CerCo

INFORMATION AND COMMUNICATION TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D3.1
Executable Formal Semantics of C

Version 1.0

Main Authors:
Brian Campbell, Randy Pollack

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Abstract We present an execution semantics of the C programming language for use in the CerCo project. It is based on the small-step inductive semantics used by the CompCert verified compiler. We discuss the extensions required for our target architecture, porting the semantics to our choice of tool, Matita, providing an equivalent executable semantics and the validation of the semantics.

Contents

1 Introduction

We present an executable formal semantics of the C programming language which will serve as the specification of the input language for the CerCo verified compiler. Our semantics is based on Leroy et. al.'s C semantics for the CompCert project [2, 3], which divides the treatment of C into two pieces. The first is an OCaml stage which parses and elaborates C into an abstract syntax tree for the simpler Clight language, based on the CIL C parser. The second part is a small step semantics for Clight formalised in the proof tool, which we have ported from Coq to the Matita theorem prover. This semantics is given in the form of inductive definitions, and so we have added a third part giving an equivalent functional presentation in Matita.

The CerCo compiler needs to deal with the constrained memory model of the target microcontroller (in our case, the 8051). Thus each part of the semantics has been extended to allow explicit handling of the microcontroller's memory spaces. *Cost labels* have also been added to the Clight semantics to support the labelling approach to cost annotations presented in a previous deliverable [1].

The following section discusses the C language extensions for memory spaces. Then the port of the two stages of the CompCert Clight semantics is described in Section 3, followed by the new executable semantics in Section 4. Finally we discuss how the semantics has been validated in Section 5.

2 Language extensions for the 8051 memory model

The choice of an extended 8051 target for the CerCo compiler imposes an irregular memory model with tight resource constraints. The different memory spaces and access modes are summarised in Figure 1 — essentially the evolution of the 8051 family has fragmented memory into four regions: one half of the 'internal' memory is fully accessible but also contains the register banks, the second half cannot be accessed by direct addressing because it is shadowed

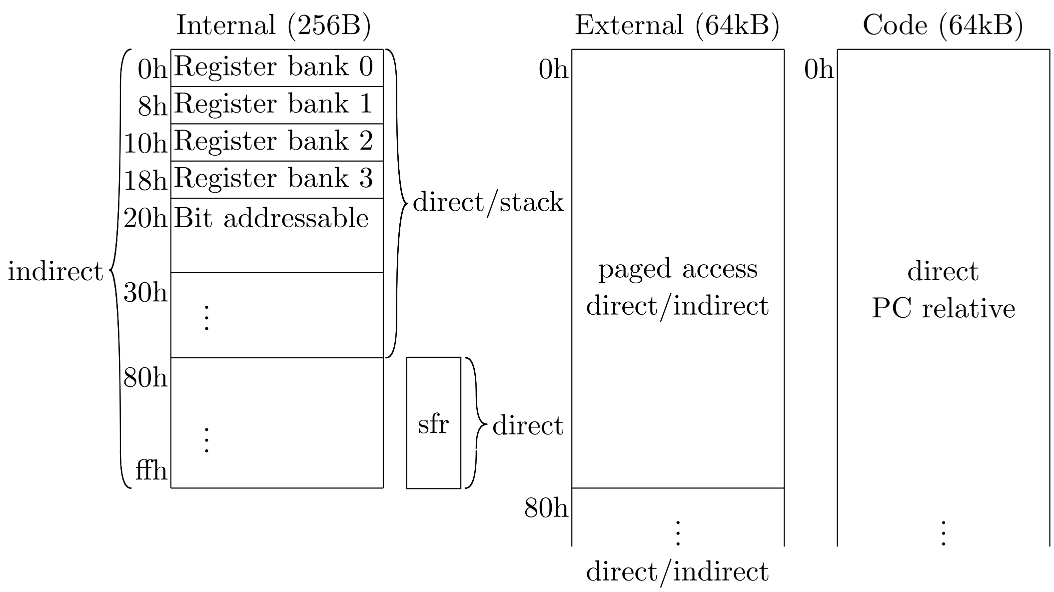

Figure 1: The extended 8051 memory model

by the 'Special Function Registers' (SFRs) for I/O; 'external memory' provides the bulk of memory in a separate address space; and the code is in its own read-only space.

