A Mechanically Verified Compiling Specification for a Realistic Compiler*

A. Dold, F. W. von Henke, V. Vialard
Abt. Künstliche Intelligenz
Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
{dold,vhenke,vincent}@informatik.uni-ulm.de

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

Abstract

We report on a large formal verification effort in mechanically proving correct a compiling specification for a realistic 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 five steps through a series of intermediate languages. In the first phase, 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 $\mathbf{C}^{int}$ 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 compiler's backend, first control structures (loops, conditionals) of the intermediate language $\mathbf{C}^{int}$ are implemented by linear assembler code with relative jumps, the infinite memory model of $\mathbf{C}^{int}$ is realized on the finite Transputer memory, and the basic $\mathbf{C}^{int}$ statements for accessing the stack and heap are implemented by a sequence of assembler instructions. The fourth phase consists of the implementation of code instructions with large and negative word operands, while the last phase is concerned with the integration of the assembly program into the memory. 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 and real target architectures.

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

Contents

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.

Verifix [GZ99, Goe97] 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. *Verifix* assumes hardware to behave correctly as described in the instruction manuals. The main achievements of the project are

- the definition of appropriate realistic notions of correctness for the specification and implementation level [GL01b, MOW99].
- the application of general approved compiler construction techniques including the use of (unverified) compiler generation tools (e.g. Lex, Yacc, BEG) [GZ99].
- to ensure the correctness of the generated code, the technique of algorithmic and a-posteriori program checking is applied using verified program checkers [GGZ98].
- the construction of a trusted initial compiler for a realistic imperative high-level sytem programming language [GH98a, GH98c] which serves as a sound bootstrapping and implementation basis (e.g. for the program checkers) and lifts proof obligations from machine code level to the more abstract source code level.
- the use of automated proof systems to support the different verification tasks of the project (e.g. [DV00, DvHPR97]).

As already noted in 1986 by Chirica and Martin [CM86], 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. The entire correctness proof of a compiler executable running on a host machine can be modularized in three tasks where every step is represented by a corresponding commuting diagram (see Fig. 1).

$$\begin{tikzcd}
\text{semantics spaces} & \mathrm{Sem}_{\mathsf{SL}} \arrow[r, harpoon, "\sqsubseteq"] & \mathrm{Sem}_{\mathbf{TL}} & \text{semantics relation } \sqsubseteq \\
\text{abstract programs} & \mathbf{SL} \arrow[u, harpoon, "{[\![\cdot]\!]_{\mathsf{SL}}}"] \arrow[r, harpoon, "\mathcal{C}"'] \arrow[d, harpoon, "\varphi^{\mathsf{SL}}_{\mathsf{SL}'}"'] & \mathbf{TL} \arrow[u, harpoon, "{[\![\cdot]\!]_{\mathsf{TL}}}"'] \arrow[d, harpoon, "\varphi^{\mathsf{TL}}_{\mathsf{TL}'}"] & \text{compiling specification (relation) } \mathcal{C} \\
\text{concrete programs repr. as } \mathbf{HL}\text{-data} & \mathbf{SL}' \arrow[r, harpoon, "{[\![\,\tau_{\mathsf{HL}}\,]\!]_{\mathsf{HL}}}"'] \arrow[d, harpoon, "\varphi^{\mathsf{SL}'}_{\mathsf{SL}''}"'] & \mathbf{TL}' \arrow[d, harpoon, "\varphi^{\mathsf{TL}'}_{\mathsf{TL}''}"] & \text{high level compiler implementation } \tau_{\mathsf{HL}} \\
\text{concrete programs repr. as } \mathbf{HML}\text{-data} & \mathbf{SL}'' \arrow[r, harpoon, "{[\![\,\tau_{\mathsf{HML}}\,]\!]_{\mathsf{HML}}}"'] & \mathbf{TL}'' & \text{low level compiler implementation } \tau_{\mathsf{HML}}
\end{tikzcd}$$

Figure 1. Three steps for correct compiler implementation

1. Specification of a compiling relation $\mathcal{C}$ between abstract source and target languages **SL** and **TL**, and compiling (specification) verification w.r.t. an appropriate semantics relation $\sqsubseteq$ between language semantics $[\![\cdot]\!]_{\mathsf{SL}}$, $[\![\cdot]\!]_{\mathsf{TL}}$. Compiling specifications $\mathcal{C}$ typically allow for more than only one target program, and also compiler program semantics may be non-deterministic. Hence, relations are utilized for this purpose ($A \rightharpoonup B$ denotes the domain of relations between A and B). Notice

that only well-formed programs (that is, static semantically correct programs) have a (dynamic) semantics.

2. Implementation of a corresponding compiler program τ_{HL} in a high level host language **HL** close to the specification language, and high level compiler implementation verification (w.r.t. $\mathcal{C}$ and to program representations $\varphi^{\mathsf{SL}}_{\mathsf{SL}'}$ and $\varphi^{\mathsf{TL}}_{\mathsf{TL}'}$). Data and program representations, which map abstract programs to concrete program representations and to their string representations (indicated by primes), are in general relations. This step corresponds to programming. Here, either a constructive approach can be applied which transforms the specification $\mathcal{C}$ into a high-level implementation by applying correctness-preserving development steps or the program is verified a-posteriori using classical program verification techniques.
3. Implementation of a corresponding compiler executable τ_{HML} written in binary host machine language **HML**, and low level compiler implementation verification (with respect to $[\![\, \tau_{\mathsf{HL}} \,]\!]_{\mathsf{HL}}$ and program representations $\varphi^{\mathsf{SL}'}_{\mathsf{SL}''}$ and $\varphi^{\mathsf{TL}'}_{\mathsf{TL}''}$). This step can be established by a trusted initial compiler, or by syntactical a-posteriori result checking. It is important to note that this last step is absolutely necessary to ensure correctness of the generated executable. If this step is neglected, security relevant intentional errors such as Trojan Horses are hard to find as demonstrated by the construction of a malicious compiler executable [Goe00, Tho84] and this might have harmful consequences.

The initial correct bootstrap compiler which has been constructed in the context of the Verifix 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, that is, source and host languages are identified (**SL** = **HL** = ComLisp). ComLisp programs are systems of mutually recursive function procedures, working on the domain of s-expressions, which is a recursive dynamic data type suitable for program and term manipulation. The target language and host machine languages are identified as well: **TL** = **HML** = Transputer. The construction process of our initial correct ComLisp compiler according to the three tasks above consists of the following steps:

- define and prove correct the compiling specification relating ComLisp programs with binary Transputer code according to a suitable correctness criterion.

- construct a correct compiler implementation in the source language ComLisp itself by applying a transformational constructive approach which builds a correct implementation from the specification by stepwise applying correctness-preserving development steps [Dol00].

- use an existing (unverified) implementation of the source language (here: some arbitrary Common Lisp compiler) to execute the program. Execute the program, apply it 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 [GL01a, Hof98]. This closes the gap between high-level implementation and executable code and lifts proof obligations from machine code level to the much more abstract level of ComLisp code.

The size and complexity of the verification task in constructing such 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 [ORSvH95] to support the verification of the compiling specification and the construction process of a compiler implementation in the source language.

This report is concerned with the mechanical verification of the compiling specification (*compiling verification*) for the ComLisp compiler (the first task above). This proof has completely been mechanized and is one of the largest case-studies in formal verification we are aware of.

There are many different possibilities in order to define what a correct compiling specification and compiler implementation means. It substantially depends on the application context in which the compiler is used. As mentioned above, one of the results of the *Verifix* project is the development of a realistic correctness criterion for sequential imperative source languages and concrete target processors which takes the finite resource limitations of the target architecture into account. Computations may possibly fail and run into errors which should be one of an acceptable error. On the other hand, a practical compiler may fail in most cases, anyway, on nearly every (sufficiently large) source program.

A practical compiler executable cannot implement every source program behaviorally equivalently on the target machine. There are trivial correct compilers which either always fail themselves or generate code which always fails. Such a compiler is of no use at all, but it is impossible, to define a rigorous notion of *useful correct implementation.* Another important fact is, that compiler constructors are not responsible if source programs are not admissible (that is, they do not meet pre-conditions such as well-formedness) which would lead to unacceptable outcomes. These considerations lead to a family of notions of correctness which is parameterized with the set Ω of possible errors, consisting of two disjoint sets of unacceptable and acceptable errors ($\Omega = A \mathbin{\dot\cup} U$). Informally, the notion states that for admissible inputs either the result of the target program semantics is correct (representation of a corresponding source program output) or the target program execution aborts with an acceptable (resource) error (see [GL01b] for the formal definitions).

For the compiling verification of the ComLisp compiler outlined in this report, we make use of a specialization of the general notion, where the set U of unacceptable errors is empty and all acceptable errors are identified. This notion then defines *preservation of partial program correctness*, that is, partial correct source programs are mapped onto partial correct target programs.

$$\begin{tikzcd}
D_i^s \arrow[r, "f_s"] \arrow[d, "\rho_i"'] \arrow[dr, phantom, "\subseteq"] & D_o^s \arrow[d, "\rho_o"] \\
D_i^t \arrow[r, "f_t"'] & D_o^t
\end{tikzcd}$$

Figure 2. Preservation of partial program correctness

We formalize operational program semantics by relations $f \subseteq D_i \times D_o$ between input and output domains. Let f_s be a source and f_t a target program semantics, and let ρ_i and ρ_o be (input and output) data representation relations which relate source language domains with target language domains. Then the notion of correctness is defined as follows:

Definition 1.1 (Preservation of Partial Correctness). *f_t is a correct translation of f_s, if the diagram in Figure 2 commutes in the following sense:*

$$\rho_i \,;\, f_t \;\subseteq\; f_s \,;\, \rho_o$$

where ; *denotes (diagrammatic) relational composition.*

Using this correctness criterion, the overall goal of the verification task outlined in this report is to show the correctness of a compiling relation $\mathcal{C}(p, q)$ relating well-formed ComLisp programs p with binary Transputer code q. The semantics of a ComLisp program (denoted by p_{ComLisp}) is a relation between input character sequences is and output character lists ol; $p_{\mathbf{TC_0}}$ denotes the semantics of binary Transputer programs (a relation between byte sequences and byte lists). Figure 3 illustrates the main theorem. Here, the input and output domain of the source language is the set of character sequences $D_i^s = seq[char]$ and character lists $D_o^s = char^*$, respectively, the input and output domain of the target language is the set of byte sequences $D_i^t = seq[byte]$ and byte lists $D_o^t = byte^*$, the data representation relations ρ_i and ρ_o are functions mapping character sequences and lists to byte sequences and lists by means of the character codes. Hence, the subset inclusion in definition 1.1 becomes an implication in the following theorem:

Theorem 1.1 (Main Correctness Theorem).
$\forall p, q, is, ol.\ wf(p) \ \wedge\ \mathcal{C}(p, q) \Rightarrow p_{\mathbf{TC_0}}(q)(char2byte(is))(char2byte(ol)) \Rightarrow p_{\text{ComLisp}}(p)(is)(ol)$

$$\begin{tikzcd}
seq[char] \arrow[r, "p_{\text{ComLisp}}"] \arrow[d, "char2byte"'] \arrow[dr, phantom, "\subseteq"] & char^* \arrow[d, "char2byte"] \\
seq[byte] \arrow[r, "p_{\mathbf{TC_0}}"'] & byte^*
\end{tikzcd}$$

Figure 3. Correctness of Compiling Specification

Since the distance between a high-level ComLisp program and the final binary Transputer code is very large, the compilation is modularized suitably into five steps such that in each phase a specific compilation or implementation aspect is realized. The purpose of this verification project is to establish a fully correct bootstrap compiler, henceforth the intermediate languages are chosen in a way such that each of the three tasks, namely the compiling verification, high-level compiler implementation and verification of the binary executable can be realized. Each compilation phase is implemented and verified separately. The five steps are chosen as follows:

$$\text{ComLisp} \rightarrow \text{SIL} \rightarrow \mathbf{C}^{int} \rightarrow \text{TASM} \rightarrow \mathbf{TC}_1 \rightarrow \mathbf{TC}_0$$

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 $\mathbf{C}^{int}$ 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 and constitute the compilers frontend. In the compilers backend, in the next step, control structures of $\mathbf{C}^{int}$ (loops, conditionals, etc.) are implemented by linear assembler code with jumps in TASM, the Cint memory model is integrated into the finite Transputer memory, and the Cint statements for accessing the stack and heap are implemented by a sequence of TASM instructions. The compilation from TASM to $\mathbf{TC}_1$ consists of the implementation of assembler instructions containing large or negative operands by a sequence of pfix/nfix chains and the transformation of the assembler code into binary Transputer code, and finally the program is integrated into the Transputer memory. Note that the last two steps are introduced only for mechanical verification purposes to logically separate the two aspects. No compilation is carried out for the last step and hence the last two phases are in fact implemented in one pass.

This report is organized as follows. In Sect. 2, other related compiler verification approaches found in the literature are briefly presented. The following sections then present the main aspects of the formalizations and verifications of each of the five compilation phases. In each section, first the formalizations of the languages, that is, their abstract syntax and semantics are outlined. Operational semantics in a structural operational style is used for ComLisp and for each of the intermediate languages in the compiler's frontend. The different abstract Transputer formalizations make use of a transition semantics. Then the specific compilation process and its correctness is described. We further provide statistics concerning the mechanical verification effort. Having proved correct each of the compilation steps, finally, the different phases have to be combined in order to accomplish the global correctness proof, relating ComLisp source programs with binary Transputer code as stated in theorem 1.1. Section 8 is concerned with this issue. Finally, in Sect. 9 we summarize our work and discuss the results of this formal verification project.

2 Related Work

Verification of compiler correctness is a much-studied area starting with the work by McCarthy and Painter in 1967 [MP67], where a simple compiler for arithmetic 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 [Pol81, Joy89, Bro92, Moo89, Cur94, BS98]. 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 [Moo89]. 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 [GMR$^+$92] 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 [Cur94] 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 [Sch99]. A (small-step) ASM semantics is used for the languages.

More recently, M. Strecker [Str02] has carried out a formal verification of the compilation process from a subset of the Java source language into Java bytecode using the Isabelle proof environment. The verification effort here is comparable to the effort in proving correct one of the compilation phases of our bootstrap compiler.

3 Compiling ComLisp to Stack Intermediate Code

3.1 ComLisp

A ComLisp program consists of

- a declaration part consisting of a list of function names F_l, a list of symbols (except NIL and T) S_l, and a list of non-atomic program constants C_l. The declaration part is syntactical sugar and semantically irrelevant, used to facilitate one pass compilation.
- a list of global variables,
- a list of possibly mutual recursive function definitions,
- a main form.

The abstract syntax of ComLisp is given as follows:

$$\begin{array}{ccl}
p & ::= & d; x_1, \ldots, x_k; f_1, \ldots, f_n; e \\
d & ::= & F_l; S_l; C_l \\
f & ::= & h(x_1, \ldots, x_m) \leftarrow e \\
e & ::= & \mathit{abort} \mid c \mid x \mid x := e \mid \mathit{progn}(e_1, \ldots, e_n) \mid \mathit{if}(e_1, e_2, e_3) \mid \mathit{do}(e_1, e_2) \mid \mathit{call}(h, e_1, \ldots, e_n) \mid \\
 & & \mathit{uop}(e) \mid \mathit{bop}(e_1, e_2) \mid \mathit{let}(x_1 = e_1, \ldots, x_n = e_n; e) \mid \mathit{list}{*}(e_1, \ldots, e_n) \mid \\
 & & \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n) \mid \mathit{read\_char} \mid \mathit{peek\_char} \mid \mathit{print\_char}(e) \\
c & ::= & i \mid \mathit{NIL} \mid T \mid a \mid sb \mid st \mid (c_1 \ . \ c_2) \quad (a \in \mathit{Char};\ sb, \mathit{NIL}, T \in \mathit{Symbol};\ st \in \mathit{String}) \\
\mathit{uop} & ::= & \mathit{car} \mid \mathit{cdr} \mid \mathit{nullp} \mid \mathit{consp} \mid \mathit{symbolp} \mid \mathit{characterp} \mid \mathit{integerp} \mid \\
 & & \mathit{length} \mid \mathit{char\_code} \mid \mathit{code\_char} \mid \mathit{symbol\_name} \mid \mathit{intern} \mid \mathit{coerce\_string} \\
\mathit{bop} & ::= & \mathit{cons} \mid + \mid * \mid < \mid \geq \mid \mathit{floor} \mid \mathit{mod} \mid \mathit{eql} \mid \mathit{aref}
\end{array}$$

where $n, m, k \geq 0; t \geq 1$, $c : SExpr$, $x, x_1, \ldots, x_n : Ident$ are identifiers, h is a function name, $f, f_1, \ldots, f_n$ are function definitions, $\mathit{uop} : Unop$ are unary operators, $\mathit{bop} : Binop$ are binary operators, and $e, e_1, \ldots, e_n$ and $p_1, \ldots, p_n$ are expressions (forms).

ComLisp statements (expressions or forms) include the *abort* statement, s-expression constants, variables, assignments, sequential composition, conditional, do-loop, call of user defined functions, call of built-in unary and binary ComLisp operators, local let-blocks, cond-instruction, and instructions for reading from the input sequence and writing to the output. The form *list∗* constructs a s-expression list from its evaluated arguments. There must be at least one argument.

The only available datatype is the type of s-expressions which are binary trees built with constructor "cons", where the leafs are either integers, characters, strings, or symbols. The set of symbols include T and NIL with $T \neq NIL$. In addition, there are sets of unary and binary operators defined on s-expressions. The 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. $+, *, floor$). ComLisp operators denote partial functions on s-expressions which is expressed by two relations: relation $v_1 : \mathit{uop} \rightarrow v_2$ for unary operators *uop*, and $v_1, v_2 : \mathit{bop} \rightarrow v$ for binary operators. For example, the first relation expresses that the application of unary operator *uop* to s-expression v_1 is defined, terminates, and yields s-expression v_2 as result.

Having defined the abstract syntax of ComLisp, we now focus on the semantics. First, the static semantics of ComLisp programs, declarations, and forms are specified by means of several well-formedness predicates for forms, function definitions, and programs. A ComLisp form is *well-formed* with respect to a function environment Γ (a list of function definitions), a local variable environment ζ (a list of formal parameters), a list of global variables γ, a list of symbols sl and a list of non-atomic constants cl, if

- the list of local and global variables are disjoint.
- all variables are declared, that is, occur in ζ or γ.
- all program constants and symbols are declared in the symbol list sl and list of constants cl, respectively.
- each user-defined function is called with the correct number of arguments (correct parameter passing) and the function identifiers are declared in Γ.

Formally, a relation $wf(e, \zeta, \gamma, \Gamma, cl, sl)$ is defined inductively on the structure of form e. Its formal definition is straightforward and omitted here.

A function environment Γ is well-formed with respect to a list of global variables γ, a list of non-atomic constants cl, and a list of symbols sl, 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 Γ, its local parameter list, γ, cl, and sl. This is specified by a predicate $wf(\Gamma, \gamma, cl, sl)$.

Finally, a ComLisp program is well-formed ($wf(p)$), if the function identifiers occuring in the function definitions correspond to the list of names in the declaration part and are disjoint, and all function definitions are well-formed with respect to the global variables, and the declarations, and the main program is well-formed w.r.t. to the function declarations, the global variables, and the empty local parameter list, the constant list, and the symbol list.

For all intermediate languages occuring 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). However, in order to illustrate the definition of a small-step structural operational semantics for a high-level language, we provide such an additional semantics for ComLisp in the appendix A.2. Notice that we do not make use of this small-step semantics for the compiling verification outlined in this report.

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):

$$state_{\mathrm{CL}} ::= seq[char] \times char^* \times (Ident \rightarrow SExpr)$$

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 $s_{var} : Ident \rightarrow SExpr$. 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 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:

$$\Gamma \vdash s : e \rightarrow (v, q)$$

where Γ is a function environment, s, q are states, and v is a value. The relation expresses 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 $\rightarrow$ 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:

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

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_{ComLisp} between input streams is and output lists ol. $p_{\text{ComLisp}}(p)(is)(ol)$ 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:

$$p_{\text{ComLisp}}(p)(is)(ol) ::= \exists v, q.\ (\Gamma \vdash (is, [], \lambda x.NIL) : e \rightarrow (v, q)) \ \wedge\ (q_{\text{output}} = ol)$$

3.2 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

- a declaration part (used to facilitate one pass compilation), consisting of a number denoting the length of the global memory, and (like for ComLisp), the list of procedure names, the list of program symbols and non-atomic constants.
- 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. The unary and binary operators are the same as for ComLisp.

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

The SIL statements have the following informal meaning:

- $abort$ immediately aborts the execution.
- $copyc(c, i)$ writes a constant c to stack relative position i.
- $copy(i, j)$ copies the content at stack relative position i to stack relative position j.
- $gcopy(g, i)$ copies the content of global memory cell g to stack relative position i.
- $copyg(i, g)$ copies the content of stack relative position i to the global memory at position g.
- $fcall(f, i)$ is a subroutine call that executes the code associated to f where the frame pointer is increased by i. After the body code has been executed the frame pointer is reset to its old value.
- $itef(i, t, f)$ executes instruction f if the content of stack relative position i is NIL, otherwise t is executed.
- $sq(s_1, \ldots, s_n)$ executes the instructions $s_1, \ldots, s_n$ in sequence.
- $uop(i)$ applies the unary operator uop to the stack cells at relative position i.
- $bop(i)$ applies the binary operator bop to the stack cells at relative positions i and $i + 1$.
- $do(i, c, b)$ executes the statement c, and then terminates if the content of stack relative position i is not NIL, otherwise executes the body b and then the loop again.

- $read\_char(i)$ writes the head of the input stream at relative position i, and removes the head from the stream
- $peek\_char(i)$, same as $read\_char(i)$ but does not alter the input stream.
- $print\_char(i)$ writes the content at relative position i to the output stream.
- $list{*}(n, i)$ construct an s-expression tree with n arguments and writes it at relative position i.

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).

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

As for ComLisp, an evaluation semantics for SIL statements is defined as the smallest relation

$$\Gamma \vdash s : cmd \rightarrow q$$

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:

$$p_{\mathbf{SIL}}(p)(is)(ol) ::= \exists q.\ (\Gamma \vdash init : s \rightarrow q)\ \wedge\ (q_{\mathrm{output}} = ol)$$

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

3.3 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. 3.5). 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.

3.4 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. 4). 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.

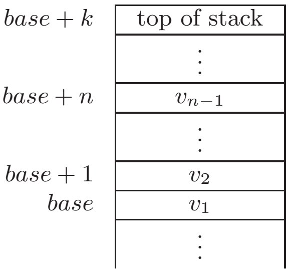


Figure 4. Parameter passing on the stack

- $\mathcal{C}_{\text{form}}(e, \gamma, \rho, k)$ 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 to be removed, leaving only the result on top (achieved by a copy instruction from position n to 0).

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

- 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:

$$\mathcal{C}_{\text{prog}}(p) ::= \mathcal{C}_{\text{defs}}(\Gamma)(\gamma); \mathcal{C}_{\text{form}}(e, \gamma, [], 0)$$

3.5 Correctness of the Compilation Process

$$\begin{tikzcd}
\mathit{seq}[\mathit{char}] \arrow[r, "p_{\mathrm{ComLisp}}"] \arrow[d, "\mathit{id}"'] \arrow[dr, phantom, "\supseteq"] & \mathit{char}^* \arrow[d, "\mathit{id}"] \\
\mathit{seq}[\mathit{char}] \arrow[r, "p_{\mathbf{SIL}}"'] & \mathit{char}^*
\end{tikzcd}$$

Figure 5. ComLisp program compilation correctness

For this first phase there are no resource restrictions and the 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* (see Fig. 5).

Theorem 3.1 (Correctness of Program Compilation).
$\forall p, \mathit{is}, \mathit{ol}.\ \mathit{wf}(p) \Rightarrow p_{\mathbf{SIL}}(\mathcal{C}_{\text{prog}}(p))(\mathit{is})(\mathit{ol}) \Rightarrow p_{\mathrm{ComLisp}}(p)(\mathit{is})(\mathit{ol})$

Unfolding $p_{\mathbf{SIL}}$ and p_{ComLisp}, 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_o relates ComLisp output states (v, σ') with SIL states s'. Figure 6 illustrates the correctness property for forms by means of a commuting diagram.

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

Figure 6. Correctness property for the compilation of ComLisp forms

The relations are parameterized with a list of global variables γ, the local compile time environment ρ, and the current top of stack position k. Relation ρ_i 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_o 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:

1. $\rho_i(\gamma, \rho, k)(\sigma, s) ::=$
   $[\forall x \in \mathit{dom}(\rho).\ (\rho(x) < k)\ \wedge\ (\sigma(x) = s_{\mathrm{local}}(s_{\mathrm{base}} + \rho(x)))]\ \wedge$
   $[\forall x \in \gamma.\ (\gamma(x) < |s_{\mathrm{global}}|)\ \wedge\ (\sigma(x) = s_{\mathrm{global}}(\gamma(x)))]\ \wedge$
   $(s_{\mathrm{input}} = \sigma_{\mathrm{input}})\ \wedge\ (s_{\mathrm{output}} = \sigma_{\mathrm{output}})$

2. $\rho_o(\gamma, \rho, k)(v, \sigma', s') ::= (s'_{\mathrm{local}}(s'_{\mathrm{base}} + k) = v)\ \wedge\ (\rho_i(\gamma, \rho, k)(\sigma', s'))$

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.

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

   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. 6 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_i 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_o and the target states and source states invariants hold. Note that the additional parameters cl and sl denoting the list of non-atomic program constants and symbols, respectively, are required for the well-formedness predicates.

Definition 3.1 (Correctness Property for Form Compilation).
$correct\_prop(\Gamma, \gamma, \zeta, \rho, cl, sl, k)(e) ::=$
$\forall \sigma, s, s'.\ wf(\Gamma, \gamma, cl, sl)\ \wedge\ wf(e, \zeta, \gamma, \Gamma, cl, sl)\ \wedge\ injective?(\rho)\ \wedge$
$(dom(\rho) = \zeta)\ \wedge\ \rho_i(\gamma, \rho, k)(\sigma, s)\ \wedge\ (\mathcal{C}_{\mathrm{defs}}(\Gamma)(\gamma) \vdash s : \mathcal{C}_{\mathrm{form}}(e, \gamma, \rho, k) \rightarrow s')$
$\Rightarrow \exists v, \sigma' : (\Gamma \vdash \sigma : e \rightarrow (v, \sigma'))\ \wedge\ \rho_o(\gamma, \rho, k)(v, \sigma', s')\ \wedge\ invar?(\rho, k)(s, s')\ \wedge\ source\_invar?(\zeta, \gamma)(\sigma, \sigma')$

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

Theorem 3.2 (Correctness of Form Compilation).
$\forall e, \Gamma, \gamma, \zeta, \rho, cl, sl, k.\ correct\_prop(\Gamma, \gamma, \zeta, \rho, cl, sl, k)(e)$

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:

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

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 3.2 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 3.1):

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_i that the output source and target states are related by ρ_o (Note that ρ_o is defined by means of ρ_i).

   (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.

Table 1. Formalization and verification statistics for the first phase

	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

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*∗.

4 Data and Operation Refinement

The second compilation phase is a classical data and operation refinement step, where ComLisp data structures (s-expressions) and operators are implemented in linear integer memory using a runtime stack and a heap for representing dynamic data structures.

4.1 A modified Semantics for SIL

To verify the compiling relation for this second compilation step, the semantics of SIL has to be slightly modified. An additional pointer, referencing the top of stack has to be integrated into the SIL state. This is required since SIL is not a (classical) stack machine with standard push and pop instructions, instead it contains copy instructions which allow arbitrary read and write access within the current stack frame. However, as SIL is only used as an intermediate language for compiling ComLisp programs, it is well-known (from the verification outlined in the last section) that any correctly compiled ComLisp program runs like a program on a classical stack machine. In order to be able to modularly verify the two compilation phases, knowledge about the first compilation phase has to be integrated into SIL's semantics. An additional stack pointer is utilized for this purpose and a weaker SIL semantics is defined. If a SIL program is not in correspondence with a stack machine behavior (that is, the stack pointer has an erroneous value), the SIL machine may step non-deterministically into any possible successor state. The modified SIL state includes the additional top-of-stack pointer of type *Int*.

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

The rules for the relation $\Gamma \vdash s : cmd \to q$ have to be updated such that the top pointer is modeled suitably. As an example how the stack top pointer is integrated into the semantics, consider the semantics for the SIL statement *copy*. It is specified by means of two rules. The first rule specifies the normal case. The top pointer always points above $s_{\mathrm{base}} + i$ and the target location j (relative to the frame pointer *base* could be equal to the top pointer plus 1. In this case, the copy instruction corresponds to a *push* operation on a classical stack machine. The pointer is updated to point to the maximum of i and j. On the other hand, the second rule specifies the erroneous behavior. In this case the semantics is defined non-deterministically and any successor state q will be allowed.

$$\frac{(s_{\text{base}} + i \leq s_{\text{top}}) \wedge (s_{\text{base}} + j \leq s_{\text{top}} + 1)}{\Gamma \vdash s : \mathit{copy}(i,j) \rightarrow s[j \leftarrow s(i), \text{top} \leftarrow s_{\text{base}} + \max(i,j)]} \qquad \frac{(s_{\text{base}} + i > s_{\text{top}}) \vee (s_{\text{base}} + j > s_{\text{top}} + 1)}{\Gamma \vdash s : \mathit{copy}(i,j) \rightarrow q}$$

The complete set of rules for this modified semantics can be found in the appendix D. Using a modified semantics for the verification of this phase, effects the global correctness proof, since the first two phases cannot be combined due to different semantics of SIL. Thus, it is required to either

- formally relate both SIL semantics or to
- repeat the verification of the first compilation phase from ComLisp to SIL using the modified SIL semantics.

