Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness

Dirk Leinenbach* Wolfgang Paul Elena Petrova †
Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
E-mail: {dirkl,wjp,petrova}@cs.uni-sb.de

Abstract

In the spirit of the famous CLI stack project [2] the Verisoft project [31] aims at the pervasive verification of entire computer systems including hardware, system software, compiler, and communicating applications, with a special focus on industrial applications. The main programming language used in the Verisoft project is C0 (a subset of C which is similar to MISRA C [20]). This paper reports on (i) an operational small steps semantics for C0 which is formalized in Isabelle/HOL [25], (ii) the formal specification of a compiler from C0 to the DLX machine language [14, 23] in Isabelle/HOL, (iii) a paper and pencil correctness proof for this compiler and the status of the formal verification effort for this proof, and (iv) the implementation of the compiler in C0 and a formal proof in Isabelle/HOL that the implementation produces the same code as the specification.

1. Introduction

Pervasive systems verification [2, 22, 31] aims at the formal verification of entire computer systems including hardware, operating system, compiler, and communicating applications. This paper deals with the compilation of a C dialect in the context of pervasive verification.

At first sight this does not seem too much of a challenge. Programming language semantics is a well established field with elegant textbooks, e.g. [24, 32]. Several formal semantics for large subsets of C have been developed [13, 26, 28]. Basic proof techniques for compiler correctness are textbook material (cf. [18, 24]). Compiler verification is an active field of research and the formal verification of compilers is reported in [5, 7, 10, 21, 30].

*Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

†Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

1.1. Requirements analysis

A closer look at the requirements imposed by pervasive systems verification gives a different picture. In order to completely bridge the gap between the verified hardware and verified software we have to meet strict restrictions. In the Verisoft project the verified VAMP processor [3, 4] is used as hardware basis. Therefore its machine language DLX is the target language of our compiler. On the other hand an operating system kernel and several application programs (e.g. the compiler itself), which are written in the source language of our compiler, have to be formally verified. This requires us to use a source language which is powerful enough to implement these programs without having too complicated semantics; the latter would make formal correctness proofs of system and application software with several thousand lines of code infeasible.

The restrictions both on source and target language imply several differences between this paper and former work in the field of compiler verification:

Dynamic data structures. In the published formal semantics for C [13, 26, 28] the authors do not address dynamic memory allocation; they state that dynamic memory handling should not be defined by the language semantics but in the form of standard libraries. The formal verification of a Java compiler handling a heap is reported in [30], but there the target machine is much closer to the source language than a DLX machine; in particular the representation of the heap is literally the same in the formalization of both languages. Usage of one of the existing Java byte code processors as hardware basis would only shift the proof obligations to the hardware verification. In the Verifix project [10] the correctness of a compiler for COMLISP, which supports some kind of dynamic memory, has been shown.

Small steps semantics and step by step simulation. In multitasking systems, C programs do not run in isolation: execution of (many) user programs and of the operating

systems is *interleaved*. In big steps semantics computations cannot be interleaved in a simple way as it is hard to argue about intermediate states. Thus big steps semantics (and *classical* Hoare logics build on top of them) *alone* do not suffice to conveniently model interleaving programs. Therefore we model source language semantics by small steps (or structured operational) semantics.

In [26] Norrish defines small step semantics for a rich subset of C. But he only reports on the verification of very small C programs and his definitions seem to be too complicated to make verification of large applications feasible.

Compiler correctness proofs with respect to small steps semantics exist on paper [18, 24]. But the proofs are carried out 'big step style' by a straightforward induction over the syntax tree, which works only for terminating programs. In the context of interleaved computations and nonterminating programs (like operating systems) it is *much* more comfortable to work with a compiler correctness statement in the form of a step by step simulation theorem as it is done for a compiler backend in [34].

Small steps semantics versus Hoare logics. In [9, 27] Hoare logics for concurrent programs have been developed but the languages considered there are not powerful enough for our purposes, e.g. they do not support function calls. Of course this does not entirely preclude the use of Hoare logics for pervasive verification. Indeed both a big steps semantics and classical Hoare logics [29] were formally defined for the programs of C0 without 'address-of' operator. Presently, the bulk of all verification work for C0 programs in the Verisoft project is done using Hoare logics; especially the verification of the compiler implementation (see Sect. 5) has been done in Isabelle/HOL using Hoare logics.

Program analysis in Hoare logics necessarily concerns the execution of terminating portions of programs. In order to handle interleaving, these results then have to be 'imported' into the small steps semantics. This import is based on classical results relating Hoare logics and small steps semantics in the style of Theorem 6.16 in [24]. In a pervasive verification effort these results have to be shown formally.

Target architecture. The VAMP processor requires data to be aligned and uses delayed branches. How to fill delay slots is textbook material [14] and correct alignment of data is not a difficult problem. Nevertheless we are not aware of proofs for these mechanisms in the literature.

Pervasiveness In [5] the specification of an optimizing compiler backend from the SSA intermediate language has been formally verified. However the machine model used there is far from being the language of a realistic processor and hence the work does not suffice to bridge the gap between software and hardware for pervasive verification. On the other hand the work from [33] describes a framework for modeling the semantics of expression evaluation including non-determinism in the evaluation order. In the context of pervasive verification, languages with such complicated semantics for expression evaluation are not desirable as they complicate correctness proofs of large programs.

In the Verifix project [10] impressive work on the verification of compilers has been done; e.g. in [34] the authors report on a beautiful paper and pencil theory for the translation of intermediate languages to real machine languages. The work has been partly formalized in the PVS prover.

If the compiler optimizes code over a window of several C statements or changes the order of statements (like in [34]), the step by step simulation property may be lost inside this window. Hence we have to restrict such optimizations near places, where computations can be interleaved, e.g. when executing system calls. Note that this already concerns delay slot filling.

