$\mathcal{C}_{\text{prog}}(p) ::= \mathcal{C}_{\text{defs}}(\Gamma)(\gamma);\mathcal{C}_{\text{form}}(e,\gamma,[],0)$$

4 Correctness of the Compilation Process

An appropriate notion of correct compilation for sequential imperative languages on a concrete target processor must take the finite resource limitations of the target architecture into account. The notion of correctness used in *Verifix* is the preservation of the observable behavior up to resource limitations. In our case correctness of the compilation process is stated as follows: for any well-formed ComLisp program p, whenever the semantics of the compiled program is defined for some input stream is and output list ol, this is also the case for p for the same is and ol:

Theorem 1 (Correctness of Program Compilation).
$\forall p, is, ol.\ wf_{\text{program}}(p) \Rightarrow (P_{\text{sem}_{\text{SIL}}}(\mathcal{C}_{\text{prog}}(p))(is)(ol) \Rightarrow P_{\text{sem}_{\text{CL}}}(p)(is)(ol))$

Unfolding $P_{\text{sem}_{\text{SIL}}}$ and $P_{\text{sem}_{\text{CL}}}$, the semantics of forms and corresponding SIL statements have to be compared. In particular, this requires relating source and target language states. ComLisp forms denote state transformers transforming a state into a result value and a result state (if defined) $\sigma \rightarrow_e (v,\sigma')$. On the other hand, SIL statements denote ordinary state transformers $s \rightarrow_s s'$. Two relations are required: one relation ρ_{in} relates ComLisp input states σ with SIL states s, while the other relation ρ_{out} relates ComLisp output states (v, σ') with SIL states s'. Figure 2 illustrates the correctness property for forms by means of a commuting diagram. The relations are parameterized with a list of global

$$\begin{tikzcd}
\mathit{state}_{\text{CL}} \ni \sigma \arrow[r, "\rightarrow_e"] \arrow[d, dash, "\rho_{\text{in}}"'] \arrow[dr, phantom, "\circlearrowleft"] & (v, \sigma') \in \mathit{SExpr} \times \mathit{state}_{\text{CL}} \arrow[d, dash, "\rho_{\text{out}}"] \\
\mathit{state}_{\text{SIL}} \ni s \arrow[r, "\rightarrow_s"] & s' \in \mathit{state}_{\text{SIL}}
\end{tikzcd}$$

Fig. 2. Correctness property for the compilation of ComLisp forms

variables γ, the local compile time environment ρ, and the current top of stack position k. Relation ρ_{in} distinguishes between local and global variables. The relative address for variables for which ρ is defined is given by $\rho(x)$, while the address of the global variables in γ is given by $\gamma(x)$. Relation ρ_{out} additionally assumes that the final value v is available at the stack top (relative address k). In addition, it is required that the input streams and the output lists of σ and s correspond. The data representation relations are defined as follows:
$\rho_{\text{in}}(\gamma, \rho, k)(\sigma, s) ::=$
 $[\forall x \in \mathit{dom}(\rho).\ (\rho(x) < k) \ \wedge\ (\sigma(x) = s_{\text{local}}(s_{\text{base}} + \rho(x)))] \ \wedge$
 $[\forall x \in \gamma.\ (\gamma(x) < |s_{\text{global}}|) \ \wedge\ (\sigma(x) = s_{\text{global}}(\gamma(x)))] \ \wedge$
 $(s_{\text{input}} = \sigma_{\text{input}}) \ \wedge\ (s_{\text{output}} = \sigma_{\text{output}})$
$\rho_{\text{out}}(\gamma, \rho, k)(v, \sigma', s') ::= (s'_{\text{local}}(s'_{\text{base}} + k) = v) \ \wedge\ (\rho_{\text{in}}(\gamma, \rho, k)(\sigma', s'))$

In order to state the correctness property for the compilation of forms two additional invariants are required:

1. The first invariant relates ComLisp input and output states. It assures that identifiers not belonging to ζ or γ (the local and global identifier lists) do not alter their values.

   $$\mathit{source\_invar?}(\zeta, \gamma)(\sigma, \sigma') ::= \forall x.\ (x \notin \zeta \ \wedge\ x \notin \gamma) \ \Rightarrow\ \sigma'(x) = \sigma(x)$$

2. The second one relates input SIL states s with output SIL states s'. It states that
   (a) the frame pointers of s and s' are identical.
   (b) the contents of all stack cells with addresses not within the range of the local environment ρ do not change from s to s'. In particular, this includes all stack cells below the current stack frame.