We have decided to use the second choice which has turned out to be much easier since nearly all of the proofs could be reused without adaption.

4.2 The intermediate language $\mathbf{C}^{int}$

$\mathbf{C}^{int}$, the next intermediate language for our bootstrap compiler, is used to represent the s-expressions in the SIL program and to define the core runtime system (a set of $\mathbf{C}^{int}$ procedures) which implements the SIL (and ComLisp) unary and binary operators. Like SIL, $\mathbf{C}^{int}$ has parameterless procedures. Programs, however, operate on integers of arbitrary size. Besides a stack with frame-pointer relative and random access, there is a random access integer array *heap* for implementing dynamic data structures. There are statements and expressions. A $\mathbf{C}^{int}$ program consists of a declaration part (the list of procedure names F_l, initial stack (st_0) and heap segments (h_0) used to represent the global variables and the SIL program symbols and non-atomic s-expression constants), a list of procedure declarations, and the main program (statement). Each procedure has to specify in its heading the amount of stack spaces it directly uses which can be determined statically (see below). This information is used in the next compilation phase, when compiling $\mathbf{C}^{int}$ to abstract Transputer assembler (TASM) but is not relevant for the compilation step outlined in this section.

$$\begin{array}{lll}
p & ::= & F_l; st_0; h_0; f_1, \ldots, f_n; s \\
f & ::= & h(\mathit{size}) \leftarrow s \\
s & ::= & \mathit{skip} \mid \mathit{abort} \mid \mathit{allocate}(e) \mid \mathit{set\_local}(e,i) \mid \mathit{set\_stack}(e_1,e_2) \mid \mathit{set\_heap}(e_1,e_2) \mid s_1; s_2 \mid \\
 & & \mathit{if}(e,s_1,s_2) \mid \mathit{if}(e,s_1) \mid \mathit{do}(s_1,e,s_2) \mid \mathit{call}(h,i) \mid \mathit{read\_char}(i) \mid \mathit{peek\_char}(i) \mid \mathit{print\_char}(i) \\
e & ::= & \mathit{heaptop} \mid \mathit{quotetop} \mid \mathit{stacktop} \mid i \mid \mathit{local}(i) \mid \mathit{stack}(e_1) \mid \\
 & & \mathit{heap}(e_1) \mid \mathit{unavailable}(e_1) \mid 2 * (e_1) \mid \mathit{op}(e_1,e_2) \\
\mathit{op} & ::= & + \mid * \mid - \mid \mathit{div} \mid \mathit{rem} \mid < \mid \geq \mid = \mid \neq
\end{array}$$

The statements consist of

- *allocate*(e): allocation and de-allocation of heap memory (by increasing or decrementing the heaptop pointer)
- statements for accessing the stack and heap ($\mathit{set\_local}(e,i)$ writes the value of expression e to stack relative position i, $\mathit{set\_stack}(e_1,e_2)$ for random access on the stack writing the value of e_1 to stack position denoted by e_2; similarly $\mathit{set\_heap}(e_1,e_2)$ for random access on the heap),
- control structures (loop, conditional, sequential composition, procedure call),
- I/O: reading from the input sequence and writing to the output list.

The expressions of $\mathbf{C}^{int}$ consist of expressions accessing the current top of heap, stack, and the top of the initial heap segment (quotetop), integer constants, local and random stack access, random heap access, and application of an unary and binary operator. In addition, the expression *unavailable*(e) has a non-deterministic semantics (it returns some arbitrary integer value if e evaluates to a defined value) and is used only in the core runtime system for memory management purpose. $\mathbf{C}^{int}$ stacks and heaps are infinite arrays which have to be implemented in finite TASM memory.

The static semantics of $\mathbf{C}^{int}$ programs, procedure definitions, and statements is specified by means of several well-formedness predicates. Informally, a statement is well-formed if every function symbol is declared, if there are no double definitions of procedure names, and if the names are different from the operator names. Each procedure specifies in its heading the amount of stack spaces it uses directly which is given by the highest relative stack index in its body. The specified amount has to be greater than or equal the actual amount.

As for SIL, a big-step semantics is specified for $\mathbf{C}^{int}$. A $\mathbf{C}^{int}$ state consists of the input stream, the output list, the stack array (consisting of the frame pointer $base : Nat$ and the stack, a function from natural numbers to integers), the heap array, and the two heap pointers *heaptop* and *quotetop* which point to heap addresses (natural numbers). The pointer *quotetop* references the end of the initial heap segment, while *heaptop* points to the index of the first free heap cell.

$$state_{\mathrm{Cint}} ::= seq[char] \times char^* \times (Nat \times (Nat \to Int)) \times (Nat \to Int) \times Adr \times Adr$$

An evaluation semantics for $\mathbf{C}^{int}$ expressions and statements is provided. Expression evaluation uses the following notation: $s : e \to v$. This expresses that expression e evaluated in state s terminates and evaluates to integer value v. Given rules for each kind of form, the semantics is defined as the smallest relation $\to$ satisfying the set of rules. Similarly, an evaluation semantics for $\mathbf{C}^{int}$ 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 $\mathbf{C}^{int}$ procedure environment Γ (a list of procedure declarations) is defined, terminates, and results in a new state q. The complete set of rules for expressions and statements is listed in the appendix E.

The semantics of a $\mathbf{C}^{int}$ program $p = F_l; st_0; h_0; \Gamma; cmd$ is given by the input/output behavior of the program defined by a relation $p_{\mathbf{C}^{int}}$ between input streams is and output lists ol. $p_{\mathbf{C}^{int}}(p)(is)(ol)$ holds if the evaluation of the main statement s in an initial state, where the input stream is given by is, the output list is empty, the base pointer is set to the length of the initial stack segment st_0, the initial stack is st_0, the initial heap segment is h_0, and both, the heaptop and quotetop pointer point to the end of the initial heap segment, terminates in a state q with output list ol.

$$p_{\mathbf{C}^{int}}(p)(is)(ol) ::= \exists q.\ (\Gamma \vdash init : cmd \to q)\ \wedge\ (q_{\mathrm{output}} = ol)$$

where the initial state is defined by $init ::= (is, [], (|st_0|, st_0), h_0, |h_0|, |h_0|)$.

4.3 Compiling SIL to $\mathbf{C}^{int}$

The main task of the compilation from SIL to $\mathbf{C}^{int}$ is a data and operation refinement step. Following classical approaches (e.g. [Jon90]), the main idea of representing an (abstract) data type D by a more concrete one Z is to introduce a subset $R \subseteq Z$ (also referred to as *data type invariant*) denoting the set of elements used to represent D together with an abstraction function ρ (also referred to as *retrieve mapping*) mapping elements satisfying the invariant R to abstract values of D. For each operation on D an implementation on Z must be defined. Correctness of operation refinement ensures that the data type invariant is preserved by the implementation and that the implementation exhibits the same behavior as the abstract operation with respect to the abstraction function ρ.

In our case, the task is to represent the s-expressions in the SIL program in linear integer memory and to define a set of $\mathbf{C}^{int}$ procedures (core runtime system) which implement the ComLisp (and SIL) unary and binary operators. An initial heap segment has to be constructed such that it is a correct representation of the non-atomic s-expressions and symbols which are specified in the declaration part of a SIL program. Analogously, an initial stack segment must be constructed which contains the global variables and a reference to the symbol table.

4.3.1 Representation of S-Expressions

Data and operation refinement for s-expressions and ComLisp operators are according to a standard scheme of mapping directed graphs to linear memory using *tagged pointers*. In $\mathbf{C}^{int}$, nodes of an s-expression tree are represented by a tag, value (t, v) pair of two consecutive stack/heap locations, where the tag provides type information on how the value is to be interpreted. Table 2 presents seven different tags and corresponding values. Atomic s-expressions have an immediate representation: the symbol NIL

Tag	Value
nil-tag(0)	0
T-tag(1)	1
sym-tag(2)	*adr*
num-tag(3)	*i*
chr-tag(4)	$[0, \ldots, 255]$
cons-tag(5)	*adr*
stg-tag(6)	*adr*

Table 2. Tags and Values

is represented by $(0, 0)$, the symbol T is represented by $(1, 1)$, a character c is represented by $(4, code(c))$, an integer constant i is represented by $(3, i)$.

For non-atomic s-expressions (symbols, strings, and binary trees (lists)), there are references into the heap. Strings are represented by $(6, adr)$, where adr is a reference to a heap cell which stores the length of the string. The heap cells above this cell consecutively store the characters of the string. The representation of a string s of length n needs $2 * (n + 1)$ memory locations. Figure 7 illustrates the representation of the string "ab".

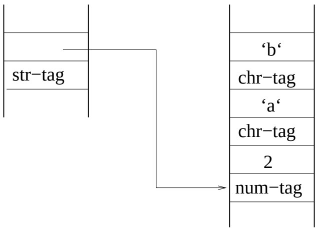


Figure 7. Representation of string "ab"

Binary trees ('cons'-nodes $(l\ .\ r)$) are represented by two consecutive pairs of nodes. The tagged pointer of a 'cons'-node is given by $(5, adr)$, where adr is a reference to a heap cell which represents the 'car'-part. The left part of Figure 8 shows the representation of cons nodes in general, while the right part illustrates the representation of list $[1, 2] \doteq (1\ .\ (2\ .\ NIL))$.

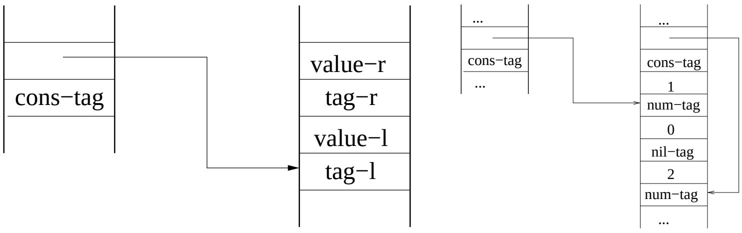


Figure 8. Representation of cons nodes and example list $[1, 2]$

For symbols, their print-names are stored uniquely in a symbol table implemented by a list of symbols. Every program symbol is represented only once within this list. The representation of a symbol is given by $(2, adr)$, where adr is a reference to a 'cons'-node the 'car' of which contains the representation of the symbol's print-name. The base address of the symbol table can be found at the absolute stack address 0.

The memory model utilized by $\mathbf{C}^{int}$ is depicted in Fig. 9. The initial stack segment contains a reference to the symbol table and the global variables. Due to the garbage collector algorithm (see Sect. 4.4), the heap above the initial segment is divided into two parts.

4.3.2 Representation Type and Abstraction Function

An arbitrary stack and heap does not necessarily represent an s-expression or a list of s-expressions. The stack or heap may contain invalid addresses and the chains of references may be cyclic. Several invariants must hold. The subset of stacks and heaps which represent s-expressions are to be characterized by a predicate *admissible*. In addition, stacks and heaps are modeled as functions from natural numbers to

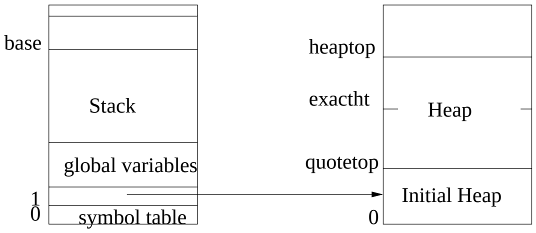


Figure 9. $\mathbf{C}^{int}$ memory model

integers. However, each program only works on finite stacks and heaps. For this purpose, admissibility is parameterized with upper stack and heap bounds, i.e. for a heap h with upper bound ht only the interval $[h(0), h(1), \ldots, h(ht))$ excluding $h(ht)$ is considered (written as (h, ht)). A valid stack/heap address is an even non-negative integer. The abstraction function to be defined below makes use of the following properties (definitions omitted):

- $admissible\_cons?(a, h, ht)$
  For a 'cons'-node to be a valid representation, the pointer chains must be acyclic (see Fig. 8). This can be assured if the references of the chain decrease which is specified by the recursive predicate $admissible\_cons?$, where (h, ht) is a heap with upper bound ht and a is a valid address.

- $admissible?((t, v), h, ht)$ specifies that a tagged value (t, v) is admissible for heap (h, ht). This is the case, if the atomic and non-atomic s-expressions are correctly represented according to Table 2. For example, if $t = 5$ (cons-tag) then v must be a valid address and the pointer chains (if any) starting with $h(v)$ and $h(v + 2)$ must be acyclic.

- $admissible?(ht, h)$: a heap is admissible if every tagged value in (h, ht) is admissible for (h, ht)

- $admissible?(st, h, ht, top)$
  Similarly, admissibility is defined for stacks with upper bounds. A stack (st, top) is admissible for (h, ht), if every tagged value in (st, top) is admissible for (h, ht).

Having introduced the necessary constraints for representing s-expressions, the abstraction function (which retrieves s-expressions from admissible stacks and heaps) can now be defined. The definition makes use of two auxiliary functions: $\Phi_{\text{string}}(adr, ht, h)$ retrieves the string at start address $h(adr)$ and function $intern$ takes a string st and yields the unique symbol the print-name of which is st.

Definition 4.1 (Abstraction Function).
Suppose $admissible?((t, v), h, ht)$ *and* $admissible?(ht, h)$.

$$\Phi((t, v), ht, h) ::= \begin{cases} t = 0: & NIL \\ t = 1: & T \\ t = 2: & intern(\Phi_{\text{string}}(h(v+1), ht, h)) \\ t = 3: & v \\ t = 4: & Char(v) \\ t = 5: & (\Phi((h(v), h(v+1)), ht, h)\ .\ \Phi((h(v+2), h(v+3)), ht, h)) \\ t = 6: & \Phi_{\text{string}}(v, ht, h) \end{cases}$$

For a stack st and a valid stack address a with $admissible?((st(a), st(a+1)), h, ht)$ *we define*

$$\Phi(st, ht, a, h) ::= \Phi((st(a), st(a+1)), ht, h)$$

Some properties concerning admissibility of stacks and heaps and the abstraction function are required in the sequel. They deal with the admissibility of stack and heap extensions and modifications. For instance, a stack remains admissible if it is modified at a valid address with an admissible tagged value:

Lemma 4.1 (Stack Modification).
Suppose $admissible?(st, h, ht, top)$ *and* $admissible?((t, v), h, ht)$. *Then for all valid addresses a with* $0 \leq a + 1 \leq top$, $admissible?(st[a \leftarrow t, (a+1) \leftarrow v], h, ht, top)$.

The next lemma states that the upper heap bound can be increased without effecting the admissibility of the pointer chains since the references of the pointer chains decrease.

Lemma 4.2 (Increasing Heap Bound, Valid Pointer Chains).
Suppose $admissible\_cons?(a, h, ht)$ *and* $ht' \geq ht$*. Then* $admissible\_cons?(a, h, ht')$*.*

A tagged value abstracts to the same s-expressions if the upper heap bound is increased:

Lemma 4.3 (Increasing Heap Bound).
Suppose $admissible?((t, v), h, ht)$ *and* $admissible?(ht, h)$ *and* $ht' \geq ht$*. Then* $admissible?((t, v), h, ht')$ *and* $\Phi((t, v), ht', h) = \Phi((t, v), ht, h)$*.*

Similar results hold, if the heap is correctly extended at the top. The new heap will be admissible and any admissible tagged value in the old heap abstracts to the same s-expression in the new heap. The following lemma is needed later for proving the correctness of heap extending operations such as *cons*.

Lemma 4.4 (Heap Extension).
Suppose $admissible?((t_1, v_1), h, ht)$ *and* $admissible?((t_2, v_2), h, ht)$ *and* $admissible?(ht, h)$*. Then*

- $admissible?(ht + 2, h[ht \leftarrow t_2, (ht + 1) \leftarrow v_2])$ *and*
- $admissible?((t_1, v_1), h[ht \leftarrow t_2, (ht + 1) \leftarrow v_2], ht + 2)$ *and*
- $\Phi((t_1, v_1), ht, h) = \Phi((t_1, v_1), ht + 2, h[ht \leftarrow t_2, (ht + 1) \leftarrow v_2])$.

4.3.3 The Compiling Relation

One of the task of this compilation is to represent the SIL program symbols and non-atomic s-expression constants in an initial heap segment and to store the global variables in the initial stack segment. For this purpose, a heap environment ζ, mapping the s-expression constants and symbols to a (tag, value) pair in the initial heap, is utilized. SIL statement compilation is straightforward and defined using a function

$$\mathcal{CC}_{\mathrm{stmt}}(\mathrm{cmd}, \zeta)$$

It is defined inductively on the structure of SIL statements. Unary and binary operators are compiled into a call of the corresponding procedure of the core runtime system, and the $list*$ operator is implemented by a sequence of calls of the "cons" procedure. Note that SIL stack position i corresponds to $\mathbf{C}^{int}$ stack positions $2i$ and $2i + 1$. The compiling relation follows the technical report [GH98b] which also contains the complete core runtime system. The complete definition of the relation is listed in the appendix F. For example, non-atomic s-expressions and symbols are compiled as follows:

$$\mathcal{CC}_{\mathrm{stmt}}(copyc(s, i), \zeta) = set\_local(tag, 2i); set\_local(val, 2i + 1) \quad \text{where} \quad (tag, val) = \zeta(s)$$

A $\mathbf{C}^{int}$ procedure heading must contain the maximum number of stack locations the procedure directly uses in its stack frame. (The information is only relevant for the compilation step to TASM code with its finite memory). This is determined by twice the highest relative stack index in the SIL function body plus one (function *maxindex*, omitted). Note that every SIL stack item consumes two $\mathbf{C}^{int}$ memory cells.

Definition 4.2 (Procedure Compilation).
$\mathcal{CC}_{\mathrm{def}}(h \leftarrow s, \zeta) = h(size) \leftarrow \mathcal{CC}_{\mathrm{stmt}}(s, \zeta) \quad where \;\; size = 2 * (maxindex(s) + 1)$
$\mathcal{CC}_{\mathrm{defs}}(f_1, \ldots, f_n, \zeta) = [\mathcal{CC}_{\mathrm{def}}(f_1, \zeta), \ldots, \mathcal{CC}_{\mathrm{def}}(f_n, \zeta)]$

Finally, the initial stack and heap segments are specified declaratively using the following relation. The construction of the initial segments including the heap environment ζ is the task of an additional compiler implementation step.

Definition 4.3 (Initial Stack and Heap).
$\mathcal{CC}_{\mathrm{decl}}(l_g, S_l, C_l, st_0, h_0, \zeta) ::=$

- $|st_0| = 2 * (l_g + 1)$

- $\mathit{even}(|h_0|)$
- $\mathit{admissible?}(|h_0|, h)$ *(initial heap is admissible)*
- $\mathit{admissible?}(st_0, h_0, |h_0|, |st_0|)$ *(initial stack is admissible w.r.t. initial heap)*
- $\forall g$ *with* $1 \leq g \leq l_g$. $\Phi(st_0, |h_0|, 2 * g, h_0) = \mathit{NIL}$ *(the global variables are initialized with NIL on the initial heap)*
- $\Phi(st_0, |h_0|, 0, h_0) = S_l$ *(position* 0 *references the symbol table)*
- ζ *associates every non-atomic constant* q *of* C_l *and symbol of* S_l *with a pair* (t, v) *such that* $\phi((t, v), |h_0|, h_0) = q$.

It remains to define the compiling relation for SIL programs; *crts* denotes the list of $\mathbf{C}^{int}$ procedures which implement the ComLisp operations (core runtime system). It also includes a garbage collector.

Definition 4.4 (Compiling SIL Programs).
Suppose $p = l_g; F_l; S_l; C_l; \Gamma; s$. *Then*

$$\begin{aligned}&\mathcal{CC}_{\text{prog}}(p, q) ::= \\ &\qquad \exists st_0, h_0, \zeta.\ \mathcal{CC}_{\text{decl}}(l_g, S_l, C_l, st_0, h_0, \zeta) \ \wedge \quad q = F_l \cup \mathit{crts}; st_0; h_0; \mathit{crts} \cup \mathcal{CC}_{\text{defs}}(\Gamma, \zeta); \mathcal{CC}_{\text{stmt}}(s, \zeta)\end{aligned}$$

4.3.4 Notes on the PVS formalization

The PVS formalization of the admissibility predicates are relatively straightforward. Besides the lemmas presented in subsection 4.3.2, a lot more additional corollaries have to be formalized and proved correct. They are all concerned with admissibility of stack and heap modifications and are to be used in the compiling correctness proofs. Furthermore, the abstraction function Φ is defined with the heavy use of PVS's predicate subtype concept. This has the consequence that many TCC's (type correctness conditions) are generated which must be proved in order to ensure type correctness of the specification. For example, the signature of the abstraction function in PVS is as follows, where type `TVal` denotes the type of tagged values.

```
phi(tv:TVal,
    ht:(even),
    h:{h1:Heap | admissible?(ht,h1) AND admissible?(tv,h1,ht)}) : RECURSIVE sexpr
```

Each time function `phi` is applied, TCC's are generated to ensure that the arguments satisfy the type constraints (admissibility). The compiling relations and functions are defined similarly as above.

4.4 Correctness of the Compilation Step

In the same way as for the first compilation phase, correctness of this compilation process is stated as follows: for any well-formed SIL 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 (Fig. 10).

$$\begin{tikzcd}
seq[char] \arrow[r, "p_{\mathbf{SIL}}"] \arrow[d, "id"'] \arrow[dr, phantom, "\circlearrowleft"] & char^* \arrow[d, "id"] \\
seq[char] \arrow[r, "p_{\mathbf{C}^{int}}"'] & char^*
\end{tikzcd}$$

Figure 10. SIL program compilation correctness

Theorem 4.1 (Correctness of Program Compilation).

$$\forall p, q, is, ol.\ (\mathit{wf}(p) \ \wedge \ \mathcal{CC}_{\text{prog}}(p, q)) \Rightarrow \ (p_{\mathbf{C}^{int}}(q)(is)(ol) \Rightarrow p_{\mathbf{SIL}}(p)(is)(ol))$$

Unfolding $p_{\mathbf{SIL}}$ and $p_{\mathbf{C}^{int}}$, the semantics of SIL statements and $\mathbf{C}^{int}$ statements have to be compared. In particular, this requires relating source and target language states. Both SIL statements and $\mathbf{C}^{int}$ statements denote state transformers. However, SIL states and $\mathbf{C}^{int}$ states differ in several respects. The principle ideas of the data representation relation are as follows (see also Fig. 9):

- global variables of SIL are represented on the stack starting at address 2. Note that at stack position $(0, 1)$ there is a reference to the start address of the symbol table.
- the base pointer of SIL has a direct correspondence with its $\mathbf{C}^{int}$ counterpart. Each SIL memory cell corresponds to two $\mathbf{C}^{int}$ cells. The pointers are related by
$$s_{\text{base}} = 2 * (l_g + 1) + 2 * \sigma_{\text{base}}.$$
- the local SIL memory (the runtime stack) corresponds to (the abstraction of) the runtime stack of $\mathbf{C}^{int}$ up to the top of stack. The abstraction is only well-defined if the stack is admissible. The reason why we have introduced a stack top pointer to SIL's semantics is that the compilation only ensures stack correspondence up to the top. $\mathbf{C}^{int}$ cells above the top do in general not correspond with the respective SIL stack cells.
- the initial heap area contains representations of SIL's non-atomic s-expression constants and symbols and they can be retrieved from the references given by the heap environment ζ. This is stated using the following relation:
$correct\_heapenv?(\zeta, S_l, C_l, h, ht) ::=$
$\quad even(ht) \ \wedge \ admissible?(ht, h) \ \wedge$
$\quad [\forall s \in (S_l \cup C_l).\ admissible?((tag, val), h, ht) \ \wedge \ \Phi((tag, val), ht, h) = s, \text{ where } (tag, val) = \zeta(s)]$
- part of the $\mathbf{C}^{int}$ core runtime system is a *stop-and-copy* garbage collector. The principle idea of this garbage collector algorithm is to reserve two non-overlapping memory areas on the heap. The first area is used to store the dynamic data structures generated during program execution, while the second one is temporarily unused. In case the first area is full, the garbage collector is invoked the next time new heap memory is to be allocated. Traversing the stack from top to the bottom and following the references into the heap, the visited datas are copied into the second heap part. References into the initial heap (the area below *quotetop*) which contains the SIL program constants and symbols are not modified. They have fixed addresses provided by the heap environment ζ at compile time. After the stack has been traversed, the second heap area is then copied back into the first area and the process continues.
- the heaptop pointer points to the first unused heap location above the second heap part. Due to the garbage collector algorithm, the current heaptop points to the first cell of the second heap part which is used as a temporary buffer for the references. In case n new cells are to be allocated, the heaptop pointer is increased by $2 * n$. The actual heaptop is referred to as *exact heaptop* and can be calculated for a $\mathbf{C}^{int}$ state s by $s_{\text{exactht}} ::= div(s_{\text{heaptop}} + s_{\text{quotetop}}, 2)$.
- an error SIL state σ ($\sigma_{\text{top}} = -2$) (that is, the SIL program has not a stack machine behavior), is related with any $\mathbf{C}^{int}$ state s, since we want (and are able) to prove correctness only for correctly compiled ComLisp programs and not for an arbitrary SIL program.
- input and output of SIL and $\mathbf{C}^{int}$ correspond.
- the symbol list must be disjoint and can be retrieved from the stack at position 0.

The above remarks lead to the following definition of the data representation relation between SIL states σ and $\mathbf{C}^{int}$ states s:

Definition 4.5 (Data Representation Relation).
$\rho(\zeta, S_l, C_l)(\sigma, s) ::=$
if $error?(\sigma)$ *then true*
else
$\quad s_{\text{heaptop}} \geq s_{\text{quotetop}} \ \wedge \ even(s_{\text{exactht}}) \ \wedge$

$$\begin{aligned}
&\quad \mathit{admissible?}(s_{\text{exactht}}, s_{\text{heap}}) \ \wedge \\
&\quad \mathit{admissible?}(s_{\text{stack}}, s_{\text{heap}}, s_{\text{exactht}}, 2 * (|\sigma_{global}| + 1 + \sigma_{top}) + 1) \ \wedge \\
&\quad \sigma_{\text{input}} = s_{\text{input}} \ \wedge \ \sigma_{\text{output}} = s_{\text{output}} \ \wedge \\
&\quad [\forall g.\ 1 \leq g \leq |\sigma_{global}| \Rightarrow \sigma_{global}.(g-1) = \Phi(s_{\text{stack}}, s_{\text{exactht}}, 2 * g, s_{\text{heap}})] \ \wedge \\
&\quad s_{\text{base}} = 2 * (|\sigma_{global}| + 1 + \sigma_{\text{base}}) \ \wedge \\
&\quad [\forall a.\ a \leq \sigma_{\text{top}} \Rightarrow \sigma_{\text{local}}(a) = \Phi(s_{\text{stack}}, s_{\text{exactht}}, 2 * (|\sigma_{global}| + 1 + a), s_{\text{heap}})] \ \wedge \\
&\quad \mathit{correct\_heapenv?}(\zeta, S_l, C_l, s_{\text{heap}}, s_{\text{quotetop}}) \ \wedge \\
&\quad \mathit{disj\_list?}(\Phi(s_{\text{stack}}, s_{\text{exactht}}, 0, s_{\text{heap}})) \\
&\mathit{endif}
\end{aligned}$$

Figure 11 illustrates the correctness property for statements by means of a commuting diagram.

$$\begin{tikzcd}
\mathit{state}_{\text{SIL}} \ni \sigma \arrow[r, "\rightarrow_{\text{SIL}}"] \arrow[d, "\rho"'] \arrow[dr, phantom, "\circlearrowleft"] & \sigma' \in \mathit{state}_{\text{SIL}} \arrow[d, "\rho"] \\
\mathit{state}_{\mathbf{C}^{int}} \ni s \arrow[r, "\rightarrow_{\mathbf{C}^{int}}"'] & s' \in \mathit{state}_{\mathbf{C}^{int}}
\end{tikzcd}$$

Figure 11. Correctness property for the compilation of SIL statements

In order to state the correctness property two technical invariants are required in addition:

1. The first invariant states that the size of the global SIL memory equals to constant g.

$$\mathit{correct\_globals?}(g, \sigma) := (|\sigma_{\text{global}}| = g).$$

   This invariant is trivially preserved since the size of the global memory does not change during program execution.

2. The second one relates $\mathbf{C}^{int}$ input state s with $\mathbf{C}^{int}$ output states s'. It states that the frame-pointers of s and s' are identical, formally: $\mathit{base\_invariant?}(s, s') ::= (s_{\text{base}} = s'_{\text{base}})$.

All ingredients have now been collected to state the correctness property for the translation of statements. The diagram in Fig. 11 has to commute in the sense of preservation of partial program correctness. The property states that if the procedure definitions and the statement are well-formed, the procedure names and the names of the core runtime system procedures are pairwise disjoint, the length of SIL's global memory equals to parameter g, the initial SIL and $\mathbf{C}^{int}$ states are related by ρ and the code resulting from compiling statement *cmd* transforms $\mathbf{C}^{int}$ state s into s', then there exists a SIL state σ' such that *cmd* evaluates in state σ to σ' and the final SIL and $\mathbf{C}^{int}$ states are related by ρ and the invariants hold.

