Compiler Verification for C0 (intermediate report)

Martin Strecker

Université Paul Sabatier, Toulouse, France *
http://www.irit.fr/~Martin.Strecker

Abstract. This paper describes formalisations of a type-safe fragment of the C programming language (called C0) and of the DLX assembly language. It then presents the definition and correctness proof of a substantial fragment of a C0-to-DLX compiler, carried out in the proof assistant Isabelle.

1 Introduction

This paper reports on a major compiler verification effort, using the proof assistant Isabelle [NPW02], that is carried out in the context of the *Verisoft*[Vera] project, which aims at demonstrating the practical feasibility of comprehensive and large-scale software verification. Here, "comprehensive" means that an entire platform is taken into consideration, ranging from processor, peripherals, operating system to application software. "Large-scale" means that software written by application developers in a general-purpose language can be verified with little effort.

This setting determined the choices that were made when selecting source and target language of the compiler. As source language, a dialect of C (which we call C0) was developed which offers most of the facilities found in "real" C (among them arrays and dynamic data structures) while omitting idiosyncrasies that would make the language difficult to analyse, not only on the meta-level, but also with regard to verification of individual programs. Indeed, from the language as it is defined now, a Hoare logic has been derived, which is the basis of a C0 verification environment [Sch04] that has already been successfully applied to major case studies such as an implementation of a BDD algorithm [Ort04].

The most significant departure from real C is that our language is strictly type safe and does not permit any form of numerical address calculation. In particular, there is no "address of" operator and no address arithmetic. On the source level, identification of subcomponents is entirely symbolic, the mapping to numeric addresses is performed by the compiler. When abandoning this principle, we would be obliged to reason on the byte level and to give up any structural inferences.

* Major parts of this work were carried out at the Technische Universität München with funding from project *Verisoft*

As target language, the assembler of the DLX processor was chosen. Even though DLX is considerably less complex than the architecture of state-of-the-art processors, it incorporates features such as pipelining which have been the object of an in-depth machine-assisted analysis [BBJ+02], so we have good reasons to believe that our approach scales up to more advanced designs.

The work presented here is still in progress (whence the subtitle "intermediate report"), but has reached a stage where the general structure of the compiler becomes clearly apparent and where the viability of the language design decisions has been confirmed by the completion of major subproofs.

After a small summary of notation of the proof assistant Isabelle (Section 2), we describe the source language C0 (Section 3 and 4) and the target language of our compiler (Section 5), before defining the compilation function (Section 6), introducing our notion of compiler correctness and commenting on the correctness proof (in Section 7). In Section 8, we conclude with a comparison with related work. Source and target language are stable in the sense that no substantial modifications are expected in the near future. The compiler definition and -proof can so far only handle "expressions" of the source language, but no "statements", which we expect to deal with in the next few months.

Due to space limitations, we can only sketch our formalisation and the resulting compiler. We cannot even introduce the definitions of all functions used in excerpts of our formalisation, but the names should be self-explaining.

2 Notation

This section introduces some notation and some elementary functions used in the following.

Isabelle's syntax is reminiscent of ML, so we will only mention a few peculiarities: Consing an element `x` to a list `xs` is written as `x#xs`. Infix `@` is the append operator, `xs ! n` selects the `n`-th element from list `xs`.

The first resp. second component of a pair is selected by `fst` resp. `snd`.

We have the usual type constructors `T1 × T2` for product and `T1 ⇒ T2` for function space. The long arrow $\Longrightarrow$ is Isabelle's meta-implication, in the following mostly used in conjunction with rules of the form $[\![P_1; \ldots; P_n]\!] \Longrightarrow C$ to express that `C` follows from the premises $P_1 \ \ldots \ P_n$. Apart from that, there is the implication $\longrightarrow$ of the HOL object logic, along with the standard connectives and quantifiers. The distinction between $\Longrightarrow$ and $\longrightarrow$ is mostly operational; in most cases, they can be considered as equivalent in the following.

The polymorphic option type

```
datatype 'a option = None | Some 'a
```

is frequently used to simulate partiality in a logic of total functions: Here, `None` stands for an undefined value, `Some x` for a defined value `x`.

Lifted to function types, we obtain the type of "partial" functions `T1 ⇀ T2`, which just abbreviates `T1 ⇒ (T2 option)`. Functions of this type are also called *maps* from `T1` to `T2`.