To make efficient use of the limited amount of memory, compilers for 8051 microcontrollers provide extra keywords to allocate global variables to particular memory spaces, and to limit pointers to address a particular space. The freely available `sdcc` compiler provides the following extensions for the 8051 memory spaces:

Attribute	Pointer size (bytes)	Memory space
`__data`	1	Internal, first half (0h – 7fh)
`__idata`	1	Internal, indirect only (80h – ffh)
`__pdata`	1	External, page access (usually 0h – 7fh)
`__xdata`	2	External, any (0h – ffffh)
`__code`	2	Code, any (0h – ffffh)
none	3	Any / Generic pointer

The generic pointers are a tagged union of the other kinds of pointers.

We intend the CerCo compiler to support extensions that are broadly compatible with `sdcc` to enable the compilation of programs with either tool. In particular, this would allow the comparison of the behaviour of test cases compiled with each compiler. Thus the C syntax and semantics have been extended with the memory space attributes listed above. The syntax follows `sdcc` and in the semantics we track the memory space that each block was allocated from and only permit access via the appropriate kinds of pointers. The details on these changes are given in the following sections.

The `sdcc` compiler also supports special variable types for accessing the SFRs, which provide the standard I/O mechanism for the 8051 family. (Note that pointers to these types are not permitted because only direct addressing of the SFRs is allowed.) We intend to use CompCert-style 'external functions' instead of special types. These are functions which are declared, but no C implementation of them is provided. Instead they are provided by the runtime or compiled directly to the corresponding machine code. This has the advantage that no changes from the CompCert semantics are required, and a compatibility library can be provided for `sdcc` if necessary. The 8051 and the `sdcc` compiler also provide bit-level access to a small region of internal memory. We do not intend to expose this feature to C programs in the CerCo compiler, and so no extension is provided for it.

Finally, we have the option of using CompCert's translation of `volatile` variable accesses to 'external' function calls. Should we need more flexible I/O than SFRs provide, then we could adopt the `sdcc` extension to allocate a variable at a particular address to provide a way to deal with memory mapped I/O in the external memory space. The translation to function calls would mean that the semantics presented here would be unaffected.

3 Port of CompCert Clight semantics to Matita

3.1 Parsing and elaboration

The first stage taken from the CompCert semantics is the parsing and elaboration of C programs into the simpler Clight language. This is based upon the CIL library for parsing, analysing and transforming C programs by Necula et. al. [5]. The elaboration provides explicit type information throughout the program, including extra casts for promotion. It also performs simplifications such as breaking up expressions with side effects into effect-free expressions along with statements to perform the effects. The transformed Clight programs are much more manageable and lack the ambiguities of C, but also remain easily understood by C programmers.

The parser has been extended with the 8051 memory spaces attributes given above. The resulting abstract syntax tree records them on global variable declarations and pointer types. However, we also need to deal with them during the elaboration process to produce all of the required type information. For example, when the address-of operator `&` is used it must decide which kind of pointer should be used. Thus the extended elaboration process keeps track of the memory space (if any) that the value of each expression resides in. Where the memory space is not known, a generic pointer will be used instead. Moreover, we also include the pointer kind when determining whether a cast must be inserted so that conversions between pointer representations can be performed.