Definition 4.6 (Correctness Property for Statement Compilation).

$$\begin{aligned}
&\mathit{correct\_prop}(\Gamma, \zeta, S_l, C_l, g)(\mathit{cmd}) ::= \\
&\forall \sigma, s, s'.\ \mathit{wf}(\Gamma, S_l, C_l, g) \ \wedge \ \mathit{wf}(\mathit{cmd}, \Gamma, S_l, C_l, g) \ \wedge \ \mathit{disj\_names?}(\Gamma, \mathit{crts}) \ \wedge \ \mathit{correct\_globals?}(g, \sigma) \ \wedge \\
&\quad \rho(\zeta, S_l, C_l)(\sigma, s) \ \wedge \ (\mathcal{CC}_{\text{defs}}(\Gamma, \zeta) \cup \mathit{crts}) \vdash s : \mathcal{CC}_{\text{stmt}}(\mathit{cmd}, \zeta) \rightarrow s' \\
&\quad \Rightarrow \exists \sigma' : (\Gamma \vdash \sigma : \mathit{cmd} \rightarrow \sigma') \ \wedge \ \rho(\zeta, S_l, C_l)(\sigma', s') \ \wedge \ \mathit{correct\_globals?}(g, \sigma') \ \wedge \ \mathit{base\_invariant?}(s, s')
\end{aligned}$$

The main obligation is to prove that this property holds for all SIL statements *cmd*:

Theorem 4.2 (Correctness of Statement Compilation).
$\forall \Gamma, \zeta, S_l, C_l, g, \mathit{cmd}.\ \mathit{correct\_prop}(\Gamma, \zeta, S_l, C_l, g)(\mathit{cmd})$

As for the verification of the first phase, in the PVS formalization, the correctness property has an additional counter argument N according to the inductive relations defining the semantics and the PVS proof of this theorem is done by measure-induction using the lexicographic combination of the counter N and the structural size of the statement *cmd* as termination measure. To suitably manage the complexity of this proof, for each kind of form a separate compilation theorem is introduced. The proof of Theorem 4.2 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 Def. 4.6):

1. First, definitions must be unfolded, the $\mathbf{C}^{int}$ statement which results from compiling a SIL statement must be executed symbolically according to its operational semantics.

2. A case-analysis according to the value of the stack top pointer of SIL must be carried out: if the pointer's value is not in correspondence with a stack machine behavior (that is, any successor SIL state is allowed in the semantics), the result state σ' is instantiated with the error state $\sigma[top \leftarrow -2]$. In most cases the proof can be easily finished since ρ (see Def. 4.5) trivially evaluates to true. In case the stack top pointer of SIL has a "correct" value, the proof continues as follows.

3. the induction hypothesis (if any) which is stated as a precondition in the compilation lemmas must be instantiated.

4. Instantiations for the SIL result state σ' (according to its semantics) must be found.

5. The consequent part of the formula must be proved:

   (a) show that statement *cmd* evaluates to the instantiated state σ'.

   (b) show (using ρ in the precondition) that the result source and target states are related by ρ

   (c) show that the invariants hold.

PVS strategies have been defined for most of the cases of the general scheme: `(UNFOLD_DEFS)` realizes the first step. The most interesting step is to show ρ (see Def. 4.5) for the result states. Strategy `(SOLVE_LOCAL)` tries to prove the correspondence of the stacks, `(SOLVE_GLOBAL)` tries to prove the correspondence of SIL's global variables with $\mathbf{C}^{int}$'s initial stack area, several strategies are utilized to prove that the modified or extended stacks are still admissible by suitably applying the lemmas from Sect. 4.3.2, and a strategy `(SYMTAB_STRAT)` tries to prove that the list of symbols is still available at stack position 0. In addition, a strategy has been defined for the error case.

In order to prove the compilation theorems for the unary and binary operators, it must be shown that the corresponding procedure of the core runtime system *crts* satisfies the correctness property. Since procedures of *crts* may call other procedures of *crts*, properties and the effect of each procedure has to be stated. Thus, classical program verification of $\mathbf{C}^{int}$ code modules is necessary. This can be realized in a modular way using code specifications: given a precondition (a $\mathbf{C}^{int}$ state predicate) P and a postcondition Q (a relation between $\mathbf{C}^{int}$ states), show that if the code is executed in a state where P holds, then the program terminates in a state s' such that relation $Q(s, s')$ holds:

$$spec?(P, cmd, Q) ::= \forall s, s'\Gamma.\ (P(s)\ \wedge\ \Gamma \cup crts \vdash s : cmd \rightarrow s') \Rightarrow Q(s, s')$$

As an example, we present the implementation of the binary operator *cons* which generates a binary tree from its arguments: $cons(c_1, c_2) ::= (c_1\ .\ c_2)$. Since *cons* extends the heap, the garbage collector may be invoked in case the memory to be allocated is not available. Thus, verification of this code requires verification of the garbage collector. Here, the properties of the garbage collector are stated axiomatically and then used in the proof of procedure *cons* (omitted here).

The code for *cons* works as follows: first, the next free heap address is calculated which is given by the exact heaptop pointer (first cell of the second heap part). It is stored at stack relative position 5. If there are no 8 additional heap cells available (twice as much as needed), the garbage collector is invoked and then the new exact heaptop is calculated. Then new heap memory is allocated by increasing the heaptop pointer. The new allocated memory is then filled by storing the first stack argument (at relative positions 0 and 1) in the 'car' part and the second stack argument (at positions $2, 3$) in the 'cdr' part. Finally the type tag of the result which is available at relative position 0 is set to 5 ('cons'-tag), and the reference to the 'car'-part is stored at position 1. The code of the $\mathbf{C}^{int}$ procedure *cons* is listed in Fig. 12.

The effect of procedure *cons* is formalized using the following lemma. If the procedure is called in a state where the stack and heap are admissible, the exact heaptop pointer points to a valid heap address, then executing the code does not modify the base pointer, input and output, and the quotetop pointer. In addition, the exact heaptop pointer points again to a valid heap address, and it is required that the code does not modify the heap below the quotetop pointer (the program constants and symbols have constant references calculated at compile time and they have to be preserved (predicate *upto_equal?*)), and the abstraction of the old stack at relative position *adr* equals to the cons-cell consisting of the abstraction of the new stack at positions *adr* and $adr + 2$.

```
"cons"(6) ←
set_local(div(heaptop + quotetop, 2), 5);
if(unavailable(8),
   call("collect_garbage", 4), set_local(div(heaptop + quotetop, 2), 5);
allocate(8);
set_local(local(0), local(5));
set_local(local(1), local(5) + 1);
set_local(local(2), local(5) + 2);
set_local(local(3), local(5) + 3);
set_local(5, 0);
set_local(local(5), 1)
```

Figure 12. $\mathbf{C}^{int}$ code of core runtime procedure *cons*

Lemma 4.5 (Effect of *cons*).
spec?$(P, call(\text{"}cons\text{"}, adr), Q)$*, where*
$P(s) ::= TRUE$ *and*
$Q(s, q) ::=$

$$\begin{array}{l}
s_{\text{heaptop}} \geq s_{\text{quotetop}} \ \wedge\ even(s_{\text{base}}) \ \wedge\ even(s_{\text{exactht}}) \ \wedge \\
admissible?(s_{\text{exactht}}, s_{\text{heap}}) \ \wedge\ admissible?(s_{\text{stack}}, s_{\text{heap}}, s_{\text{exactht}}, s_{\text{base}} + adr + 3) \\
\ \Rightarrow \\
\quad q_{\text{base}} = s_{\text{base}} \ \wedge\ q_{\text{quotetop}} = s_{\text{quotetop}} \ \wedge\ s_{\text{input}} = s_{\text{input}} \ \wedge\ q_{\text{output}} = s_{\text{output}} \ \wedge \\
\quad upto\_equal?(s_{\text{heap}}, q_{\text{heap}}, q_{\text{quotetop}}) \ \wedge\ q_{\text{heaptop}} \geq q_{\text{quotetop}} \ \wedge\ even(q_{\text{exactht}}) \ \wedge \\
\quad admissible?(q_{\text{exactht}}, q_{\text{heap}}) \ \wedge\ admissible?(q_{\text{stack}}, q_{\text{heap}}, q_{\text{exactht}}, q_{\text{base}} + adr + 1) \ \wedge \\
\quad q_{\text{stack}}(q_{\text{base}} + adr) = 5 \ \wedge \\
\quad (\forall a.\ a < q_{\text{base}} + adr \ \wedge\ even(a) \ \Rightarrow\ \Phi(q_{\text{stack}}, q_{\text{exactht}}, a, q_{\text{heap}}) = \Phi(s_{\text{stack}}, s_{\text{exactht}}, a, s_{\text{heap}})) \ \wedge \\
\quad \Phi(q_{\text{stack}}, q_{\text{exactht}}, q_{\text{base}} + adr, q_{\text{heap}}) = \\
\quad\ \ (\Phi(s_{\text{stack}}, s_{\text{exactht}}, s_{\text{base}} + adr, s_{\text{heap}})\ .\ \Phi(s_{\text{stack}}, s_{\text{exactht}}, s_{\text{base}} + adr + 2, s_{\text{heap}})))
\end{array}$$

The proof of this code specification is by symbolically executing the $\mathbf{C}^{int}$ code in Fig. 12 and by applying lemmas from Sect. 4.3.2. As above, PVS strategies have been defined to support these tasks.

Similar code specifications have to be established and proved for each of the *crts* procedures. These specifications are then used to prove the compilation theorems for the corresponding unary and binary operators. The most complicated proof obligation is the proof of the specification of procedure *intern* which realizes the corresponding ComLisp operator. Operator *intern* takes a string ‘s’ as argument and returns the unique symbol the print-name of which is ‘s’. The print-names of the program symbols are stored within a symbol table (a list of symbols with base address at stack position 0). *intern* applied to a string which is stored within the symbol list returns the reference to that symbol. On the other hand, *intern* applied to a new symbol requires extending the list with this new symbol. Admissibility of the extended heap must be proved.

4.4.1 Statistics

We present some statistics concerning the formalization and verification effort in PVS for this compilation step. Table 3 summarizes the results. The compiling specification is formalized in 3 theories consisting of the formalization of the representation type and abstraction function, the complete core runtime system, and the compiling relations. Most of the proof obligations occur in the theory which specifies the representation type and the abstraction function. Compiling verification is modularized in 19 theories for the verification of the procedures of the core runtime system and one theory which contains the compilation theorems. Most effort lies in the verification of the procedure *intern* (1320 proof steps). The large interactive effort is due to the kind of chosen semantics. A relational semantics as used in this verification project is easy to understand and to formalize and is suitable for typical compiling verification proofs. However, program verification involves a lot of symbolic execution steps of code for which such a declarative approach is not optimal since many term rewriting, replacing and deleting steps are necessary while carrying out the proof. Here, of course, either a specialized verification calculus for verifying $\mathbf{C}^{int}$ code modules or a more operational interpreter semantics would be desirable.

Table 3. Formalization and verification statistics for the second phase

	PVS theories	LOC	proof obligations	proof steps
spec. of languages	8	1046	163	867
compiling specification	3	1072	305	1761
compiling verification	20	1268	399	6874
list,map libraries	6	448	112	776
	37	3834	979	10278

The library theories stating well-known properties for lists, lists of tuples, and maps have in part already been utilized for the verification of the first compilation phase. In addition, the theories for s-expressions and operators have also been reused for this step.

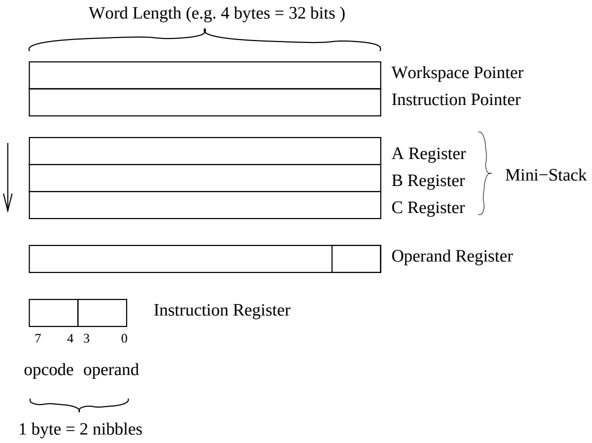
5 Transputer Backend: Generating Assembler Code

The first phase of the Transputer backend is concerned with the implementation of $\mathbf{C}^{int}$ control structures (loops, conditionals) by linear TASM assembler code with relative jumps, the realization of the basic $\mathbf{C}^{int}$ statements and expressions for stack and heap access by TASM code sequences, the mapping of the $\mathbf{C}^{int}$ runtime model consisting of the infinite stack and heap arrays onto the finite Transputer memory, and the implementation of procedures using a jump table of subroutine entry points and a stack to save the return addresses and frame pointers.

5.1 Transputer Assembler (TASM)

The purpose of our Transputer formalization is not to provide a complete model of the processor but rather to specify the components and properties which are relevant for the compilation process.

The Transputer base model is a (mini)-stack machine with byte instructions and word values. It has a RISC-like instruction set with 16 direct instructions with 4-bit opcodes and 4-bit operands. Large operands have to be loaded to the operand register using **pfix**/**nfix**-chains. An extended set of instructions (operations) can be invoked using the **opr** instruction. Operations have no direct operands; they find their arguments in registers (see Fig. 13). There is no operating system required, instead a bootloader and hardware initializations bring the Transputer in a regular state which enables program loading and execution.



Direct Operation Codes

0x	0000	j	jump
1x	0001	ldlp	load local pointer
2x	0010	pfix	prefix
3x	0011	ldnl	load non local
4x	0100	ldc	load constant
5x	0101	ldnlp	load non local pointer
6x	0110	nfix	negative prefix
7x	0111	ldl	load local
8x	1000	adc	add constant
9x	1001	call	call
Ax	1010	cj	conditional jump
Bx	1011	ajw	adjust workspace
Cx	1100	eqc	equal to constant
Dx	1101	stl	store local
Ex	1110	stnl	store non local
Fx	1111	opr	operate

Figure 13. Transputer base model and direct function codes. The Transputer-state consists of the registers Areg, Breg and Creg, which form a mini stack with top Areg, the operand register Oreg, the instruction pointer (program counter) Iptr, the workspace pointer Wptr, various flags like the error flag, some more registers and the memory Mem. The registers contain Word valued quantities. The memory is byte or word addressable.

In addition to the direct functions, a set of one-byte and two-byte operations are used in the compiling

Operations	opr-Code	Description
`rev`	`opr` 0	exchange Areg and Breg
`diff`	`opr` 4	difference
`add`	`opr` 5	addition
`gcall`	`opr` 6	computed (absolute) jump
`in`	`opr` 7	input via a link
`gt`	`opr` 9	arithmetic greater than test
`wsub`	`opr` 10	word memory subscript
`out`	`opr` 11	output via a link
`sub`	`opr` 12	subtraction

Table 4. One byte Transputer operations

Operations	opr-Code	Description
`seterr`	`opr` 16	set the error flag
`csub0`	`opr` 19	check subscript
`xdbl`	`opr` 29	convert single to double
`rem`	`opr` 31	remainder
`div`	`opr` 44	integer quotient
`not`	`opr` 50	1 complement
`xor`	`opr` 51	bitwise exclusive or
`bcnt`	`opr` 52	word/byte calculation
`wcnt`	`opr` 63	byte/word calculation
`shr`	`opr` 64	shift right
`shl`	`opr` 65	shift left
`mint`	`opr` 66	load `MinInt`
`and`	`opr` 70	bitwise and
`or`	`opr` 75	bitwise or
`mul`	`opr` 83	multiplication

Table 5. Two byte Transputer operations

specification. Two-byte operations will actually produce a 2-byte prefix-chain, first loading the operation code into the operand register and then calling the `opr` direct function. The one-byte and two-byte operations are listed in Table 4 and 5, respectively. Type *Cmd* combines the TASM instructions and operations which are utilized for implementing $\mathbf{C}^{int}$ code:

$$\begin{array}{rcl} \mathit{Cmd} & ::= & \mathit{instr}\ w \mid \mathit{opr} \qquad (w \in \mathit{Word}) \\ \mathit{instr} & ::= & \texttt{j} \mid \texttt{ldlp} \mid \texttt{ldnl} \mid \texttt{ldc} \mid \texttt{ldnlp} \mid \texttt{ldl} \mid \texttt{adc} \mid \texttt{cj} \mid \texttt{eqc} \mid \texttt{stl} \mid \texttt{stnl} \\ \mathit{opr} & ::= & \texttt{rev} \mid \texttt{diff} \mid \texttt{add} \mid \texttt{gcall} \mid \texttt{in} \mid \texttt{gt} \mid \texttt{wsub} \mid \texttt{out} \mid \texttt{sub} \mid \texttt{seterr} \mid \texttt{csub0} \mid \\ & & \texttt{xdbl} \mid \texttt{rem} \mid \texttt{div} \mid \texttt{not} \mid \texttt{xor} \mid \texttt{bcnt} \mid \texttt{wcnt} \mid \texttt{shr} \mid \texttt{shl} \mid \texttt{mint} \mid \texttt{and} \mid \texttt{or} \mid \texttt{mul} \end{array}$$

For the abstract assembler of TASM, the only available datatype is the type of machine words with word-size operands, hence no operand register is required. Transputers are parameterized with the number of bytes of a machine word ($\mathit{Byte} ::= [0, \ldots, 255]$). For the compilation step outlined in this paper a concrete Transputer model is chosen where each word contains four bytes (32-bit):

$$Word ::= [-2^{31}, \ldots, 2^{31} - 1]$$

Since the address space is dense, that is, the number of bytes per word is a power of 2, every word is a valid (byte) address. Thus, for our Transputer model we have $\mathit{ByteAddr} ::= Word$. Word addresses are addresses on word boundaries, that is, addresses which can be divided by 4:

$$WordAddr ::= \lambda(w : Word) : rem(w, 4) = 0$$

In order to calculate word addresses, in the Transputer manual [Inm88] a mapping

$$\mathit{Index} : \mathit{WordAddr} \times \mathit{Word} \rightarrow \mathit{Word}$$

is utilized. For a word address x and a word y, $\mathit{Index}(x, y)$ calculates the word address y words past the base address x. Since each word contains four bytes, *Index* can be defined as

$$\mathit{Index\_def} : \text{AXIOM} \quad \forall w, i.\ \mathit{Word}(w + 4 * i) \Rightarrow \mathit{Index}(w, i) = (w + 4 * i)$$

Note that we do not specify *Index* in case of overflow. Often memory is addressed relative to the workspace pointer. As a shorthand notation, we sometimes write

$$\mathit{Wsp}(adr) ::= \mathit{Mem}(\mathit{Index}(\mathsf{Wptr}, adr))$$

that is, the contents of the memory cell *adr* words past the workspace pointer Wptr.

The semantics of a machine program executed on some processor is typically specified operationally by means of transitions of machine configurations where a configuration consists of the machine components which are relevant for the considered model and abstraction level of the processor. For our TASM model, the machine configuration (state) consists of a subset of the components of the base model illustrated in Fig. 13. In addition, for modelling I/O the state contains an input byte sequence and an output byte list. The state includes the following components:

- the three register mini-stack (Areg, Breg, Creg) with top register Areg
- the workspace pointer Wptr pointing to a valid word address
- the error flag Eflg
  This flag is used in order to indicate error situations (like overflow) or the result of test operations. We suppose that the HaltOnErrorFlag is set, which has the effect that the TASM machine will stop whenever Eflg is set.
- the memory (a mapping from machine words (addresses) to machine words),
- an input sequence of bytes
- an output list of bytes
- the instruction pointer
  The instruction pointer is represented symbolically such that the program m is partitioned into two parts u and $i \cdot v$ with $m = u \circ (i \cdot v)$, where the instruction i is the next one to be executed. For instructions which do not alter the flow of control, the new partition after the execution of i will be $m = (u \cdot i) \circ v$. In case the second list of the partition is empty and the error flag is not set, the TASM machine will regularly stop. Such a configuration will be referred to as *final.*

Figure 14 illustrates the TASM state (configuration). Its type is given by:

$$\mathit{conf}_{\mathrm{TASM}} ::= \\ (\mathit{Word} \times \mathit{Word} \times \mathit{Word}) \times \mathit{WordAddr} \times \mathit{bool} \times (\mathit{Word} \to \mathit{Word}) \times \mathit{seq}[\mathit{byte}] \times \mathit{byte}^* \times \mathit{Cmd}^* \times \mathit{Cmd}^*)$$

In the following, c_{Areg}, c_{Breg}, c_{Creg}, c_{Wptr}, c_{Eflg}, c_{Mem}, c_{In}, c_{Out}, c_{PrA}, c_{PrB} denote the respective (state) components of a configuration c.

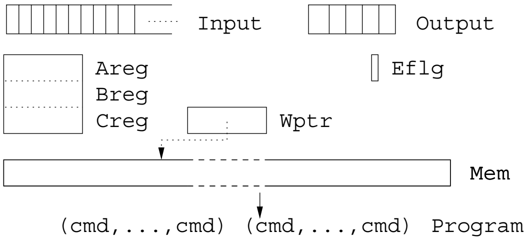


Figure 14. Machine configuration of TASM

To specify the arithmetic operations, we use total operators on words. For example, addition on words is specified using a total function $\mathit{plus} : \mathit{Word} \times \mathit{Word} \Rightarrow \mathit{Word}$. We do not specify wrapped around addition but only specify *plus* in case there is no overflow. In this case, the addition on words equals to the standard addition + on integer values:

$$\mathit{plusovfl}(x, y) ::= \neg\, \mathit{Word}(x + y) \qquad \mathit{plus\_def} : \text{AXIOM} \quad \forall x, y.\ \neg \mathit{plusovfl}(x, y) \Rightarrow \mathit{plus}(x, y) = x + y$$

The other operators such as *mult*, *div* are defined in a similar way. Only those properties are modeled which are required for the compilation step from $\mathbf{C}^{int}$ to TASM.

The semantics of the TASM instructions is given by specifying the effects of each instruction (of type *Cmd*) using a relation

$$\mathit{Effects}(c_1, c_2)$$

which directly follows the Z-like specification of the instructions in the Transputer manual [Inm88]. The complete specification of the effects can be found in the appendix G, where a rule-based notation for the single effects is utilized: we write $c : \texttt{cmd} \rightarrow q$ for $\mathit{Effects}(c, q)$, where the instruction to be executed in configuration c is given by `cmd` (that is, $c_{\mathrm{PrB}} = (\texttt{cmd} \cdot v)$ for some instruction sequence v). For example, the effect of TASM instruction `rev` which swaps the contents of Areg and Breg is specified as follows:

$$c : \texttt{rev} \rightarrow c[\mathit{Areg} := c_{\mathrm{Breg}}, \mathit{Breg} := c_{\mathrm{Areg}}]$$

This specifies the single effect of each TASM instruction. More information concerning this formalization can be found in subsection 5.1.1.

The n-step relation starting from some configuration c_1 up to a final configuration c_2 is defined using relation $\mathit{Rc}(c_1, c_2)$, where $\mathit{Effects}^*$ denotes the reflexive, transitive closure of *Effects*.

$$\mathit{Rc}(c_1, c_2) ::= \mathit{Effects}^*(c_1, c_2) \ \wedge \ \mathit{final?}(c_2)$$

A TASM program consists of two data modules (word sequences) (which are the result of compiling the $\mathbf{C}^{int}$ initial stack and heap segments), a list of code modules (the compiled $\mathbf{C}^{int}$ procedures), and a main code sequence (the main program):

$$\mathit{TASM}_{\mathrm{prg}} ::= \mathit{Word}^* \times \mathit{Word}^* \times (\mathit{Cmd}^*)^* \times \mathit{Cmd}^*$$

For a TASM program p its components are denoted by p_{data1}, p_{data2}, p_{modules}, and p_{main}. The semantics of a program is its input/output behavior (a relation between an input byte sequence and an output byte list): starting in some initial TASM state c with input sequence bs, the relation holds if the machine regularly terminates in a final state c' which has bl as its output list:

$$p_{\mathbf{TASM}}(p)(bs)(bl) ::= \ \exists c, c'.\ \mathit{init\_state?}(p)(c) \ \wedge \ c_{\mathrm{In}} = bs \ \wedge \ \mathit{Rc}(c, c') \ \wedge \ c'_{\mathrm{Out}} = bl$$

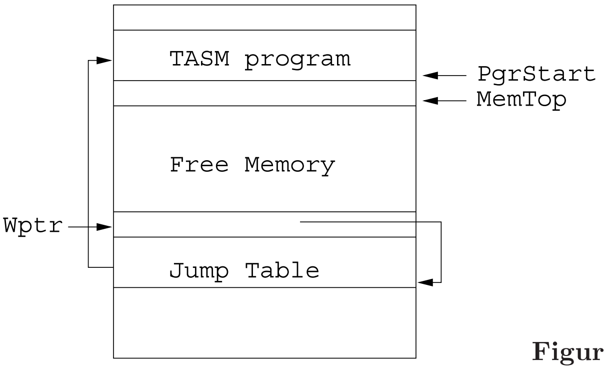


Figure 15. Initial TASM memory

It remains to define the initial TASM state. Before the TASM program can be executed, it is supposed that the program is loaded into memory (in the region above *MemTop*) and that the start addresses of the TASM code modules are available through a subroutine jump table which is located in the region below the workspace pointer Wptr. Therefore, this jump table and memory partition must be specified in the initial TASM configuration (see Fig. 15). More specifically, an initial TASM state (specified by predicate $\mathit{init\_state?}(p)(c)$) is a configuration c, where

- the symbolic program counter is given by $c_{\mathrm{PrA}} = p_{\mathrm{modules}}$ and $c_{\mathrm{PrB}} = p_{\mathrm{main}}$, that is, the first instruction to be executed is the first instruction of p_{main}.
- the error flag is false ($\neg c_{\mathrm{Eflg}}$)

- the output list is empty
- $c_{\text{Mem}}(c_{\text{Wptr}})$ is a word address (the start address of the jump table)
- $c_{\text{Mem}}(c_{\text{Wptr}}) + 4n$ is a word (n is the size of the module sequence)
- $c_{\text{Mem}}(c_{\text{Wptr}}) + 4n < c_{\text{Wptr}}$ (the jump table region is below Wptr)
- $\forall i.\ 1 \leq i \leq n \Rightarrow c_{\text{Mem}}(\mathit{Index}(c_{\text{Mem}}(c_{\text{Wptr}}), i)) = \mathit{PgrStart} + \mathit{startPos}(m_i)$ (the procedure start address can be retrieved from the jump table). This property is needed even for this abstraction level, since absolute jumps (`gcall`) are used in the compiling specification for procedure calls and returns. For this reason, the low-level concept of a program which is stored in memory has to be partly modeled on the abstraction level of TASM. The address *PgrStart* denotes the start address of the program in memory. Note that a global constraint of the TASM model is: $\mathit{PgrStart} > \mathit{MemTop}$.

5.1.1 Remarks on the PVS formalization

For the effect of some instructions, the Transputer manual [Inm88] specifies the content of some register to be undefined. This is modelled by choosing some arbitrary word value using the non-deterministic choice operator `choose`. Other instructions have preconditions in order to be applicable. For example, the shift left operation (`shl`) requires Areg to contain an unsigned word smaller than the wordlength (= 32). The manual specifies that the behavior of the Transputer is undefined if these preconditions do not hold. In our model this behavior is represented using non-determinism by allowing the machine to step in any successor state, that is, relation $\mathit{Effects}(c_1, c_2)$ holds for all c_2. The instruction `rev` for example is specified in the manual as follows:

```
rev            #00        swap
-------------------------------
 Areg' = Breg
 Breg' = Creg
 Iptr' = NextInstr
```

In PVS, this is modeled: (Note that operation ++ denotes list concatenation).

```
% Effects of TASM instructions in PVS
Effects(c1,c2) : bool =
 NOT(Eflg(c1)) AND cons?(PrB(c1)) AND
  CASES car(PrB(c1)) OF
   opr(o) :
    CASES o OF
     rev:  c2 = c1 WITH  [Areg:=Breg(c1), Breg:=Areg(c1),
                          PrA:= PrA(c1) ++ car(PrB(c1)), PrB:= cdr(PrB(c1))],
     add:  c2 = c1 WITH  [Areg:=plus(Breg(c1), Areg(c1)), Breg:=Creg(c1),
                          Creg:=choose(Word?),
                          PrA:= PrA(c1) ++ car(PrB(c1)), PrB:= cdr(PrB(c1)),
                          Eflg:=plusovfl(Breg(c1),Areg(c1))]
     [...]
    ENDCASES
```

This specifies the single effect of each instruction. The complete specification of the single effects (in a more readable rule-based notation) can be found in appendix G. The definition of the n-step relation starting from some configuration `c1` up to a final configuration `c2` is straightforward. For a symbolic execution of a code sequence (which is later required for the compiling verification) we will make use of the following intuitive corollary in combination with a PVS strategy which then tries to symbolically execute a single TASM instruction:

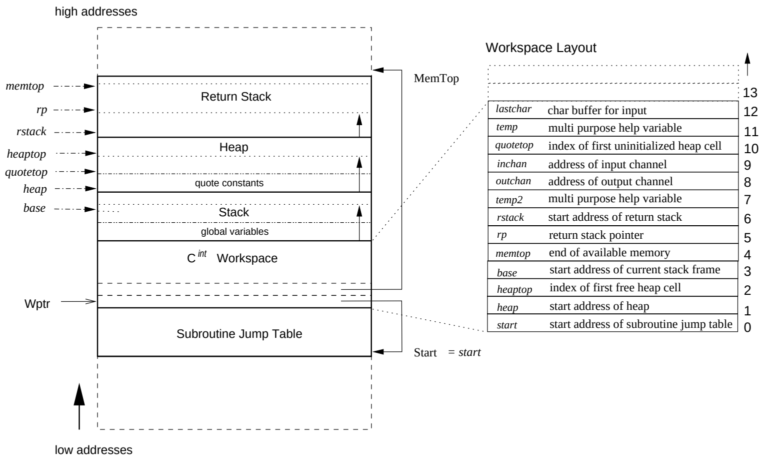
```
R_definition: COROLLARY ∀(c1,c2:CONFC):
 Rc(c1,c2) =
 (IF final?(c1) THEN c1 = c2 ELSE ∃ c3: Effects(c1,c3) ∧ Rc(c3,c2) ENDIF)
```

Furthermore, we need the property which states that the machine will stop, in case the current state is an error state:

```
error_stop: COROLLARY ∀(c1,c2:CONFC):
  (Eflg(c1) ∧ NOT(final?(c1))) ⇒ NOT(Rc(c1,c2))
```

5.2 Compiling $\mathbf{C}^{int}$ to TASM

The objective of the compilation from $\mathbf{C}^{int}$ to TASM is to implement the control structures (loops, conditionals, sequential composition) by linear TASM assembler code, and to realize the basic $\mathbf{C}^{int}$ statements and expression for stack and heap access by a TASM code sequence (a kind of macro expansion). In addition, the runtime model of $\mathbf{C}^{int}$ consisting of the infinite stack and heap arrays has to be mapped onto the finite Transputer memory. Procedures are compiled into TASM code modules and calls are implemented by means of a jump table of subroutine entry points and a stack to store the return addresses. The memory map of the $\mathbf{C}^{int}$ runtime model is illustrated in Fig. 16.



lastchar	char buffer for input	12
temp	multi purpose help variable	11
quotetop	index of first uninitialized heap cell	10
inchan	address of input channel	9
outchan	address of output channel	8
temp2	multi purpose help variable	7
rstack	start address of return stack	6
rp	return stack pointer	5
memtop	end of available memory	4
base	start address of current stack frame	3
heaptop	index of first free heap cell	2
heap	start address of heap	1
start	start address of subroutine jump table	0

Figure 16. Memory Map of $\mathbf{C}^{int}$ runtime model

A translated $\mathbf{C}^{int}$ program uses the Transputer memory between *Start* and *MemTop* which is calculated by the boot loader at program load time. For our TASM model, *MemTop* is used as a global parameter. It points to the second last available memory location. The compilation makes use of 13 system variables that can be considered as additional registers some of which are used as pointers to the specific memory areas. They are located in the Transputer workspace with workspace pointer relative access (`ldl, stl`). *MemTop* is stored to variable $memtop = 4$ at workspace location 4. The jump table used for procedure calls the start address of which is stored in system variable $start = 0$, is also constructed by the boot loader. As stated in the last section, in the initial TASM state the jump table has to be represented. The stack area starts just above the system variables at workspace location 13 ($stack = 13$), while the region for implementing $\mathbf{C}^{int}$'s heap starts at the address pointed by $heap = 1$. To implement procedure calls, a stack of return addresses is maintained which starts at the address given by $rstack = 6$. The top of this stack is accessible through system variable $rp = 5$ at workspace location 5. Absolute $\mathbf{C}^{int}$ addressing to the stack ($set\_stack, stack$) can be implemented by workspace pointer relative addressing.