Records can be understood as named tuples. We use them, for example, for defining the program state in Section 4.1. A record with fields `c1 ... cn` and respective component values `v1 ... vn` is created by ⦇ `c1 = v1, ... cn = vn` ⦈. Selection of a component `c` from record `r` is just function application `c r`. Record update is written as `r` ⦇`c := v` ⦈.

3 Source Language: C0

3.1 Types

We have elementary types Boolean, integers and pointers and composite types `Struct`, consisting of a list of pairs component name / type, and arrays of a fixed size. Thus, `Arr n T` is the array of length `n` with component types `T`.

```
datatype ty
  = Boolean
  | Integer
  | Struct ((cname × ty) list)
  | Arr nat ty
  | Ptr tname
```

As in all languages where pointers play a role, we have to be able to reference objects of a type which is structurally at least at large as the pointer type itself – otherwise it would not be possible, for example, to build up recursive structures. We therefore introduce an indirection: Pointer types do not reference other types, but a type name `tname`.

The correspondence between type names and types is established by a type name environment `tname_env` which maps type names to types.

```
types tname_env = tname ⇀ ty
```

We inductively define a predicate `wf_type` that checks the well-formedness of a type with respect to a type environment. It verifies, among others, that component names in structs are unique and that type names are defined in the environment.

3.2 Values

We use structured values provided with a type tag, which in a sense made precise in Section 3.4 allow us to recognise the type of the value. Apart from elementary values matching the elementary types, we have struct values (a list of pairs component name / value) and arrays (list of component values). For locations, from which address values are constructed, we use an uninterpreted type `loc`.

typedecl *loc* — locations, i.e. abstract references on objects

datatype *val*
```
  = Bool bool    — Boolean value
  | Intg int     — integer value, name
  | Structv ((cname × val) list)
  | Arrv (val list)
  | Null         — null reference
  | Addr loc     — addresses, i.e. locations of objects
```

A remark about integer values is in order: Even though we use unbounded integers in the definition of *val*, we suppose that we have a finite architecture in which only values up to a certain size are representable. In fact, our whole development is parameterised by a constant *wlen_bit*. Arithmetic operations (Section 4.2 and 5) are modulo, and memory is limited accordingly.

3.3 Expressions and Statements

We introduce two uninterpreted types of names and function names. Since it can be decided by type checking during preprocessing whether a variable is local or global, we have chosen to distinguish global and local variable names syntactically. In addition, there is a result variable *Res* to which the return value of a function is assigned. This makes the definition of semantics more convenient. Translating a `return` statement to a corresponding assignment is again an issue for a preprocessor.

datatype *vname*
```
  = GVName name  — global variable name
  | LVName name  — local variable name
  | Res          — function result variable
```

Based on a type *binop* of binary and *unop* of unary operators, we can now define the type of expressions. Expressions are parameterised with a type variable for which we will later substitute *ty*, so we will mainly use expressions with type attributes, which turn out to be handy when defining the compiler.

datatype *'a expr*
```
  = VarAcc vname 'a                        — variable access
  | ArrAcc ('a expr) ('a expr) 'a          — array access
  | StructAcc ('a expr) cname 'a           — structure access
  | BinOp binop ('a expr) ('a expr) 'a     — binary operation
  | UnOp unop ('a expr) 'a                 — unary operation
  | Deref ('a expr) 'a                     — pointer dereference
  | Lit val 'a                             — literal value
  | NullPtr 'a
```

We have made some concessions to the syntax in order to be able to easily reason about programs written in our language. As a consequence, expressions are side-effect free. While the statements of the language, defined below, are mostly standard, function call *SCall* is limited to the format *x = f(ps)*, where *ps* is a list of parameter expressions and *x* a variable. Obviously, expressions with nested function calls can be decomposed into a sequence of function calls of the above form and assignments.

Pointer allocation reserves storage for a type *tname* and assigns the pointer to an expression which is supposed to yield an lvalue. Thus, *PAlloc a tn* corresponds to something like `a = new(tn)`.