   $\mathit{invar?}(\rho, k)(s, s') ::=$
    $s_{\text{base}} = s'_{\text{base}} \ \wedge$
    $\forall \mathit{adr}.\ \mathit{adr} < k \wedge \mathit{adr} \notin \mathit{ran}(\rho) \Rightarrow s_{\text{local}}(s_{\text{base}} + \mathit{adr}) = s'_{\text{local}}(s'_{\text{base}} + \mathit{adr}) \ \wedge$
    $\forall \mathit{adr}.\ \mathit{adr} < s_{\text{base}} \Rightarrow s_{\text{local}}(\mathit{adr}) = s'_{\text{local}}(\mathit{adr})$

   This property is required to ensure that for function and operator calls the computed values of the arguments are still available (and not overwritten) when the operator is applied or the function body is executed.

All ingredients have now been collected to state the correctness property for the translation of forms. The diagram in Fig. 2 has to commute in the sense of preservation of partial program correctness. The property states that if the function environment and the ComLisp form is well-formed, the compile time environment ρ is injective and its domain corresponds to the local variable list ζ, the initial ComLisp and SIL states are related by ρ_{in} and the code resulting from compiling form e transforms SIL state s into s', then there exists a value v and ComLisp state σ' such that e evaluates in state σ to (v, σ') and the final ComLisp and SIL states are related by ρ_{out} and the target states and source states invariants hold:

Definition 1 (Correctness Property for Form Compilation).
$\mathit{correct\_prop}(\Gamma, \gamma, \zeta, \rho, k)(e) ::=$
$\forall \sigma, s, s'.\ \mathit{wf}_{\text{proc}}(\Gamma, \gamma)\ \wedge\ \mathit{wf}(e, \zeta, \gamma, \Gamma)\ \wedge\ \mathit{injective?}(\rho)\ \wedge\ (\mathit{dom}(\rho) = \zeta)\ \wedge$
$\quad \rho_{\text{in}}(\gamma, \rho, k)(\sigma, s)\ \wedge\ (\mathcal{C}_{\text{defs}}(\Gamma)(\gamma) \vdash s : \mathcal{C}_{\text{form}}(e, \gamma, \rho, k) \to s')$
$\quad\quad \Rightarrow \exists v, \sigma' : (\Gamma \vdash \sigma : e \to (v, \sigma'))\ \wedge\ \rho_{\text{out}}(\gamma, \rho, k)(v, \sigma', s')\ \wedge$
$\quad\quad\quad \mathit{invar?}(\rho, k)(s, s')\ \wedge\ \mathit{source\_invar?}(\zeta, \gamma)(\sigma, \sigma')$

The main obligation is to prove that this property holds for each kind of form:

Theorem 2 (Correctness of Form Compilation).
$\forall e, \Gamma, \gamma, \zeta, \rho, k.\ \mathit{correct\_prop}(\Gamma, \gamma, \zeta, \rho, k)(e)$

In the PVS formalization, the correctness property has an additional counter argument N according to the inductive relations defining the semantics. This additional argument is required here since we prove that the target semantics implies the source semantics but the compilation is defined structurally on the source language. If we would prove the other way round, rule induction (without a counter argument) would suffice. The PVS proof of this theorem is done by measure induction (a variant of well-founded induction) using the lexicographic combination of the counter N and the structural size of form e as termination measure:

$$(N', e') < (N, e) ::= (N' < N \vee (N' = N \wedge \mathit{size}(e') < \mathit{size}(e)))$$

This measure ensures that for each kind of form the induction hypothesis is applicable. To suitably manage the complexity of this proof, for each kind of form a separate compilation theorem is introduced. The proof of Theorem 2 is then carried out by case analysis and application of the compilation theorems.