The Compiling Relation

The compilation from $\mathbf{C}^{int}$ to TASM must take the finite machine resources into account. A $\mathbf{C}^{int}$ program may contain data of arbitrary size and can be arbitrary large, however TASM programs may only contain words, and the size of the program is limited.

For compiling expressions only the Transputer mini-stack is available. Thus, only expressions which can be translated with 3 registers are compilable. We further suppose that a binary expression is compiled by first compiling the left operand and then compiling the right operand. Then for example, the expression $((((a + b) + c) + d) + e)$ is compilable, but $(a + (b + (c + d)))$ is not compilable. For specifying and verifying the correctness of expression compilation, a predicate *compilable?* has to be explicitly specified. $compilable?(e, n)$ yields true if the expression e can be compiled to a stack machine of maximum size n. For binary operations $op(e_1, e_2)$ this means, that e_1 must be compilable with at most 3 registers, and e_2 must be compilable with at most 2 registers. However, for the verification we will need also the property that in case there are only 2 registers available for e_1 then e_2 must be an atomic expression.

Definition 5.1 (Compilable Expressions).
compilable$?(e, n)$ ::=

- *if* $n = 0$ *then false*
- *if e is atomic then true*
- *if e is an unary operation* $op(e')$ *then compilable*$?(e', n)$
- *if e is a binary operation* $op(e_1, e_2)$ *then compilable*$?(e_1, n)$ *and compilable*$?(e_2, n-1)$ *and if* $n = 2$ *then* e_2 *must be atomic*

Based on *compilable*? for expressions, a relation *compilable*$?(s)$ for statements is defined. It's definition is straightforward and omitted. For statements such as *set_stack*(e_1, e_2) it is required that e_1 is compilable with at most 3 registers and e_2 is compilable with at most 2, since the value of e_1 must be preserved.

The compiling relations for expressions and statements make use of a global environment

$$\varphi =< \psi, s_{\text{size}}, h_{\text{size}} >$$

consisting of a procedure environment ψ mapping procedure names to jump table indices, and the size of the initial stack and heap segment. Expression compilation is defined using an inductive relation

$$\mathcal{CC}_{\text{expr}}(e, \varphi, \sigma, m)$$

where e is the expression to be compiled, φ the global environment, σ the stack frame size found in $\mathbf{C}^{int}$ procedure definition headings, and m is the TASM code which implements e. The relation is defined following the technical report [GH98b]. However, it differs for the expression *stack*(e). Here we have found an error in the original compiling specification [GH98b] and have corrected the code. This issue will be discussed later. The complete compiling relation for expressions is listed in the appendix H.1. In a similar way, an inductive relation

$$\mathcal{CC}_{\text{stmt}}(cmd, \varphi, \sigma, m)$$

is defined for statement compilation. As for expression compilation similar errors have been found in the original specification for *allocate* and *set_stack*. The specification listed in the appendix H.2 contains the corrected code. In the next step, procedures have to be compiled. A procedure environment Γ (a list of $\mathbf{C}^{int}$ procedure definitions) is compiled into a list of TASM code modules using a relation

$$\mathcal{CC}_{\text{defs}}(\Gamma, \varphi, l)$$

where φ is the global environment, and l is the list of code modules. It is defined inductively on Γ by means of the following two rules:

$$\frac{\mathcal{CC}_{\text{stmt}}(body, \varphi, size, m),\ \ Word?(size),\ \mathcal{CC}_{\text{defs}}(\Gamma, \varphi, l)}{\mathcal{CC}_{\text{defs}}([h(size) \leftarrow body] \cdot \Gamma, \varphi, (entrycode(size) \cdot m \cdot exitcode) \cdot l)} \qquad \mathcal{CC}_{\text{defs}}([], \varphi, [])$$

Each compiled procedure has an entry code and an exit code which are used for the following purposes: the entry code saves the return address (found in register Areg when the entry code is executed) and the current frame pointer (found in Breg) on the return stack at positions pointed to by *Wsp*(rp) and *Wsp*$(rp) + 1$ respectively. Note that the code for a procedure call will leave the return address in Areg and the frame offset in Breg. Then the frame pointer is increased by the number stored in register Breg (the offset i from a call instruction *call*(h, i)), and the return stack top pointer rp is increased by 2. In case of an overflow into the memory area above *MemTop*, the program is terminated irregularly (by executing a `seterr` instruction). A stack overflow check is performed using the frame size which is specified in the procedure's heading. The exit code performs the complementary task: it restores the frame pointer from the return stack, pops the return address and jumps to it. The entry and exit codes are listed in the appendix H.3. A procedure call is implemented by an indexed jump `gcall` via a jump table of procedure start addresses. The start address of the table is located in system variable *start*.

The procedure environment ψ provides the index for a given procedure identifier. First the offset i is pushed onto the Transputer's mini-stack, then the procedure's start address is pushed and finally a jump to this address is performed:

$$\mathcal{CC}_{\text{stmt}}(\mathit{call}(h,i),\varphi,\sigma,\texttt{ldc}\ i;\texttt{ldl}\ \mathit{start};\texttt{ldnl}\ \psi(h);\texttt{gcall})\quad \text{if } 0 \le i < \sigma$$

It remains to specify the compilation of $\mathbf{C}^{int}$ programs. A $\mathbf{C}^{int}$ program specifies initial stack and heap segments which are compiled into two corresponding TASM data modules using the relation

$$\mathcal{CC}_{\text{data}}(il,wl) ::= |il| = |wl| \ \wedge\ \forall i < |il|.\ (\mathit{Word?}(il.i)\ \wedge\ il.i = wl.i)$$

where il is a list of integers and wl is the corresponding list of words. A $\mathbf{C}^{int}$ procedure list Γ is related with a global procedure environment ψ as follows:

$$\mathcal{CC}_{\text{env}}(\Gamma,\psi) ::= \forall f. f \in \Gamma \Rightarrow 1 \le \psi(f) \le |\Gamma| \wedge (\Gamma.\psi(f)-1)`1 = f$$

The property states that for each procedure with identifier f its jump table index given by $\psi(f)$ is a number between 1 and $|\Gamma|$ and the index (minus 1) specifies the position of f within the procedure list Γ.

The compilation of a $\mathbf{C}^{int}$ program $p ::= F_l; st_0; h_0; f_1,\ldots,f_n; \mathit{main}$ is specified by relation $\mathcal{CC}_{\text{prg}}$ with the following rule:

$$\frac{\begin{array}{c}\mathcal{CC}_{\text{data}}(st_0,d_1),\mathcal{CC}_{\text{data}}(h_0,d_2),\\ \varphi =< \psi, |st_0|, |h_0| >, \Gamma = [f_1,\ldots,f_n]\\ \mathcal{CC}_{\text{env}}(\Gamma,\psi),\\ \mathcal{CC}_{\text{defs}}(\Gamma,\varphi,l),\\ \mathcal{CC}_{\text{stmt}}(\mathit{main},\varphi,\mathit{maxindex}(\mathit{main}),m)\end{array}}{\mathcal{CC}_{\text{prg}}(p,(d_1;d_2;l;\mathit{initcode}(\mathit{maxindex}(\mathit{main}))\cdot m))}$$

Here, $\mathit{initcode}(\mathit{size})$ specifies the $\mathbf{C}^{int}$ initialization code sequence which at runtime initializes the processor to a valid (initial) $\mathbf{C}^{int}$ state. It partitions the available memory into the segments stack, heap, and return address stack, sets up the pointers *base*, *rp*, *rstack*, *quotetop*, *heaptop*, initializes the character buffer *lastchar* with -1, and copies the initial stack and heap segments into the stack and heap regions. Furthermore, the input and output channels are initialized. Then the entry code for the main procedure is called which performs also a stack overflow check using the maximum index of the code ($\mathit{maxindex}(\mathit{main})$). The initialization code can be found in the technical report [GH98b], it is omitted for the compiling verification presented in this paper. Instead, its effect is specified axiomatically (see next section). This completes the compiling specification from $\mathbf{C}^{int}$ to TASM.

5.3 Correctness of the Compilation Process

For this compilation step, the correctness is stated as follows (see also Fig. 17): for any well-formed $\mathbf{C}^{int}$ program p, whenever the semantics of the compiled program q is defined for some input character resp. byte stream is and output list ol, this is also the case for p for the same is and ol:

Theorem 5.1 (Correctness of Program Compilation).

$$\forall p,q,is,ol.\ (\mathit{wf}(p)\ \wedge\ \mathcal{CC}_{\text{prg}}(p,q)) \Rightarrow\ p_{\mathbf{TASM}}(q)(\mathit{char2byte}(is))(\mathit{char2byte}(ol)) \Rightarrow p_{\mathbf{C}^{int}}(p)(is)(ol)$$

$$\begin{tikzcd}
\mathit{seq}[\mathit{char}] \arrow[r, "p_{\mathbf{C}^{int}}"] \arrow[d, "\mathit{char2byte}"'] & \mathit{char}^* \arrow[d, "\mathit{char2byte}"] \\
\mathit{seq}[\mathit{byte}] \arrow[r, "p_{\mathbf{TASM}}"'] \arrow[ur, phantom, "\subseteq"] & \mathit{byte}^*
\end{tikzcd}$$

Figure 17. Correctness of $\mathbf{C}^{int}$ program compilation

Unfolding the definitions of $p_{\mathbf{C}^{int}}$ and $p_{\mathbf{TASM}}$, the semantics of $\mathbf{C}^{int}$ statements and expressions and TASM instruction sequences have to be compared. In order to have an inductive argument, a stronger property for arbitrary statements, expressions and arbitrary start and final states is required. In particular, this requires relating $\mathbf{C}^{int}$ states with TASM machine configurations by means of a data representation relation ρ. $\mathbf{C}^{int}$ state components such as stacks and heaps have to be related with the corresponding memory areas on the Transputer according to the memory map depicted in Fig. 16. The data representation relation is parameterized with the size of the current stack frame fs. The principal ideas of this data represenation relation is as follows:

- the pointers *base*, *start*, *rstack*, *rp*, *memtop*, and *heap* which are stored in the corresponding system variables at workspace locations $0, 1, \ldots, 12$ must point to valid word addresses.
- the contents of the registers *heaptop* and *quotetop* correspond to the value of the $\mathbf{C}^{int}$ heaptop and quotetop pointers.
- the segments used for the stack, heap, and return address stack are modeled by inequalities specifying their start and end addresses.
- the input stream of $\mathbf{C}^{int}$ corresponds to the input stream of TASM, in case the character buffer *lastchar* contains the value -1. Otherwise the buffer contains the last character read from the input stream and which has to be added to the TASM input stream in order to correspond to $\mathbf{C}^{int}$'s input stream. The buffer is required for the implementation of the *peek_char* statement which reads a character but does not remove it from the input stream. The output lists of $\mathbf{C}^{int}$ and TASM have to correspond directly.
- the heap area of TASM up to the beginning of the return address stack must be equal to the $\mathbf{C}^{int}$ heap area. In particular, this requires that all heap entries of $\mathbf{C}^{int}$ are words.
- the base pointer of TASM ($Wsp(base)$) corresponds to the $\mathbf{C}^{int}$ base pointer relative to the start of the TASM stack area
- the stack area of TASM which starts at $Wsp(stack) = Wsp(13)$ must be equal to the $\mathbf{C}^{int}$ stack area up to the beginning of the heap area.
- the input and output TASM channels are fixed.

This leads to the following formal definition of ρ:

Definition 5.2 (Relating $\mathbf{C}^{int}$ states with TASM configurations).
$\rho(fs)(\sigma, s) ::=$
$\quad \mathsf{LET}\ Wsp = \lambda w.\ s_{\mathrm{Mem}}(Index(s_{\mathrm{Wptr}}, w))\ \mathsf{IN}$
$\quad WordAddr?(Wsp(base)) \ \wedge\ WordAddr?(Wsp(start)) \ \wedge\ WordAddr?(Wsp(rstack) \ \wedge$
$\quad WordAddr?(Wsp(rp)) \ \wedge\ WordAddr?(Wsp(memtop)) \ \wedge\ WordAddr?(Wsp(heap)) \ \wedge\ Word?(fs) \ \wedge$
$\quad \sigma_{\mathrm{heaptop}} = Wsp(heaptop) \ \wedge\ \sigma_{\mathrm{quotetop}} = Wsp(quotetop) \ \wedge$
$\quad minword < s_{\mathrm{Wptr}} \ \wedge\ s_{\mathrm{Wptr}} + 4 * stack \leq Wsp(base) \ \wedge\ Wsp(base) + 4 * fs \leq Wsp(heap) \ \wedge$
$\quad Wsp(heap) + 4 * Wsp(heaptop) \leq Wsp(rstack) \wedge Wsp(rstack) \leq Wsp(rp) \wedge Wsp(rp) \leq Wsp(memtop) \wedge$
$\quad Wsp(memtop) + 8 < maxword \ \wedge\ Word?(Wsp(memtop) + 8 - s_{\mathrm{Wptr}}) \ \wedge$
$\quad (\mathsf{IF}\ Wsp(lastchar) = -1\ \mathsf{THEN}\ char2byte(\sigma_{\mathrm{input}}) = s_{\mathrm{In}}$
$\quad \mathsf{ELSE}\ byte?(Wsp(lastchar)) \ \wedge\ char2byte(\sigma_{\mathrm{input}}) = add(Wsp(lastchar), s_{\mathrm{In}})$
$\quad \mathsf{ENDIF}) \ \wedge\ char2byte(\sigma_{\mathrm{output}}) = s_{\mathrm{Out}} \ \wedge$
$\quad (\forall ha.\ Wsp(heap) + 4 * ha < Wsp(rstack) \Rightarrow Word?(ha) \wedge \sigma_{\mathrm{heap}}(ha) = s_{\mathrm{Mem}}(Index(Wsp(heap), ha))) \wedge$
$\quad Wsp(base) = s_{\mathrm{Wptr}} + 4 * (stack + \sigma_{\mathrm{base}}) \ \wedge$
$\quad (\forall sa.\ s_{\mathrm{Wptr}} + 4 * (stack + sa) < Wsp(heap) \Rightarrow Word?(stack + sa) \ \wedge\ \sigma_{\mathrm{stack}}(sa) = Wsp(stack + sa)) \wedge$
$\quad Wsp(inchan) = Index(minword, 4) \ \wedge\ Wsp(outchan) = minword$

Having defined the data representation relation we now can focus on the correctness of the compilation. Since statement compilation makes use of expression compilation, the correctness of the latter has to be considered first.

In order to have an inductive argument for the correctness proof of statement compilation, we have introduced a counter into TASM configurations (denoted by c_{count}). For TASM instructions which do change the flow of control (jumps, conditional jumps, absolute jumps) the counter is increased by 1, for the other instructions it remains the same. This proof technique is used here, since a pure structural induction on the statements does not work. For procedure calls, for example, the structure of the procedure body is not necessarily smaller than the call statement. In the compiled code, however, at least one jump instruction will be used. Hence, the counter will be decreased. A lexicographic ordering consisting of the counter and the structural size of the statement will then be used to formulate an induction principle.

Correctness of Expression Compilation

$\mathbf{C}^{int}$ expressions evaluated in some state may yield a value but do not effect the state, that is, there are no side effects. However, the TASM code for expressions may also change the memory and has other effects on the TASM configurations (for example, the symbolic instruction pointer). The code for expression compilation, however, only uses the two system variables *temp* and *temp*2 in order to save intermediate values. No other memory locations will be changed. The correctness property for expression compilation is illustrated in Fig. 18.

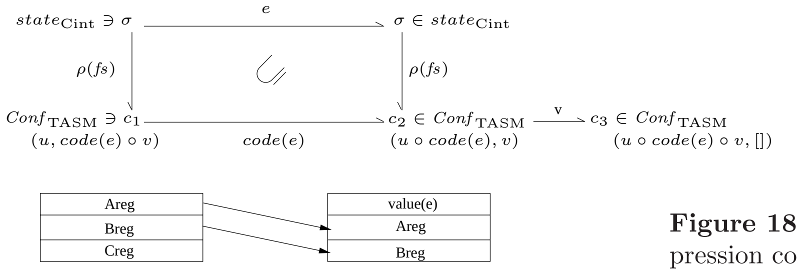

$state_{\text{Cint}} \ni \sigma$
e
$\sigma \in state_{\text{Cint}}$
$\rho(fs)$
$\rho(fs)$
$Conf_{\text{TASM}} \ni c_1$
$(u, code(e) \circ v)$
$code(e)$
$c_2 \in Conf_{\text{TASM}}$
$(u \circ code(e), v)$
v
$c_3 \in Conf_{\text{TASM}}$
$(u \circ code(e) \circ v, [])$
Areg
Breg
Creg
value(e)
Areg
Breg

Figure 18. Correctness property for $\mathbf{C}^{int}$ expression compilation

The correctness property makes use of code contexts u, v around the code of interest (the result of compilation). This more stronger property is necessary in order to apply the induction hypothesis for composed programs in the correctness proof. The property states that some TASM code m is a correct implementation of an expression e, if the program $p = u \cdot (m \circ v)$ for some u and v is executed in a start state c_1 which is related with $\mathbf{C}^{int}$ state σ by ρ and terminates in some final state c_3, then there exists an intermediate state c_2 such that $p = (u \circ m) \cdot v$ (that is, m is completely executed) and there is a transition from c_2 to final state c_3 and c_2 is related with σ by ρ and expression e evaluates in state σ to some value which is available in stack register Areg (the top of the mini-stack). In case e is compilable with at most two registers the old value of Areg in state c_1 is preserved in Breg in state c_2 and in case e is atomic, even the value of Breg is preserved and available in register Creg. The last properties are necessary in order to prove the correctness of binary expression compilation since the value of the left operand must be preserved when evaluating the right operand. In addition, the code m may have an effect on the system variables *temp* and *temp2* (expressed in predicate *expression_memory_effect?*). As stated above, a counter has been introduced into TASM configurations for proof technical reasons. The counter in state c_2 will be at least as large as in the start state c_1 since some of the expression code contains relative jumps. For the source level an additional invariant (*base_respects_env?*) is required which relates $\mathbf{C}^{int}$'s frame pointer with the global environment $\varphi =< \psi, s_{\text{size}}, h_{\text{size}} >$: the size of the initial $\mathbf{C}^{int}$ stack segment s_{size} must be smaller or equal the frame pointer *base*. This property holds in the initial $\mathbf{C}^{int}$ state (since the frame pointer is initialized with the size of the initial stack) and is preserved throughout the execution of $\mathbf{C}^{int}$ statements (the frame pointer will never point below its initial position). Formally, the definition is as follows:

Definition 5.3 (Correctness property for $\mathbf{C}^{int}$ expressions).

$$\begin{array}{l}
\mathcal{CE}(e, m, \varphi, \mathit{fs}) ::= \ \forall u, v, c_1, c_3, \sigma, Z. \\
\quad \mathit{base\_respects\_env?}(\sigma, \varphi) \ \wedge \ \rho(\mathit{fs})(\sigma, c_1) \ \wedge \ \neg {c_1}_{\text{Eflg}} \ \wedge \\
\quad {c_1}_{\text{PrA}} = u \ \wedge \ {c_1}_{\text{PrB}} = m \circ v \ \wedge \ {c_1}_{\text{count}} = Z \ \wedge \ \mathit{Rc}(c_1, c_3) \\
\quad \Rightarrow \ \exists c_2, Y. \ \neg {c_2}_{\text{Eflg}} \ \wedge \ {c_2}_{\text{PrA}} = u \circ m \ \wedge \ {c_2}_{\text{PrB}} = v \ \wedge \ Y \leq Z \ \wedge \ {c_2}_{\text{count}} = Y \ \wedge \ \mathit{Rc}(c_2, c_3) \ \wedge \\
\qquad \rho(\mathit{fs})(\sigma, c_2) \ \wedge \ \sigma : e \ \rightarrow \ {c_2}_{\text{Areg}} \ \wedge \ (\mathit{compilable?}(e, 2) \ \Rightarrow \ {c_2}_{\text{Breg}} = {c_1}_{\text{Areg}}) \ \wedge \\
\qquad (\mathit{atom?}(e) \ \Rightarrow \ {c_2}_{\text{Creg}} = {c_1}_{\text{Breg}}) \ \wedge \ \mathit{expression\_memory\_effect?}(c_1, c_2)
\end{array}$$

The obligation to prove is that property $\mathcal{CE}$ holds for any expression e and compiled code m:

Lemma 5.1 (Correctness of Expression Compilation).

$$\forall e, \varphi, fs, m.\ \mathcal{CC}_{\text{expr}}(e, \varphi, fs, m) \ \Rightarrow\ \mathcal{CE}(e, m, \varphi, fs)$$

The proof is by (a pure) structural induction on the structure of e. To suitably manage the complexity of this proof, for each kind of expression a separate compilation lemma is introduced. For example, the compilation lemma for the binary expression $e_1 + e_2$ is as follows:

compile_plus : LEMMA
$\mathcal{CE}(e_1, m_1, \varphi, fs) \ \wedge\ \mathcal{CE}(e_2, m_2, \varphi, fs) \ \wedge\ compilable?(e_2, 2) \ \Rightarrow\ \mathcal{CE}(e_1 + e_2, m_1 \cdot m_2 \cdot \texttt{add}, \varphi, fs)$

The proof of lemma 5.1 is then carried out by application of the compilation lemmas. Most of the proofs of the compilation lemmas follow a similar scheme according to the structure of the correctness property $\mathcal{CE}$:

1. First, definitions must be unfolded, and the compiled code for the expression must be executed symbolically. In the inductive case (for unary and binary expressions) the induction hypothesis must be suitably instantiated. A PVS strategy has been defined to symbolically execute one TASM instruction. Basically this strategy uses the property `R_definition` (see Section 5.1). However, some of the TASM instructions have applicability conditions. It has to be proved that these conditions are satisfied. Most of these conditions are simply proved by expanding the definition of ρ.

2. After the TASM code has been executed, the final TASM state and counter value have to be instantiated for the existentially quantified variables c_2 and Y.

3. the consequent part of the formula must be proved: to show that the flag is not set and that the symbolic instruction pointer consisting of the left part PrA and the right part PrB has the correct value and that the counter value is equal or less the starting value is trivial in most of the cases. In addition it must be shown, that expression e evaluates to the value stored in register Areg. This is done by symbolically evaluating e. More interesting is the proof that $\rho(fs)(\sigma, c_2)$ holds and that the old mini-stack values are preserved. For proving $\rho(fs)(\sigma, c_2)$ the definitions of the 'old' ρ and the new one are expanded. Then for proving the correspondence of the stack and heap areas PVS strategies have been defined: `rho-stack` and `rho-heap` try to automatically prove the properties. Finally one has to show *expression_memory_effect?*, that is, the memory has not changed (only the system variables *temp* and *temp*2 may have changed). A strategy `expression-memory` has been defined to prove this property.

Correctness of Statement Compilation

The correctness notion for statement compilation follows the notion for expressions. $\mathbf{C}^{int}$ statements denote ordinary state transformers, hence, different to the notion used for expression compilation the source state might possibly change. The notion is illustrated in Fig. 19.

$$\begin{tikzcd}
state_{\text{Cint}} \ni \sigma \arrow[r, "cmd"] \arrow[d, "\rho(fs)"'] \arrow[dr, phantom, "\circlearrowleft"] & \tau \in state_{\text{Cint}} \arrow[d, "\rho(fs)"] & \\
Conf_{\text{TASM}} \ni c_1 \arrow[r, "code(cmd)"'] & c_2 \in Conf_{\text{TASM}} \arrow[r, "\text{v}"] & c_3 \in Conf_{\text{TASM}} \\
(u, code(cmd) \circ v) & (u \circ code(cmd), v) & (u \circ code(cmd) \circ v, [])
\end{tikzcd}$$

Figure 19. Correctness property for $\mathbf{C}^{int}$ statement compilation

The main difference however, is the fact, that additional invariants are necessary ensuring that specific TASM memory areas such as the jump table, for example, are not modified. In addition, $\mathbf{C}^{int}$ procedure defintions have to be related with the TASM program and the global compile time environment φ by means of a 'fit' predicate. The addtional invariants are as follows:

- *jump_table*?(p, c):
the start addresses of the modules of p can be found in the jump table the start address of which is *Mem*(*Wptr*). This is also stated in the initial TASM state and the compiled $\mathbf{C}^{int}$ code preserves

this property. This target level invariant is proved by showing that the jump table memory area is not changed.

- *register_invariant?*:
  this invariant states that the workspace pointer and the contents of the system variables *heap*, *start*, *rstack*, and *memtop* do not change.

- *stack_invariant?*:
  this invariant is required to prove the correctness of the compilation of *call*. It states that the return address stack pointers *rstack* (the start address of the stack) and *rp* (the stack top pointer) do not change, and that the contents of the return address stack starting from address $Wsp(rstack)$ up to $Wsp(rp)$ do not change. This invariant ensures that the return address of the call and the frame pointer which are stored on the stack are preserved.

- $fit?(\Gamma, \varphi, u, p)$:
  This property relates the $\mathbf{C}^{int}$ procedure list Γ with the global environment φ and the TASM code sequence u and TASM program p. It is also only needed for the proof of the compilation of *call*:

  $fit?(\Gamma, \varphi, u, p) ::=$ $\mathsf{LET}\ \psi = \varphi`1\ \mathsf{IN}$
  $(flatten(p_{\text{modules}}) \circ p_{\text{main}}) = u \ \wedge\ |\Gamma| = |p_{\text{modules}}| \ \wedge$
  $\forall f.\ f \in \Gamma \Rightarrow 1 \leq \psi(f) \leq |\Gamma| \ \wedge\ \mathsf{LET}\ (g, stksize, body) = nth(\Gamma, \psi(f) - 1)\ \mathsf{IN}$
  $g = f \ \wedge\ Word?(stksize) \ \wedge\ \exists x, y, m.\ x \cdot entrycode(stksize) \cdot m \cdot exitcode \cdot y = u \ \wedge$
  $PgrLength(x) = startPos(p_{\text{modules}}, \psi(f)) \ \wedge\ \mathcal{CC}_{\text{stmt}}(body, \varphi, stksize, m)$

  This first conjunct expresses that the TASM code sequence u corresponds to TASM program p consisting of the module sequence and the main program. The second conjunct states that the length of the $\mathbf{C}^{int}$ procedure list Γ is equal to the length of the module list. The third conjunct states that for each procedure f in Γ, the jump table index $\psi(f) - 1$ denotes the position of f in Γ, and for each procedure f there exists a TASM code sequence in u consisting of the entry code, the code for the procedure's body, and the exit code. The start position of the code for f is given by its position within the module sequence.

Putting these invariants together, the correctness property $\mathcal{CS}$ looks as follows:

Definition 5.4 (Correctness Property for Statements).