We introduce some pretty-printing syntax: Assignment can be written as *l::= r*, statement composition as *c1;;c2*, a conditional as *If (b) ct Else ce* and a while loop as *While (b) c*.

datatype *'a stmt*

```
  = Skip                                  — empty statement
  | Ass ('a expr) ('a expr)               — assignment    (_::=_)
  | PAlloc ('a expr) tname                — Pointer allocation
  | SCall vname fname ('a expr list)       — function call  ( _ ::=_.'(_'))
  | Comp ('a stmt) ('a stmt)              — Composition   (_;; _)
  | Cond ('a expr) ('a stmt) ('a stmt)      (If '(_') _ Else _)
  | Loop ('a expr) ('a stmt)                (While '(_') _)
```

From these ingredients, we can gradually built up a program. Type declarations *tdecl* and variable declarations *vdecl* associate type resp. variable names to types. A function definition contains a function name, a function declaration and a function body. The latter is just a single statement. As mentioned in Section 3.3, the return value is determined by what has been assigned to the special variable *Res* in the function body.

A function declaration consists of the list of (formal) parameter declarations, local variable declarations and the return type of the function.

We can now gather these components and define the structure of programs:

types

— program: type decls, global var. decls, function definitions

'a prog = tdecl list × vdecl list × 'a fdefn list

3.4 Typing

Terms have to be well-typed, according to the typing relation presented in the following. Once we have defined this relation, we can make sure that certain run-time errors do not arise, by displaying an invariant that is maintained by all program runs. This invariant plays an essential role in the compiler correctness proof, since it allows to conclude, among others, that structures allocated on the heap do not change their type dynamically, so offsets for accessing substructures remain the same throughout the lifetime of these structures.

The type system is essentially defined by recursion on the structure of terms. Unfortunately, terms do not necessarily possess a unique type, because `NullPtr` may have several pointer types, therefore the typing relation does not immediately yield a type inference algorithm.

Type of a value Typing is defined relative to a program (which contains, among others, all the information about types of functions), to a local variable environment `lenv`, which maps variable names to types, and to a heap type, which maps locations to type names. The latter may seem surprising, as the type of elements in the heap is not known statically. In fact, we only need the heap type for our proof of type soundness. For static type checking, the heap type is assumed to be empty - see the typing rules for statements.

types
```
lenv   = vname ⇀ ty
heap_type = (loc ⇀ tname)
```

We first define the typing relation `ty_lit` for literal values. For lack of space, we can only show some representative rules:

consts
```
ty_lit :: (heap_type × val × ty) set
```

inductive `ty_lit` **intros**
```
TLBool: (hptp, Bool b, Boolean) ∈ ty_lit
TLNull: (hptp, Null, Ptr tn) ∈ ty_lit
TLAddr: hptp l = Some tn   ⟹ (hptp, Addr l, Ptr tn) ∈ ty_lit
```

Note that for typing purposes, the `hptp` will always be the empty map, so by rule `TLAddr`, we will not be able to type address values statically.

Typing Expressions and Statements We essentially have three typing judgements, for typing

- expressions, written as `P, E, hptp` $\vdash$ `e` $\surd$, which says that for program `P`, local variable environment `E` and heap type `hptp`, expression `e` is well-typed. Remember that `ty expr` expressions are attributed with a type.
- "left value expressions", written as `P, E, hptp` $\vdash_l$ `e` $\surd$, which says that expression `e` represents a valid left value.
- statements, written as `P, E`$\vdash$`c` $\surd$, which says that statement `c` is well-typed.

Typing of expressions is mostly standard. Typing of literal values is reduced to the typing relation `ty_lit` introduced above. Again a few representative rules:

inductive `ty_expr P E hptp` **intros**
```
WTLit:   (hptp, lv, T) ∈ ty_lit ⟹ P, E, hptp ⊢ Lit lv T √
```

```
WTDeref: ⟦P, E, hptp⊢ e√; expr_ty e = Ptr tn; tnenv P tn = Some T⟧
         ⟹ P, E, hptp⊢ (Deref e T) √
```

Valid left value expressions are a subset of well-typed expressions, namely either variable access, array access, struct access or dereference.

Also typing of statements is mostly standard. Note that in typing rules involving expressions or left values, we now use the *empty* heap type, which makes static type checking possible, such as in the following rule:

```
inductive wt_stmt intros
  WTAss: ⟦ P, E, empty ⊢l l √;
           P, E, empty ⊢ e √; expr_ty l = expr_ty e ⟧
  ⟹  P, E ⊢ (l::=e)√
```

We finish by characterising well-formedness of a program, *wf_prog*. In order to be well-formed, a program has to satisfy certain structural and some typing properties, in particular:

- Type declarations, global variable declarations and function definitions have to be unique, i.e. there must not be two declarations of the same category with the same name. Types, variables and functions have separate name spaces, so a global variable may have the same name as a function.
- All the types occurring in type and variable declarations are well-formed.
- All function definitions are well-formed; in particular, their bodies are well-typed statements.