Thus the elaboration turns the following C code

```c
int g(int *x) { return 5; }

int f(__data int *x, int *y) {
  return x==y ? g(x) : *x;
}

__data int i = 1;

int main(void) {
  return f(&i, &i);
}
```

into the Clight program below:

```
int g(int *x) { return 5; }

int f(__data int * x, int * y)
{
  int t;
  if (x == (__data int * )y) {
    t = g((int * )x);
  } else {
    t = *x;
  }
  return t;
}

int main(void)
{
  int t;
  t = f(&i, (int * )(&i));
  return t;
}
```

The expression in `f` had to be broken up due to the call to `g`, and casts have been added to change between generic pointers and pointers specific to the `__data` section of memory. The underlying data structure also has types attached to every expression, but these are impractical to show in source form.

Note that the translation from C to Clight is not proven correct — instead it effectively forms a semi-formal part of the whole C semantics. We can have some confidence in the code, however, because it has received testing in the CerCo prototype, and it is very close to the version used in CompCert. We can also perform testing of the semantics without involving the rest of the compiler because we have an executable semantics. Moreover, the cautious programmer could choose to inspect the generated Clight code, or even work entirely in the Clight language.

3.2 Small-step inductive semantics

The semantics for Clight itself has been ported from the Coq development used in CompCert to Matita for use in CerCo. Details about the original big-step formalisation of Clight can be found in Leroy and Blazy [3] (including a discussion of the translation from C in §4.1), although we started from a later version with a small-step semantics and hence support for `goto` statements. Several parts of the semantics were shared with other parts of the CompCert development, notably:

- the representation of primitive values (integers, pointers and undefined values, but not structures or unions) and operations on them,
- traces of I/O events,
- a memory model that keeps conceptually distinct sections of memory strictly separate (assigning 'undefined behaviour' to a buffer overflow, for instance),

- results about composing execution steps of arbitrary small-step semantics,
- data structures for local and global environments, and
- common error handling constructs, in particular an error monad.

We anticipate a similar arrangement for the CerCo verified compiler, although this means that there may be further changes to the common parts of the semantics later in the project to harmonise the stages of the compiler. In particular, some of data structures for environments are just preliminary definitions for developing the semantics.

The main body of the small-step semantics is a number of inductive definitions giving details of the defined behaviour for casts, expressions and statements. Expressions are side-effect free in Clight and only produce a value as output. In our case we also need a trace of any cost labels that are 'evaluated' so that we will be able to give fine-grained costs for the execution of compiled conditional expressions.

As an example of one of the expression rules, consider an expression which evaluates a variable, `Expr (Evar id) ty`. A variable is an example of a class of expressions called *lvalues*, which are roughly those expressions which can be assigned to. Thus we use a general rule for lvalues,

```
ninductive eval_expr (ge:genv) (e:env) (m:mem) : expr → val → trace → Prop :=
...
  | eval_Elvalue: ∀a,ty,psp,loc,ofs,v,tr.
      eval_lvalue ge e m (Expr a ty) psp loc ofs tr →
      load_value_of_type ty m psp loc ofs = Some ? v →
      eval_expr ge e m (Expr a ty) v tr
```

where the auxiliary relation `eval_lvalue` yields the location of the value, `psp,loc,ofs`, consisting of memory space, memory block, and offset into the block, respectively. The expression can thus evaluate to the value `v` if `v` can be loaded from that location. One corresponding part of the `eval_lvalue` definition is

```
...
with eval_lvalue (*(ge:genv) (e:env) (m:mem)*) :
        expr → memory_space → block → int → trace → Prop :=
  | eval_Evar_local: ∀id,l,ty.
      get ??? id e = Some ? l →
      eval_lvalue ge e m (Expr (Evar id) ty) Any l zero E0
...
```

simply looks up the variable in the local environment. The offset is zero because all variables are given their own memory block to prevent the use of stray pointers. A similar rule handles global variables, with an extra check to ensure that no local variable has the same name. Note that the two relations are defined using mutual recursion because `eval_lvalue` uses `eval_expr` for the evaluation of the pointer expression in the dereferencing rule.

Casts also have an auxiliary relation to specify the allowed changes, and operations on values (including the changes in representation performed by casting) are given as functions.