Most of the proofs of the compilation theorems follow a similar scheme according to the structure of the correctness property (see Definition 1):

1. First, definitions must be unfolded and the SIL statement which results from compiling the ComLisp form must be "executed" symbolically according to the operational SIL semantics.
2. The induction hypothesis (stated as a precondition in the compilation lemmas) must be instantiated.
3. Instantiations for the result value v and result state σ' (existentially quantified variables) of the ComLisp form must be found.

4. The consequent part of the formula must be proved. This reduces to showing four properties:
   (a) show that form e evaluates to the instantiated value and result state.
   (b) show with the help of precondition ρ_{in} that the output source and target states are related by ρ_{out} (Note that ρ_{out} is defined by means of ρ_{in}).
   (c) show that the target state invariant holds.
   (d) show that the source state invariant holds.

PVS strategies have been defined for some of the cases of the general scheme. These strategies enable the (nearly) automatic discharge of the respective cases. The proofs of most of the compilation lemmas are relatively straightforward and follow directly the scheme. However, some of the compilation theorems are tedious, in particular the theorems for function call, let-form, and *list∗*. They make use of an additional lemma which relates sequences of ComLisp forms with SIL statement sequences. Due to lack of space we cannot go into the details of the proofs. All the proofs have been completely accomplished using PVS.

Statistics

We present some statistics concerning the formalization and verification effort for this compilation step. Table 1 summarizes the results. First of all, we have extended the built-in PVS library with additional functions and properties for lists, and with a new theory for association lists (finite maps). This library has already been reused for other verification tasks. There are 7 additional PVS theories with 621 lines of PVS specification code (LOC), 139 obligations to prove including all type correctness conditions generated by the system. These obligations are proved interactively by invoking 1048 proof steps. The specifications of the languages ComLisp and SIL including the definition of s-expressions and corresponding unary and binary operators involve 7 theories. Not surprisingly, the most effort lies in the verification of the compiling specification: 30 proof obligations (mainly the compiling theorems) have been proved in more than 1600 proof steps. Most work has been put into the verification of the compilation theorems for function call, *let*, and *list∗*. Although strategies for parts of the proofs have been developed, the number of manual steps is quite high and shows that this verification task is by no means trivial.

It is hard to give an estimation of the amount of work invested in the final verification, since we started the verification on a smaller subset of ComLisp in order to experiment with different styles of semantics and find the necessary invariants, and then incrementally extended this subset and tried to rerun and adapt the already accomplished proofs. A coarse estimation of the total formalization and verification effort required for the compiling specification for all 4 compilation phases is about 3 person-years.

Related Work

Verification of compiler correctness is a much-studied area starting with the work by McCarthy and Painter in 1967 [13], where a simple compiler for arith-

Table 1. Formalization and verification statistics

	PVS theories	LOC	proof obligations	proof steps
spec. of languages	7	759	139	575
compiling specification	1	122	36	95
compiling verification	1	219	30	1617
list, alist library	7	621	139	1048
	16	1721	344	3335

metic expressions has been proved correct. Many different approaches have been taken since then, usually with mechanized support to manage the complexity of the specifications and the proofs, for example [1,2,4,12,14,17]. Most of the approaches only deal with the correctness of the compiling specification, while the approach taken in the *Verifix* project also takes care of the implementation verification, even on the level of binary machine code. Another difference of our approach is that we are concerned with the compilation of "realistic" source languages and target architectures. A ComLisp implementation of the ComLisp compiler as well as a binary Transputer executable is available.

Notable work in this area with mechanized support is CLInc's verified stack of system components ranging from a hardware-processor up to an imperative language [14]. Both the compiling verification and the high-level implementation (in ACL2 logic which is a LISP subset) have been carried out with mechanized support using the ACL2 prover. Using our compiler, correct binary Transputer code could be generated.

The impressive VLISP project [10] has focused on a correct translation for Scheme. However, although the necessity of also verifying the compiler implementation has been expressed this has explicitly been left out. Proofs were accomplished without mechanized support.

P. Curzon [4] considers the verification of the compilation of a structured assembly language, Vista, into code for the VIPER microprocessor using the HOL system. Vista is a low-level language including arithmetic operators which correspond directly to those available on the target architecture.