4 Semantics

4.1 Defining the state

We can now define the program state, which will be used in the definition of the evaluation relation of Section 4.2. Essentially, the program state just consists of a heap (which is a map from locations to values) and a variable assignment (which is a map from variable names to values).

```
types heap  = loc ⇀ val
      vars = vname ⇀ val
record
  state  =
  heap_s :: heap
  vars_s :: vars
```

Some of the fine structure of the state is influenced by our decision to syntactically separate global and local variables. In previous attempts at defining the semantics, we split the *vars_s* component into a "global variable" and a "local variable" component. This led to a very cumbersome handling of simple

operations like looking up the current variable value, which had to take into account that global variables are shadowed by local variables. The current approach is much smoother because an issue of the "static semantics" is handled by preprocessing and not shifted into the dynamic semantics.

Manipulation of the variable component of the state plays a major role during function call: When the new function is called, the current actual parameters are assigned to the formal parameters of the called function. When the called function returns, the old variables are reinstated and the result value is copied back. The heap remains unaffected from these operations.

4.2 Evaluation

Evaluation of expressions and execution of statements can be cleanly separated because expressions are side-effect free. The execution relation is given in the style of a "big-step" semantics. Because expression evaluation always terminates, it is most convenient to define evaluation as a function, as done in the following. Strictly speaking, also an evaluation function has a big-step flavor, as it is not possible to "interrupt" evaluation at subexpressions.

A few remarks about the outcome of a computation are in order: Both evaluation and execution may fail, which is modeled with Isabelle's `option` type, where the result `None` indicates failure. This choice is appropriate as long as no more fine-grained analysis of the reason for a failure is required (such as "division by zero" or "array out of bounds"). Replacing `option` by a more appropriate type is not difficult.

Expression evaluation in functional style In the following, we essentially define three functions:

- Evaluation of expressions (resp. expression lists)
- Reduction of "left value expressions" to canonical form
- Update of a left value expression in canonical form with a value

The first (expression evaluation) reduces an expression to a value (or, more precisely to a `val option`, because evaluation may fail). Its definition, by primitive recursion over the expression, is standard. Possible causes for failure are array access out of bounds, access to an undefined component of a struct (which can be shown to be impossible in well-typed programs), any failure occurring during evaluation of operators, and null pointer dereference.

Traditionally, in an assignment `el ::= er`, the left value `el` is computed in analogy to the right value `er` and yields an address where the value obtained for `er` can be inserted. For assignments to complex left expressions, like `a[i] = er`, it then seems inevitable to engage into involved address arithmetic, like computing the size of the component type of `a` and multiplying it with `i` to obtain a displacement from the base address of `a`.

Our treatment of "left values" is motivated by our desire to work with an abstract model of composite values, i.e. to conceive a composite value as a collection of more elementary values which can be selected and modified structurally. Computation of an lvalue by `levalf` therefore does not produce an address, but is an expression in canonical form which displays the structure of the value in which an update has to be performed. The update itself is carried out by recursively traversing this structure and constructing the new value in bottom-up fashion. To convey an idea, we present the case of array access:

consts
```
  levalf :: [state, 'a expr] ⇒ 'a expr option
```
primrec
```
  levalf s (ArrAcc e i T) =
     (case (levalf s e) of
       None ⇒ None
     | Some ev ⇒
       (case (evalf (Some s) i) of
          None ⇒ None
        | Some iv ⇒ Some (ArrAcc ev (Lit iv (expr_ty i)) T)))
```

4.3 Execution of Statements

Execution of statements is defined by means of a transition relation `exec`, written as `P⊢s -c-> s'`, where `P` is a program (required for looking up function definitions), `s` is the start state, `c` the statement to be executed, and `s'` is the result state. To be precise, `s` and `s'` are of type `state option`, where `None` is an error condition that is propagated to the end of the program.