$\mathcal{CS}(cmd, m, \varphi, fs, \Gamma, p, Z) ::=\ \forall u, v, c_1, c_3, \sigma.$
$wf(cmd, \Gamma) \ \wedge\ wf(\Gamma) \ \wedge\ \mathcal{CC}_{\text{stmt}}(cmd, \varphi, fs, m) \ \wedge\ fit?(\Gamma, \varphi, u \circ m \circ v, p) \ \wedge\ base\_respects\_env?(\sigma, \varphi) \ \wedge$
$\neg {c_1}_{\text{Eflg}} \ \wedge\ {c_1}_{\text{PrA}} = u \ \wedge\ {c_1}_{\text{PrB}} = m \circ v \ \wedge\ {c_1}_{\text{count}} = Z \ \wedge\ jump\_table?(p, c_1) \ \wedge\ \rho(fs)(\sigma, c_1) \ \wedge\ Rc(c_1, c_3)$
$\Longrightarrow$
$\exists c_2, \tau, Y.\ \neg {c_2}_{\text{Eflg}} \ \wedge\ {c_2}_{\text{PrA}} = u \circ m \ \wedge\ {c_2}_{\text{PrB}} = v \ \wedge\ Y \leq Z \ \wedge\ {c_2}_{\text{count}} = Y \ \wedge$
$Rc(c_2, c_3) \ \wedge\ \Gamma \vdash \sigma : cmd \rightarrow \tau \ \wedge\ \rho(fs)(\tau, c_2) \ \wedge$
$register\_invariant?(c_1, c_2) \wedge base\_respects\_env?(\tau, \varphi) \wedge stack\_invariant?(c_1, c_2) \wedge jump\_table?(p, c_2)$

Note that in the definition of $\mathcal{CS}$ the counter Z is a argument of $\mathcal{CS}$ in order to formulate an induction principle for the main proof obligation:

Theorem 5.2 (Correctness of Statement Compilation).

$$\forall cmd, m, \varphi, fs, \Gamma, p, Z.\ \mathcal{CS}(cmd, m, \varphi, fs, \Gamma, p, Z)$$

The proof of the theorem is again by measure induction using the lexicographic combination of the counter Z and the structural size of the statement *cmd*. As for expression compilation, for each kind of $\mathbf{C}^{int}$ statement a separate compilation theorem is introduced and the proof of theorem 5.2 is then by application of the different compilation theorems. The proofs of the different compilation theorems again follow a similar scheme according to the definition of $\mathcal{CS}$ which is quite similar to $\mathcal{CE}$. Since more invariants are involved in $\mathcal{CS}$, the proofs are more tedious comparable to the proofs carried out for expression compilation. In addition to the PVS strategies used for the proofs of expression compilation, strategies for the new invariants have been defined. The most tedious proof is for the *call* statement:

compile_call : LEMMA

	PVS theories	LOC	proof obligations	proof steps
spec. of transputer	4	915	117	757
compiling specification	3	521	58	98
compiling verification	7	833	79	4480
	14	2269	254	5335

Table 6. Formalization and verification statistics for the third phase

$$\begin{aligned} & f \in \Gamma \ \wedge \ (\forall m_1, W : W < X \Rightarrow \mathcal{CS}(get(f,\Gamma)`3, m_1, \varphi, get(f,\Gamma)`2, \Gamma, p, W)) \\ & \quad \Rightarrow \mathcal{CS}(call(f, \mathit{offset}), m, \varphi, fs, \Gamma, p, X) \end{aligned}$$

Since procedure's are compiled into a sequence *entrycode*(*size*) · *body_code* · *exitcode* the effects of the entry and exit code must be taken into account. It therefore makes sense to introduce separate auxiliary lemmas stating the effects of both the entry and exit code. The proof for lemma *compile_call* then makes use of these lemmas.

Initialization Code

According to the compiling specification for programs, an initialization TASM code is executed before the code for the main statement is started. As sketched in Sect. 5.2, the purpose of the initialization code is as follows: it defines the partition of the TASM memory into the segments stack, heap, and return address stack, initializes the system variables and copies the initial stack and heap segments (the two data modules of the TASM program) into the lower stack and heap regions. Thus, the overall effect of the code is to initialize the processor to a valid initial $\mathbf{C}^{int}$ state (see Fig. 20). This effect is specified axiomatically (see appendix H.4) as we are mainly interested in verifying the compilation process. However, the axiom can be turned into a lemma by "executing" the initialization code and by proving the properties. One problem will arise: copying the data modules into the stack and heap regions is implemented by means of loops. Hence, assembler code with loops must be verified - symbolic execution will not be sufficient. The problem can be solved by applying classical program verification techniques for object code verification (see for example [Yu93]).

$$\begin{tikzcd}
 & \sigma \in \mathit{init\_state?}_{\mathrm{Cint}} \arrow[d, "\rho(\mathit{size})"] \\
\begin{array}{l} \mathit{init\_state?}_{\mathrm{TASM}} \ni c_1 \\ (p_{\mathrm{modules}}, \mathit{initcode}(\mathit{size}) \circ v) \end{array} \arrow[r, "\mathit{initcode}(\mathit{size})"] & \begin{array}{l} c_2 \in \mathit{Conf}_{\mathrm{TASM}} \\ (p_{\mathrm{modules}} \circ \mathit{initcode}(\mathit{size}), v) \end{array}
\end{tikzcd}$$

Figure 20. Effect of Initialization code

The proof of the main correctness theorem of this phase (5.1) is then accomplished using the axiom for the initialization code and theorem 5.2 applied to the main $\mathbf{C}^{int}$ statement.

5.3.1 Statistics.

We present some statistics concerning the formalization and verification effort in PVS for the compilation from $\mathbf{C}^{int}$ to TASM. Table 6 summarizes the results. The specification of the TASM Transputer model involves 4 PVS theories with 915 lines of specification code (LOC). There are 117 proof obligations to prove for this specification including all type correctness conditions (TCCs). These obligations are proved interactively using the PVS proof checker by manually invoking 757 proof steps. Most of the effort has been put into the proof of the compilation of the *call* statement (1826 proof steps).

More interesting is the fact that during the verification process we have found an error in the given compiling specification. This error is hard to find when doing only paper-and-pencil proofs. The nature of this error is as follows: a necessary overflow check has been omitted in the following way: the code for expression *stack*(e) and for the statement *set_stack* and the procedure entry code uses the TASM instruction `wsub` to efficiently calculate the word address $\mathit{Index}(\mathsf{Areg}, \mathsf{Breg}) = \mathsf{Areg}\ + 4 * \mathsf{Breg}$. However, no address overflow checks in the address calculation are applied. The error flag is not set in case $\mathsf{Areg}\ + 4 * \mathsf{Breg}$ is not a word. For (very) large expressions e, for example, the code will calculate a wrong address. The code is easy to correct: instead of using the efficient operation `wsub` the address can be calculated explicitly using the standard arithmetic operators `mult` and `plus`. These operators set the error flag in case there is an overflow and the machine will halt. One could argue that such large expressions or values will never occur. However, here we explicitly have to express the conditions to ensure correctness of the compiling specification. It is not possible to formally express that the value of an arbitrary expression is of acceptable size.

6 Transputer Backend: Implementing Large Word Constants

The purpose of this compilation step is to implement large and negative word constants occurring in TASM assembler instructions by means of pfix/nfix chains and to transform the assembly code mnemonics to 4-bit opcode in order to generate a sequence of instruction bytes for every assembler instruction. The $\mathbf{TC}_1$ machine corresponds to TASM but has additional `pfix` and `nfix` instructions and works on instruction byte sequences rather than on assembler instructions. A $\mathbf{TC}_1$ instruction now is simply a byte. The machine has an additional operand register Oreg used to load large operands which are loaded by means of `pfix` and `nfix` instructions.

Instruction decoding takes place before an instruction is executed. Note that an instruction byte i consists of the operand (least significant 4 bits) and the operation code (most significant nibble). The operand of i is loaded to the least significant nibble of the operand register,

$$\mathsf{Oreg}' ::= \mathsf{Oreg}\ \vee (i \wedge 15)$$

while the operation code is determined by

$$\mathit{Code}' ::= (i \wedge 240).$$

The least significant 4 bits of Oreg are guaranteed to be 0 before the decoding takes place. The instructions of $\mathbf{TC}_1$ are executed in the same manner, i.e. they have the same semantics. The only difference is that the instruction byte now includes the operand whereas in the TASM assembler the (word) operand is explicit (like in **ldc** w, for example). The prefix instructions `pfix` and `nfix` are used to fill the operand register Oreg nibble by nibble. After loading the operand nibble of the current instruction to the operand register, the `pfix` instruction then shifts the operand register one nibble to the left, while `nfix` first complements the content of Oreg and then shifts to the left. Both instructions leave $0's$ in the least significant nibble of the operand register. All other instructions clear the entire operand register ($\mathsf{Oreg}' = 0$).

A $\mathbf{TC}_1$ program consists of two data modules (word sequences), a list of code modules, and a main code sequence (the main program):

$$\mathit{TC1}_{\mathrm{prg}} ::= \mathit{Word}^* \times \mathit{Word}^* \times (\mathit{byte}^*)^* \times \mathit{byte}^*$$

The configuration (state) of the $\mathbf{TC}_1$ machine corresponds to the TASM state. The differences are as follows: In the same way like for TASM the instruction pointer is represented symbolically such that the program m is partitioned into two parts u and $i \cdot v$ such that $m = u \circ (i \cdot v)$, where the instruction i is the next one to be executed. In contrast to TASM where u and v are sequences of assembler instructions, in $\mathbf{TC}_1$ u and v are sequences of bytes (see Figure 21). The configuration type is given by:

$\mathit{conf}_{\mathrm{TC}_1} ::=$
$(\mathit{Word} \times \mathit{Word} \times \mathit{Word}) \times \mathit{WordAddr} \times \mathit{Word} \times \mathit{bool} \times (\mathit{Word} \rightarrow \mathit{Word}) \times \mathit{seq}[\mathit{byte}] \times \mathit{byte}^* \times \mathit{byte}^* \times \mathit{byte}^*$

In the following, c_{Areg}, c_{Breg}, c_{Creg}, c_{Wptr}, c_{Oreg}, c_{Eflg}, c_{Mem}, c_{In}, c_{Out}, c_{PrA}, c_{PrB} denote the respective (state) components of a configuration c.

Similarly as for TASM, the effects of the instructions are specified using a relation *Effects*. For example, if the instruction code of the current instruction i to be executed is 32 ($(i \wedge 240) = 32$) (a `pfix` instruction) then its effect in state c is given by

$$c : \texttt{pfix} \rightarrow c[\mathit{Oreg} := \mathit{shiftL}(c_{\mathrm{Oreg}}, 4)]$$

The `nfix` instruction has the following meaning:

$$c : \texttt{nfix} \rightarrow c[\mathit{Oreg} := \mathit{shiftL}(\mathit{bitNOT}(c_{\mathrm{Oreg}}), 4)]$$

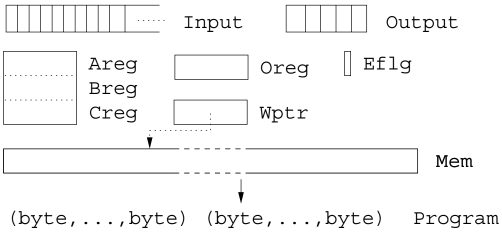


Figure 21. Machine configuration of $\mathbf{TC}_1$

Note that the current instruction is determined by decoding the first instruction of the second instruction byte sequence c_{PrB} (the symbolic instruction pointer).

The n-step relation $Rc(c_1, c_2)$ from some starting configuration c_1 to some final configuration c_2 is defined as for TASM. As for TASM, the semantics of a $\mathbf{TC}_1$ program is its input/output behavior: starting in an initial state which corresponds to the initial TASM state with input byte sequence *bs*, the relation holds, if the machine regularly terminates in a final state which has *bl* as its output list:

$$p_{\mathbf{TC_1}}(bs)(bl) ::= \ \exists c, c'.\ init\_state?(p)(c) \ \wedge \ c_{\mathrm{In}} = bs \ \wedge \ Rc(c, c') \ \wedge \ c'_{\mathrm{Out}} = bl$$

6.1 Compiling TASM to TC$_1$

Prefix chains are to be generated in order to transform an abstract TASM assembler instruction $opr(e)$ on large or negative operands $e \in Word$ to an equivalent sequence

$$\texttt{pfix}(e_0) \cdot \texttt{pfix}(e_1) \cdots [\texttt{pfix} \mid \texttt{nfix}](e_{n-1}) \cdot opr(e_n)$$

of $length(e) + 1$ $\mathbf{TC}_1$ instructions; $e_i \in [0, \ldots, 15]$ and $n = length(e)$. Note that $length(e) + 1$ equals to the number of bytes of the concrete instruction sequence.

Following the definition given in the Transputer manual [Inm88], the following recursive function computes the prefix chain (a byte list) where i is a byte and w is a word. Note that $\texttt{pfix} = 32$ and $\texttt{nfix} = 96$ denote the instruction codes of the prefix instructions.

$$prefix(i, w) ::= \begin{cases} [bitOR(i, w)] & \text{if } 0 \le w < 16 \\ prefix(\texttt{pfix}, shiftR(w, 4)) \cdot bitOR(i, bitAND(w, 15)) & \text{if } w \ge 16 \\ prefix(\texttt{nfix}, shiftR(bitNOT(w), 4)) \cdot bitOR(i, bitAND(w, 15)) & \text{if } w < 0 \end{cases}$$

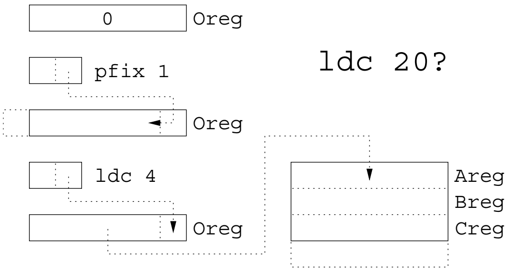


Figure 22. Construction of the pfix/nfix chain for `ldc` 20

Figure 22 shows the implementation of the TASM instruction `ldc` 20. Since the operand of this instruction is large (≥ 16) it has to be loaded to the operand register by means of a `pfix`(1) instruction. Shifting the operand (1) one nibble to the left results in the number 16. Then adding 4 to 16 yields the operand 20. Thus, the byte list generated for *prefix*(64, 20) is given by [33, 68], where 64 is the instruction code of `ldc`.

For specifying the compiling relation for this step, one principal problem arises: there are $\mathbf{TC}_1$ configurations which do not have a corresponding TASM configuration. For example, according to the effects of the instructions, $\mathbf{TC}_1$ may non-deterministically step in any possible successor state, if preconditions of instructions are not met. There are bytes which do not denote the instruction code of an abstract TASM instruction. For this purpose, we introduce a pseudo TASM instruction `err` which has a completely non-deterministic effect and compiles to any byte sequence. It is never generated and only introduced as a means to relate $\mathbf{TC}_1$ states with TASM states in case of erroneous state transitions.

As a specific example, notice that a $\mathbf{TC}_1$ jump instruction can jump within a pfix/nfix chain. However, for TASM we have defined a non-deterministic behavior in this case, allowing the machine to step in any successor state. For the last compilation phase from $\mathbf{C}^{int}$ to TASM we (implicitly) proved that such a situation will never occur and hence all the jumps are well defined. Otherwise, we would not have been able to establish the correctness of that phase.

Using function *prefix*, we now can define the compiling relations for this step.

$$\mathcal{CC}_{\mathrm{Cmd}}(i, bl)$$

relates TASM instructions i with corresponding byte sequences bl in the following way:

- *instr w*
$$\mathcal{CC}_{\mathrm{Cmd}}(instr\ w, prefix(InstrCode(instr), w))$$
- one and two-byte operations *op*
$$\mathcal{CC}_{\mathrm{Cmd}}(op, prefix(InstrCode(\texttt{opr}), OprCode(op)))$$
- pseudo instruction `err`
$$\mathcal{CC}_{\mathrm{Cmd}}(\texttt{err}, bl),\ \text{for any byte list } bl.$$

The instruction and operation codes are listed in Figure 13 and Tables 4 and 5, respectively.

Note that the length (in bytes) of a TASM assembler instruction i (written as $|i|$) is defined by the length of the associated pfix/nfix chain. Function $|.|$ is used in the compiling specification from $\mathbf{C}^{int}$ to TASM in order to calculate relative jump distances (see Appendix H). A simple proof then shows that the length of the byte list generated for instruction i equals the length of i:

$$\mathcal{CC}_{\mathrm{Cmd}}(i, bl) \Rightarrow\ |bl| = |i|$$

A list of TASM instructions is compiled by compiling each instruction in the list and appending the resulting byte sequences.

$$\frac{\mathcal{CC}_{\mathrm{Cmd}}(i_1, b_1), \ldots, \mathcal{CC}_{\mathrm{Cmd}}(i_n, b_n)}{\mathcal{CC}_{\mathrm{Cmds}}([i_1, \ldots, i_n], b_1 \cdots b_n)}$$

Analogously, a list of TASM code modules is compiled:

$$\frac{\mathcal{CC}_{\mathrm{Cmds}}(m_1, b_1), \ldots, \mathcal{CC}_{\mathrm{Cmds}}(m_n, b_n)}{\mathcal{CC}_{\mathrm{def}}([m_1, \ldots, m_n], [b_1, \ldots b_n])}$$

Finally, TASM programs p are compiled into $\mathbf{TC}_1$ programs q:

$$\mathcal{CC}_{\mathrm{prog}}(p, q) ::= [q_{\mathrm{data1}} = p_{\mathrm{data1}}\ \wedge\ q_{\mathrm{data2}} = p_{\mathrm{data1}}\ \wedge\ \mathcal{CC}_{\mathrm{def}}(p_{\mathrm{modules}}, q_{\mathrm{modules}})\ \wedge\ \mathcal{CC}_{\mathrm{Cmds}}(p_{\mathrm{main}}, q_{\mathrm{main}})]$$

6.2 Correctness of the Assembler

For this compilation step, the correctness is stated as follows: for any TASM program p, whenever the semantics of the compiled program q is defined for some input byte stream bs and output byte list bl, this is also the case for p for the same bs and bl:

Theorem 6.1 (Correctness of TASM Program Compilation).

$$\forall p, q, bs, bl.\ \mathcal{CC}_{\mathrm{prog}}(p, q)\ \Rightarrow\ p_{\mathbf{TC_1}}(q)(bs)(bl) \Rightarrow p_{\mathbf{TASM}}(p)(bs)(bl)$$

Similar as for the other compilation phases, in order to prove this theorem, TASM and $\mathbf{TC}_1$ instructions have to be compared. In particular, this requires relating TASM configurations with $\mathbf{TC}_1$ configurations. The configurations directly correspond in most of their state components. They only differ in the following components:

- symbolic instruction pointer (c_{PrA},c_{PrB}).
  In TASM, c_{PrA} and c_{PrB} are assembler instruction sequences, while in $\mathbf{TC}_1$ these are sequences of bytes. $\mathbf{TC}_1$ and TASM sequences are related by $\mathcal{CC}_{\text{Cmds}}$. Note that any $\mathbf{TC}_1$ sequence can be related with a corresponding TASM sequence due to the TASM pseudo instruction `err`. For any byte, it either corresponds to an abstract TASM instruction or it corresponds to `err`.
- $\mathbf{TC}_1$ configurations contain the additional operand register.
  The correct execution of the pfix/nfix chain calculated by function *prefix* depends heavily on the a-priori content of the operand register. An essential requirement here is that the operand register is always 0 when an abstract TASM instruction is to be executed.

This leads to the following definition of the data representation relation ρ between TASM states s and $\mathbf{TC}_1$ states q

$\rho(s, q) ::=$

$$q_{\text{Areg}} = s_{\text{Areg}} \ \wedge\ q_{\text{Breg}} = s_{\text{Breg}} \ \wedge\ q_{\text{Creg}} = s_{\text{Creg}} \ \wedge\ q_{\text{Wptr}} = s_{\text{Wptr}} \ \wedge\ q_{\text{Oreg}} = 0 \ \wedge\ q_{\text{Eflg}} = s_{\text{Eflg}} \ \wedge$$
$$q_{\text{Mem}} = s_{\text{Mem}} \ \wedge\ q_{\text{In}} = s_{\text{In}} \ \wedge\ q_{\text{Out}} = s_{\text{Out}} \ \wedge\ \mathcal{CC}_{\text{Cmds}}(s_{\text{PrA}}, q_{\text{PrA}}) \ \wedge\ \mathcal{CC}_{\text{Cmds}}(s_{\text{PrB}}, q_{\text{PrB}})$$

We have to prove in particular that the $\mathbf{TC}_1$ machine simulates the TASM machine. Since one abstract TASM instruction is compiled into a sequence of $\mathbf{TC}_1$ instructions, the proof will be a classical $1 : n$ simulation proof for state machines. More specifically, according to the compiling specification, one TASM instruction i is compiled into a pfix/nfix chain (of length ≥ 0) followed by another instruction which corresponds to i. Instead of using the general n-step relation $Rc(c_1, c_2)$ which holds for a starting configuration c_1 and a termating configuration c_2, we define a special n-step relation which follows the idea of the compilation: $R_{\text{stop}}(s, q)$ holds, if $\mathit{Effects}^*(s, q)$ holds, where $\mathit{Effects}^*$ denotes the reflexive and transitive closure of *Effects*, and if there is a configuration sequence $s = c_1, c_2, \ldots, c_n = q$ such that in each configuration $c_2, c_3, \ldots, c_{n-1}$ the operand register is not cleared and in state $c_n = q$ the register finally is cleared. This corresponds to a pfix/nfix chain followed by some other instruction since all but the prefix instructions will clear the operand register after they have been executed. The $1 : n$ L-simulation theorem (preservation of partial correctness) is as follows (see also Fig. 23). Note that the compiling relation $\mathcal{CC}_{\text{Cmds}}$ is part of the definition of ρ.

Theorem 6.2 ($\mathbf{TC}_1$ L-simulates TASM).

$$\forall s, t, \sigma.\ R_{\text{stop}}(s, t) \ \wedge\ \rho(\sigma, s) \ \Rightarrow \exists \tau.\ \rho(\tau, t) \ \wedge\ \mathit{Effects}(\sigma, \tau)$$

$$\begin{tikzcd}
\mathit{Conf}_{\text{TASM}} \ni \sigma \arrow[r, "i"] \arrow[d, "\rho"'] & \tau \in \mathit{Conf}_{\text{TASM}} \arrow[d, "\rho"] \\
\mathit{Conf}_{\text{TC}_1} \ni s \arrow[r, "bl"] & t \in \mathit{Conf}_{\text{TC}_1}
\end{tikzcd}$$

where $\mathcal{CC}_{\text{Cmd}}(i, bl)$

Figure 23. $\mathbf{TC}_1$ $1 : n$ simulates TASM

In order to accomplish the simulation proof, the main effort lies in the proof of the following *prefix property*: the property expresses that when executing the pfix/nfix chain

$$\texttt{pfix}(e_0) \cdot \texttt{pfix}(e_1) \cdots [\texttt{pfix} \mid \texttt{nfix}](e_{n-1}) \cdot \mathit{opr}(e_n)$$

without the last instruction $\mathit{opr}(e_n)$ then the operand register Oreg will be loaded with the value

$$((e \gg 4) \ll 4) \quad \text{which equals to } (e \wedge 240),$$

where $\gg$ and $\ll$ denote the right and left shift operations, respectively. Formally stated ($\mathit{lead}(l)$ returns list l without the last element):

Lemma 6.1 (Prefix Lemma).

$$\forall s, u, i, w.\ s_{\text{Oreg}} = 0 \ \wedge\ \neg s_{\text{Eflg}} \ \wedge\ s_{\text{PrB}} = \mathit{lead}(\mathit{prefix}(i, w)) \cdot u$$
$$\Rightarrow \mathit{Effects}^*(s, s[\mathit{Oreg} ::= \mathit{shiftL}(\mathit{shiftR}(w, 4), 4), \mathit{PrA} ::= s_{\text{PrA}} \cdot \mathit{lead}(\mathit{prefix}(i, w)), \mathit{PrB} ::= u])$$

The main proof of this compilation phase (6.1) is then straightforward and by induction using the transitive closure of the n-step relation R_{stop} and $\mathit{Effects}^*$ for TASM. The complete proof effort for this compilation step comprises approximately 600 manual proof interactions.

7 Transputer Backend: Program in Memory

The $\mathbf{TC}_1$ machine presented in the last section is very close to the Transputer base model as introduced in Sect. 5.1 and illustrated in Figure 13. Instructions are bytes consisting of the 4-bit operation code and the 4-bit operand. What remains to be implemented is the abstract symbolic instruction pointer represented by two byte sequences $PrA \cdot PrB$ where the current instruction to be executed is the first one of PrB. Instead of using an instruction register the contents of which points to a valid program address, the complete program m is made explicit in machine $\mathbf{TC}_1$ by the concatenation of the two byte sequences $m = PrA \cdot PrB$. Hence, the task of this final step is to integrate this program into the Transputer memory which then contains both the binary code as well as word datas. In Section 5.1 we already had to specify that the start addresses of the program modules are available through a subroutine jump table. A global constant *PgrStart* denotes the start address of the program in memory. This concept has to be specified on the level of TASM since the compilation of $\mathbf{C}^{int}$ procedures makes use of an absolute jump (`gcall`).

A program of the Transputer base model $\mathbf{TC}_0$ corresponds exactly to a $\mathbf{TC}_1$ program: it consists of two data modules (word sequences), a list of code modules, and a main code sequence (the main program):

$$TC0_{\mathrm{prg}} ::= Word^* \times Word^* \times (byte^*)^* \times byte^*$$

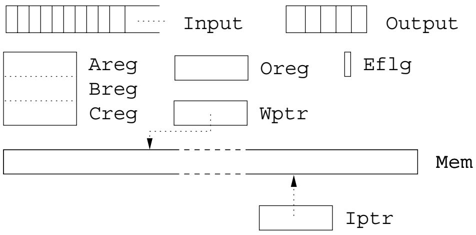


Figure 24. Machine configuration of $\mathbf{TC}_0$

The configurations of both machines differ only in the representation of the instruction pointer: $\mathbf{TC}_0$ configurations include an instruction register of type *Word* while $\mathbf{TC}_1$ configurations use the symbolic instruction pointer given by the two components c_{PrA} and c_{PrB}. Figure 24 illustrates the machine configuration of $\mathbf{TC}_0$. Its type is given by

$conf_{\mathrm{TC}_0} ::=$
$(Word \times Word \times Word) \times WordAddr \times Word \times bool \times (Word \rightarrow Word) \times seq[byte] \times byte^* \times Word$

In the following, c_{Areg}, c_{Breg}, c_{Creg}, c_{Wptr}, c_{Oreg}, c_{Eflg}, c_{Mem}, c_{In}, c_{Out}, and c_{Iptr} denote the respective (state) components of a configuration c.

The effect of each instruction is specified using a relation *Effects$_0$*. For the more abstract machine $\mathbf{TC}_1$ a terminating configuration is one where c_{PrB} is empty. In order to compare the two machines, we thus need a corresponding termination notion for machine $\mathbf{TC}_0$. For this purpose, the machine has a global constant *PgrLength* denoting the length of the machine program. This constant is later identified with the sum of the lengths of the $\mathbf{TC}_1$ code sequences c_{PrA} and c_{PrB}. The machine then "terminates" in state c, if $c_{\mathrm{Iptr}} = plus(PgrStart, PgrLength)$, that is, the instruction pointer points to the end of the program, where *PgrStart* denotes the global constant inherited from the definition of TASM and *plus* is defined as the addition on integers, in case there is no overflow (see Sect. 5.1). An instruction is now retrieved from memory at the address where the instruction pointer points to:

$$ByteMem(c_{\mathrm{Mem}})(c_{\mathrm{Iptr}})$$

Here, *ByteMem* denotes another (equivalent) view of the machine memory where byte values are stored at (byte) addresses. The (normal) update of the instruction pointer is simply defined by

$$c_{\mathrm{Iptr}}' = plus(c_{\mathrm{Iptr}}, 1).$$

The effects of $\mathbf{TC}_0$ and $\mathbf{TC}_1$ instructions are equal. The n-step relation $Rc(c_1, c_2)$ from some starting configuration c_1 to some final configuration c_2 is defined as for $\mathbf{TC}_1$. In this case, a final configuration is a terminating configuration where the error flag is not set. Program semantics for machine $\mathbf{TC}_0$ is defined by a predicate $p_{\mathbf{TC_0}}$ in the same way as for $\mathbf{TC}_1$.

Since only the machine configurations are different and both machines are defined on identical code, no compilation step is necessary for this phase. Correctness is stated as for the last step: for any program p, whenever the semantics of the program is defined on machine $\mathbf{TC}_0$ for some input byte stream bs and output byte list bl, this is also the case for p on machine $\mathbf{TC}_1$ for the same bs and bl:

Theorem 7.1 (Correctness of the final Step).

$$\forall p, bs, bl.\ p_{\mathbf{TC_0}}(p)(bs)(bl) \Rightarrow p_{\mathbf{TC_1}}(p)(bs)(bl)$$

As for the other compilation phases, first of all, the configurations of both machines are to be related by means of a data representation relation ρ. Here, this reduces to relate the symbolic instruction pointer $(c_{\text{PrA}}, c_{\text{PrB}})$ of $\mathbf{TC}_1$ with the instruction register c_{Iptr} of $\mathbf{TC}_0$. In particular, it is required

- that either the instruction pointer points to a valid program address or a terminating configuration has been reached and that
- the program $m = c_{\text{PrA}} \cdot c_{\text{PrB}}$ is available in memory with start address *PgrStart*.