The only new expression in our semantics is the cost label which wraps around another expression. It does not change the result, but merely augments the trace with the given label to identify the branches taken in conditional expressions so that accurate cost information can be attached to the program:

```
  | eval_Ecost: ∀a,ty,v,l,tr.
      eval_expr ge e m a v tr →
      eval_expr ge e m (Expr (Ecost l a) ty) v (tr++Echarge l)
```

As the expressions are side-effect free, all of the changes to the state are performed by statements. The state itself is represented by records of the form

```
ninductive state: Type :=
  | State:
      ∀f: function.
      ∀s: statement.
      ∀k: cont.
      ∀e: env.
      ∀m: mem. state
  | Callstate:
      ∀fd: fundef.
      ∀args: list val.
      ∀k: cont.
      ∀m: mem. state
  | Returnstate:
      ∀res: val.
      ∀k: cont.
      ∀m: mem. state.
```

During normal execution the state contains the currently executing function's definition (used to find `goto` labels and also to check whether the function is expected to return a value), the statement to be executed next, a continuation value to be executed afterwards (where successor statements and details of function calls and loops are stored), the local environment mapping variables to memory locations[1] and the current memory state. The function call and return states appear to store less information because the details of the caller are contained in the continuation.

An example of the statement execution rules is the assignment rule (corresponding to the C syntax `a1 = a2`),

```
ninductive step (ge:genv) : state → trace → state → Prop :=
  | step_assign: ∀f,a1,a2,k,e,m,psp,loc,ofs,v2,m',tr1,tr2.
      eval_lvalue ge e m a1 psp loc ofs tr1 →
      eval_expr ge e m a2 v2 tr2 →
      store_value_of_type (typeof a1) m psp loc ofs v2 = Some ? m' →
      step ge (State f (Sassign a1 a2) k e m)
           (tr1++tr2) (State f Sskip k e m')
...
```

which can be read as:

- if `a1` can evaluate to the location `psp,loc,ofs`,
- `a2` can evaluate to a value `v2`, and

[1]In the semantics all variables are allocated, although the compiler may subsequently allocate them to registers where possible.

- storing `v2` at location `psp,loc,ofs` succeeds, yielding the new memory state `m'`, then
- the program can step from the state about to execute `Sassign a1 a2` to a state with the updated memory `m'` about to execute the no-op `Sskip`.

This rule would be followed by one of the rules to which replaces the `Sskip` statement with the 'real' next statement constructed from the continuation `k`. Note that the only true side-effect here is the change in memory — the local environment is initialised once and for all on function entry, and the only events appearing in the trace are cost labels used purely for accounting. At present this imposes an ordering due to the cost labels. Should this prove too restrictive we may change it to produce a set of labels encountered.

The Clight language provides input and output effects through 'external' functions and the step rule

```
| step_external_function: ∀id,targs,tres,vargs,k,m,vres,t.
   event_match (external_function id targs tres) vargs t vres →
   step ge (Callstate (External id targs tres) vargs k m)
        t (Returnstate vres k m)
```

which allows the function to be invoked with and return any values subject to the enforcement of the typing rules in `event_match`, which also provides the trace.

Cost label statements prefix the trace with the given label, similar to the cost label expressions above.

4 Executable semantics

We have added an equivalent functional version of the Clight semantics that can be used to animate programs. The definitions roughly follow the inductive semantics, but are necessarily rearranged around pattern matching of the relevant parts of the state rather than presenting each case separately.