We only discuss the case of pointer allocation, for which we need, as auxiliary notion, an allocation function that is defined non-constructively. The only property about allocation that we are interested in is that, whenever `new_Addr` returns a defined location, this location is unused in the current heap.

constdefs
```
  new_Addr :: heap ⇒ loc option
  new_Addr h ≡ SOME a. (a = None) ∨ (∃ l. a = Some l ∧ h l = None)
```

Several refinements are conceivable here. For example, with a minor extension of the definition of `state`, a qualitative measure (e.g. "amount of storage still available") could be associated with the heap. By associating a size to types, the heap size could be modified appropriately at each instance of allocation.

Pointer allocation `PAlloc e tn` (corresponding to `e ::= new tn`) first looks for a new free address in the heap. Provided this is successful, a new state is created with a heap in which the new location is initialised with the default values of the type of the pointer. In this new state, the lvalue of `e` is computed and the new pointer assigned to it. We have to initialise the new location in order to achieve type soundness.

```
PAlloc:
    ⟦ new_Addr (heap_s s) = Some l;
       (tnenv P tn) = Some T;
       sa = s (|heap_s := (heap_s s)(l ↦ default_val T) |);
       so' = updatef sa (levalf sa e) (Some (Addr l)) ⟧
⟹ P⊢Some s -(PAlloc e tn)-> so'
```

5 Assembly Language

DLX [MP98] is a processor in the tradition of the MIPS processor families [PH98]. Its main purpose is academic and didactic, and so it is an order of magnitude less complex than modern commercial architectures, but it contains *in nuce* all essential aspects, such as pipelining, so we are confident that our development can be scaled up to more realistic processors.

DLX offers a simple, regular structure with 32 general purpose registers of 32 bits. In our context, we will also assume that we have a homogeneous main memory. In particular, we do not bother to model a memory hierarchy with caches etc. Apart from that, however, the question arises where to draw a borderline between numeric operations and bitstring manipulations. Should registers and memory contain numeric values or bitstrings?

We have opted for a mixed model: Registers contain integer values, because these are often incremented by numeric values, for example when calculating indices. Memory is assumed to consist of bytes, i.e. bitstrings, which is convenient to accommodate different addressing modes (word, half-word, ...). We therefore need conversion functions when transferring values from registers to memory and vice versa, but also for certain register operations, such as shift.

We can now define the state of the DLX machine, which essentially consists of three components: the register set `gprs`, a program counter `pcp` and main memory `mm`.

The execution relation `execR` is the transitive closure of the execution of individual instructions. Due to its regularity, the DLX instruction can be defined succinctly with the aid of a few functions interpreting, for example, arithmetic and comparison instructions.

6 Compiler Definition

Before defining the compilation function itself, we introduce some auxiliary notions which will be used as parameters of the compiler, in particular:

- *register maps*, describing register allocation
- *memory layout*, describing a partitioning of memory into different areas

This separation of concerns will not only make the compiler more readable; it will also allow more clever implementations to be used, such as a sophisticated

register allocation strategy. In fact, our compiler (and its correctness proof) only depends on abstract properties of the above-mentioned parameters and not on a particular implementation.

6.1 Register Maps

A register map defines which registers will receive the results of which subexpression. Note that the DLX machine only offers a limited number of registers, some of which are special purpose and may not be overwritten. If a register map that respects these constraints can be found, the compilation function can safely assign subexpression results to the corresponding registers. Otherwise, compilation fails, since our compiler does not handle register spilling.

For the purpose of defining register maps, expressions will be interpreted as trees whose nodes are identified by paths. By extending a path with numbers $0 \ldots n$, respectively, the n children of an expression can be addressed. A register map associates register names to paths:

types
```
treepath = nat list
regmap = treepath ⇒ regname
```

When traversing expression trees, we assume that certain registers are "used", which means that they hold values that will later be used and thus may not be modified. For a register map to be *in scope*, the set of registers currently in use has to be a superset of the set of reserved (special purpose) registers, the current path may not be among the used registers and the register selected by the map must not exceed the number of registers available.

constdefs
```
regmap_in_scope :: [treepath, regmap, regname set] ⇒ bool
regmap_in_scope pth rmp used ==
set reserved_regs ⊆ used ∧ (rmp pth) ∉ used ∧ (rmp pth) < regcount
```

We define a register map to be *valid* for an expression, if it is in scope for the expression root and the subexpressions are valid again. We just present the case for array access, the other term constructors are similar:

consts
```
valid_regmap :: [treepath, regmap, regname set, ty expr] ⇒ bool
```
primrec
```
valid_regmap pth rmp used (ArrAcc e i T) =
   (regmap_in_scope pth rmp used ∧
    valid_regmap (0#pth) rmp used e ∧
    valid_regmap (1#pth) rmp (insert (rmp (0#pth)) used) i)
```

Note that we traverse expression trees from left to right, and that the registers assigned to siblings at the left of a node are marked as "used", so they may not be used again in the current tree. As we will see later, this also fixes the order in which subexpressions are evaluated by the code that the compiler emits. A

more flexible strategy is conceivable, but comes at the price of considerably complicating the definition and correctness proof of the compiler.