Implementation correctness. For pervasive systems verification a verified compiling specification is not sufficient. Additionally, one needs a verified implementation of the compiler to be sure that properties proven for applications on the source code level also hold for the compiled code. In the Verifix project the correct implementation of a compiler on the machine code level has been verified [12].

Integration of solutions. As pointed out in the previous paragraphs several subproblems related to our work have been solved in a similar or even more general way in other work. But in the context of pervasive verification an essential part of the verification effort has to be invested in the combination of the individual solutions into one framework. In addition to the impressive work of the CLI stack project [2], early work from Joyce [17] discusses problems imposed by the formal combination of a verified compiler with verified hardware. Nevertheless the source language considered in his work is not more than a toy example and the theorems which have been proven there would not fit in our framework of pervasive verification as they are carried out 'big step style'. To the best of our knowledge our work is the first which integrates all the separate solutions into a single framework that meets the needs of pervasive verification of complex systems.

1.2. Overview of the paper

In Sect. 2 we sketch syntax and semantics of C0 and prove a folklore theorem about small steps semantics. In Sect. 3 we outline a non-optimizing code generation algorithm, which has been specified in Isabelle/HOL and implemented in C0. In Sect. 4 we sketch a paper and pencil proof for a compiler correctness statement in the form of a step by

step simulation theorem. This proof depends on the folklore theorem of Sect. 2.3. In Sect. 5 we report on a formal correctness proof for the C0 implementation of the compiler. In Sect. 6 we report on the status of the formal verification effort. Section 7 concludes and discusses further work.

2. The language C0

2.1. Informal description

Semantics for the full C language are complex [13, 26, 28]. In contrast, formal semantics for Pascal can be written down in a few pages [16]. The use of all features of C leads to an error prone programming style. In safety or security critical applications one therefore tends to use C in a more restricted way that makes it more similar to Pascal [20]. The language C0 can be classified as such a Pascal-like restriction of C. We highlight some design choices.

Types. Types are fully recursive. For the pervasive verification of the compiler together with the DLX hardware the range of elementary types (bool, char, int, unsigned int) must be finite. *Scalar types* are elementary types or pointer types. Values of variables with scalar type are called *scalar values*. If t, t_1, ..., t_s are types, n is a number, and n_1, ..., n_s are names, then $*t$, $t[n]$, and $struct\{n_1 : t_1, \ldots, n_s : t_s\}$ are aggregate types. Thus pointers are typed. Unions are not allowed. We define the (abstract) size of types: for scalar types t we set $size(t) = 1$, for arrays $size(t[n]) = n \cdot size(t)$, and for structures $size(struct\{n_1 : t_1, \ldots, n_s : t_s\}) = \sum_i size(t_i)$.

Expressions and short circuit evaluation. Variable names and literals are expressions. If e and i are expressions and n_j is a name, then $e[i]$, $e.n_j$, dereferencing $*e$, and the 'address-of' operator $\&e$ are also expressions. Thus we permit the setting of pointers to subvariables of aggregate variables; pointer arithmetic however is forbidden. Expressions in C0 do not have side effects; thus we do not permit function calls as part of expressions.

The evaluation of the logical *and* (&&) and *or* (||) operators is done using short circuit evaluation. Thus for an expression $e_1 \&\& e_2$ the evaluation of e_2 is skipped in case that e_1 evaluates to *false*, and $e_1 \&\& e_2$ is evaluated to *false*. Similarly for $e_1 || e_2$ the evaluation of e_2 is skipped if e_1 evaluates to *true*. In this case $e_1 || e_2$ evaluates to *true*.

Statements. There is a separate function call statement of the form $e = f(e_1, \ldots, e_p)$. Each function contains is a single return statement at the end of the body. For later reference we present the production rules for statements where s, s_1, and s_2 are statements, e, e', and e_i are expressions, and t is a type (*comp* s_1 s_2 means sequencing of statements):

$$s \rightarrow e = e' \mid e = new\ t \mid while\ e\ do\ s \mid if\ e\ do\ s_1\ else\ s_2 \mid e = f(e_1, \ldots, e_p) \mid return\ e \mid skip \mid comp\ s_1\ s_2$$

Dynamic memory. Memory is allocated with the Pascal *new* construct. Deallocation of memory is not needed for the definition of the semantics but for the handling of finite memories in pervasive verification. We plan to implement garbage collection.

2.2. Formal small steps semantics

For functions $m : [0 : n-1] \rightarrow W$ and numbers a and d with $a < d < n$ we define $m_d(a) = m(a+d-1) \ldots m(a)$. Thus if m models a memory (of size n and element type W) and a is an address, then $m_d(a)$ is the content of d locations starting at address a. For lists l, l_1, and l_2 and single elements a we denote by $hd(l)$ the first element of l, by $last(l)$ the last element, and by $tl(l)$ a list containing all but the first element; $a \in l$ is *true* iff a is an element of list l. We denote the empty list by $[\,]$, concatenation by $l_1 \circ l_2$, and consing by $a; l_1$. For computations of the C0 machine we denote by c^i the configuration after i steps of the machine.

Often it is more convenient to talk about lists of statements instead of statement trees made up of *skip* and *comp* statements. Thus we define a function $s2l(s)$ which converts a statement tree into a list of statements as follows: $s2l(skip) = [\,]$, $s2l(comp\ s_1\ s_2) = s2l(s_1) \circ s2l(s_2)$, and for other statements x we set $s2l(x) = x$. Because $s2l(s)$ only converts *skip* and *comp* statements it returns the list of 'top-level' statements. Subtrees of conditional and while statements remain unchanged. We will often omit the $s2l$ function where the meaning is obvious, e.g. in expressions like $hd(s2l(c.pr))$ or $s2l(c.pr) = a; b$.

We will elaborate here only on definitions that cannot be literally copied from textbooks on semantics or which we need to reference explicitly in later parts of this text.