The compilation of PROLOG into WAM has been realized through a series of refinement steps and has been mechanically verified using the KIV system [18]. A (small-step) ASM semantics is used for the languages.

5 Concluding Remarks

In this paper we have reported on an ongoing effort in constructing a correct bootstrap compiler for a subset of Common Lisp into binary Transputer code. We have focused on the formal, mechanically supported verification of the compiling specification of the first compilation phase. The verification of the second phase, the translation from SIL to C^{int}, where s-expressions and their operators are implemented in linear memory (classical data and operation refinement), is also completed. Current work is concerned with the verification of the compiler back-end, namely, the compilation from C^{int} into abstract Transputer assembler code TASM. The standard control structures of C^{int} must be implemented by conditional and unconditional jumps, and the state space must be realized on the concrete Transputer memory. Hence, this step is again a data refinement process to be verified. The verification of the last compilation phase, where abstract Transputer assembler is compiled into binary Transputer code (TC) has already been accomplished following approved verification techniques [15]: starting from a (low-level) base model of the Transputer, where programs are a part of the memory, a series of abstraction levels is constructed allowing different views on the Transputer's behavior and the separate treatment of particular aspects.

We have demonstrated that the formal, mechanized verification of a non-trivial compiler for a (nearly) realistic programming language into a real target architecture is feasible with state-of-the-art prover technology.

Acknowledgements

The construction of the initial correct ComLisp compiler is joint work with the project partners from the university of Kiel. We thank them for their constructive collaboration and many discussions on this subject. The constructive criticism and suggestions provided by the anonymous referees have been a great help to improve this paper.

References

1. E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd Int. Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*. Springer, 1998.
2. M. Broy. Experiences with Software Specification and Verification using LP, the Larch Proof Assistant. Technical report, Digital Systems Research Center, 1992.
3. L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
4. Paul Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.
5. Axel Dold. *Formal Software Development using Generic Development Steps.* Logos-Verlag, Berlin, 2000. Dissertation, Universität Ulm.
6. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In *Proceedings of the Poster Session of CC'96 - International Conference on Compiler Construction.* ida, 1996. TR-Nr.: R-96-12.
7. Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.

8. Wolfgang Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. of the 9th Int. Colloquium on Numerical Analysis and Computer Sciences with Applications, Plovdiv, Bulgaria*, 2000.
9. G. Goos and W. Zimmermann. Verification of Compilers. In B.Steffen E.-R. Olderog, editor, *Correct System Design*, volume 1710 of *LNCS*, pages 201–230. Springer-Verlag, 1999.
10. J. D. Guttman, L. G. Monk, J. D. Ramsdell, W. M. Farmer, and V. Swarup. A Guide to VLISP, A Verified Programming Language Implementation. Technical Report M92B091, The MITRE Corporation, Bedford, MA, September 1992.
11. Ulrich Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998.
12. J.J. Joyce. A Verified Compiler for a Verified Microprocessor. Technical Report 167, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, March 1989.
13. J. McCarthy and J.A. Painter. Correctness of a Compiler for Arithmetical Expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science.* American Mathematical Society, 1967.
14. J S. Moore. A Mechanically Verified Language Implementation. *Journal of Automated Reasoning*, 5(4), 1989.
15. Markus Müller-Olm. *Modular Compiler Verification*, volume 1283 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 1997. PhD Thesis.
16. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
17. W. Polak. *Compiler Specification and Verification*, volume 124 of *LNCS*. Springer-Verlag, 1981.
18. Gerhard Schellhorn. *Verifikation abstrakter Zustandsmaschinen.* PhD thesis, Unversität Ulm, 1999.

A Semantics of ComLisp Forms

For a state s, we denote the input stream of s by s_{input}, the output list of s by s_{output}, and the variable state of s by $s_{\text{var}} : \mathit{Ident} \to \mathit{SExpr}$. In the following to increase readability, we often write simply s instead of s_{var}; $s[x \leftarrow v]$ denotes the modification of s_{var} at x by v.