With the expression compilation function `comp_expr` to be defined later, we can prove the following theorem which captures the essence of `valid_regmap`: When executing the code resulting from compiling an expression `e`, thereby passing from a DLX state `sdlx` to `sdlx'`, we know that the set of registers `used` remains unaltered, provided the register map `rmp` is valid for this `used` set.

```
⟦ execR (comp_expr pth rmp lo fn rv e) sdlx sdlx';
   valid_regmap pth rmp used e ⟧
 ⟹ invariant_used_regs used sdlx sdlx'
```

6.2 Memory Layout

Remember that the DLX machine assumes that there is a single, homogeneous memory. However, for executing compiled code, memory is traditionally partitioned into several areas, such as a call stack, a heap for dynamic data and a global variable segment. To a certain extent, we can determine in advance the dimensions of these partitions, or at least parts thereof. Thus, we know the number of global variables and their types, so we can precompute the size of the global variable segment. In a similar vein, we can determine the size of each stack frame. All this information will be collected in a structure we call *memory layout*, and it is the second major parameter of our compilation function.

We define `layout` as

record

```
  layout =
  addr_space :: addr interval          — entire address space
  globs_area :: addr interval          — global variables
  stack_area :: addr interval
  heap_area  :: addr interval
  gvmp       :: vname ⇒ addr           — global variable map
  lresvoff   :: fname ⇒ vname ⇒ int    — local and result variable offset
  frame_size :: fname ⇒ nat            — maximal size of function frame
```

Here, an interval is just a pair defining a lower bound (inclusive) and an upper bound (exclusive).

We define a predicate, `valid_layout_prog`, that captures the validity of a layout with respect to the declarations of a program, and which is a conjunction of several predicates, such as `valid_layout_areas`:

constdefs

```
  addr_space_ub :: int
  addr_space_ub == 2^(wlen_bit - 1)
```

```
addr_space_lb :: int
addr_space_lb == - (2^(wlen_bit - 1))

valid_layout_areas :: layout ⇒ bool
valid_layout_areas lo ==
 (¬ Null_addr  ∈I (addr_space lo)) ∧
 (Null_addr  ∈I (addr_space_lb, addr_space_ub)) ∧
 (interv_contained (addr_space lo) (addr_space_lb, addr_space_ub)) ∧
 (intervs_contained [globs_area lo, stack_area lo, heap_area lo]
                    (addr_space lo)) ∧
 (intervs_disjoint [globs_area lo, stack_area lo, heap_area lo])
```

which says that the address of the null pointer is not contained in the address space, but that the null pointer address is nevertheless representable by words of length `wlen_bit`. The same holds for the address space itself. The partitions for global variable area, stack area and heap are contained in the address space, and they are disjoint.

Apart from the sizes of the memory areas, it is possible to determine in advance the addresses of global variables (function `gvmp`) and the offset of variables in function frames (`lresvoff`). In this case, we have similar requirements concerning disjointness of areas and containment in the proper interval as sketched above. The memory allocated for an element is computed, depending on its type, by a function `asize`.

6.3 Compilation Function

We now have the main ingredients to define the compilation function itself:

```
comp_expr :: [treepath, regmap, layout, fname, bool, ty expr]
              ⇒ instr list
```

The parameters are

- a path indicating in which subexpression of the original expression we are,
- a register map which tells us where to store the result of the current subexpression,
- the memory layout,
- the name of the function in which the expression to be compiled resides (necessary for computing offsets)
- a boolean indicating whether to compile the expression as right value (RV, so that we obtain the value itself) or as left value (LV, so that we obtain a pointer to the value)
- finally, the expression to be compiled

The function returns a list of assembly instructions. Take as example the case of binary expressions:

```
comp_expr pth rmp lo fn rv (BinOp bop e0 e1 T) =
    (comp_expr (0#pth) rmp lo fn RV e0) @
```

```
(comp_expr (1#pth) rmp lo fn RV e1) @
(comp_binop bop (rmp pth) (rmp (0#pth)) (rmp (1#pth)))
```

The function recursively traverses the expression to be compiled, first generating the code for subexpressions, then combining the partial results, for example by adding them if the expression is an addition operation. The registers holding the partial results are determined by the register map: If, for example, `pth` is the current expression, then its first subexpression addressed by `0#pth` will be stored in `rmp (0#pth)`.