The first property is formalized using a predicate

$$\mathit{PgrAddr?}(a) ::= (\mathit{PgrStart} \leq a < \mathit{plus}(\mathit{PgrStart}, \mathit{PgrLength}))$$

The second property is parameterized with the memory m and two byte sequences p_1 and p_2:

$$\begin{aligned} &\mathit{PgrInMem?}(m)(p_1, p_2) ::= \\ &\qquad (|p_1| + |p_2| = \mathit{PgrLength}) \ \wedge\ [\forall i < \mathit{PgrLength}.\ \mathit{ByteMem}(m)(\mathit{PgrStart} + i) = (p_1 \cdot p_2).i] \end{aligned}$$

This leads to the following definition of the data representation relation ρ between $\mathbf{TC}_1$ states s and $\mathbf{TC}_0$ states q:

$$\begin{aligned} &\rho(s, q) ::= \\ &\quad (\mathit{PgrAddr?}(q_{\text{Iptr}}) \ \vee\ \mathit{terminated?}(q_{\text{Iptr}})) \wedge \\ &\quad q_{\text{Areg}} = s_{\text{Areg}} \ \wedge\ q_{\text{Breg}} = s_{\text{Breg}} \ \wedge\ q_{\text{Creg}} = s_{\text{Creg}} \ \wedge\ q_{\text{Wptr}} = s_{\text{Wptr}} \ \wedge\ q_{\text{Oreg}} = s_{\text{Oreg}} \ \wedge \\ &\quad q_{\text{Eflg}} = s_{\text{Eflg}} \ \wedge\ q_{\text{Mem}} = s_{\text{Mem}} \ \wedge\ q_{\text{In}} = s_{\text{In}} \ \wedge\ q_{\text{Out}} = s_{\text{Out}} \ \wedge \\ &\quad \mathit{PgrInMem?}(q_{\text{Mem}})(s_{\text{PrA}}, s_{\text{PrB}}) \ \wedge\ q_{\text{Iptr}} = \mathit{plus}(\mathit{PgrStart}, |s_{\text{PrA}}|) \end{aligned}$$

As for the last phase, we must prove that the lower level machine simulates the more abstract one. In this case, it is an easy 1 : 1 simulation proof: one step of $\mathbf{TC}_0$ corresponds to one step of $\mathbf{TC}_1$. However, one additional important requirement is necessary to accomplish the simulation proof: it must be assured that the $\mathbf{TC}_0$ program does not modify itself while being executed. In fact, a memory write access into the program area is possible on the $\mathbf{TC}_0$ level. However, as this memory area starts above *MemTop* (*PgrStart* > *MemTop*), the semantics of TASM and $\mathbf{TC}_1$ are defined non-deterministically in this case (see the semantics of the TASM store instructions in Appendix G). The simulation theorem (depicted in Fig. 25) is then as follows:

$$\begin{tikzcd} \mathit{Conf}_{\text{TC}_1} \ni \sigma \arrow[r, "i"] \arrow[d, "\rho"'] & \tau \in \mathit{Conf}_{\text{TC}_1} \arrow[d, "\rho"] \\ \mathit{Conf}_{\text{TC}_0} \ni s \arrow[r, "i"] & t \in \mathit{Conf}_{\text{TC}_0} \end{tikzcd}$$

Figure 25. $\mathbf{TC}_0$ 1 : 1 simulates $\mathbf{TC}_1$

Theorem 7.2 ($\mathbf{TC}_0$ simulates $\mathbf{TC}_1$).

$$\forall s, t, \sigma.\ \rho(\sigma, s) \ \wedge\ \mathit{Effects}_0(s, t) \ \Rightarrow\ \exists \tau.\ \mathit{Effects}(\sigma, \tau) \wedge\ \rho(\tau, t)$$

The proof of the simulation theorem is by a large case-analysis on the different kind of instructions. Since most of the cases are proved in the same way, PVS proof strategies are defined which enable a tremendous reduction of the manual proof effort. Altogether, the verification effort for this last phase comprises approximately 350 proof steps.

8 Combining the Compilation Phases

In the last sections, the correctness of each of the five compilation steps has been outlined. However, what is needed in order to accomplish the correctness of the complete compilation process from ComLisp down to binary Transputer code as stated in the main theorem 1.1 is a combination of the single proofs. The commuting diagrams of the different compilation phases have to be stacked as depicted in Fig. 26. The different correctness theorems for program compilation must be combined logically in order to prove the main correctness conjecture. First, the combined compiling relation from ComLisp down to $\mathbf{TC}_0$ has to be defined: Let p be a ComLisp program, q be a $\mathbf{TC}_0$ program, p' be a $\mathbf{C}^{int}$ program, p'' be a TASM program and p''' be a $\mathbf{TC}_1$ program.

$$\begin{tikzcd}
seq[char] \arrow[r, "p_{\text{ComLisp}}"] \arrow[d, "id"'] \arrow[dr, phantom, "\circlearrowleft"] & char^* \arrow[d, "id"] \\
seq[char] \arrow[r, "p_{\mathbf{SIL}}"'] \arrow[d, "id"'] \arrow[dr, phantom, "\circlearrowleft"] & char^* \arrow[d, "id"] \\
seq[char] \arrow[r, "p_{\mathbf{C}^{int}}"'] \arrow[d, "char2byte"'] \arrow[dr, phantom, "\circlearrowleft"] & char^* \arrow[d, "char2byte"] \\
seq[byte] \arrow[r, "p_{\mathbf{TASM}}"'] \arrow[d, "id"'] \arrow[dr, phantom, "\circlearrowleft"] & byte^* \arrow[d, "id"] \\
seq[byte] \arrow[r, "p_{\mathbf{TC_1}}"'] \arrow[d, "id"'] \arrow[dr, phantom, "\circlearrowleft"] & byte^* \arrow[d, "id"] \\
seq[byte] \arrow[r, "p_{\mathbf{TC_0}}"'] & byte^*
\end{tikzcd}$$

Figure 26. Combining the commuting diagrams

Definition 8.1 (Global compiling relation).

$$\mathcal{C}(p,q) ::= \exists p', p'', p'''.\ \mathcal{CC}_{prog}(\mathcal{C}_{prog}(p), p') \ \wedge\ \mathcal{CC}_{prog}(p', p'') \ \wedge\ \mathcal{CC}_{prog}(p'', p''') \ \wedge\ q = p'''$$

The proof of the main theorem 1.1 is then as follows:

Proof: In order to be able to apply the compilation correctness theorems of the different phases, namely the theorems 3.1, 4.1, 5.1, 6.1, 7.1, it must be proved that the programs resulting from the first two compilation phases are well-formed:

- for any ComLisp program p:
$$wf(p) \Rightarrow wf(\mathcal{C}_{\text{prog}}(p))$$
- for any SIL program p and $\mathbf{C}^{int}$ program q:
$$wf(p) \wedge\ \mathcal{CC}_{\text{prog}}(p,q) \Rightarrow wf(q)$$

Consider the proof of the first lemma. In order to show that a well-formed ComLisp program is compiled into a well-formed SIL program it suffices to show that well-formed ComLisp forms are compiled into well-formed SIL statements:

$$(wf(e, \zeta, \gamma, \Gamma, cl, sl) \ \wedge\ dom(\rho) = \zeta) \ \Rightarrow wf(\mathcal{C}_{\text{form}}(e, \gamma, \rho, k), \mathcal{C}_{\text{defs}}(\Gamma)(\gamma), sl, cl, |\gamma|)$$

The proof of this property is by measure-induction using the size of the expression e as measure.

Analogously, in order to show that well-formed SIL programs are compiled into well-formed $\mathbf{C}^{int}$ programs, we have to show, that

- well-formed SIL statements are compiled into well-formed $\mathbf{C}^{int}$ statements (by measure-induction)
- the core runtime system (the set of $\mathbf{C}^{int}$ procedures implementing the ComLisp operators) is well-formed. For each procedure it must be showed that its body is well-formed and the specified stack index is greater or equal the maximum index of the body. (The proof is by unfolding the definition of *maxindex* for each procedure).

□

9 Discussion, Results

In this paper we have reported on the construction of a correct realistic bootstrap compiler compiling a subset of Common Lisp into binary Transputer code. We have focused on the formal, mechanically supported verification of the compiling specification. The formal verification has been carried out completely using the PVS specification and verification system.

It is hard to give an estimation of the amount of work invested in the final verification, since we started the verification first 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 verification for all phases is about three person-years. This case study in formal verification is, so far, one of the largest one found in the literature. Comparable formalizations are Clinc's stack of system components (using the Boyer-Moore prover) [Moo89] and the Prolog to WAM case study done with the KIV system [Sch99].

What did we learn from this verification project?

First of all, using a mechanical proof system, we have found several errors in the original specification [GH98b] which have not been recognized so far. A serious error occurred in the compiler's back-end where necessary overflow checks have been omitted which will lead to erroneous code for large values. It was an easy task to correct the errors. However, as we are constructing an initial correct compiler this has the consequence that the compiler implementation must be corrected too and its verification both on the high and lower level must be repeated. However, this effort is small comparable to the total verification effort due to the sophisticated modular verification technique [Hof98] for low-level implementation verification.

Second, as stated above, we have started this verification project on easier sublanguages and abstract machine architectures in order to experiment with this kind of verification problems and to learn about the representation of the languages, specifications, the appropriate choice of semantics, induction principles, invariants, and proof techniques. For instance, in order to accomplish compiling correctness proofs, a counter has been introduced into the semantics of statements to formulate a suitable induction principle for the chosen notion of correctness. Then we stepwise increased the complexity of the verification problems by extending the languages and machine architectures trying to reuse specifications and proofs as much as possible. This engineering methodology has turned out to be a suitable technique in order to carry out such a large new verification project. Of course, for a new similar compiling verification project the effort would be much smaller since one can profit from the gained experience.

Third, the choice of the intermediate languages and compilation phases originally has been made in order to accomplish the compiler implementation verification especially on the lower level. However, in order to carry out the compiling verification, one would had defined some of the intermediate languages slightly different. Hence there is a trade-off between compiling verification and compiler verification. The compiling verification of some of the phases requires knowledge about the compilation of the previous phase, that is, the phases are not completely independent of each other. The general situation is as follows: suppose we consider two compilation phases ($L_{i-1} \rightarrow L_i$ and $L_i \rightarrow L_{i+1}$) as part of a compilation process from some source language L_1 to some target language L_n:

$$L_1 \rightarrow \cdots L_{i-1} \rightarrow L_i \rightarrow L_{i+1} \rightarrow \cdots L_n$$

In general, the set of target programs q which are a possible compilation of a source program $p \in L_{i-1}$ is a real subset of the target language: $\{q \in L_i \mid \mathcal{CC}(p,q)\} \subset L_i$. This subset often has specific properties which are necessary and must be explicitly expressed in order to establish the correctness of the next subsequent phase. Suppose P is such a property and suppose that P is a precondition for the correctness of the second compilation phase from L_i to L_{i+1}. In order to combine the correctness proofs of the two phases to a correctness proof of $L_{i-1} \rightarrow L_{i+1}$ there are different possibilities:

1. Show that the property holds for each possible compilation of a L_{i-1} program: for all $p \in L_{i-1}$ and $q \in L_i$, if $\mathcal{CC}(p,q)$ then the property P holds for q. Often P is a statically decidable property such as special well-formed conditions. However, sometimes P expresses dynamic properties of the code such as the fact that some memory region is not changed when evaluating the code. In this case, P could be introduced as an additional (target) invariant into the correctness proof of the first phase or an additional theorem can be established showing that the target code fulfills the dynamic property P.

2. an alternative method is to introduce non-determinism into the semantics of L_i, that is, to weaken the semantics of L_i in such a way, that in case P does not hold some arbitrary successor state will be allowed. Since P is a property which holds for the first phase, it should be easy to accomplish the correctness proof with the weaker semantics. The proof becomes harder, however, since it must be explicitly proved that the non-deterministic cases (the chaotic cases) will never occur when symbolically executing the compiled code. Otherwise, it would not be possible to prove the correctness of the compilation from L_{i-1} to L_i since some arbitrary L_i state can not be related with some specific L_{i-1} state. This non-determinism is then utilized in order to accomplish the proof of the subsequent compilation phase from L_i to L_{i+1}. For the correctness of this phase, we do not need the precondition P any more. At the level of L_{i+1} there is a defined state transition in the case P does not hold. However, for each such target L_{i+1} state we are then trivially able to find a L_i state transition (since every transition is allowed). In particular, we may choose a L_i state τ such that it corresponds to the target state t with respect to the data representation relation ($\rho(\tau, t)$) which relates L_i states with L_{i+1} states. In some cases it might be possible however, that there are L_{i+1} states which do not have a corresponding L_i state. For these cases we then have to make ρ more total in a way such that we are always able to find a corresponding source state τ. This second method has the advantage that no additional invariant is required and no additional precondition must be established for the compiled code. We just have introduced some knowledge (the property P) into the semantics of L_i. This reflects the fact that only the subset of L_i is actually considered which results from compiling L_{i-1} programs. Certainly, the semantics of the source language L_1 and the final target language L_n are fixed. However, introducing non-determinism into the semantics of the intermediate languages $L_2, L_3, \ldots, L_{n-1}$ when necessary, is an allowed and useful technique.

For our initial bootstrap compiler we have *exclusively* utilized the second proof technique both in the front-end and back-end since it has been turned out that it is much more suitable to enable a modular verification and easy combination of the different phases. The only preconditions which are necessary to combine the phases are easy statically decidable well-formedness conditions. In the front-end, for example, an arbitrary SIL program does not have a stack machine behavior, although the language has been defined in order to be used as a stack machine. However, a compiled ComLisp program has in fact a stack machine behavior. We solved this problem by slightly introducing non-determinism into the semantics of SIL. For SIL, an additional top-of-stack pointer has been introduced to model normal and abnormal stack machine behavior. In the back-end, the correctness of the compilation from TASM to $\mathbf{TC}_1$ depends on correct jump distances. Whenever a relative TASM jump (such as `cj` w or `j` w) has a jump distance w which does not start at the beginning of an abstract TASM instruction (that is, a jump within a pfix/nfix chain), then the machine is defined to behave non-deterministically such that any successor state will be allowed. For $\mathbf{TC}_1$ a jump within a pfix/nfix chain is certainly allowed but in this case, we can easily find a TASM state which corresponds to the $\mathbf{TC}_1$ state. Changing the semantics of SIL and TASM for example in order to establish the correctness of one compilation phase required to repeat the verification of the previous phase. In our case, nearly all of the proofs could be reused, a few of them required minor modifications.

Fourth, for all phases, we have utilized a relational semantics. For the languages in the front-end, natural semantics (big-step semantics) has been used, and for the Transputer base model and its abstractions, a transition semantics has been employed. The main advantage of this style of semantics is its readability and it is easy to validate against informal language descriptions. In addition, the semantics is straightforward to formalize in a system like PVS and has proved to be a suitable vehicle for compiling verification.

Fifth, in order to decrease the manual verification effort of the compilation theorems, proof strategies have heavily been employed. This technique could successfully been applied here since the proofs of some of the compiling theorems (for a specific compilation phase) follow a similar scheme. Nevertheless, the total manual interaction effort to accomplish the proofs is very high (approx. 18,000 proof steps).

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

Acknowledgements

We like to thank our colleagues of the *Verifix* project for their constructive collaboration and many discussions on this subject. We debt very special thanks to Hans Langmaack for his inspiring ideas and uncountably many contributions, and for his leading influence on the project.

References

[Bro92] M. Broy. Experiences with Software Specification and Verification using LP, the Larch Proof Assistant. TR 93, Digital Systems Research Center, Palo Alto, 1992.

[BS98] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, *23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 17–35. Springer-Verlag, 1998.

[CM86] 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.

[Cur94] P. Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.

[Dol00] Axel Dold. *Formal Software Development using Generic Development Steps*. Logos-Verlag, Berlin, 2000. PhD Thesis, University of Ulm, Germany.

[DV00] Axel Dold and Vincent Vialard. Formal Verification of a Compiler Back-end Generic Checker Program. In *Proc. of the Andrei Ershov Third International Conference Perspectives of System Informatics (PSI'99)*, number 1755 in LNCS, pages 470–480. Springer-Verlag, 2000.

[DvHPR97] A. Dold, F.W. von Henke, H. Pfeifer, and H. Rueß. Formal Verification of Transformations for Peephole Optimization. In P. Lucas J. Fitzgerald, C.B. Jones, editor, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, volume 1313 of *LNCS*, pages 459–472, 1997.

[GGZ98] W. Goerigk, T. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, pages 108 – 122, Wien, New York, 1998. Springer Verlag.

[GH98a] W. Goerigk and U. Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Techn. Rep. 9812, Inst. f. Informatik, Univ. Kiel, 1998.

[GH98b] W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *LNCS*, pages 122 – 136, 1998.

[GH98c] W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Techn. Rep. 9713, Inst. f. Informatik, Univ. Kiel, 1998.

[GL01a] W. Goerigk and H. Langmaack. Compiler Implementation Verification and Trojan Horses. In D. Bainov, editor, *Proc. 9th International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, 2001.

[GL01b] W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Technical Report 2015, Inst. f. Informatik, Univ. Kiel, 2001.

[GMR$^+$92] 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.

[Goe97] W. Goerigk. Towards Rigorous Compiler Implementation Verification. In R. Berghammer and F. Simon, editors, *Workshop on Programming Languages and Fundamentals of Programming*, pages 118 – 126, Avendorf, Germany, 1997.

[Goe00] W. Goerigk. Compiler Verification Revisited. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer, 2000.

[GZ99] G. Goos and W. Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201 – 230. Springer Verlag, 1999.

[Hof98] U. Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection.* PhD thesis, Rep. 9814 Technical Faculty, University of Kiel, 1998.

[Inm88] Inmos Limited. *Transputer instruction set: A compiler writer's guide*, 1988.

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

[Joy89] 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.

[Moo89] J S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4), 1989.

[MOW99] M. Müller-Olm and A. Wolf. On Excusable and Inexcusable Failures: Towards an Adequate Notion of Translation Correctness. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods FM'99*, volume 1709 of *LNCS*, pages 1107–1127, Toulouse, France, 1999. Springer-Verlag.

[MP67] J. McCarthy and J. A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science.* American Mathematical Society, 1967.

[ORSvH95] 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.

[Pol81] W. Polak. *Compiler Specification and Verification*, volume 124 of *LNCS.* Springer-Verlag, New York, NY, USA, 1981.

[Sch99] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen.* PhD thesis, University of Ulm, 1999.

[Str02] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS.* Springer Verlag, 2002.

[Tho84] K. Thompson. Reflections on Trusting Trust. *CACM*, 27(8):761–763, 1984.

[Yu93] Yuan Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. Technical report, Digital Systems Research Center, Palo Alto, USA, 1993.

A Semantics of ComLisp

A.1 A Natural Semantics

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 $s_{\text{var}} : \mathit{Ident} \rightarrow \mathit{SExpr}$. 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 $v_1 : \mathit{uop} \rightarrow v_2$ for unary operators *uop*, and $v_1, v_2 : \mathit{bop} \rightarrow v$ 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. Note that there is no rule for the *abort* expression, since for any state s there is no state q and value v with $\Gamma \vdash s : \mathit{abort} \rightarrow (v, q)$.

- constants, variables

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

- assignment:

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

- sequential composition:

$$\Gamma \vdash s : \mathit{progn}([]) \rightarrow (\mathit{NIL}, s) \qquad \frac{\Gamma \vdash s : e \rightarrow (v, q)}{\Gamma \vdash s : \mathit{progn}(e) \rightarrow (v, q)} \qquad \frac{\begin{array}{c} n \geq 2 \\ \Gamma \vdash s : e_1 \rightarrow (v_1, q_1) \\ \Gamma \vdash q_1 : \mathit{progn}(e_2, \ldots, e_n) \rightarrow (v, q) \end{array}}{\Gamma \vdash s : \mathit{progn}(e_1, \ldots, e_n) \rightarrow (v, q)}$$

- conditional:

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

- do-loop:

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

- call of user-defined functions:

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

- 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:

$$\frac{\Gamma \vdash s : e \rightarrow (v, q)}{\Gamma \vdash s : \mathit{list}{*}(e) \rightarrow (v, q)} \qquad \frac{\begin{array}{c}\Gamma \vdash s : e_1 \rightarrow (v_1, q_1)\\ \Gamma \vdash q_1 : \mathit{list}{*}(e_2, \ldots, e_n) \rightarrow (v_2, q)\ (n \geq 2)\end{array}}{\Gamma \vdash s : \mathit{list}{*}(e_1, \ldots, e_n) \rightarrow (\mathit{cons}(v_1, v_2), q)}$$

- cond form:

$$\Gamma \vdash s : \mathit{cond}() \rightarrow (\mathit{NIL}, s) \qquad \frac{\begin{array}{c}\Gamma \vdash s : p_1 \rightarrow (v_1, q_1) \quad (v_1 \neq \mathit{NIL})\\ \Gamma \vdash q_1 : e_1 \rightarrow (v, q)\end{array}}{\Gamma \vdash s : \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n) \rightarrow (v, q)}$$

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

- input/output:

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

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

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

A.2 A Structural Operational Semantics

We provide a (dynamic) small-step semantics, also called structural operational semantics (SOS) for the language ComLisp. Note that this semantics is not used for the compiling verification outlined in this report.

The basic idea is to define a set of axioms and inference rules in a bottom-up style: execution of smaller program parts is integrated into the execution of larger program parts. In contrast to a big-step semantics, SOS does not describe state transitions for entire expressions or programs. An SOS semantics explicitly defines a term-rewriting system that rewrites the program during execution until the empty program is reached.

A ComLisp state consists of the input stream of s denoted by s_{input}, the output list of s denoted by s_{output}, and the variable state of s denoted by $s_{\mathrm{var}} : [\mathit{Ident} \rightarrow \mathit{SExpr}]^*$. The variable state is given by a stack (list) of mappings, associating s-expressions with identifiers. In order to execute a function call the stack of mappings is extended with the new local environment, which is popped from the stack after the function body has been evaluated.

ComLisp expressions may have side-effects, hence they denote state transitions transforming states to pairs of result values (s-expressions) and result state. The semantics of Comlisp forms makes use of two kinds of transitions:

- $< e, s > \rightarrow\ (v, q)$ denotes a termination step of the rewriting system: (completely) evaluating expression e in state s yields the value v and result state q.
- $< e, s > \rightarrow < e', s' >$ denotes an intermediate evaluation step: execution of e in s yields a new expression e' to be evaluated and a succeeding state s'.

For the definition of the semantics, we make the following assumptions:

- 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 the first occurrence of x by v.
- the user-defined functions are available in a global constant environment Γ which is omitted in the rules for readability purposes. That is, instead of $\Gamma \vdash < e, s > \rightarrow < e', s' >$ we simply write $< e, s > \rightarrow < e', s' >$.
- ComLisp operators denote partial functions on s-expressions which is expressed by two relations: relation $< \mathit{uop}, v_1 > \rightarrow\ v_2$ for unary ComLisp operators *uop*, and $< \mathit{bop}, v_1, v_2 > \rightarrow\ v$ for binary operators *bop*.

- *push*(E, s) extends the variable state of s by E: $s[s_{\text{var}} := E \cdot s_{\text{var}}]$; *pop*$(s)$ removes the first element of the variable state of s.

The formalization of the small-step semantics is heavily influenced by the formalization of our big-step (natural) semantics for ComLisp. In particular, the axioms (atomic rules) of the big-step semantics used for the atomic ComLisp expressions become axioms in the small-step semantics (e.g. the rules for constants and variables). The structural big-step inference rules for the non-atomic ComLisp expressions are transformed into rules which explicitly specify the left-to-right evaluation of expressions. There are at least two rules for each composed expression: one rule for an intermediate evaluation step stating that if a sub-expression e is rewritten to some other sub-expression e' while changing the state from s to s' then the expression *context*(e) is rewritten to *context*(e') with the same state change:

$$\frac{< e, s > \rightarrow < e', s' >}{< \mathit{context}(e), s > \rightarrow < \mathit{context}(e'), s' >}$$

The other rule then specifies the terminating case:

$$\frac{< e, s > \rightarrow \ (v, q)}{< \mathit{context}(e), s > \rightarrow \ (\mathit{result\_value}, \mathit{result\_state})}$$

The SOS inference rules which specify the semantics of ComLisp expressions are defined as follows:

- constants, variables

$$< c, s > \rightarrow \ (c, s) \qquad < x, s > \rightarrow \ (s(x), s)$$

- assignment: (two rules)

$$\frac{< e, s > \rightarrow < e', q >}{< x := e, s > \rightarrow < x := e', q >} \qquad \frac{< e, s > \rightarrow \ (v, q)}{< x := e, s > \rightarrow \ (v, q[x \leftarrow v])}$$

- sequential composition:

$$< \mathit{progn}([]), s > \rightarrow \ (\mathit{NIL}, s) \qquad < \mathit{progn}(e), s > \rightarrow < e, s >$$

for $n \geq 2$:

$$\frac{< e_1, s > \rightarrow < e_1', q >}{< \mathit{progn}(e_1, \ldots, e_n), s > \rightarrow < \mathit{progn}(e_1', \ldots, e_n), q >}$$

$$\frac{< e_1, s > \rightarrow \ (v_1, q_1)}{< \mathit{progn}(e_1, \ldots, e_n), s > \rightarrow < \mathit{progn}(e_2, \ldots, e_n), q_1 >}$$

- conditional:

$$\frac{< e_1, s > \rightarrow < e_1', q >}{< \mathit{if}(e_1, e_2, e_3), s > \rightarrow < \mathit{if}(e_1', e_2, e_3), q >}$$

$$\frac{< e_1, s > \rightarrow \ (NIL, q)}{< \mathit{if}(e_1, e_2, e_3), s > \rightarrow < e_3, q >} \qquad \frac{< e_1, s > \rightarrow \ (v, q) \quad (v \neq NIL)}{< \mathit{if}(e_1, e_2, e_3), s > \rightarrow < e_2, q >}$$

- do-loop:

$$\frac{< c, s > \rightarrow < c', q >}{< do(c, \mathit{body}), s > \rightarrow < do(c', \mathit{body}), q >}$$

$$\frac{< c, s > \rightarrow \ (v, q) \quad (v \neq NIL)}{< do(c, \mathit{body}), s > \rightarrow \ (v, q)}$$

$$\frac{< c, s > \rightarrow \ (NIL, q)}{< do(c, \mathit{body}), s > \rightarrow < \mathit{progn}(\mathit{body}, do(c, \mathit{body})), q >}$$

- unary operators (*uop* denotes some ComLisp unary operator):

$$\frac{< e, s > \rightarrow < e', q >}{< \mathit{uop}(e), s > \rightarrow < \mathit{uop}(e'), q >} \qquad \frac{\begin{array}{c}< e, s > \rightarrow \ (v, q) \\ < \mathit{uop}, v > \rightarrow \ v'\end{array}}{< \mathit{uop}(e), s > \rightarrow \ (v', q)}$$

- binary operators (*bop* denotes some ComLisp binary operator):

$$\frac{< e_1, s > \to < e_1', s' >}{< bop(e_1, e_2), s > \to < bop(e_1', e_2), s' >} \qquad \frac{< e_1, s > \to\ (v_1, q_1)}{< bop(e_1, e_2), s > \to < bop(v_1, e_2), q_1 >}$$

$$\frac{< e_2, s > \to < e_2', s' >}{< bop(v, e_2), s > \to < bop(v, e_2'), s' >} \qquad \frac{\begin{array}{c}< e_2, s > \to\ (v_2, s_2)\\ < bop, v_1, v_2 > \to\ v\end{array}}{< bop(v_1, e_2), s > \to\ (v, s_2)}$$

- call of user-defined functions:
  we make use of an additional auxiliary parameter E denoting the local variable environment (a list of associations) which is constructed during the left-to-right evaluation of the function parameters. Note that the extension of the environment E does not change the state q; E is an additional parameter. In particular this means that each parameter e_i is evaluated in the same environment. After the function parameters have been evaluated completely, the variable state (the stack of mappings) is extended with the constructed environment and then the body of the function is evaluated. We further suppose that $f(x_1, \ldots, x_n) \leftarrow body\ \in \Gamma$.