Configurations. The components of configurations c of an abstract C0 machine are: (i) a tree of C0 statements $c.pr$ called the program rest, (ii) a type table $c.tt$ storing type information, (iii) a function table $c.ft$ containing declarations of functions (type, parameters, local variables, body), (iv) a number $c.rd$ called the recursion depth, (v) a run time stack $c.lms$, which is a mapping from the set $[0 : c.rd]$ into the set of so-called *memory frames* (memory frame $c.lms(0)$ stores the global variables), (vi) a mapping from $[0 : c.rd]$ to so-called *return destinations*, which stores where the results of function calls have to be saved, and (vii) the heap memory $c.hm$; this is a memory frame storing the variables on the heap.

We annotate statements in the function table with unique identifiers. This allows us to relate the occurrence of a certain statement in the dynamic program rest with the corresponding statement in the function table. We denote by $code(s)$ the code generated for statement s, by $cad(s)$ the first address occupied by this code, by $csize(s)$ the number of instructions in $code(s)$, and by $ead(s) = cad(s) + csize(s)$ the first address after the code.

Memory model. The 'address-of' operator forces us to use a relatively explicit, low level memory model in the style of [26]. Memory frames m have the following components: (i) the number $m.n$ of variables in m, (ii) a function $m.name$ mapping variable numbers $0 \leq i < m.n$ to their names (not used for variables on the heap), (iii) a function $m.ty$ mapping variable numbers to their type. This permits us to define the size of the memory frame m as $size(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$. (iv) a content function $m.ct$ mapping the set $[0 : size(m) - 1]$ into the set of scalar values. For each elementary type the set of possible values is finite. The values of pointer variables are not addresses in the form of natural numbers but variables and subvariables, which are formally defined in the next paragraph.

Variables and subvariables. *Variables* of configuration c are pairs $V = (m, i)$ where m is a memory frame of c and $i < m.n$ is the number of the variable in the frame. The type of a variable (m, i) is defined by $ty((m, i)) = m.ty(i)$. A *selector* is a sequence $s = s_1 \cdots s_t$ where each component s_i is either of the form $[k]$ where k is a number (selecting an array component) or of the form $.n$ where n is the name of a structure component. If V is a variable and the selector s is consistent with the type of V, then $U = V\ s$ is a *subvariable*. E.g. if x is bound to (m, i) then $(m, i)\ [5]\ .n$ is a subvariable of x which corresponds to the C0 expression `x[5].n`. Variables are subvariables with empty selector.

Abstract base addresses and values. For variables $V = (m, i)$ we define their base address with respect to frame m by $ba(V) = \sum_{j<i} size(ty((m, j))$. For array components $U[k]$ of subvariables U of type $t[n]$ we define $ba(U[k]) = ba(U) + k \cdot size(t)$. For structure components $U.n_k$ of subvariables U of type $struct\{n_1 : t_1, \ldots, n_s : t_s\}$ we define $ba(U.n_k) = ba(U) + \sum_{j<k} size(ty(t_j))$. The value of a subvariable in memory m is defined by the content function as $va(U) = m.ct_{size(ty(U))}(ba(U))$.

From this point on, the definition of the semantics can be completed essentially using definitions straight from the textbooks. In particular we formalize the visibility rules by a function $bind(x, c)$ binding occurrences of variable names x to variables (m, i) in configuration c. If $bind(x, c) = (m, i)$, i.e. x is bound to variable (m, i), then the subvariable $x_{(s)}$ is bound to $bind(x\ \ s, c) = (m, i)\ \ s$. Then the value, base address, and type of x in configuration c are defined as $va(x, c) = va(bind(x, c))$, $ba(x, c) = ba(bind(x, c))$, and $ty(x, c) = ty(bind(x, c))$.

Now for left side expressions e a *simultaneous* recursive definition of $va(e, c)$ and $bind(e, c)$ (binding expressions to subvariables) is straightforward. Base address and type for e are defined as $ba(e, c) = ba(bind(e, c))$ and $ty(e, c) = ty(bind(e, c))$. The value and type of right side expressions are defined similarly.

Program rest. We highlight some parts of the definition of the program rest for the next configuration c' of c. Let the old program rest begin with statement s, i.e. $c.pr = s; r$. In most cases s is simply executed and the new program rest is $c'.pr = r$. In three cases the length of the program rest can grow: (i) If $s = while\ e\ do\ s'$ and $va(e, c) = true$ then the new program rest is $c'.pr = s'; s; r$. (ii) If $s = if\ e\ do\ s_1\ else\ s_2$ then the new program rest is $c'.pr = s_1; r$ or $c'.pr = s_2; r$. (iii) If s is a call of function f and b is the body of f then the new program rest is $c'.pr = b; r$.

2.3. A folklore theorem

In the correctness proof for the C0 compiler in Sect. 4 we will have to show that for each step c^i of a C0 computation there exists a step number $s(i)$ such that the program counters of the corresponding DLX configuration $d^{s(i)}$ match $cad(hd(c^i.pr))$. For the induction step of this proof consider a situation where $hd(c^i.pr) = s$. In order to prove the correctness of the program counters we will need to know the head $s' = hd(c^{i+1}.pr)$ of the new program rest after doing one step of the C0 machine. If s is a conditional statement, it is easy to show that the head of the new program rest is the first statement of the *if* part and the *else* part respective of the outcome of the test. A similar proof can be done if s is a *while* statement or a function call. In all other cases (e.g. assignments or *new* statements) s' is the second statement of the original program rest $c^i.pr$, i.e. the statement which followed s in the original program rest. For this proof we will need information about the possible program rests of C0 computations.

For C0 computations that start from a valid initial configuration (i.e. the program rest is initialized with the body of the main function) we prove a strong invariant on the program rest: statements x in the program rest, except returns, are always followed by a unique *successor* statement $succ(x)$ which is statically determined by the function table. To formalize this we need additional definitions.