7 Compiler Correctness

Roughly speaking, our correctness theorem states that execution of the assembly code obtained from compiling an expression yields a result that is "comparable" to the result obtained when evaluating the expression in C0, provided both executions start in "corresponding" states.

Before getting into details, we have to note that compilation from C0 to DLX involves a non-trivial state refinement, so we first have to make more precise what we mean by "corresponding states". We do this by defining a predicate `refines_state`, which says that (for a given program `P`) a DLX state refines a C0 state if the global variable area of the DLX state refines the global variables of `P`, similarly for the heap, and if the current stack frame refines the local variables of the C0 state.

For lack of space, we only look at the global variable part. We define:

constdefs

```
refines_vars :: [vars, dlx_state, lenv, vname ⇒ addr, locmap] ⇒ bool
refines_vars vmap sdlx E advmap lmp ==
∀ vn ∈ (dom vmap).
 (let val = the (vmap vn);
      ad = (advmap vn);
      T = the (E vn)
  in mem_read_asc (mm sdlx) ad (asize T) = (alloc lmp val))

refines_globs :: [vars, dlx_state, lenv, layout, locmap] ⇒ bool
refines_globs gvmap sdlx E lo lmp  ==
   refines_vars gvmap sdlx E (gvmp lo) lmp
```

Roughly, this definition says that the global variable component `gvmap` of the C0 state (sending variable names to values) is well represented in DLX state `sdlx` if for all variables `vn` in the domain of `gvmap`, reading the memory of `sdlx` at the address assigned to `vn` by the variable layout yields the same list of bytes as the value of `vn` under `gvmap`. Here, the function `alloc` gives the byte representation of a value `val`.

With these definitions, we can now proceed to discussing the correctness theorem for expression compilation. Expressed verbally, it says:

– Provided the program under consideration is well-formed and the expression `e` to be compiled is well-typed;
– provided compilation yields a list of instructions *instrs* which make the DLX machine change from state *sdlx* to *sdlx'*;
– provided the register map and the memory layout are valid
– provided *sdlx* is the refinement of a C0 state *ssrc*

then, if we evaluate `e` and obtain a definite value `v` (i.e., no error),

– either we have compiled a right value `v` which, provided the type of the expression is simple, is stored in the correct register, as predicted by the register map
– or we have compiled a left value. In this case, evaluation of the expression as left value must yield a left expression *le* which identifies an address (*addr_lv*) from which the byte pattern corresponding to value `v` can be read.

Since we do not allow function calls in expressions and expression evaluation therefore leads to progressively smaller terms, we can carry out a proof by induction over `e`. The proof is complex, on the one hand because many invariants have to be established for the subterms for the induction hypothesis to be applicable, but also because a lot of low-level address reasoning has to be carried out, which seems hard to mechanise.

8 Conclusions

Our work stands in a long tradition of language semantics and compiler correctness proofs. Starting in the 1970s [MP67], increasingly complex languages have been investigated. A very large development, however without machine support, is the Scheme compiler described in [GMR+92]. The Pascal compiler formalization reported in [SCSW97,Ste98], using the Z language, is interesting because of its industrial context.

Since full compiler verification for a general-purpose programming language is a complex and time-consuming task (approx. 4 months for the development described here), several authors have considered translation validation in which the correctness of translation of individual programs is show (as opposed to the correctness of the compiler itself). They mostly consider restricted languages, such as data flow languages which are essentially loop-free [PSS00], or have concentrated on specific phases such as optimization [Nec00,ZPF+02].

A project which is the scientific forerunner of the *Verisoft* project, but on a smaller dimension, has been carried out using the ACL2 theorem prover in the late 1980s. The compiler [You89] translates from a simple procedural language to a high-level assembler. Here, and also in later work, using procedural [Cur93] or quasi-functional source languages like Lisp [DV01] in the *Verifix* project [Verb], the memory model is restricted to simply typed or array variables, which is a significant simplification as compared to the dynamic heap considered above.