4.1 Expressions

The code corresponding to the variable lookup definitions on page 7 is

```
nlet rec exec_expr (ge:genv) (en:env) (m:mem) (e:expr) on e : res (val×trace) :=
match e with
[ Expr e' ty ⇒
  match e' with
  [ ...
  | Evar _ ⇒
      do ⟨l,tr⟩ ← exec_lvalue' ge en m e' ty;
      do v ← opt_to_res ? (load_value_of_type' ty m l);
      OK ? ⟨v,tr⟩
...
  ]
]
and exec_lvalue' (ge:genv) (en:env) (m:mem) (e':expr_descr) (ty:type) on e'
 : res (memory_space × block × int × trace) :=
  match e' with
```

```
    [ Evar id ⇒
        match (get ...id en) with
        [ None ⇒ do ⟨sp,l⟩ ← opt_to_res ? (find_symbol ? ? ge id);
                    OK ? ⟨⟨⟨sp,l⟩,zero⟩,E0⟩ (* global *)
        | Some loc ⇒ OK ? ⟨⟨⟨Any,loc⟩,zero⟩,E0⟩ (* local *)
        ]
  ...
```

where the result is placed in an error monad (the `res` type constructor) so that *undefined behaviour* such as dereferencing an invalid pointer can be rejected. We use `do` notation similar to Haskell and CompCert, where

```
  do x ← e; e'
```

means evaluate `e` and if it yields a value then bind that to `x` and evaluate `e'`, and otherwise propogate the error.

4.2 Statements

Evaluating a step of a statement is complicated by the presence of the 'external' functions for I/O, which can return arbitrary values. These are handled by a resumption monad, which on encountering some I/O returns a suspension. When the suspension is applied to a value the evaluation of the semantics is resumed. Resumption monads are a standard tool for providing denotational semantics for input [4] and interleaved concurrency [6, Chapter 12]. The definition also incorporates errors, and uses a coercion to automatically transform values from the plain error monad.

The definition of the monad is:

```
ninductive IO (output:Type) (input:output → Type) (T:Type) : Type :=
| Interact : ∀o:output. (input o → IO output input T) → IO output input T
| Value : T → IO output input T
| Wrong : IO output input T.

nlet rec bindIO (O:Type) (I:O → Type) (T,T':Type)
                (v:IO O I T) (f:T → IO O I T') on v : IO O I T' :=
match v with
[ Interact out k ⇒ (Interact ??? out (λres. bindIO O I T T' (k res) f))
| Value v' ⇒ (f v')
| Wrong ⇒ Wrong O I T'
].
```

Note that the type of the input value is dependent on the output value. This enables us to ensure that the input is always well-typed. An alternative approach is a check in the semantics, but this causes programs to fail in a way that has no counterpart in the inductive semantics.

The execution of assignments is straightforward,

```
nlet rec exec_step (ge:genv) (st:state) on st : (IO io_out io_in (trace × state)) :=
match st with
[ State f s k e m ⇒
  match s with
  [ Sassign a1 a2 ⇒
```

```
    ! ⟨l,tr1⟩ ← exec_lvalue ge e m a1;
    ! ⟨v2,tr2⟩ ← exec_expr ge e m a2;
    ! m' ← store_value_of_type' (typeof a1) m l v2;
    ret ? ⟨tr1++tr2, State f Sskip k e m'⟩
...
```

where `!` is used in place of `do` due to the change in monad. The content is essentially the same as the inductive rule given on page 8.

Most other rules are similar translations of the inductive semantics. The handling of external calls uses the

```
do_io : ident → list eventval → IO eventval io_out eventval
```

function to suspend execution:

```
...
| Callstate f0 vargs k m ⇒
  match f0 with
  [ ...
  | External f argtys retty ⇒
      ! evargs ← err_to_io_sig ...(check_eventval_list vargs (typlist_of_typelist argtys));
      ! evres ← do_io f evargs (proj_sig_res (signature_of_type argtys retty));
      ret ? ⟨Eextcall f evargs (mk_eventval ? evres), Returnstate (mk_val ? evres) k m⟩
  ]
...
```

The rest of the code after `do_io` is included in the suspension returned.

Together with functions to provide the initial state for a program and to detect a final state we can write a function to run the program up to a given number of steps. Similarly, a corecursive function can return the entire execution as a stream of trace and state pairs.