Let s be a statement, which is uniquely identified by its ID, and let *fb* be the function body which contains s. We define the *father* of s in the following way: If $s \in s2l(fb)$, which means that s is a top-level statement of *fb*, $fa(s)$ is undefined. If there exists a while state-

ment $w = \mathit{while}\ e\ \mathit{do}\ lb$ such that $s \in \mathit{s2l}(lb)$ we define $fa(s) = w$. Similarly we define $fa(s) = c$ if there exists a conditional statement $c = \mathit{if}\ e\ \mathit{do}\ s_1\ \mathit{else}\ s_2$ such that $s \in \mathit{s2l}(s_1) \vee s \in \mathit{s2l}(s_2)$. By induction we define the i-th father of a statement s by $fa_0(s) = s$ and $fa_i(s) = fa_{i-1}(fa(s))$ for $i > 0$.

Next we define the *environment* of statements s: If $fa(s) = \mathit{if}\ e\ \mathit{do}\ s_1\ \mathit{else}\ s_2$ we define $env(s) = \mathit{s2l}(s_1)$ if $s \in \mathit{s2l}(s_1)$ and $env(s) = \mathit{s2l}(s_2)$ if $s \in \mathit{s2l}(s_2)$. If $fa(s) = \mathit{while}\ e\ \mathit{do}\ lb$ we define $env(s) = \mathit{s2l}(lb)$. If the father of s is undefined we set $env(s) = \mathit{s2l}(\mathit{fb})$.

For a list l and an element e with $e \in l$ we define by induction the *next* element by $next(e, l) = hd(tl(l))$ if $e = hd(l)$ and $next(e, l) = next(e, tl(l))$ otherwise. For a statement s we define its *direct successor* by $dsucc(s) = next(s, env(s))$. The direct successor is undefined if s is the last statement in $env(s)$. If s is not the last statement of a function body (in this case it would be a *return* statement) the *successor* of s is defined recursively by:

$$succ(s) = \begin{cases} dsucc(s) & : \ dsucc(s) \text{ is defined} \\ fa(s) & : \ dsucc(s) \text{ is undefined} \wedge \\ & \quad fa(s) \text{ is a } \mathit{while} \text{ statement} \\ succ(fa(s)) & : \ \text{otherwise} \end{cases}$$

Syntax trees have finite depth so the recursion terminates. We call the last father statement in this recursive process *senior* and define formally: $sen(s) = sen(fa(s))$ iff $dsucc(s)$ is undefined and the father of s is a conditional statement. Otherwise we define $sen(s) = s$.

Theorem 2.1 *For all steps c^i of a C0 computation the following invariant holds: If $s \in c^i.pr$ and s is not a return statement then $next(s, c^i.pr) = succ(s)$, i.e. s is always followed by its successor statement.*

We prove this theorem by induction on the step number i. By the definition of $succ(s)$ it is easy to show that the invariant holds for all subtrees of the program rest which are part of the function table $c.ft$. Because the initial program rest consists only of the body of the main function, which is a part of $c.ft$, there is not much to show for the induction start. For the induction step let the program rest be $c^i.pr = s; r$. We do a case split on the head s of the program rest, which will be executed in the next step.

1. If s is an *assign*, a *while* with *false* condition, a *return*, or a *new* the induction step is easy because we just remove the first statement from the program rest: $c^{i+1}.pr = tl(c^i.pr)$. Here the invariant obviously also holds for the new program rest.

2. If $s = \mathit{while}\ e\ \mathit{do}\ lb$ and $va(e, c^i) = \mathit{true}$ we have $c^{i+1}.pr = lb; \mathit{while}\ e\ \mathit{do}\ lb; r$. Because lb is a part of the function table it is easy to show that the invariant holds for the lb part of the new program rest. The other part of the program rest remained the same so there is nothing to show there. The crucial point is to show that the *while* statement is the correct successor for the last statement of the loop body, formally: $succ(last(\mathit{s2l}(lb))) = \mathit{while}\ e\ \mathit{do}\ lb$. This follows directly from the second case of the definition of $succ(s)$.

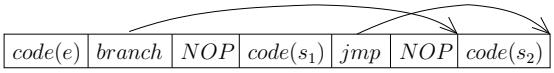

Figure 1. $code(\mathit{if}\ e\ \mathit{do}\ s_1\ \mathit{else}\ s_2)$

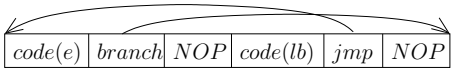

Figure 2. $code(\mathit{while}\ e\ \mathit{do}\ lb)$

3. If $s = \mathit{if}\ e\ \mathit{do}\ s_1\ \mathit{else}\ s_2$ the new program rest is $c^{i+1}.pr = s_1; r$ or $c^{i+1}.pr = s_2; r$. Let x denote $last(\mathit{s2l}(s_1))$ or $last(\mathit{s2l}(s_2))$, respectively. Here the crucial point is to show $next(x, c^{i+1}.pr) = succ(x)$. By the third case of the definition of $succ$ we know that $succ(x) = succ(s)$. Therefore we can conclude with help of the induction hypothesis $next(x, c^{i+1}.pr) = next(s, c^i.pr) = succ(s) = succ(x)$ which proves this case of the induction step.

4. If s is a function call of function f the induction step is easy. Let *fb* be the body of function f. The new program rest then is $c^{i+1}.pr = \mathit{fb}; r$. For r the invariant holds by induction hypothesis and for *fb* because *fb* is a subtree of the function table. The interesting case is again the transition from *fb* to r, namely that $hd(\mathit{s2l}(r))$ is the successor of $last(\mathit{s2l}(\mathit{fb}))$. In this special case there is nothing to show because $last(\mathit{s2l}(\mathit{fb}))$ is a return statement and the invariant does not state anything for return statements.

3. Non-optimizing compilation

The target machine is a DLX machine as described in [23] without special purpose registers or interrupts. The delayed branch mechanism is realized by two program counters, which we will call here PC and DPC (delayed PC).