ComLisp operators denote partial functions on s-expressions which is expressed by two relations: relation $v_1 : \mathit{uop} \to v_2$ for unary operators uop, and $v_1, v_2 : \mathit{bop} \to v$ for binary operators. For example, the first relation states that the application of unary operator uop to s-expression v_1 is defined, terminates, and yields s-expression v_2 as result.

– constants, variables

$$\Gamma \vdash s : c \to (c, s) \qquad \Gamma \vdash s : x \to (s(x), s)$$

– assignment:

$$\frac{\Gamma \vdash s : e \to (v, q)}{\Gamma \vdash s : x := e \to (v, q[x \leftarrow v])}$$

– sequential composition:

$$\Gamma \vdash s : \mathit{progn}([]) \rightarrow (\mathit{NIL}, s) \qquad \frac{\Gamma \vdash s : e \rightarrow (v, q)}{\Gamma \vdash s : \mathit{progn}(e) \rightarrow (v, q)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s : e_1 \rightarrow (v_1, q_1) \\ \Gamma \vdash q_1 : \mathit{progn}(e_2, \ldots, e_n) \rightarrow (v, q)\end{array}}{\Gamma \vdash s : \mathit{progn}(e_1, \ldots, e_n) \rightarrow (v, q)} \quad \text{if } n \geq 2$$

– conditional:

$$\frac{\Gamma \vdash s : e_1 \rightarrow (\mathit{NIL}, q_1) \ ; \ \Gamma \vdash q_1 : e_3 \rightarrow (v, q)}{\Gamma \vdash s : \mathit{if}(e_1, e_2, e_3) \rightarrow (v, q)}$$

$$\frac{\Gamma \vdash s : e_1 \rightarrow (v_1, q_1); \ \Gamma \vdash q_1 : e_2 \rightarrow (v, q)}{\Gamma \vdash s : \mathit{if}(e_1, e_2, e_3) \rightarrow (v, q)} \quad \text{if } v_1 \neq \mathit{NIL}$$

– while loop:

$$\frac{\begin{array}{c}\Gamma \vdash s : c \rightarrow (v_1, q_1) \ (v_1 \neq \mathit{NIL}) \\ \Gamma \vdash q_1 : \mathit{body} \rightarrow (v_2, q_2) \\ \Gamma \vdash q_2 : \mathit{while}(c, \mathit{body}) \rightarrow (v, q)\end{array}}{\Gamma \vdash s : \mathit{while}(c, \mathit{body}) \rightarrow (v, q)} \qquad \frac{\Gamma \vdash s : c \rightarrow (\mathit{NIL}, q)}{\Gamma \vdash s : \mathit{while}(c, \mathit{body}) \rightarrow (\mathit{NIL}, q)}$$

– call of user-defined functions:

$$\frac{\begin{array}{c}[f(x_1 \cdots x_n) \leftarrow \mathit{body}] \in \Gamma \ (n \geq 1) \\ \Gamma \vdash q_i : e_i \rightarrow (v_i, q_{i+1}) \ (1 \leq i \leq n) \\ \Gamma \vdash q_{n+1}[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] : \mathit{body} \rightarrow (v, r)\end{array}}{\Gamma \vdash q_1 : \mathit{call}(f, e_1, \ldots, e_n) \rightarrow (v, r[x_1 \leftarrow q_{n+1}(x_1), \ldots, x_n \leftarrow q_{n+1}(x_n)])}$$

$$\frac{\begin{array}{c}[f() \leftarrow \mathit{body}] \in \Gamma \\ \Gamma \vdash s : \mathit{body} \rightarrow (v, q)\end{array}}{\Gamma \vdash s : \mathit{call}(f, ()) \rightarrow (v, q)}$$

– built-in unary and binary operators:

$$\frac{\begin{array}{c}\Gamma \vdash s : e \rightarrow (v_1, q) \\ v_1 : \mathit{uop} \rightarrow v\end{array}}{\Gamma \vdash s : \mathit{uop}(e) \rightarrow (v, q)} \qquad \frac{\begin{array}{c}\Gamma \vdash s : e_1 \rightarrow (v_1, q_1) \\ \Gamma \vdash q_1 : e_2 \rightarrow (v_2, q) \\ v_1, v_2 : \mathit{bop} \rightarrow v\end{array}}{\Gamma \vdash s : \mathit{bop}(e_1, e_2) \rightarrow (v, q)}$$