5 Validation

We have used two methods to validate our executable semantics: we have proven them equivalent to the inductive semantics of Section 3.2, and we have animated small examples of key areas.

5.1 Equivalence to inductive semantics

To show that the executable semantics are sound with respect to the inductive semantics we need to prove that any value produced by each function satisfies the corresponding relation, modulo errors and resumption. To deal with these monads we lift the properties required. In particular, for the resumption monad we ignore error values, require the property when a value is produced, and quantify over any interaction with the outside world:

```
nlet rec P_io O I (A:Type) (P:A → Prop) (v:IO O I A) on v : Prop :=
match v return λ_.Prop with
[ Wrong ⇒ True
| Value z ⇒ P z
| Interact out k ⇒ ∀v'.P_io O I A P (k v')
].
```

We can use this lifting with the relations from the inductive semantics to state soundness properties:

```
ntheorem exec_step_sound: ∀ge,st.
  P_io ??? (λr. step ge st (\fst r) (\snd r)) (exec_step ge st).
```

The proofs of these theorems use case analysis over the state, a few lemmas to break up the expressions in the monad and the other soundness results to form the corresponding derivation in the inductive semantics.

We experimented with a different way of specifying soundness using dependent types:

```
nlet rec exec_step (ge:genv) (st:state) on st
 : (IO eventval io_out (Σr:trace × state. step ge st (\fst r) (\snd r))) :=
```

Note the Σ type for the result of the function, which shows that successful executions are sound with respect to the inductive semantics. Matita automatically generates proof obligations for each case due to a coercion between the types

`option (res T)` and `res (Σx:T. P x)`

(where a branch marked `None` would generate a proof obligation to show that it is impossible, although the semantics do not use this feature). This is intended to mimic Sozeau's Russell language and elaboration into Coq [7]. The coercion also triggers an automatic mechanism in Matita to add equalities for each pattern matched. The proofs are essentially the same as before.

However, the soundness proofs then pervade the executable semantics, making rewriting in the correctness proofs more difficult. We decided to keep the soundness results separate, partly because of the increased difficulty of using the resulting terms in proofs, and partly because they are of little consequence once equivalence has been shown.

The completeness results requiring a dual lifting which requires the term to reduce to a particular value, allowing for resumptions with existential quantification:

```
nlet rec yieldsIO (A:Type) (a:IO io_out io_in A) (v':A) on a : Prop :=
match a with
[ Value v ⇒ v' = v
| Interact _ k ⇒ ∃r.yieldsIO A (k r) v'
| _ ⇒ False
].
```

We then show the completeness theorems, such as

```
ntheorem step_complete: ∀ge,s,tr,s'.
  step ge s tr s' → yieldsIO ? (exec_step ge s) ⟨tr,s'⟩.
```

by case analysis on the inductive derivation and a mixture of reduction and rewriting. Thus we know that executing a step in these semantics is equivalent to a step in the inductive semantics.

Showing the equivalence of whole program execution is a little trickier. Our executable semantics produces a coinductive `execution` which is really a tree of executions, branching at each I/O resumption on the input value:

$$
\begin{array}{lcl}
 & & \mathsf{e\_step}\ t_i\ s_i \to\ \cdots\ \mathsf{e\_stop}\ t_j\ i\ m \\
 & \nearrow & \qquad \vdots \\
\mathsf{e\_step}\ \mathsf{E0}\ s_0 \to \mathsf{e\_step}\ t_1\ s_1 \to \cdots \mathsf{e\_interact}\ o_1\ k_1 & \to & \mathsf{e\_step}\ t_i'\ s_i' \to\ \cdots\ \mathsf{e\_step}\ t_j\ s_j \to \cdots \\
 & \searrow & \qquad \vdots \\
 & & \mathsf{e\_step}\ t_i''\ s_i'' \to \cdots\ \mathsf{e\_wrong}
\end{array}
$$

Each `e_step` comes with the trace (often the empty trace, E0) and current state. Each branch corresponds to calling the continuation k_1 with a different input value. We use the `single_exec_of` predicate to identify single executions from these trees, essentially fixing a stream of input values.