As compiler frontend we use a non-verified C program. Translation and verification starts from abstract syntax trees. Expressions are translated using an algorithm similar to the 'Aho-Ullman' algorithm [1]. Statement translation is straightforward; in Figs. 1 and 2 we have sketched the code which is generated for conditional and while statements. Delay slots are filled by NOPs.

In the formal verification process, we consider two versions of the code generation function: (i) an Isabelle/HOL function $code(s)$ mapping C0 statements s to assembler code and (ii) its C0 implementation. The former is much

more appropriate to prove the simulation theorem, but the latter is what one is interested in for pervasive verification.

In Sect. 4 we sketch the proof of a step by step simulation theorem between C0 programs p and the compiled code $code(p)$ and in Sect. 5 we report on a formal proof that the C0 implementation of the code generation function generates the same code as the Isabelle function.

3.1. Alignment and allocation

For natural numbers n and d we use the notation $\lceil n \rceil_d = \lceil n/d \rceil \cdot d$. This is the smallest multiple m of d such that m is greater or equal than n.

Alignment. We define inductively the alignment $alg(t)$ of types t. Variables of type t will be aligned at addresses which are multiples of $alg(t)$. For elementary types we define $alg(bool) = alg(char) = 1$, $alg(int) = 4$, and $alg(*t) = 4$ for pointer types. For arrays we set $alg(t[n]) = alg(t)$ and for structures $alg(struct\{n_1 : t_1, \ldots, n_s : t_s\}) = max\{alg(t_i) | 1 \leq i \leq s\}$.

Displacement and allocated size. The allocated size $asize(t)$ of a type t specifies, how many bytes will be occupied by a variable of type t in the target machine. For scalar types we set $asize(t) = alg(t)$. For aggregate types t we define inductively and simultaneously the allocated size and the displacement $dspl(t, i)$ of their components. For array types $t = t'[n]$ we set $dspl(t, i) = i \cdot \lceil asize(t') \rceil_{alg(t')}$ and $asize(t) = dspl(t, n-1) + asize(t')$. For struct types $t = struct\{n_1 : t_1, \ldots, n_s : t_s\}$ we set $asize(t) = dspl(t, s) + asize(t_s)$ and define the displacement by $dspl(t, 1) = 0$ and $dspl(t, i+1) = \lceil dspl(t, i) + asize(t_i) \rceil_{alg(t_{i+1})}$.

Displacement of variables in memory frames m is almost computed like the displacement of components in structures. Because the first three words (12 bytes) of memory frames store return address, address for the result and a pointer to the previous frame, we set $dspl(m, 0) = \lceil 12 \rceil_{alg(ty(m,0))}$ and $dspl(m, i+1) = \lceil dspl(m, i) + asize(ty(m, i)) \rceil_{alg(ty(m,i+1))}$. For memory frames the allocated size is defined by $asize(m) = dspl(m, m.n-1) + asize(ty(m, m.n-1))$.

Allocated base address. We will specify a so-called allocation function aba which specifies for frames, subvariables, and left side expressions of C0 machines at which address they are stored in the DLX machine. For variables x of memory frames m we define $aba(x) = aba(m) + dspl(bind(x))$. For subvariables U of type $t[n]$ component $t[i]$ is allocated at address $aba(U[i]) = aba(U) + dspl(t, i)$. Completely analogous for subvariables U of type $struct\{n_1 : t_1, \ldots, n_s : t_s\}$ component $U.n_i$ gets the allocated base address $aba(U.n_i) = aba(U) + dspl(t, i)$. As left side expressions always correspond to subvariables in an obvious way we extend the definition of aba to left side expressions.

Code generation for function calls will guarantee that the base address of frames $aba(m)$ is a multiple of four. Code generation for new statements will guarantee that the allocated base address $aba(hm, j)$ is a multiple of $alg(ty((hm, j)))$. These definitions permit to show:

Lemma 3.1 *For all subvariables U we have: $aba(U)$ is a multiple of $alg(ty(U))$.*

We prove this lemma first for variables. For subvariables we proceed by induction on the length of the subvariable selectors s. Applied for the scalar types the lemma allows us to prove formally the absence of misalignment interrupts.

4. Correctness of the compiler specification

4.1. Simulation relation and main theorem

We have to define a simulation relation between configurations c of the C0 machine and configurations d of the DLX machine. DLX configurations are composed of (i) two program counters $d.PC$ and $d.DPC$ implementing the delayed branch mechanism (see [23]), (ii) the memory $d.m : \{0,1\}^{32} \rightarrow \{0,1\}^8$, and (iii) the general purpose register file $d.GPR : \{0,1\}^5 \rightarrow \{0,1\}^{32}$. As the set of variables of a C0 machine changes with new statements, function calls, and returns, and because garbage collectors can change the allocated base address of heap variables, the simulation relation must be parametrized by the current allocation function aba. Thus the consistency predicate $consis(c, aba, d)$ states that with the allocation function aba the DLX configuration d encodes the C0 configuration c. The predicate requires control consistency $c\text{-}consis(c, d)$ and data consistency $d\text{-}consis(c, aba, d)$.

Control consistency states that the DLX program counters point to the code of the first statement in the current program rest: $d.DPC = cad(hd(c.pr))$ and $d.PC = d.DPC + 4$.

Data consistency is a conjunction of four predicates:

- code consistency $code\text{-}consis(c, d)$: Code consistency requires that the compiled code of the C0 program is stored at address 0 of the DLX machine, i.e. it forbids self-modifying code.

- value consistency $v\text{-}consis(c, aba, d)$: This predicate requires for all elementary subvariables U that the C0 and the DLX machine store the same value, formally[1]:

[1] Actually the representations of scalar values in the C0 and the assembler machine differ. In Isabelle/HOL we have to apply an additional conversion function which we omitted here.