– let block:

$$\frac{\begin{array}{c}\Gamma \vdash q_i : e_i \rightarrow (v_i, q_{i+1}) \ (1 \leq i \leq n) \\ \Gamma \vdash q_{n+1}[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] : e \rightarrow (v, r)\end{array}}{\Gamma \vdash q_1 : \mathit{let}(x_1 = e_1, \ldots, x_n = e_n; e) \rightarrow (v, r[x_1 \leftarrow q_{n+1}(x_1), \ldots, x_n \leftarrow q_{n+1}(x_n)])}$$

$$\frac{\Gamma \vdash s : e \rightarrow (v, q)}{\Gamma \vdash s : \mathit{let}([], e) \rightarrow (v, q)}$$

– *list∗* operator:

$$\frac{\Gamma \vdash s : e \to (v, q)}{\Gamma \vdash s : \mathit{list*}(e) \to (v, q)} \qquad \frac{\begin{array}{c}\Gamma \vdash s : e_1 \to (v_1, q_1) \\ \Gamma \vdash q_1 : \mathit{list*}(e_2, \ldots, e_n) \to (v_2, q)\end{array}}{\Gamma \vdash s : \mathit{list*}(e_1, \ldots, e_n) \to (\mathit{cons}(v_1, v_2), q)}$$

– case form:

$$\Gamma \vdash s : \mathit{cond}() \to (\mathit{NIL}, s) \qquad \frac{\begin{array}{c}\Gamma \vdash s : p \to (v_1, q_1) \quad (v_1 \neq \mathit{NIL}) \\ \Gamma \vdash q_1 : e \to (v, q)\end{array}}{\Gamma \vdash s : \mathit{cond}(p \leftarrow e) \to (v, q)}$$

$$\frac{\begin{array}{c}\Gamma \vdash s : p_1 \to (\mathit{NIL}, q_1) \\ \Gamma \vdash q_1 : \mathit{cond}(p_2 \leftarrow e_2, \ldots, p_n \leftarrow e_n) \to (v, q)\end{array}}{\Gamma \vdash s : \mathit{cond}(p_1 \leftarrow e_1, \ldots, p_n \leftarrow e_n) \to (v, q)}$$

– input/output:

$$\Gamma \vdash s : \mathit{read\_char} \to (\mathrm{first}(s_{\mathrm{input}}), s[s_{\mathrm{input}} := \mathrm{rest}(s_{\mathrm{input}})])$$

$$\Gamma \vdash s : \mathit{peek\_char} \to (\mathrm{first}(s_{\mathrm{input}}), s)$$

$$\frac{\Gamma \vdash s : e \to (v, q) \ (v \in \mathit{char})}{\Gamma \vdash s : \mathit{print\_char}(e) \to (v, q[q_{\mathrm{output}} := q_{\mathrm{output}} {+\!\!+} v])}$$

B Semantics of SIL Statements

In the following, $|l|$ denotes the length of list l, and $l(i)$ denotes the ith element of l for $(0 \leq i < |s|)$. s_{local}, s_{global}, s_{base} denote the respective state components in state s. To increase readability, we simply write $s(i)$ for the relative local access $s_{\mathrm{local}}(s_{\mathrm{base}} + i)$, and write $s[i \leftarrow v]$ for $s[s_{\mathrm{local}}(s_{\mathrm{base}} + i) \leftarrow v]$.

– copy constant:

$$\Gamma \vdash s : \mathit{copyc}(c, i) \to s[i \leftarrow c]$$

– copy local:

$$\Gamma \vdash s : \mathit{copy}(i, j) \to s[j \leftarrow s(i)]$$

– copy from global to local memory:

$$\Gamma \vdash s : \mathit{gcopy}(g, i) \to s[i \leftarrow s_{\mathrm{global}}(g)] \qquad \text{if } g < |s_{\mathrm{global}}|$$

– copy from local to global memory:

$$\Gamma \vdash s : \mathit{copyg}(i, g) \to s[s_{\mathrm{global}}(g) \leftarrow s(i)] \qquad \text{if } g < |s_{\mathrm{global}}|$$