More recently, a more complex memory organisation has been taken into account. The Isabelle formalisations in [Ohe01,Kle03] give a rather comprehensive account of the Java source resp. bytecode language. Based on them, a compiler has been developed [Str02], whose proof offers some difficulties (because of object orientation), but which is altogether simpler than the C0 to DLX compiler because Java bytecode is a very high-level assembler, featuring instructions such as field access (no address calculation required) and method invocation (no call stack management necessary).

In [Nor98], a very detailed and faithful account of the C language is given. Among others, explicit address calculation is possible, due to a notion of "size" of elements of a type. Furthermore, the intricacies of control flow in C are studied in depth. This work is remarkable in its precision, but shows the difficulties of reasoning about programs at this level of detail. For the reasons laid out in Section 1, we have refrained from following this path.

What remains to be done? On the one hand, the proof has to be finished, either based on the big-step semantics presented in this paper, or based on a small-step semantics, as suggested in [LP04]. Furthermore, we are well aware that our implementation approach is "naive" in the sense that modern compilers use more refined intermediate representations, such as SSA graphs. A first lead has been taken by the formalization of these structures in [BG04]; the *Concert* project [Con] aims at the formal study of optimisation in compilers. We hope to be able to integrate these approaches in our general framework.

Acknowledgements

The language C0 described here has been developed in close collaboration with Norbert Schirmer, to whom I am also grateful for critical comments on the compiler. The C0 language definition has benefited from discussions with Tobias Nipkow, Dirk Leinenbach, Wolfgang Paul and Elena Petrova, and the formalization of the DLX assembler from suggestions by Sven Beyer, Mark Hillebrand and Tom in der Rieden.

References

BBJ+02. C. Berg, S. Beyer, C. Jacobi, D. Kröning, and D. Leinenbach. Formal verification of the VAMP microprocessor (project status). In *Symposium on the Effectiveness of Logic in Computer Science (ELICS02)*, 2002.

BG04. Jan Olaf Blech and Sabine Glesner. A formal correctness proof for code generation from SSA form in Isabelle/HOL. In *Proceedings 3. Arbeitstagung Programmiersprachen (ATPS), 34. Jt. GI*, 2004.

Con. http://www-sop.inria.fr/lemme/concert.

Cur93. Paul Curzon. A verified Vista implementation. Technical Report 311, University of Cambridge, Computer Laboratory, September 1993. Available from http://www.cl.cam.ac.uk/Research/HVG/vista/.

DV01. A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In *Proc. FSTTCS 2001*, December 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, September 1992.

Kle03. Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

LP04. Dirk Leinenbach and Wolfgang Paul. Translation from C0 to DLX. Technical report, Verisoft Internal Report, 2004.

MP67. John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In J. T. Schwartz, editor, *Proceedings Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, RI, 1967.

MP98. Silvia Mueller and Wolfgang Paul. *Computer Architecture*. Springer, 1998.

Nec00. George Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, 2000.

Nor98. Michael Norrish. *C formalized in HOL*. PhD thesis, Cambridge University, December 1998.

NPW02. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

Ohe01. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. http://www4.in.tum.de/~oheimb/diss/.

Ort04. Veronika Ortner. Verification of BDD Algorithms. Master's thesis, Technische Universität München, 2004. http://www.veronika.langlotz.info.

PH98. David A. Patterson and John L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann, second edition, 1998.

PSS00. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation: From Signal to C. In *Correct System Design - Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 231–255. Springer Verlag, 2000.

Sch04. Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In Gerwin Klein, editor, *Proc. NICTA Workshop on OS Verification 2004*, 2004. ID: 0400002T.1, to appear, available from http://www4.in.tum.de/~schirmer.

SCSW97. David Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In *FME'97: Formal Methods: Their Industrial Application and Strengthened Foundations*, Lecture Notes in Computer Science, 1997.

Ste98. Susan Stepney. Incremental development of a high integrity compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98)*, November 1998.

Str02. Martin Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2002.

Vera. http://www.verisoft.de/StartPage.html.

Verb. http://www.verifix.de/.

You89. William D. Young. A mechanically verified code generator. Technical Report 37, Computational Logic Inc., January 1989.

ZPF+02. L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. In *2nd Workshop on Runtime Verification*, volume 70 of *ENTCS*, pages 180–201, 2002.