$va(U, c) = d.m_{asize(ty(U))}(aba(U))$. Thus subvariable U is stored in the memory m of the DLX machine at the $asize(ty(U))$ bytes starting at address $aba(U)$.

- pointer consistency $p\text{-}consis(c, aba, d)$: This predicate requires for reachable subvariables U and pointer variables p pointing to U that the value stored at the allocated address of variable p in the DLX machine is the allocated base address of U, i.e. $d.m_4(aba(p)) = aba(U)$. This defines a subgraph isomorphism between the reachable portions of the heaps of the C0 machine and the DLX machine.

- stack consistency $s\text{-}consis(c, d)$: This is a predicate about the implementation of the run time stack and the content of some special registers. Informally it states that (i) $d.GPR(30)$ is the stack pointer, (ii) $d.GPR(29)$ is the heap pointer, (iii) the first three words of each frame in the DLX machine store the return address, the address for the result, and a pointer to the previous frame, and (iv) that for all $i+j = c.rd$ the return address stored in the j-th stack frame matches the address of the statement which follows the i-th $return$ statement in the program rest.

The main theorem of the compiler correctness proof is stated below. It is proven by induction on the steps t of the C0 machine and we sketch some details of the induction step in the following sections.

Theorem 4.1 *For all C0 computations c^0, c^1, ... there is a computation d^0, d^1, ... of the DLX machine, a sequence aba^0, aba^1, ... of allocation functions, and a sequence $s(0)$, $s(1)$, ... of step numbers, such that for all t it holds: $consis(c^t, aba^t, d^{s(t)})$, i.e. t steps of the C0 machine are simulated by $s(t)$ steps of the DLX machine.*

4.2. Correct expression evaluation

Expression translation uses the Aho-Ullman algorithm [1] unless a different order is prescribed by the short circuit evaluation rule. For every subexpression u of an expression e we first compute a flag $R(u) \in \{0, 1\}$ indicating whether a right value ($R(u) = 1$) or a left value has to be computed. We also compute the *size* $s(u)$ of expressions u as the number of nodes in the abstract syntax tree for u. For right values the *value* of the expression has to be computed and for left values the *address* of the expression is needed. The actual code $code(u)$ is generated by a recursive function $cgen_{\mathrm{e}}(u, D, F)$ where D is the destination register and F is a list of registers which may be used for the evaluation of u; we require $D \notin F$.

We state the main invariant for expression translation. Assume $d\text{-}consis(c, aba, d)$ and $code(u) = cgen_{\mathrm{e}}(u, D, F)$. Let d' be the DLX configuration reached by running $code(u)$ starting in configuration d. Then the content of the destination register D in configuration d' is (i) $d'.GPR(D) = aba(u)$ iff $R(u) = 0$, (ii) $d'.GPR(D) = 0^{32-8\cdot asize(ty(u,c))}\, va(u, c)$ iff $R(u) = 1$ and $ty(u, c)$ is elementary, or (iii) $d'.GPR(D) = aba(va(u, c))$ iff $R(u) = 1$ and $ty(u, c)$ is a pointer type.

4.3. Correctness of statement translation

For data consistency, the induction step of the main theorem follows standard textbook arguments, and will not be treated here. For control consistency we use Theorem 2.1 and do the induction step of the proof by case distinction on the first statement of the program rest $s = hd(c^i.pr)$.

If s is a *while* statement *while e do lb* and $va(e, c^i) = true$ we know that the head s' of the new program rest is the first statement of the loop body $s' = hd(c^{i+1}.pr) = hd(lb)$. In this case we just have to show that the expression e is correctly evaluated (see Sect. 4.2) and that the branch instruction behind $code(e)$ (cf. Fig. 2) is correctly executed, here: that the branch is *not* taken. Let d' be the configuration of the DLX machine after executing the branch instruction. Then we know that $d'.DPC = cad(hd(c^{i+1}.pr))$.

If $s =$ *if e do* s_1 *else* s_2 the new program rest depends on the value of the condition $va(e, c^i)$. If it is $true$ the head of the new program rest $hd(c^{i+1}.pr)$ is $hd(s_1)$ otherwise it is $hd(s_2)$. The proof is similar to the *while* case: we just need to show that in the DLX configuration right after executing the branch the program counter points to the address of s_1 and s_2, respectively.

If s is a $return$ statement the head of the new program rest is the statement s' which follows s in the current program rest $c^i.pr$. By the last part of the definition of stack consistency we know that the return address on the stack which is used to jump back at the end of $code(s)$ equals $cad(s')$. Now it is easy to show: $d'.DPC = cad(s')$.

For all other types of statements s the proof is more complicated. Let $s' = succ(s)$ be the successor statement of s. First we show that after the execution of $code(s)$ the value of the program counter is $d'.DPC = ead(s)$ and that $hd(c^{i+1}.pr) = s'$. Now we have to distinguish two cases:

First we consider the case that s is *not* the last statement of $env(s)$. Then the successor of s is $s' = dsucc(s)$ by the first case of the definition of $succ(s)$. By a simple proof on the code generation function we can show that $code(s')$ is located directly behind $code(s)$. Therefore we can easily conclude $d'.DPC = ead(s) = cad(s')$ which proves control consistency for this case.

In the second case s is the last statement of $env(s)$. If $fa(s)$ is a *while* statement the proof is simple because we know that $s' = fa(s)$ and just need to show that the jump instruction after the loop body (cf. Fig. 2) is correctly executed. Otherwise $fa(s)$ is a conditional state-

ment and the proof is more complicated because the definition of $succ(s)$ is recursive. By finite induction on j we prove that there exist t_j such that after doing t_j DLX steps starting from configuration d the program counter points directly behind the code of the j-th father of s, formally $d^{t_j}.DPC = ead(fa_j(s))$. The induction start ($d'.DPC = ead(fa_0(s))$) is already proven because $fa_0(s) = s$. In the induction step we conclude from $j-1$ to j and know that $d^{t_{j-1}}.DPC = ead(fa_{j-1}(s))$. If $fa_{j-1}(s)$ is the last statement in the *else* part of $fa_j(s)$ there is nothing to show because the last addresses of both coincide (cf. Fig. 1): $ead(fa_{j-1}(s)) = ead(fa_j(s))$. If $fa_{j-1}(s)$ is the last statement in the *if* part we just need to show that the jump instruction behind the *if* part (correctly) jumps to the address $ead(fa_j(s))$ directly behind the code of $fa_j(s)$. The induction ends at $sen(s)$ because $succ(s) = dsucc(sen(s))$.