However, the inductive semantics divides program behaviours into four categories which have *individual* (co)inductive descriptions:

- successfully terminating executions;
- programs which eventually diverge (with an empty trace);
- programs which keep interacting in some way (with an infinite trace)[2]; and
- programs which *go wrong*.

We cannot constructively decide which of these categories an execution can fit into because the properties they describe are undecidable. Hence we follow CompCert's approach for showing that one of the behaviours always exists using classical logic. Thus we characterise the executions, then show the existence of the inductive semantics' behaviour that matches. We limit the scope of classical reasoning by taking the relevant axioms as hypotheses:

```
ntheorem exec_inf_equivalence:
  ∀classic:(∀P:Prop.P ∨ ¬P).
  ∀constructive_indefinite_description:(∀A:Type. ∀P:A→Prop. (∃x. P x) → Σx:A. P x).
  ∀p,e. single_exec_of (exec_inf p) e →
   ∃b.execution_matches_behavior e b ∧ exec_program p b.
```

5.2 Animation of simple C programs

We are currently working with a development version of Matita which (temporarily) does not support extraction to OCaml code. Hence to animate a program we first parse it with CIL and produce a Matita term in text format representing the program, then interpret it within Matita.

This process is rather laborious, so we have concentrated on testing small programs which exercised areas of the semantics which depart from CompCert. In particular, we tested several aspects of the handling of memory spaces and the casting of pointers. Together with the results in the previous sections we gain considerable confidence that the semantics describe

[2]In our setting this includes passing through cost labels as well as I/O.

the behaviour of programs properly. Nevertheless, we intend to experiment with larger C programs once extraction is available.

To give a concrete example, the following C program reads an integer using an 'external' function and returns its factorial as the program's exit value:

```
int get_input(void);

int main(void) {
  int i = get_input();
  int r = 1;
  int j;

  for (j = 2; j<=i; j++)
    r = r * j;

  return r;
}
```

The Clight code is essentially the same, and the Matita term is:

```
ndefinition myprog := mk_program fundef type
 [mk_pair ?? (succ_pos_of_nat 132 (* get_input *))
            (External (succ_pos_of_nat 132) Tnil (Tint I32 Signed));
  mk_pair ?? (succ_pos_of_nat 133 (* main *)) (Internal (
    mk_function (Tint I32 Signed ) [] [mk_pair ?? (succ_pos_of_nat 134) (Tint I32 Signed);
                                   mk_pair ?? (succ_pos_of_nat 135) (Tint I32 Signed);
                                   mk_pair ?? (succ_pos_of_nat 136) (Tint I32 Signed)]
      (Ssequence
      (Scall (Some ? (Expr (Evar (succ_pos_of_nat 134)) (Tint I32 Signed)))
        (Expr (Evar (succ_pos_of_nat 132))
          (Tfunction Tnil (Tint I32 Signed)))
        [])
      (Ssequence
      (Sassign (Expr (Evar (succ_pos_of_nat 135)) (Tint I32 Signed))
        (Expr (Econst_int (repr 1)) (Tint I32 Signed)))
      (Ssequence
      (Sfor (Sassign (Expr (Evar (succ_pos_of_nat 136)) (Tint I32 Signed))
              (Expr (Econst_int (repr 2)) (Tint I32 Signed)))
        (Expr (Ebinop Ole
          (Expr (Evar (succ_pos_of_nat 136)) (Tint I32 Signed))
          (Expr (Evar (succ_pos_of_nat 134)) (Tint I32 Signed)))
          (Tint I32 Signed ))
        (Sassign (Expr (Evar (succ_pos_of_nat 136)) (Tint I32 Signed))
          (Expr (Ebinop Oadd
            (Expr (Evar (succ_pos_of_nat 136)) (Tint I32 Signed))
            (Expr (Econst_int (repr 1)) (Tint I32 Signed)))
            (Tint I32 Signed)))
        (Sassign (Expr (Evar (succ_pos_of_nat 135)) (Tint I32 Signed))
          (Expr (Ebinop Omul
            (Expr (Evar (succ_pos_of_nat 135)) (Tint I32 Signed))
            (Expr (Evar (succ_pos_of_nat 136)) (Tint I32 Signed)))
            (Tint I32 Signed)))
      )
      (Sreturn (Some ? (Expr (Evar (succ_pos_of_nat 135))
```

```
                      (Tint I32 Signed)))))))
  ))]
  (succ_pos_of_nat 133)
  [].
```