  - parameterless functions:

$$< f(), s > \to < body, s >$$

  - evaluation of parameters

$$\frac{< e_1, s > \to < e_1', s' >}{< f(e_1, \ldots, e_n), s > \to < f(e_1', \ldots, e_n), s' >}$$

$$\frac{< e_1, s > \to\ (v_1, q_1)}{< f(e_1, \ldots, e_n), s > \to < f([], e_2, \ldots, e_n), q_1, [x_1 \leftarrow v_1] >}$$

$$\frac{< e_2, s > \to < e_2', q >}{< f([], e_2, \ldots, e_n), s, E > \to < f([], e_2', \ldots, e_n), q, E >}$$

$$\frac{< e_2, s > \to\ (v_2, q_2)}{< f([], e_2, \ldots, e_n), s, E > \to < f([], [], e_3, \ldots, e_n), q_2, [x_2 \leftarrow v_2] \cdot E >}$$

$$\vdots$$

  - evaluation of function body:
    Note that in the following rules the function body is used as an additional parameter in order to recognize the end of the evaluation of the body.

$$< f([], [], \ldots, []), q, E > \to < f([], [], \ldots, []), push(E, q), body >$$

$$\frac{< body, s > \to < body', s' >}{< f([], [], \ldots, []), s, body > \to < f([], [], \ldots, []), s', body' >}$$

$$\frac{< body, s > \to\ (v, q)}{< f([], [], \ldots, []), s, body > \to\ (v, pop(q))}$$

- let block:

  - empty parameter list

$$< let((); e), s > \to < e, s >$$

  - evaluation of parameters: similar as for function calls
  - evaluation of the let-body

$$< let([], [], \ldots, []; e), s, E > \to < let([], [], \ldots, []; e), push(E, s) >$$

$$\frac{< e, s > \to < e', s' >}{< let([], [], \ldots, []; e), s > \to < let([], [], \ldots, []; e'), s' >}$$

$$\frac{< e, s > \to\ (v, q)}{< let([], [], \ldots, []; e), s > \to\ (v, pop(q))}$$

- *list∗* operator:
  Here, an additional environment parameter E is used in order to save the values of the evaluated expressions.

$$< \mathit{list}{*}(e), s > \rightarrow < e, s >$$

$$\frac{< e_1, s > \rightarrow < e_1', s' >}{< \mathit{list}{*}(e_1, \ldots, e_n), s > \rightarrow < \mathit{list}{*}(e_1', \ldots, e_n), s' >}$$

$$\frac{< e_1, s > \rightarrow\ (v_1, q_1)}{< \mathit{list}{*}(e_1, \ldots, e_n), s > \rightarrow < \mathit{list}{*}([], e_2, \ldots, e_n), q_1, [v_1] >}$$

$$\frac{< e_2, s > \rightarrow < e_2', q >}{< \mathit{list}{*}([], e_2, \ldots, e_n), s, E > \rightarrow < \mathit{list}{*}([], e_2', \ldots, e_n), q, E >}$$

$$\frac{< e_2, s > \rightarrow\ (v_2, q_2)}{< \mathit{list}{*}([], e_2, \ldots, e_n), s, E > \rightarrow < \mathit{list}{*}([], [], e_3, \ldots, e_n), q_2, E \cdot v_2 >}$$

$$\vdots$$

$$< \mathit{list}{*}([], [], \ldots, []), q, E > \rightarrow\ (\mathit{mkcons}(E), q) \quad \text{where}$$

$$\mathit{mkcons}(v_1, \ldots, v_n) ::= \mathit{cons}(v_1, \mathit{cons}(v_2, \ldots, \mathit{cons}(v_{n-1}, v_n) \ldots))$$

- cond form:

$$< \mathit{cond}(), s > \rightarrow\ (\mathit{NIL}, s)$$

$$\frac{< p_1, s > \rightarrow < p_1', s' >}{< \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n), s > \rightarrow < \mathit{cond}(p_1' \rightarrow e_1, \ldots, p_n \rightarrow e_n), s' >}$$

$$\frac{< p_1, s > \rightarrow\ (v, q) \quad (v \neq NIL)}{< \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n), s > \rightarrow < e_1, q >}$$

$$\frac{< p_1, s > \rightarrow\ (NIL, q)}{< \mathit{cond}(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n), s > \rightarrow < \mathit{cond}(p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n), q >}$$

- input/output:

$$< \mathit{read\_char}, s > \rightarrow\ (\mathrm{first}(s_{\mathrm{input}}), s[s_{\mathrm{input}} := \mathrm{rest}(s_{\mathrm{input}})])$$

$$< \mathit{peek\_char}, s > \rightarrow\ (\mathrm{first}(s_{\mathrm{input}}), s)$$

$$\frac{< e, s > \rightarrow < e', s' >}{< \mathit{print\_char}(e), s > \rightarrow < \mathit{print\_char}(e'), s' >}$$

$$\frac{< e, s > \rightarrow\ (v, q)}{< \mathit{print\_char}(e), s > \rightarrow\ (v, q[q_{\mathrm{output}} := q_{\mathrm{output}} {+\!\!+} v])}$$

Finally, the semantics of a ComLisp program $p ::= d; x_1, \ldots, x_k; f_1, \ldots, f_n; e$ with declaration part d, global variables $x_1, \ldots, x_k$, function definitions $\Gamma := f_1, \ldots, f_n$ and main program (expression) e is given by a relation between an input stream *is* and output list *ol* as follows:

$$P_{\mathrm{sem}_{\mathrm{CL}}}(p)(\mathit{is}, \mathit{ol}) ::= \exists v, q.\ (\Gamma \vdash < e, \mathit{init} > \rightarrow\ (v, q))\ \wedge\ (q_{\mathrm{output}} = \mathit{ol})$$

where the initial state *init* is given by

$$\mathit{init} ::= (s_{\mathrm{input}} := \mathit{is}, s_{\mathrm{output}} := [], s_{\mathrm{var}} := [\lambda x.\mathit{NIL}])$$

B Semantics of SIL

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 $s_{\text{local}}(s_{\text{base}} + i)$, and write $s[i \leftarrow v]$ for $s[s_{\text{local}}(s_{\text{base}} + i) \leftarrow v]$.

- copy constant, copy local

$$\Gamma \vdash s : \mathit{copyc}(c, i) \rightarrow s[i \leftarrow c] \qquad \Gamma \vdash s : \mathit{copy}(i, j) \rightarrow s[j \leftarrow s(i)]$$

- copy from global to local memory:

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

- copy from local to global memory:

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

- conditional:

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

- sequential composition:

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

- function call:

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

- unary/binary operators:

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

- do-loop:

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

- input/output:

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

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

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

- $\mathit{list}*$

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

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

C Compiling ComLisp to SIL

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

- $\mathcal{C}_{\text{form}}(\mathit{abort},\gamma,\rho,k) = \mathit{abort}$
- $\mathcal{C}_{\text{form}}(c,\gamma,\rho,k) = \mathit{copyc}(c,k)$
- $\mathcal{C}_{\text{form}}(x,\gamma,\rho,k) = \begin{cases} \mathit{copy}(\rho(x),k) & \text{if } \rho(x) \text{ is defined} \\ \mathit{gcopy}(\gamma(x),k) & \text{otherwise} \end{cases}$
- $\mathcal{C}_{\text{form}}(x := e,\gamma,\rho,k) = \begin{cases} \mathit{sq}(\mathcal{C}_{\text{form}}(e,\gamma,\rho,k),\mathit{copy}(k,\rho(x))) & \text{if } \rho(x) \text{ is defined} \\ \mathit{sq}(\mathcal{C}_{\text{form}}(e,\gamma,\rho,k),\mathit{copyg}(k,\gamma(x))) & \text{otherwise} \end{cases}$
- $\mathcal{C}_{\text{form}}(\mathit{progn}([]),\gamma,\rho,k) = \mathit{copyc}(\mathit{NIL},k)$
- $\mathcal{C}_{\text{form}}(\mathit{progn}(e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \vdots \\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{if}(e_1,e_2,e_3),\gamma,\rho,k) = \mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \mathit{itef}(k,\mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k),\mathcal{C}_{\text{form}}(e_3,\gamma,\rho,k)) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{do}(e_1,e_2),\gamma,\rho,k) = \mathit{do}(k,\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k),\mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k))$
- $\mathcal{C}_{\text{form}}(\mathit{call}(f,()),\gamma,\rho,k) = \mathit{fcall}(f,k)$
- $\mathcal{C}_{\text{form}}(\mathit{call}(f,e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \vdots \\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k+n-1) \\ \mathit{fcall}(f,k) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{uop}(e),\gamma,\rho,k) = \mathit{sq}(\mathcal{C}_{\text{form}}(e,\gamma,\rho,k),\mathit{uop}(k))$
- $\mathcal{C}_{\text{form}}(\mathit{bop}(e_1,e_2),\gamma,\rho,k) = \mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \mathcal{C}_{\text{form}}(e_2,\gamma,\rho,k+1) \\ \mathit{bop}(k) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{let}((),e),\gamma,\rho,k) = \mathcal{C}_{\text{form}}(e,\gamma,\rho,k)$
- $\mathcal{C}_{\text{form}}(\mathit{let}(x_1 = e_1,\ldots,x_n = e_n;e),\gamma,\rho,k) =$
  $\mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \vdots \\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,(k+n-1)) \\ \mathcal{C}_{\text{form}}(e,\gamma,\rho[x_1 \leftarrow k,\ldots x_n \leftarrow (k+n-1)],k+n) \\ \mathit{copy}(k+n,k) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{list}{*}(e_1,\ldots,e_n),\gamma,\rho,k) = \mathit{sq}\left(\begin{array}{l} \mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k) \\ \vdots \\ \mathcal{C}_{\text{form}}(e_n,\gamma,\rho,k+n-1) \\ \mathit{list}{*}(n,k) \end{array}\right)$
- $\mathcal{C}_{\text{form}}(\mathit{cond}(),\gamma,\rho,k) = \mathit{copyc}(\mathit{NIL},k)$
- $\mathcal{C}_{\text{form}}(\mathit{cond}(p_1 \to e_1,\ldots,p_n \to e_n),\gamma,\rho,k) =$
  $\mathit{sq}(\mathcal{C}_{\text{form}}(p_1,\gamma,\rho,k),$
  $\mathit{itef}(k,\mathcal{C}_{\text{form}}(e_1,\gamma,\rho,k),\mathcal{C}_{\text{form}}(\mathit{cond}(p_2 \to e_2,\ldots,p_n \to e_n),\gamma,\rho,k)))$
- $\mathcal{C}_{\text{form}}(\mathit{read\_char},\gamma,\rho,k) = \mathit{read\_char}(k)$

- $\mathcal{C}_{\text{form}}(\mathit{peek\_char}, \gamma, \rho, k) = \mathit{peek\_char}(k)$
- $\mathcal{C}_{\text{form}}(\mathit{print\_char}(e), \gamma, \rho, k) = \mathit{sq}(\mathcal{C}_{\text{form}}(e, \gamma, \rho, k), \mathit{print\_char}(k))$

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.

D A Modified Semantics for SIL

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}, s_{top} denote the respective state components in state s. To increase readability, we simply write $s(i)$ for the relative local access $s_{\text{local}}(s_{\text{base}}+i)$, and write $s[i \leftarrow v]$ for $s[s_{\text{local}}(s_{\text{base}}+i) \leftarrow v]$. In the rules for the erroneous cases (the machine does not behave like a stack machine) the semantics is defined non-deterministically and q denotes some arbitrary state.

- copy constant:
  $\Gamma \vdash s : \mathit{copyc}(c, i) \to s[i \leftarrow c, \text{top} \leftarrow s_{\text{base}} + i]$ if $s_{\text{base}} + i \leq s_{\text{top}} + 1$
  $\Gamma \vdash s : \mathit{copyc}(c, i) \to q$ otherwise
- copy local:

$$\frac{(s_{\text{base}} + i \leq s_{\text{top}}) \wedge (s_{\text{base}} + j \leq s_{\text{top}} + 1)}{\Gamma \vdash s : \mathit{copy}(i, j) \to s[j \leftarrow s(i), \text{top} \leftarrow s_{\text{base}} + \max(i, j)]} \qquad \frac{(s_{\text{base}} + i > s_{\text{top}}) \vee (s_{\text{base}} + j > s_{\text{top}} + 1)}{\Gamma \vdash s : \mathit{copy}(i, j) \to q}$$

- copy-pop local:

$$\frac{(s_{\text{base}} + i \leq s_{\text{top}}) \wedge (i \geq j)}{\Gamma \vdash s : \mathit{copy\_pop}(i, j) \to s[j \leftarrow s(i), \text{top} \leftarrow s_{\text{base}} + j]} \qquad \frac{(s_{\text{base}} + i > s_{\text{top}}) \vee (i < j)}{\Gamma \vdash s : \mathit{copy\_pop}(i, j) \to q}$$

- copy from global to local memory:

$$\frac{g < |s_{\text{global}}| \wedge s_{\text{base}} + i \leq s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{gcopy}(g, i) \to s[i \leftarrow s_{\text{global}}(g), \text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{g < |s_{\text{global}}| \wedge s_{\text{base}} + i > s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{gcopy}(g, i) \to q}$$

- copy from local to global memory:

$$\frac{g < |s_{\text{global}}| \wedge s_{\text{base}} + i \leq s_{\text{top}}}{\Gamma \vdash s : \mathit{copyg}(i, g) \to s[s_{\text{global}}(g) \leftarrow s(i), \text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{g < |s_{\text{global}}| \wedge s_{\text{base}} + i > s_{\text{top}}}{\Gamma \vdash s : \mathit{copyg}(i, g) \to q}$$

- conditional:

$$\frac{\begin{array}{c} s_{\text{base}} + i \leq s_{\text{top}} \\ s(i) = \mathit{NIL} \\ \Gamma \vdash s : f \to q \end{array}}{\Gamma \vdash s : \mathit{itef}(i, t, f) \to q} \qquad \frac{\begin{array}{c} s_{\text{base}} + i \leq s_{\text{top}} \\ s(i) \neq \mathit{NIL} \\ \Gamma \vdash s : t \to q \end{array}}{\Gamma \vdash s : \mathit{itef}(i, t, f) \to q} \qquad \frac{s_{\text{base}} + i > s_{\text{top}}}{\Gamma \vdash s : \mathit{itef}(i, t, f) \to q}$$

- sequential composition:

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

- function call:

$$\frac{\begin{array}{c} s_{\text{base}} + i \leq s_{\text{top}} + 1 \\ \Gamma \vdash s[\mathit{base} \leftarrow s_{\text{base}} + i] : \mathit{body} \to q \wedge q_{\text{top}} \geq q_{\text{base}} \end{array}}{\Gamma \vdash s : \mathit{fcall}(h, i) \to q[\text{base} \leftarrow s_{base}, \text{top} \leftarrow s_{\text{base}} + i]} \qquad \text{if } [h \leftarrow \mathit{body}] \in \Gamma$$

$$\frac{s_{\text{base}} + i > s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{fcall}(h, i) \to q} \qquad \frac{\begin{array}{c} s_{\text{base}} + i \leq s_{\text{top}} + 1 \\ \Gamma \vdash s[\mathit{base} \leftarrow s_{\text{base}} + i] : \mathit{body} \to q \wedge q_{\text{top}} < q_{\text{base}} \end{array}}{\Gamma \vdash s : \mathit{fcall}(h, i) \to r} \qquad \text{if } [h \leftarrow \mathit{body}] \in \Gamma$$

- unary operators:

$$\frac{\begin{array}{c} s_{\text{base}} + i \leq s_{\text{top}} \\ s(i) : \mathit{uop} \to v \end{array}}{\Gamma \vdash s : \mathit{uop}(i) \to s[i \leftarrow v, \text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{s_{\text{base}} + i > s_{\text{top}}}{\Gamma \vdash s : \mathit{uop}(i) \to q}$$

- binary operators

$$\frac{\begin{array}{c} s_{\text{base}} + i + 1 \leq s_{\text{top}} \\ s(i), s(i+1) : \mathit{bop} \to v \end{array}}{\Gamma \vdash s : \mathit{bop}(i) \to s[i \leftarrow v, \text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{s_{\text{base}} + i + 1 > s_{\text{top}}}{\Gamma \vdash s : \mathit{bop}(i) \to q}$$

- do-loop:

$$\frac{\begin{array}{c} \Gamma \vdash s : c \to q \\ q_{\text{base}} + i \leq q_{\text{top}} \\ (q(i) \neq \mathit{NIL}) \end{array}}{\Gamma \vdash s : \mathit{do}(i, c, b) \to q} \qquad \frac{\begin{array}{c} \Gamma \vdash s : c \to r \\ r_{\text{base}} + i \leq r_{\text{top}} \\ (r(i) = \mathit{NIL}) \\ \Gamma \vdash r : b \to t \\ \Gamma \vdash t : \mathit{do}(i, c, b) \to q \end{array}}{\Gamma \vdash s : \mathit{do}(i, c, b) \to q} \qquad \frac{\begin{array}{c} \Gamma \vdash s : c \to q \\ q_{\text{base}} + i > q_{\text{top}} \end{array}}{\Gamma \vdash s : \mathit{do}(i, c, b) \to r}$$

- read

$$\frac{s_{\text{base}} + i \leq s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{read\_char}(i) \to s[i \leftarrow \text{first}(s_{\text{input}}), \text{input} \leftarrow \text{rest}(s_{\text{input}}), \text{top} \leftarrow s_{\text{base}} + i]}$$

$$\frac{s_{\text{base}} + i > s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{read\_char}(i) \to q}$$

- peek

$$\frac{s_{\text{base}} + i \leq s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{peek\_char}(i) \to s[i \leftarrow \text{first}(s_{\text{input}}), \text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{s_{\text{base}} + i > s_{\text{top}} + 1}{\Gamma \vdash s : \mathit{peek\_char}(i) \to q}$$

- print

$$\frac{s_{\text{base}} + i \leq s_{\text{top}}}{\Gamma \vdash s : \mathit{print\_char}(i) \to s[\text{output} \leftarrow s_{\text{output}} {+\!\!+} s(i), \text{top} \leftarrow s_{\text{base}} + i]} \qquad \text{if } s(i) \in \mathit{char}$$

$$\frac{s_{\text{base}} + i > s_{\text{top}}}{\Gamma \vdash s : \mathit{print\_char}(i) \to q}$$

- *list∗*

$$\frac{n \geq 2 \wedge s_{\text{base}} + i + n - 1 \leq s_{\text{top}}}{\Gamma \vdash s : \mathit{list}{*}(n, i) \to s[i \leftarrow \mathit{cons}(s(i), \ldots \mathit{cons}(s(i+n-2), s(i+n-1)) \ldots), \text{top} \leftarrow s_{\text{base}} + i]}$$

$$\frac{s_{\text{base}} + i \leq s_{\text{top}}}{\Gamma \vdash s : \mathit{list}{*}(1, i) \to s[\text{top} \leftarrow s_{\text{base}} + i]} \qquad \frac{s_{\text{base}} + i + n - 1 > s_{\text{top}} \ (n \geq 1)}{\Gamma \vdash s : \mathit{list}{*}(n, i) \to q}$$

E Semantics of $\mathbf{C}^{int}$

In the following, s_{stack}, s_{heap}, s_{base}, s_{quotetop}, s_{heaptop} denote the respective state components in state s.

E.1 Expressions

Note that we have defined a non-deterministic semantics for binary operator $\neq$. In case $e_1 \neq e_2$ some arbitrary non-zero value is returned.

- atomic expressions

$$s : \text{heaptop} \rightarrow s_{\text{heaptop}} \quad s : \text{stacktop} \rightarrow s_{\text{base}} \quad s : \text{quotetop} \rightarrow s_{\text{quotetop}}$$

$$s : i \rightarrow i \quad s : \mathit{local}(i) \rightarrow s_{\text{stack}}(s_{\text{base}} + i)$$

- unary expressions

$$\frac{s : e_1 \rightarrow v \quad (v \geq 0)}{s : \text{stack}(e_1) \rightarrow s_{\text{stack}}(v)} \quad \frac{s : e_1 \rightarrow v \quad (v \geq 0)}{s : \text{heap}(e_1) \rightarrow s_{\text{heap}}(v)} \quad \frac{s : e_1 \rightarrow v}{s : \text{unavailable}(e_1) \rightarrow w} \quad \frac{s : e_1 \rightarrow v}{s : 2 * (e_1) \rightarrow 2 * v}$$

- binary expressions

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2}{s : (e_1 \text{ op } e_2) \rightarrow (v_1 \text{ op } v_2)} \quad \text{op} \in \{+, -, *\}$$

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2 \ (v_2 \neq 0)}{s : (e_1 \text{ op } e_2) \rightarrow (v_1 \text{ op } v_2)} \quad \text{op} \in \{\text{div}, \text{rem}\}$$

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2}{s : e_1 < e_2 \rightarrow i} \quad (\text{If } v_1 < v_2 \text{ Then } i = 1 \text{ Else } i = 0) \quad (\text{analogous for } \geq, =)$$

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2}{s : e_1 \neq e_2 \rightarrow i} \quad (\text{If } v_1 = v_2 \text{ Then } i = 0 \text{ Else } i = z \text{ for some } z \neq 0)$$

E.2 Statements

The semantics of *print_char* is defined non-deterministically, in case the value at stack relative position i is not a character. In this case, some arbitrary character is appended to the output list.

- skip $\Gamma \vdash s : \mathit{skip} \rightarrow s$
- frame-pointer relative access on stack

$$\frac{s : e \rightarrow v}{\Gamma \vdash s : \mathit{set\_local}(e, i) \rightarrow s[s_{\text{stack}}(s_{\text{base}} + i) \leftarrow v]}$$

- random write access on stack

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2 \ (v_2 \geq 0)}{\Gamma \vdash s : \mathit{set\_stack}(e_1, e_2) \rightarrow s[s_{\text{stack}}(v_2) \leftarrow v_1]}$$

- random write access on heap

$$\frac{s : e_1 \rightarrow v_1 \quad s : e_2 \rightarrow v_2 \ (v_2 \geq 0)}{\Gamma \vdash s : \mathit{set\_heap}(e_1, e_2) \rightarrow s[s_{\text{heap}}(v_2) \leftarrow v_1]}$$

- sequential composition

$$\frac{\Gamma \vdash s : c_1 \rightarrow r \quad \Gamma \vdash r : c_2 \rightarrow q}{\Gamma \vdash s : c_1; c_2 \rightarrow q}$$

- conditional

$$\frac{s : e \rightarrow 0 \quad \Gamma \vdash s : c_2 \rightarrow q}{\Gamma \vdash s : \mathit{if}(e, c_1, c_2) \rightarrow q} \qquad \frac{s : e \rightarrow v \ (v \neq 0) \quad \Gamma \vdash s : c_1 \rightarrow q}{\Gamma \vdash s : \mathit{if}(e, c_1, c_2) \rightarrow q}$$

- simple conditional

$$\frac{s : e \to 0}{\Gamma \vdash s : \mathit{if}(e, c_1) \to s} \qquad \frac{s : e \to v \ (v \neq 0) \quad \Gamma \vdash s : c_1 \to q}{\Gamma \vdash s : \mathit{if}(e, c_1) \to q}$$

- do loop

$$\frac{\Gamma \vdash s : c \to q \quad q : e \to v \ (v \neq 0)}{\Gamma \vdash s : \mathit{do}(c, e, b) \to q} \qquad \frac{\begin{array}{c}\Gamma \vdash s : c \to r \\ r : e \to 0 \\ \Gamma \vdash r : b \to t \\ \Gamma \vdash t : \mathit{do}(c, e, b) \to q\end{array}}{\Gamma \vdash s : \mathit{do}(c, e, b) \to q}$$

- read character

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

- peek character

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

- print character

$$\frac{0 \leq s_{\text{stack}}(s_{\text{base}} + i) \leq 255}{\Gamma \vdash s : \mathit{print\_char}(i) \to s[\text{output} \leftarrow s_{\text{output}} +\!\!+ s_{\text{stack}}(s_{\text{base}} + i)]}$$

$$\frac{s_{\text{stack}}(s_{\text{base}} + i) < 0 \ \vee \ s_{\text{stack}}(s_{\text{base}} + i) > 255}{\Gamma \vdash s : \mathit{print\_char}(i) \to s[\text{output} \leftarrow s_{\text{output}} +\!\!+ c]} \quad (c \in \mathit{char})$$

- procedure call

$$\frac{\Gamma \vdash s[\text{base} \leftarrow s_{\text{base}} + i] : \mathit{body} \to q}{\Gamma \vdash s : \mathit{call}(h, i) \to q[\text{base} \leftarrow s_{\mathit{base}}]} \qquad \text{if } [h(\mathit{size}) \leftarrow \mathit{body}] \in \Gamma$$

- modification of heaptop pointer

$$\frac{s : e \to v \ \wedge \ (s_{\text{heaptop}} + v \geq 0)}{\Gamma \vdash s : \mathit{allocate}(e) \to s[\text{heaptop} \leftarrow s_{\text{heaptop}} + v]}$$

F Compiling SIL to $\mathbf{C}^{int}$

The complete definition of the compiling function $\mathcal{CC}_{\text{stmt}}$ compiling SIL statements to $\mathbf{C}^{int}$ statements using a heap environment ζ is as follows:

- $\mathcal{CC}_{\text{stmt}}(\mathit{abort}, \zeta) = \mathit{abort}$
- constants
  - $\mathcal{CC}_{\text{stmt}}(\mathit{copyc}(n, i), \zeta) = \mathit{set\_local}(3, 2i); \mathit{set\_local}(n, 2i + 1)$
  - $\mathcal{CC}_{\text{stmt}}(\mathit{copyc}(c, i), \zeta) = \mathit{set\_local}(4, 2i); \mathit{set\_local}(\mathit{code}(c), 2i + 1)$
  - $\mathcal{CC}_{\text{stmt}}(\mathit{copyc}(\mathit{NIL}, i), \zeta) = \mathit{set\_local}(0, 2i); \mathit{set\_local}(0, 2i + 1)$
  - $\mathcal{CC}_{\text{stmt}}(\mathit{copyc}(T, i), \zeta) = \mathit{set\_local}(1, 2i); \mathit{set\_local}(1, 2i + 1)$
  - $\mathcal{CC}_{\text{stmt}}(\mathit{copyc}(s, i), \zeta) = \mathit{set\_local}(\mathit{tag}, 2i); \mathit{set\_local}(\mathit{val}, 2i + 1)$
    where $(\mathit{tag}, \mathit{val}) = \zeta(s)$ (for non-atomic s-expressions and symbols)
- $\mathcal{CC}_{\text{stmt}}(\mathit{copy}(i, j), \zeta) = \mathit{set\_local}(\mathit{local}(2i), 2j); \mathit{set\_local}(\mathit{local}(2i + 1), 2j + 1)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{copy\_pop}(i, j), \zeta) = \mathit{set\_local}(\mathit{local}(2i), 2j); \mathit{set\_local}(\mathit{local}(2i + 1), 2j + 1)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{gcopy}(i, j), \zeta) = \mathit{set\_local}(\mathit{stack}(2i + 2), 2j); \mathit{set\_local}(\mathit{stack}(2i + 3), 2j + 1)$

- $\mathcal{CC}_{\text{stmt}}(\mathit{copyg}(i,j),\zeta) = \mathit{set\_stack}(\mathit{local}(2i), 2j+2); \mathit{set\_stack}(\mathit{local}(2i+1), 2j+3)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{fcall}(f,i),\zeta) = \mathit{call}(f, 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{uop}(i),\zeta) = \mathit{call}(``\mathit{uop}'', 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{bop}(i),\zeta) = \mathit{call}(``\mathit{bop}'', 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{itef}(i,s_1,s_2),\zeta) = \mathit{if}(\mathit{local}(2i) \neq 0, \mathcal{CC}_{\text{stmt}}(s_1,\zeta), \mathcal{CC}_{\text{stmt}}(s_2,\zeta))$
- $\mathcal{CC}_{\text{stmt}}(\mathit{sq}(s_1,\ldots,s_n),\zeta) = \mathcal{CC}_{\text{stmt}}(s_1,\zeta);\ldots;\mathcal{CC}_{\text{stmt}}(s_n,\zeta)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{while}(i,s_1,s_2),\zeta) = \mathit{while}(\mathcal{CC}_{\text{stmt}}(s_1,\zeta), \mathit{local}(2i) \neq 0, \mathcal{CC}_{\text{stmt}}(s_2,\zeta))$
- $\mathcal{CC}_{\text{stmt}}(\mathit{read\_char}(i),\zeta) = \mathit{call}(``\mathit{read\_char}'', 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{peek\_char}(i),\zeta) = \mathit{call}(``\mathit{peek\_char}'', 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{print\_char}(i),\zeta) = \mathit{call}(``\mathit{print\_char}'', 2i)$
- $\mathcal{CC}_{\text{stmt}}(\mathit{list*}(n,i),\zeta) = \mathit{call}(``\mathit{cons}'', 2n+2i-2);\ldots;\mathit{call}(``\mathit{cons}'', 2i)$

G Effects of TASM instructions

In the following, c_{PrB} is supposed to be non-empty, i.e, there is at least one instruction to be executed, $\neg c_{\text{Eflg}}$ (i.e. the error flag is not set in configuration c). To increase readability, we omit the symbolic instruction pointer for all linear instructions where the pointer is updated regularly such that it points to the next instruction in the sequence. For these instructions, we have $\mathit{PrA} := c_{\text{PrA}} \cdot \mathit{car}(c_{\text{PrB}})$ and $\mathit{PrB} := \mathit{cdr}(c_{\text{PrB}})$. To non-deterministically choose some word value, we write ? which denotes $\mathit{choose?}(\mathit{Word?})$.

- swap Areg and Breg
$$c : \texttt{rev} \rightarrow c[\mathit{Areg} := c_{\text{Breg}}, \mathit{Breg} := c_{\text{Areg}}]$$
- addition (similar for `sub`, `mul`)
$$c : \texttt{add} \rightarrow c[\mathit{Areg} := \mathit{plus}(c_{\text{Breg}}, c_{\text{Areg}}), \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?, \mathit{Eflg} := \mathit{plusovfl}(c_{\text{Breg}}, c_{\text{Areg}})]$$
- integer division (similar for remainder `rem`)
$$\frac{c_{\text{Areg}} = 0 \vee (c_{\text{Areg}} = -1 \wedge c_{\text{Breg}} = \mathit{minword})}{c : \texttt{div} \rightarrow c[\mathit{Areg} := ?, \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?, \mathit{Eflg} := T]}$$
$$\frac{c_{\text{Areg}} \neq 0 \wedge (c_{\text{Areg}} \neq -1 \vee c_{\text{Breg}} \neq \mathit{minword})}{c : \texttt{div} \rightarrow c[\mathit{Areg} := \mathit{div}(c_{\text{Breg}}, c_{\text{Areg}}), \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?]}$$
- difference
$$c : \texttt{diff} \rightarrow c[\mathit{Areg} := \mathit{minus}(c_{\text{Breg}}, c_{\text{Areg}}), \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?]$$
- bitwise and (similar for `or`, `xor`)
$$c : \texttt{and} \rightarrow c[\mathit{Areg} := \mathit{bitAND}(c_{\text{Breg}}, c_{\text{Areg}}), \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?]$$
- 1 complement
$$c : \texttt{not} \rightarrow c[\mathit{Areg} := \mathit{bitNOT}(c_{\text{Areg}})]$$
- shift left (similar for `shr`)
$$\frac{0 \leq c_{\text{Areg}} < \mathit{wordlength}}{c : \texttt{shl} \rightarrow c[\mathit{Areg} := \mathit{shiftL}(c_{\text{Breg}}, c_{\text{Areg}}), \mathit{Breg} := c_{\text{Creg}}, \mathit{Creg} := ?]}$$
$$\frac{c_{\text{Areg}} < 0 \vee c_{\text{Areg}} \geq \mathit{wordlength}}{c : \texttt{shl} \rightarrow q}$$

- arithmetic greater than test

$$c : \texttt{gt} \rightarrow c[\mathit{Areg} := (c_{\mathrm{Breg}} > c_{\mathrm{Areg}}), \mathit{Breg} := c_{\mathrm{Creg}}, \mathit{Creg} :=?]$$

- byte/word calculation

$$c : \texttt{wcnt} \rightarrow c[\mathit{Areg} := \mathit{shiftR}(c_{\mathrm{Areg}}, 2), \mathit{Breg} := \mathit{bitAND}(c_{\mathrm{Areg}}, 3), \mathit{Creg} := c_{\mathrm{Breg}}]$$

- load minword

$$c : \texttt{mint} \rightarrow c[\mathit{Areg} := \mathit{minword}, \mathit{Breg} := c_{\mathrm{Areg}}, \mathit{Creg} := c_{\mathrm{Breg}}]$$

- word memory subscript

$$\frac{\mathit{WordAddr?}(c_{\mathrm{Areg}})}{c : \texttt{wsub} \rightarrow c[\mathit{Areg} := \mathit{Index}(c_{\mathrm{Areg}}, c_{\mathrm{Breg}}), \mathit{Breg} := c_{\mathrm{Creg}}, \mathit{Creg} :=?]}$$

$$\frac{\neg\, \mathit{WordAddr?}(c_{\mathrm{Areg}})}{c : \texttt{wsub} \rightarrow q}$$

- input via a link. Note that only the case is specified where only one byte is read at the specific channel $\mathit{minword} + (\mathit{bytesperword} = 4) * 4$. The value read from the input stream is stored at the address specified by c_{Creg} which must be cleared. The address must be smaller or equal than MemTop.

$$\frac{c_{\mathrm{Areg}} = 1, c_{\mathrm{Breg}} = \mathit{Index}(\mathit{minword}, 4), \mathit{WordAddr?}(c_{\mathrm{Creg}}), c_{\mathrm{Mem}}(c_{\mathrm{Creg}}) = 0, c_{\mathrm{Creg}} \leq \mathit{MemTop}}{c : \texttt{in} \rightarrow c\left[\begin{array}{c} \mathit{Areg}, \mathit{Breg}, \mathit{Creg} :=? \\ \mathit{Mem} := c_{\mathrm{Mem}}[c_{\mathrm{Creg}} := \mathit{first}(c_{\mathrm{In}})] \\ \mathit{In} := \mathit{rest}(c_{\mathrm{In}}) \end{array}\right]}$$

In case the preconditions are not satisfied, any successor configuration is allowed: $c : \texttt{in} \rightarrow q$.