4.4. Delay slot filling

Consider the translation of *while e do lb*. After $code(lb)$ one has to insert a jump to $cad(code(e))$. Without optimization the delay slot after the jump is filled with a NOP instruction. In non-optimized code for expression evaluation the actual value of e is computed from scratch. Thus $code(e)$ starts either with the load of a variable or with the generation of a constant. It never starts with a control instruction. Thus we can always copy the first instruction of $code(e)$ into the delay slot after the jump and change the destination of the jump to the second instruction of $code(e)$.

The correctness of this optimization is proven on the *assembler level*: a DLX machine with the optimized code simulates a DLX with the non-optimized code step by step *except* at the filled delay slot; there three steps are simulated by two steps. These steps belong however to two different statements of the C0 machine: (i) the last statement of the loop body lb and (ii) the *while* statement which follows $last(lb)$ in the program rest. Combined with the compiler correctness, we obtain that the optimized code simulates the C0 machine step by step except for the two statements identified above. This causes complications in a single situation: if computation is interleaved exactly at the boundary between these statements (i.e. $last(lb)$ is an I/O operation). The easiest solution is not to fill delay slots in this case.

5. Correct compiler implementation

As already pointed out in Sect. 1.1 verification of the compiling specification is not enough; additionally we need to show that the compiler implementation in C0 is correct. This proof has been done in Isabelle/HOL using the verification environment from [29], which is based on Hoare logic and uses an automatic verification condition generator (VCG).

In the verification environment C0 expressions are shallow embedded into Isabelle/HOL and the state space of the C0 program is represented as an Isabelle record. The state includes global and local variables and several functions which model the heap. The range of elementary variables is infinite. Thus, in order to be able to use the results, proven with the verification environment, in the context of pervasive verification it is necessary to add guards to statements. The guards will ensure that the values of variables and expressions are always in a fixed range. The heap can contain structures and its representation follows ideas from [6]. Pointers are modeled by an abstract type ref. There is one heap function of type $ref \rightarrow t$ for each component of type t of a structure in the heap; this keeps field updates in the heap disjoint and simplifies verification. The VCG applies Hoare rules automatically to a Hoare triple $\{P\}c\{Q\}$ until the program c is completely processed; thereby it computes the weakest precondition WP. After applying the VCG it remains to prove that the precondition of the Hoare triple fulfills the weakest precondition: $P \subseteq WP$.

We formulate the verification goals inside the Hoare triples using *abstraction functions* which abstract the current state from concrete C0 variables to abstract HOL types [19]. For example we can formulate an abstraction function $List(\sigma, s, h, l)$ for a list data structure in state σ with start pointer s and heap function h by $List(\sigma, s, h, [\,]) = (s = Null)$ and $List(\sigma, s, h, (p; ps)) = (s = p \wedge s \neq Null \wedge List(\sigma, h(s), h, ps) \wedge h(last(ps)) = Null)$. This abstraction function states that the list of references which is reached by following the start reference up to $Null$ equals the abstract Isabelle list l.

Such abstraction functions exist for all relevant data structures of the compiler implementation. The top-level abstraction functions are (i) $C0_{prog}(\sigma, tt, ft, gv)$ which states that the state σ encodes the C0 program which is in Isabelle represented via type table tt, function table ft, and a list of global variables gv and (ii) $ASM_{code}(\tau, al)$ which states that the state τ contains a list of DLX instructions al. Using these abstraction functions we can formulate the top-level correctness theorem for the compiler implementation.

Theorem 5.1 *Let $p = (tt, ft, gv)$ be a C0 program and let σ be the initial state of the compiler implementation where the data structures are initialized with the concrete representation of program p. Then we know that, after executing the C0 function $cimpl()$ which implements the compiler, the state encodes exactly that list of DLX instructions which is specified by the compiling specification via $code(tt, ft, gv)$. Formally:*

$$\{C0_{prog}(\sigma, tt, ft, gv)\}$$
`Call cimpl();`
$$\{ASM_{code}(\tau, code(tt, ft, gv))\}$$

The partial correctness of this theorem has been proven in Isabelle/HOL (at the time of this writing still without guards). Note that we presently do not prove the correctness of the compiler frontend, but assume that the initial state of the compiler implementation contains the correct abstract syntax tree for the input program. Eventually this has to be formally verified.

The main verification efforts concern the points, that the implementation and specification are programs written in an imperative and a functional programming language, respectively. The equivalence of while loops to recursive function (often nested) must be shown. Absence of pointers in the specification language makes some data structures very different from the implementation and hence, abstraction functions and verification more complex. In some cases functions in the specification are equivalent to a combination of several C0 functions. Then specifications of the latter demand additional functions to specify the behavior of the C0 functions.

6. Status of the formal verification effort

At the time of this writing (June 2005) a considerable part of the formal compiler verification has already been done in Isabelle: (i) Theorem 2.1 has been proven, (ii) the compiler implementation (over 1 000 lines of C0 code) has been proven correct using Hoare logic for partial correctness without guards (iii) for several types of expressions we have proven formally that the generated assembler code simulates the expression evaluation of the C0 semantics, and (iv) large parts of the proof from Sect. 4.3 have been verified. We plan to finish the formal verification of the C0 compiler (without delay slot filling) in the fall of 2005.