– conditional:

$$\frac{\begin{array}{c}s(i) = \mathit{NIL} \\ \Gamma \vdash s : f \to q\end{array}}{\Gamma \vdash s : \mathit{itef}(i, t, f) \to q} \qquad \frac{\begin{array}{c}s(i) \neq \mathit{NIL} \\ \Gamma \vdash s : t \to q\end{array}}{\Gamma \vdash s : \mathit{itef}(i, t, f) \to q}$$

– sequential composition:

$$\frac{\Gamma \vdash s : c \rightarrow q}{\Gamma \vdash s : sq(c) \rightarrow q} \qquad \frac{\begin{array}{c}\Gamma \vdash s : c_1 \rightarrow q_1 \\ \Gamma \vdash q_1 : sq(c_2, \ldots, c_n) \rightarrow q\end{array}}{\Gamma \vdash s : sq(c_1, \ldots, c_n) \rightarrow q} \quad \text{if } n \geq 2$$

– function call:

$$\frac{\Gamma \vdash s[s_{\text{base}} \leftarrow s_{\text{base}} + i] : \mathit{body} \rightarrow q}{\Gamma \vdash s : \mathit{fcall}(h, i) \rightarrow q[q_{\text{base}} \leftarrow s_{base}]} \quad \text{if } [h \leftarrow \mathit{body}] \in \Gamma$$

– unary/binary operators:

$$\frac{s(i) : \mathit{uop} \rightarrow v}{\Gamma \vdash s : \mathit{uop}(i) \rightarrow s[i \leftarrow v]} \qquad \frac{s(i), s(i+1) : \mathit{bop} \rightarrow v}{\Gamma \vdash s : \mathit{bop}(i) \rightarrow s[i \leftarrow v]}$$

– while loop:

$$\frac{\Gamma \vdash s : c \rightarrow q \ (q(i) = \mathit{NIL})}{\Gamma \vdash s : \mathit{while}(i, c, b) \rightarrow q} \qquad \frac{\begin{array}{c}\Gamma \vdash s : c \rightarrow r \ (r(i) \neq \mathit{NIL}) \\ \Gamma \vdash r : b \rightarrow t \\ \Gamma \vdash t : \mathit{while}(i, c, b) \rightarrow q\end{array}}{\Gamma \vdash s : \mathit{while}(i, c, b) \rightarrow q}$$

– input/output:

$$\Gamma \vdash s : \mathit{read\_char}(i) \rightarrow s[i \leftarrow \text{first}(s_{\text{input}}), s_{\text{input}} \leftarrow \text{rest}(s_{\text{input}})]$$

$$\Gamma \vdash s : \mathit{peek\_char}(i) \rightarrow s[i \leftarrow \text{first}(s_{\text{input}})]$$

$$\Gamma \vdash s : \mathit{print\_char}(i) \rightarrow s[s_{\text{output}} \leftarrow s_{\text{output}} {+\!\!+} s(i)] \quad \text{if } s(i) \in \mathit{char}$$

– *list**

$$\frac{n \geq 2}{\Gamma \vdash s : \mathit{list}{*}(n, i) \rightarrow s[i \leftarrow \mathit{cons}(s(i), \ldots \mathit{cons}(s(i+n-2), s(i+n-1)) \ldots)]}$$

$$\Gamma \vdash s : \mathit{list}{*}(1, i) \rightarrow s$$

C Compilation of ComLisp Forms

$\mathcal{C}_{\text{form}}(e, \gamma, \rho, k)$ is defined inductively on e:

- $\mathcal{C}_{\text{form}}(\mathit{abort}, \gamma, \rho, k) = \mathit{abort}$
- $\mathcal{C}_{\text{form}}(c, \gamma, \rho, k) = \mathit{copyc}(c, k)$
- $\mathcal{C}_{\text{form}}(x, \gamma, \rho, k) = \begin{cases} \mathit{copy}(\rho(x), k) & \text{if } \rho(x) \text{ is defined} \\ \mathit{gcopy}(\gamma(x), k) & \text{otherwise} \end{cases}$
- $\mathcal{C}_{\text{form}}(x := e, \gamma, \rho, k) = \begin{cases} \mathit{sq}(\mathcal{C}_{\text{form}}(e, \gamma, \rho, k), \mathit{copy}(k, \rho(x))) & \text{if } \rho(x) \text{ is defined} \\ \mathit{sq}(\mathcal{C}_{\text{form}}(e, \gamma, \rho, k), \mathit{copyg}(k, \gamma(x))) & \text{otherwise} \end{cases}$
- $\mathcal{C}_{\text{form}}(\mathit{progn}([]), \gamma, \rho, k) = \mathit{copyc}(\mathit{NIL}, k)$

- $\mathcal{C}_{\text{form}}(\mathit{progn}(e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \vdots\\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k)\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{if}(e_1,e_2,e_3),\gamma,\rho,k) = \mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \mathit{itef}(k,\mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k),\mathcal{C}_{\text{form}}(e_3,\gamma,\rho,k))\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{while}(e_1,e_2),\gamma,\rho,k) = \mathit{while}(k,\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k),\mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k))$
- $\mathcal{C}_{\text{form}}(\mathit{call}(f,()),\gamma,\rho,k) = \mathit{fcall}(f,k)$
- $\mathcal{C}_{\text{form}}(\mathit{call}(f,e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \vdots\\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k+n-1)\\ \mathit{fcall}(f,k)\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{uop}(e),\gamma,\rho,k) = \mathit{sq}(\mathcal{C}_{\text{form}}(e,\gamma,\rho,k),\mathit{uop}(k))$
- $\mathcal{C}_{\text{form}}(\mathit{bop}(e_1,e_2),\gamma,\rho,k) = \mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k+1)\\ \mathit{bop}(k)\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{let}((),e),\gamma,\rho,k) = \mathcal{C}_{\text{form}}(e,\gamma,\rho,k)$
- $\mathcal{C}_{\text{form}}(\mathit{let}(x_1=e_1,\ldots,x_n=e_n;e),\gamma,\rho,k) =$
  $\mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \vdots\\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,(k+n-1))\\ \mathcal{C}_{\text{form}}(e,\gamma,\rho[x_1\leftarrow k,\ldots x_n\leftarrow(k+n-1)],k+n)\\ \mathit{copy}(k+n,k)\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{list{*}}(e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\begin{pmatrix}\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k)\\ \vdots\\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k+n-1)\\ \mathit{list{*}}(n,k)\end{pmatrix}$
- $\mathcal{C}_{\text{form}}(\mathit{cond}(),\gamma,\rho,k) = \mathit{copyc}(\mathit{NIL},k)$
- $\mathcal{C}_{\text{form}}(\mathit{cond}(p_1\to e_1,\ldots,p_n\to e_n),\gamma,\rho,k) =$
  $\mathit{sq}(\mathcal{C}_{\text{form}}(p_1,\gamma,\rho,k),$
  $\mathit{itef}(k,\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k),\mathcal{C}_{\text{form}}(\mathit{cond}(p_2\to e_2,\ldots,p_n\to e_n),\gamma,\rho,k)))$
- $\mathcal{C}_{\text{form}}(\mathit{read\_char},\gamma,\rho,k) = \mathit{read\_char}(k)$
- $\mathcal{C}_{\text{form}}(\mathit{peek\_char},\gamma,\rho,k) = \mathit{peek\_char}(k)$
- $\mathcal{C}_{\text{form}}(\mathit{print\_char}(e),\gamma,\rho,k) = \mathit{sq}(\mathcal{C}_{\text{form}}(e,\gamma,\rho,k),\mathit{print\_char}(k))$

Some remarks to this definition:

- for sequences $\mathit{progn}(e_1,\ldots,e_n)$ all values are stored at the same relative position in the current stack frame since the value of a sequence is the value of its rightmost subexpression e_n. The intermediate values do not have to be preserved.
- for the compilation of a *let*-form, new variable bindings are allocated in the current stack frame beginning at relative position k. The body is evaluated to return its value at $k+n$ where n is the number of new local bindings. A final copy instruction moves the result value to the desired location k.
- a *cond*-form is compiled into a nested conditional.