- output via a link. Note that only the case is specified where one single byte is written to the output stream. Furthermore, the output channel is fixed.

$$\frac{c_{\mathrm{Areg}} = 1, c_{\mathrm{Breg}} = \mathit{minword}, \mathit{WordAddr?}(c_{\mathrm{Creg}})}{c : \texttt{out} \rightarrow c\left[\begin{array}{c} \mathit{Areg}, \mathit{Breg}, \mathit{Creg} :=?, \\ \mathit{In} := \mathsf{IF}\ \mathit{byte?}(c_{\mathrm{Mem}}(c_{\mathrm{Creg}}))\ \mathsf{THEN}\ c_{\mathrm{Out}} \cdot c_{\mathrm{Mem}}(c_{\mathrm{Creg}})\ \mathsf{ELSE}\ c_{\mathrm{Out}} \cdot b \quad (b \in \mathit{byte}) \end{array}\right]}$$

If the preconditions are not met `out` is defined as follows: $c : \texttt{out} \rightarrow q$.

- computed absolute jump

$$\frac{\begin{array}{c} c_{\mathrm{count}} > 0, c_{\mathrm{PrA}} \cdot c_{\mathrm{PrB}} = a \cdot b \\ c_{\mathrm{Areg}} = \mathit{PgrStart} + |a| \\ \mathit{Word?}(\mathit{PgrStart} + |c_{\mathrm{PrA}}| + 1) \end{array}}{c : \texttt{gcall} \rightarrow c[\mathit{Areg} := \mathit{PgrStart} + |c_{\mathrm{PrA}}| + 1, \mathit{PrA} := a, \mathit{PrB} := b, \mathit{count} := c_{\mathrm{count}} - 1]}$$

- set the error flag

$$c : \texttt{seterr} \rightarrow c[\mathit{Eflg} := T]$$

- check subscript

$$c : \texttt{csub0} \rightarrow c[\mathit{Areg} := c_{\mathrm{Breg}}, \mathit{Breg} := c_{\mathrm{Creg}}, \mathit{Creg} :=?, \mathit{Eflg} := (c_{\mathrm{Breg}} \geq_{\mathrm{unsigned}} c_{\mathrm{Areg}})]$$

- convert single to double

$$c : \texttt{xdbl} \rightarrow c[\mathit{Breg} := \mathsf{IF}\ c_{\mathrm{Areg}} < 0\ \mathsf{THEN}\ -1\ \mathsf{ELSE}\ 0, \mathit{Creg} := c_{\mathrm{Breg}}]$$

- load constant

$$c : \texttt{ldc}\ w \rightarrow c[\mathit{Areg} := w, \mathit{Breg} := c_{\mathrm{Areg}}, \mathit{Creg} := c_{\mathrm{Breg}}]$$

- load local

$$\frac{\mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Wptr}}, w))}{c : \texttt{ldl}\ w \rightarrow c[\mathit{Areg} := c_{\mathrm{Mem}}(\mathit{Index}(c_{\mathrm{Wptr}}, w)), \mathit{Breg} := c_{\mathrm{Areg}}, \mathit{Creg} := c_{\mathrm{Breg}}]}$$

$$\frac{\neg\, \mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Wptr}}, w))}{c : \texttt{ldl}\ w \rightarrow q}$$

- store local

$$\frac{\mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Wptr}}, w)) \ \wedge\ (\mathit{Index}(c_{\mathrm{Wptr}}, w) \leq \mathit{MemTop})}{c : \texttt{stl}\ w \rightarrow c\left[\begin{array}{c} \mathit{Areg} := c_{\mathrm{Breg}}, \mathit{Breg} := c_{\mathrm{Creg}}, \mathit{Creg} :=? \\ \mathit{Mem} := c_{\mathrm{Mem}}[\mathit{Index}(c_{\mathrm{Wptr}}, w) := c_{\mathrm{Areg}}] \end{array}\right]}$$

  In case the preconditions are not met, any successor configuration is allowed: $c : \texttt{stl}\ w \rightarrow q$.

- load local pointer

$$c : \texttt{ldlp}\ w \rightarrow c[\mathit{Areg} := \mathit{Index}(c_{\mathrm{Wptr}}, w), \mathit{Breg} := c_{\mathrm{Areg}}, \mathit{Creg} := c_{\mathrm{Breg}}]$$

- add constant

$$c : \texttt{adc}\ w \rightarrow c[\mathit{Areg} := \mathit{plus}(c_{\mathrm{Areg}}, w), \mathit{Eflg} := c_{\mathrm{Eflg}} \vee \mathit{plusovfl}(c_{\mathrm{Areg}}, w)]$$

- equal to constant

$$c : \texttt{eqc}\ w \rightarrow c[\mathit{Areg} := (c_{\mathrm{Areg}} = w)]$$

- jump

$$\frac{\begin{array}{c} c_{\mathrm{count}} > 0,\ c_{\mathrm{PrA}} \cdot c_{\mathrm{PrB}} = a \cdot b \\ |a| = |c_{\mathrm{PrA}}| + |\texttt{j}\ w| + w \end{array}}{c : \texttt{j}\ w \rightarrow c[\mathit{Areg}, \mathit{Breg}, \mathit{Creg} :=?, \mathit{PrA} := a, \mathit{PrB} := b, \mathit{count} := c_{\mathrm{count}} - 1]}$$

  For a jump within a TASM instruction (that is, the corresponding pfix/nfix chain), the effect is specified non-deterministically:

$$\frac{c_{\mathrm{count}} > 0,\ (\forall\ a, b.\ (c_{\mathrm{PrA}} \cdot c_{\mathrm{PrB}} = a \cdot b)\ \Rightarrow\ (|a| \neq |c_{\mathrm{PrA}}| + |\texttt{j}\ w| + w))}{c : \texttt{j}\ w \rightarrow q}$$

- conditional jump

$$\frac{\begin{array}{c} c_{\mathrm{count}} > 0,\ (c_{\mathrm{Areg}} = 0),\ c_{\mathrm{PrA}} \cdot c_{\mathrm{PrB}} = a \cdot b \\ |a| = |c_{\mathrm{PrA}}| + |\texttt{cj}\ w| + w \end{array}}{c : \texttt{cj}\ w \rightarrow c[\mathit{PrA} := a, \mathit{PrB} := b, \mathit{count} := c_{\mathrm{count}} - 1]}$$

$$\frac{c_{\mathrm{count}} > 0,\ (c_{\mathrm{Areg}} \neq 0)}{c : \texttt{cj}\ w \rightarrow c[\mathit{Areg} := c_{\mathrm{Breg}}, \mathit{Breg} := c_{\mathrm{Creg}}, \mathit{Creg} :=?, \mathit{count} := c_{\mathrm{count}} - 1]}$$

$$\frac{c_{\mathrm{count}} > 0,\ (c_{\mathrm{Areg}} = 0),\ (\forall a, b.\ (c_{\mathrm{PrA}} \cdot c_{\mathrm{PrB}} = a \cdot b)\ \Rightarrow\ (|a| \neq |c_{\mathrm{PrA}}| + |\texttt{cj}\ w| + w))}{c : \texttt{cj}\ w \rightarrow q}$$

- load non local

$$\frac{\mathit{WordAddr?}(c_{\mathrm{Areg}}), \mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Areg}}, w))}{c : \texttt{ldnl}\ w \rightarrow c[\mathit{Areg} := c_{\mathrm{Mem}}(\mathit{Index}(c_{\mathrm{Areg}}, w))]}$$

$$\frac{\neg\, \mathit{WordAddr?}(c_{\mathrm{Areg}}) \vee \neg\, \mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Areg}}, w))}{c : \texttt{ldnl}\ w \rightarrow q}$$

- store non local

$$\frac{\mathit{WordAddr?}(c_{\mathrm{Areg}}),\ \mathit{WordAddr?}(\mathit{Index}(c_{\mathrm{Areg}},w)),\ (\mathit{Index}(c_{\mathrm{Areg}},w)\leq \mathit{MemTop})}{c:\texttt{stnl}\ w\rightarrow c\begin{bmatrix}\mathit{Areg}:=c_{\mathrm{Creg}},\mathit{Breg},\mathit{Creg}:=?\\ \mathit{Mem}:=c_{\mathrm{Mem}}[\mathit{Index}(c_{\mathrm{Areg}},w):=c_{\mathrm{Breg}}]\end{bmatrix}}$$

If the preconditions are not met, the effect is non-deterministic: $c:\texttt{stnl}\ w\rightarrow q$.

- load non local pointer

$$\frac{\mathit{WordAddr?}(c_{\mathrm{Areg}})}{c:\texttt{ldnlp}\ w\rightarrow c[\mathit{Areg}:=\mathit{Index}(c_{\mathrm{Areg}},w)]}$$

$$\frac{\neg\,\mathit{WordAddr?}(c_{\mathrm{Areg}})}{c:\texttt{ldnlp}\ w\rightarrow q}$$

H Compiling $\mathbf{C}^{int}$ to TASM

Note that *base*, *temp*, *heap*, ... are the names for the system variables which denote numbers from 0 to 13 (see Fig. 16). The code contains several overflow checks which ensure that, for example, heap addresses are within the bounds, that is, they are positive numbers below *heaptop*. In the following $\varphi=<\psi,s_{\mathrm{size}},h_{\mathrm{size}}>$ denotes the global environment, where ψ maps procedure identifiers to jump table indices. For more explanations concerning this specification we refer to the technical report [GH98c].

H.1 Expressions

- $\mathcal{CC}_{\mathrm{expr}}(\mathit{heaptop},\varphi,\sigma,\texttt{ldl}\ \mathit{heaptop})$
- $\mathcal{CC}_{\mathrm{expr}}(\mathit{quotetop},\varphi,\sigma,\texttt{ldl}\ \mathit{quotetop})$
- $\mathcal{CC}_{\mathrm{expr}}(\mathit{stacktop},\varphi,\sigma,\texttt{stl}\ \mathit{temp};\texttt{ldl}\ \mathit{base};\texttt{ldlp}\ \mathit{stack};\texttt{diff};\texttt{wcnt};\texttt{rev};\texttt{stl}\ \mathit{temp2};\texttt{ldl}\ \mathit{temp};\texttt{rev})$
- $\mathcal{CC}_{\mathrm{expr}}(i,\varphi,\sigma,\texttt{ldc}\ i)$ (if i is a word)
- $\mathcal{CC}_{\mathrm{expr}}(\mathit{local}(i),\varphi,\sigma,\texttt{ldl}\ \mathit{base};\ \texttt{ldnl}\ i)$ (if $i\in\mathit{Word}$ and $0\leq i<\sigma$)
- absolute stack access (simpler code for literals i)
  $\mathcal{CC}_{\mathrm{expr}}(\mathit{stack}(i),\varphi,\sigma,\texttt{ldl}\ \mathit{stack}+i)$ (if $i\in\mathit{Word}$ and $0\leq i\leq s_{\mathrm{size}}$)
- absolute stack access
  Let $\mathit{stack\_code}::=\texttt{xdbl};\texttt{rev};\texttt{cj}\ 2;\texttt{seterr};\texttt{add};\texttt{ldc}\ 4;\texttt{mul};\texttt{ldlp}\ \mathit{stack};\texttt{add};\texttt{stl}\ \mathit{temp};$
  $\texttt{ldl}\ \mathit{base};\texttt{ldl}\ \mathit{temp};\texttt{gt};\texttt{cj}\ -14;\texttt{ldl}\ \mathit{temp};\texttt{ldnl}\ 0$

$$\frac{\mathcal{CC}_{\mathrm{expr}}(e,\varphi,\sigma,m)\ \wedge\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\mathrm{expr}}(\mathit{stack}(e),\varphi,\sigma,m\cdot\mathit{stack\_code})}$$

- heap access

$$\frac{\mathcal{CC}_{\mathrm{expr}}(e,\varphi,\sigma,m)\ \wedge\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\mathrm{expr}}(\mathit{heap}(e),\varphi,\sigma,m\cdot\texttt{ldl}\ \mathit{heaptop};\texttt{csub0};\texttt{ldl}\ \mathit{heap};\texttt{wsub};\texttt{ldnl}\ 0)}$$

- unavailable

$$\frac{\mathcal{CC}_{\mathrm{expr}}(e,\varphi,\sigma,m)\ \wedge\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\mathrm{expr}}(\mathit{unavailable}(e),\varphi,\sigma,m\cdot\texttt{ldl}\ \mathit{heaptop};\texttt{add};\texttt{ldl}\ \mathit{heap};\texttt{wsub};\texttt{ldl}\ \mathit{rstack};\texttt{gt})}$$

- operators

$$\frac{\mathcal{CC}_{\mathrm{expr}}(e,\varphi,\sigma,m)\ \wedge\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\mathrm{expr}}(2*(e),\varphi,\sigma,m\cdot\texttt{ldc}\ 2;\texttt{mul})}$$

$$\frac{\mathcal{CC}_{\mathrm{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\mathrm{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \mathit{compilable?}(e_1,3)\wedge\mathit{compilable?}(e_2,2)}{\mathcal{CC}_{\mathrm{expr}}(e_1*e_2,\varphi,\sigma,m_1\cdot m_2\cdot\texttt{mul})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1+e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{add})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1-e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{sub})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(\textit{div}(e_1,e_2),\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{div})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(\textit{rem}(e_1,e_2),\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{rem})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1<e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{rev};\mathtt{gt})}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1\geq e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{rev};\mathtt{gt};\mathtt{eqc}\ 0)}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1=e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{diff};\mathtt{eqc}\ 0)}$$

$$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2)\ \wedge\ \textit{compilable?}(e_1,3)\wedge\textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{expr}}(e_1\neq e_2,\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{diff})}$$

H.2 Statements

In the following $|c|$ denotes the length of TASM code sequence c in bytes which is determined by the number of instructions and the length of the associated *pfix*/*nfix* chain for each instruction.

- $\mathcal{CC}_{\text{stmt}}(\textit{skip},\varphi,\sigma,[])$
- $\mathcal{CC}_{\text{stmt}}(\textit{abort},\varphi,\sigma,\mathtt{seterr})$
- heap pointer adjustment
  $\textit{allocate\_code} ::= \mathtt{ldl}\ \textit{heaptop};\mathtt{add};\mathtt{stl}\ \textit{heaptop};\mathtt{ldl}\ \textit{heaptop};\mathtt{ldc}\ 4;\mathtt{mul};\mathtt{ldl}\ \textit{heap};$
  $\mathtt{add};\mathtt{ldl}\ \textit{rstack};\mathtt{gt};\mathtt{cj}\ 2;\mathtt{seterr};\mathtt{ldl}\ \textit{quotetop};\mathtt{ldl}\ \textit{heaptop};\mathtt{gt};\mathtt{cj}\ 2;\mathtt{seterr}$

  $$\frac{\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m),\ \textit{compilable?}(e,3)}{\mathcal{CC}_{\text{stmt}}(\textit{allocate}(e),\varphi,\sigma,m\cdot\textit{allocate\_code})}$$

- relative stack access

  $$\frac{\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m),\ \textit{compilable?}(e,3),\ \textit{Word?}(i),0\leq i<\sigma}{\mathcal{CC}_{\text{stmt}}(\textit{set\_local}(e,i),\varphi,\sigma,m\cdot\mathtt{ldl}\ \textit{base};\mathtt{stnl}\ i)}$$

- absolute stack access (special code for literals)

  $$\frac{\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m),\ \textit{compilable?}(e,3),\ \textit{Word?}(i),0\leq i<s_{\text{size}}}{\mathcal{CC}_{\text{stmt}}(\textit{set\_stack}(e,i),\varphi,\sigma,m\cdot\mathtt{stl}\ \textit{stack}+i)}$$

- absolute stack access
  $\textit{access\_code} ::= \mathtt{xdbl};\mathtt{rev};\mathtt{cj}\ 2;\mathtt{seterr};\mathtt{add};\mathtt{ldc}\ 4;\mathtt{mul};\mathtt{ldlp}\ \textit{stack};\mathtt{add};\mathtt{stl}\ \textit{temp};$
  $\mathtt{ldl}\ \textit{base};\mathtt{ldl}\ \textit{temp};\mathtt{gt};\mathtt{cj}\ -14;\mathtt{ldl}\ \textit{temp};\mathtt{stnl}\ 0$

  $$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2),\ \textit{compilable?}(e_1,3),\ \textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{stmt}}(\textit{set\_stack}(e_1,e_2),\varphi,\sigma,m_1\cdot m_2\cdot\textit{access\_code})}$$

- heap access

  $$\frac{\mathcal{CC}_{\text{expr}}(e_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{expr}}(e_2,\varphi,\sigma,m_2),\ \textit{compilable?}(e_1,3),\ \textit{compilable?}(e_2,2)}{\mathcal{CC}_{\text{stmt}}(\textit{set\_heap}(e_1,e_2),\varphi,\sigma,m_1\cdot m_2\cdot\mathtt{ldl}\ \textit{heaptop};\mathtt{csub0};\mathtt{ldl}\ \textit{heap};\mathtt{wsub};\mathtt{stnl}\ 0)}$$

- sequential composition
$$\frac{\mathcal{CC}_{\text{stmt}}(s_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{stmt}}(s_2,\varphi,\sigma,m_2)}{\mathcal{CC}_{\text{stmt}}(s_1;s_2,\varphi,\sigma,m_1\cdot m_2)}$$

- two way branch
$$\frac{\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m_e),\mathcal{CC}_{\text{stmt}}(s_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{stmt}}(s_2,\varphi,\sigma,m_2),\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\text{stmt}}(\mathit{if}(e,s_1,s_2),\varphi,\sigma,m_e\cdot \texttt{cj}\ |m_1|+|\texttt{j}|m_2||\cdot m_1\cdot \texttt{j}\ |m_2|\cdot m_2)}$$

- one way branch
$$\frac{\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m_e),\mathcal{CC}_{\text{stmt}}(s,\varphi,\sigma,m),\ \mathit{compilable?}(e,3)}{\mathcal{CC}_{\text{stmt}}(\mathit{if}(e,s),\varphi,\sigma,m_e\cdot \texttt{cj}\ |m|\cdot m)}$$

- do loop
$$\frac{\begin{array}{c}\mathcal{CC}_{\text{expr}}(e,\varphi,\sigma,m_e),\mathcal{CC}_{\text{stmt}}(s_1,\varphi,\sigma,m_1),\mathcal{CC}_{\text{stmt}}(s_2,\varphi,\sigma,m_2),\ \mathit{compilable?}(e,3)\\ \mathit{do\_code}=m_1\cdot m_e\cdot \texttt{eqc}\ 0;\texttt{cj}\ l_1\cdot m_2\cdot \texttt{j}\ -l_2\\ l_1=|m_2\cdot \texttt{j}-l_2|,\ l_2=|\mathit{do\_code}|\end{array}}{\mathcal{CC}_{\text{stmt}}(\mathit{do}(s_1,e,s_2),\varphi,\sigma,\mathit{do\_code})}$$

- procedure call
$\mathcal{CC}_{\text{stmt}}(\mathit{call}(h,i),\varphi,\sigma,\texttt{ldc}\ i;\texttt{ldl}\ \mathit{start};\texttt{ldnl}\ \psi(h);\texttt{gcall})$, if $0\le i<\sigma$

- read character
$\mathit{read\_code}$ ::= $\texttt{ldl}\ \mathit{lastchar};\texttt{eqc}\ -1;\texttt{cj}\ 6;\texttt{ldc}\ 0;\texttt{stl}\ \mathit{lastchar};\texttt{ldlp}\ \mathit{lastchar};$
$\texttt{ldl}\ \mathit{inchan};\texttt{ldc}\ 1;\texttt{in};\texttt{ldl}\ \mathit{lastchar};\texttt{ldl}\ \mathit{base};\texttt{stnl}\ i;\texttt{ldc}\ -1;\texttt{stl}\ \mathit{lastchar}$

  $\mathcal{CC}_{\text{stmt}}(\mathit{read\_char}(i),\varphi,\sigma,\mathit{read\_code})$ if $0\le i<\sigma\wedge \mathit{Word?}(i)$

- peek character
$\mathit{peek\_code}$ ::= $\texttt{ldl}\ \mathit{lastchar};\texttt{eqc}\ -1;\texttt{cj}\ 6;\texttt{ldc}\ 0;\texttt{stl}\ \mathit{lastchar};\texttt{ldlp}\ \mathit{lastchar};$
$\texttt{ldl}\ \mathit{inchan};\texttt{ldc}\ 1;\texttt{in};\texttt{ldl}\ \mathit{lastchar};\texttt{ldl}\ \mathit{base};\texttt{stnl}\ i$

  $\mathcal{CC}_{\text{stmt}}(\mathit{peek\_char}(i),\varphi,\sigma,\mathit{peek\_code})$ if $0\le i<\sigma\wedge \mathit{Word?}(i)$

- print character

  $\mathcal{CC}_{\text{stmt}}(\mathit{print\_char}(i),\varphi,\sigma,\texttt{ldl}\ \mathit{base};\texttt{ldnlp}\ i;\texttt{ldl}\ \mathit{outchan};\texttt{ldc}\ 1;\texttt{out})$ (if $0\le i<\sigma\wedge \mathit{Word?}(i)$)

H.3 Entry/Exit code for Procedures

Notice that in the last line of the entry code starting with instruction `ldc` 4, that is, the stack overflow check, we present a slightly different code compared to the original specification [GH98c]. The original code using a `ldnlp` σ instruction is erroneous, since no overflow check is carried out for this instruction.

entrycode(σ) ::=
`ldl` *rp*; `stnl` 0; (save return address)
`ldl` *base*; `ldl` *rp*; `stnl` 1; (save frame pointer)
`ldl` *base*; `wsub`; `stl` *base*; (adjust frame pointer)
`ldl` *rp*; `ldnlp` 2; `stl` *rp*; (adjust return stack pointer)
`ldl` *rp*; `ldl` *memtop*; `gt`; `cj` 2; `seterr`; (check return stack overflow)
`ldc` 4; `ldc` σ; `mul`; `ldl` *base*; `add`; `ldl` *heap*; `gt`; `cj` 2; `seterr` (check stack overflow)

exitcode ::=
`ldl` *rp*; `ldnlp` -2; `stl` *rp*; (adjust return stack pointer (pop))
`ldl` *rp*; `ldnl` 1; `stl` *base*; (restore frame pointer)
`ldl` *rp*; `ldnl` 0; `gcall` (jump to return address)

H.4 Effect of Initialization Code

Let $\varphi =< \psi, s_{\mathrm{size}}, h_{\mathrm{size}} >$ denote the global environment and p be a TASM program. Then the effect of the initialization code is specified by means of the following axiom:

$$
\begin{array}{l}
\mathit{semantics\_of\_initcode} : \mathsf{AXIOM}\\
\forall v, c_1, c_3, \varphi, p, \mathit{stksize} :\ \ Word?(\mathit{stksize}) \ \wedge\ \mathit{init\_state}?(p)(c_1) \ \wedge\ p_{\mathrm{main}} = \mathit{initcode}(\mathit{stksize}) \cdot v \ \wedge\\
{c_1}_{\mathrm{PrA}} = \mathit{flatten}(p_{\mathrm{modules}}) \ \wedge\ {c_1}_{\mathrm{PrB}} = \mathit{initcode}(\mathit{stksize}) \cdot v \ \wedge\ Rc(c_1, c_3) \ \wedge\ s_{\mathrm{size}} = |p_{\mathrm{data1}}| \ \wedge\ h_{\mathrm{size}} = |p_{\mathrm{data2}}|\\
\quad \Rightarrow\ \exists c_2, Y.\ \mathsf{LET}\ \mathit{Wsp} = \lambda w.\ {c_2}_{\mathrm{Mem}}(\mathit{Index}({c_2}_{\mathrm{Wptr}}, w))\ \mathsf{IN}\\
\qquad \neg {c_2}_{\mathrm{Eflg}} \ \wedge\ {c_2}_{\mathrm{PrA}} = {c_1}_{\mathrm{PrA}} \cdot \mathit{initcode}(\mathit{stksize}) \ \wedge\ {c_2}_{\mathrm{PrB}} = v \ \wedge\ Y \leq {c_1}_{\mathrm{count}} \ \wedge\\
\qquad {c_1}_{\mathrm{count}} = Y \wedge Rc(c_2, c_3) \wedge {c_2}_{\mathrm{Wptr}} = {c_1}_{\mathrm{Wptr}} \wedge \mathit{WordAddr}?(\mathit{Wsp}(\mathit{start})) \wedge \mathit{WordAddr}?(\mathit{Wsp}(\mathit{rstack})) \wedge\\
\qquad \mathit{WordAddr}?(\mathit{Wsp}(\mathit{memtop})) \ \wedge\ \mathit{WordAddr}?(\mathit{Wsp}(\mathit{heap})) \ \wedge\ \mathit{Word}?(s_{\mathrm{size}}) \ \wedge\ \mathit{Word}?(h_{\mathrm{size}}) \wedge\\
\qquad \mathit{minword} < {c_2}_{\mathrm{Wptr}} \ \wedge\ {c_2}_{\mathrm{Wptr}} + 4 * \mathit{stack} \leq \mathit{Wsp}(\mathit{base}) \ \wedge\ \mathit{Wsp}(\mathit{base}) = {c_2}_{\mathrm{Wptr}} + 4 * \mathit{stack} + 4 * s_{\mathrm{size}} \wedge\\
\qquad \mathit{Wsp}(\mathit{base}) + 4 * \mathit{stksize} \leq \mathit{Wsp}(\mathit{heap}) \ \wedge\ \mathit{Wsp}(\mathit{lastchar}) = -1 \ \wedge\ \mathit{Wsp}(\mathit{heaptop}) = h_{\mathrm{size}} \wedge\\
\qquad \mathit{Wsp}(\mathit{quotetop}) = \mathit{Wsp}(\mathit{heaptop}) \ \wedge\ \mathit{Wsp}(\mathit{heap}) + 4 * h_{\mathrm{size}} \leq \mathit{Wsp}(\mathit{rstack}) \wedge\\
\qquad \mathit{Wsp}(\mathit{rp}) = \mathit{Wsp}(\mathit{rstack}) \ \wedge\ \mathit{Wsp}(\mathit{rp}) \leq \mathit{Wsp}(\mathit{memtop}) \ \wedge\ \mathit{Wsp}(\mathit{memtop}) + 8 < \mathit{maxword} \wedge\\
\qquad \mathit{Wsp}(\mathit{outchan}) = \mathit{minword} \ \wedge\ \mathit{Wsp}(\mathit{inchan}) = \mathit{Index}(\mathit{minword}, 4) \ \wedge\ \mathit{Wsp}(\mathit{memtop}) = \mathit{MemTop} \wedge\\
\qquad {c_2}_{\mathrm{In}} = {c_1}_{\mathrm{In}} \ \wedge\ {c_2}_{\mathrm{Out}} = {c_1}_{\mathrm{Out}} \ \wedge\ \mathit{Word}?(\mathit{stack} + s_{\mathrm{size}}) \ \wedge\ \mathit{Word}?(\mathit{Wsp}(\mathit{memtop}) + 8 - {c_2}_{\mathrm{Wptr}}) \wedge\\
\qquad (\forall sa.\ sa < s_{\mathrm{size}} \Rightarrow \mathit{Word}?(\mathit{stack} + sa) \ \wedge\ \mathit{Wsp}(\mathit{stack} + sa) = \mathit{nth}(p_{\mathrm{data1}}, sa)) \wedge\\
\qquad (\forall ha.\ ha < h_{\mathrm{size}} \Rightarrow \mathit{Word}?(\mathit{Wsp}(\mathit{heap}) + 4 * ha) \ \wedge\ {c_2}_{\mathrm{Mem}}(\mathit{Wsp}(\mathit{heap}) + 4 * ha) = \mathit{nth}(p_{\mathrm{data2}}, ha)) \wedge\\
\qquad (\forall a.\ a \leq {c_2}_{\mathrm{Wptr}} \Rightarrow {c_2}_{\mathrm{Mem}}(a) = {c_1}_{\mathrm{Mem}}(a))
\end{array}
$$