The formal proof (without guards) for partial correctness of the compiler implementation has about 900 lemmas and a total of 20 000 proof steps. In the Verisoft project a software model checker is being integrated into Isabelle/HOL. This model checker will be used to discharge the guards which ensure that elementary variables always are in the correct range. We expect that most of the guards for the compiler implementation will be discharged automatically.

7. Summary and further work

We have sketched a paper and pencil correctness proof of a compiler which deals with (i) delay slots, (ii) alignment, (iii) dynamic heap, (iv) pointers to subvariables, and (v) function calls. This proof has been done in the context of pervasive systems verification with all additional requirements from Sect. 1.1 in mind. Especially arguments about small steps semantics have become mandatory. The equation systems of the correctness proof (alignment, consistency, etc.) seem natural and are amazingly short. This is not a matter of a beauty contest: the construction of formal correctness proofs is an engineering effort and short proofs are formalized much faster than longer ones. The formal verification of the compiler specification is work in progress but will hopefully be finished during the next months.

For the pervasive verification effort in Verisoft the compiler correctness proof has to deal with resource limitations on the target machine (e.g. restricted memory size). Thus the top level correctness theorem has to be extended with additional requirements on the C0 computations (e.g. limits on recursion depth or dynamic memory consumption) which guarantee that properties proven on the source language level also hold for the compiled code in *all* cases.

We have briefly sketched a formal correctness proof for the compiler implementation using a Hoare logic for partial correctness; addition of guards and total correctness are future work. Furthermore the following steps are planned in the Verisoft project in the context of compiler verification:

1. Bootstrap correctness: As we want to use our C0 compiler on a real system we eventually need to have a binary 'bootstrap' version of it [12]. The binary version of the compiler may be generated by compiling the C0 implementation of the compiler with the gcc. In order to show the correctness of the bootstrap compiler we plan to use translation validation. Additionally we can compile the compiler implementation by executing the compiler specification and increase the trust in the compiler's bootstrap version.

2. Operating system kernel verification: The correctness statement for CVM-style micro kernels [11] is a step by step simulation theorem between a parallel model (using a C0 machine for the kernel and virtual DLX machines as user processes) and a physical DLX machine. The proof depends both on the compiler correctness proof presented here and the virtual memory simulation theorem from [8, 15].

3. Inline assembler: For the implementation of operating system kernels inline assembler code is necessary e.g. for I/O. Thus we plan to extend the C0 compiler presented in this paper by support for $C0_{\mathrm{A}}$ which is C0 extended by inline assembler code. The formal semantics of $C0_{\mathrm{A}}$ have already been defined.

Acknowledgments

For helpful discussions, constructive criticism, and suggestions the authors thank Willem-Paul de Roever, Wolf Zimmermann, and Sabine Glesner.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume II: Compiling. Prentice-Hall, 1973.

[2] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning (JAR)*, 5(4):411–428, Dec. 1989.

[3] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP processor. In D. Geist and E. Tronci, editors, *CHARME'03*, Lecture Notes in Computer Science (LNCS), pages 51–65. Springer, 2003.

[4] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Putting it all together: Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 7, 2005. To appear.

[5] J. O. Blech and S. Glesner. A formal correctness proof for code generation from SSA form in Isabelle/HOL. In P. Dadam and M. Reichert, editors, *GI Jahrestagung (2)*, volume 51 of *Lecture Notes in Informatics*, pages 449–458. Gesellschaft für Informatik, 2004.

[6] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[7] P. Curzon. The verified compilation of Vista programs. In *1st ProCoS Working Group Meeting*. Gentofte, Denmark, January 1994.

[8] I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *CHARME'05*, LNCS. Springer, 2005. To appear.

[9] W. de Roever, U. Hanneman, J. Hooiman, Y. Lakhneche, M. Poel, J. Zwiers, and F. de Boer. *Concurrency Verification*. Cambridge University Press, Cambridge, UK, 2001.

[10] A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In R. Hariharan, M. Mukund, and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Sience*, volume 2245 of *LNCS*, pages 144–155. Springer, 2001.

[11] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *TPHOLs'05*, LNCS. Springer, 2005. To appear.

[12] 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.

[13] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Börger, G. Jäger, H. K. Büning, S. Martini, and M. M. Richter, editors, *CSL'92*, volume 702 of *LNCS*, pages 274–308. Springer, 1993.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[15] M. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[16] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica (ACTA)*, 2:335–355, 1973.

[17] J. J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis*, volume 408 of *LNCS*, pages 177–201. Springer, 1989.

[18] J. Loeckx, K. Mehlhorn, and R. Wilhelm. *Foundations of Programming Languages*. B.G. Teubner Stuttgart, 1986.

[19] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *CADE'03*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[20] The Motor Industry Software Reliability Association (MISRA). *MISRA-C:2004 — Guidelines for the use of the C language in critical systems*. Motor Industry Research Association (MIRA), Ltd., UK, 2004.

[21] J. S. Moore. A mechanically verified language implementation. *JAR*, 5(4):461–492, 1989.

[22] J. S. Moore. A grand challenge proposal for formal methods: A verified stack. In B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2003.

[23] S. M. Mueller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[24] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[26] M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, Dec. 1998.

[27] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the Association for Computing Machinery (ACM)*, 19(5):279–285, 1976.

[28] N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[29] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *LPAR'04*, volume 3452 of *LNCS*, pages 398–414. Springer, 2005.

[30] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 63–77. Springer Verlag, 2002.

[31] The Verisoft Consortium. The Verisoft project. `http://www.verisoft.de/`, 2003.

[32] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.

[33] W. Zimmermann and A. Dold. A framework for modelling the semantics of expression evaluation with abstract state machines. In E. R. Egon Boerger, Angelo Gargantini, editor, *ASM'03*, volume 2589 of *LNCS*, pages 391–406. Springer Verlag, 2003.

[34] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science*, 3(5):504–567, May 1997.