We can use the definitions in the `Animation.ma` file to reduce the term for a given input (5, in this case; executing a maximum of 40 steps):

```
nremark exec: result ? (exec_up_to myprog 40 [EVint (repr 5)]).
nnormalize; (* you can examine the result here *)
@; nqed.
```

The result state records the interaction (the `EVextcall`) and the return result (120 in least-significant-bit first binary):

```
result (res (list event×state))
  (OK (list event×state)
   ⟨[(EVextcall (p1 (p0 (p1 (p0 (p0 (p0 (p0 one))))))) []
        (EVint (mk_int (pos (p1 (p0 one))) (inrg_mod (pos (p1 (p0 one)))))))],
    Returnstate (Vint (mk_int (pos (p0 (p0 (p0 (p1 (p1 (p1 one)))))))
                      (inrg_mod (pos (p0 (p0 (p0 (p1 (p1 (p1 one))))))))))
      Kstop MEM ⟩)
```

A Description of the Code

The files ported from CompCert were based on version 1.6, with some minor details taken from 1.7.1 (in particular, the parser and the method of building infinite traces for the equivalence proof).

The majority of the semantics is given as Matita source files. The exception is the changes to the CIL based parser, which is presented as a patch to a preliminary version of the CerCo prototype compiler. This patch provides both the extensions to the parser and a pretty printer to produce a usable Matita term representing the program.

acc-0.1.spaces.patch	Changes to early prototype compiler for parsing

Ancilliary definitions

Files corresponding to CompCert.

Coqlib.ma	Minor definitions ported from CompCert
Errors.ma	The error monad
Floats.ma	Axiomatised floating point numbers
Globalenvs.ma	Global environments
Integers.ma	Integers modulo powers of two
Maps.ma	Finite maps (used in particular for local environments)
Smallstep.ma	Generic definitions and lemmas for small step semantics

Files specific to this development.

binary/positive.ma	Binary positive numbers
binary/Z.ma	Binary integers
extralib.ma	Extensions to Matita's library

Inductive semantics ported from CompCert

AST.ma	Minor syntax definitions intended for several compiler stages
Values.ma	Definitions for values manipulated by Clight programs
Mem.ma	Definition of the memory model
Events.ma	I/O events
CostLabel.ma	Definition of cost labels
Csyntax.ma	Clight syntax trees
Csem.ma	Clight inductive semantics

Executable semantics

IOMonad.ma	Definitions of I/O resumption monad
Cexec.ma	Definition of the executable semantics
CexecSound.ma	Soundness of individual steps
CexecComplete.ma	Completeness of individual steps
CexecEquiv.ma	Equivalence of whole program executions
Animation.ma	Definitions to help test the semantics

References

[1] Roberto M. Amadio, Nicolas Ayache, Yann Régis-Gianas, Kayvan Memarian, and Ronan Saillard. Compiler design and intermediate languages. Deliverable 2.1, Project FP7-ICT-2009-C-243881 CerCo.

[2] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[3] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[4] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[5] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction: 11th International Conference (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[6] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[7] Matthiue Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag, 2007.