CerCo

INFORMATION AND COMMUNICATION TECHNOLOGIES (ICT) PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

# Report n. D3.3
# Executable Formal Semantics of front-end intermediate languages

Version 1.0

Main Authors:
Brian Campbell

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

**Abstract** We report on the formalization of the front-end intermediate languages for the CerCo project's compiler using executable semantics in the Matita proof assistant. This includes common definitions for fresh identifier handling, $n$-bit arithmetic and operations and testing of the semantics. We also consider embedding invariants into the semantics for showing correctness properties.

This work was Task 3.3 of the CerCo project and the languages that were formalized were first described in Deliverable 2.1 [1]. They are used by the formalized front-end in Task 3.2 to provide the syntax for the transformations, and to test them by animating programs in the executable semantics. It will also be a crucial part of the specifications for the correctness results in Task 3.4.

# Contents

**1 Introduction** **3**
1.1 Revisions to the prototype compiler . . . 4

**2 Definitions common to several languages** **4**
2.1 Identifiers . . . 4
2.2 Machine integers and arithmetic . . . 5
2.3 Front-end operations . . . 5
2.4 Presentation of small-step executable semantics . . . 5

**3 Clight modifications** **6**

**4 Cminor** **6**

**5 RTLabs** **7**

**6 Testing** **7**

**7 Embedding invariants** **8**
7.1 Cminor block depth . . . 8
7.2 Identifier invariants . . . 9

**8 Conclusion** **10**

# 1 Introduction

This work formalizes the front-end intermediate languages from the CerCo prototype compiler, described in previous deliverables [1, 2]. The front-end of the compiler is summarized in Figure 1 including the intermediate languages and the compiler passes described in the accompanying Deliverable 3.2. We have also refined parts of the formal development that are used for several of the languages in the compiler.

The input language to the formalized front-end is the Clight language. The executable semantics for this language were presented in a previous deliverable [3]. Here we will report on some minor changes to its semantics made to better align it with the whole development.

The formalization of each language takes the form of definitions for abstract syntax and functions providing a small-step executable semantics. This is done in the Calculus of Inductive Constructions (CIC), as implemented in the Matita proof assistant. These definitions will be essential for the correctness proofs of the compiler in Task 3.4.

Finally, we will report on work to add several invariants to the languages. This activity overlaps with Task 3.4 on the correctness of the compiler front-end. However, the use of dependent types mean that this work is tied closely to the definition of the languages and the transformations of the front-end in Task 3.2. By considering it now we can experiment with and judge its impact on the formal semantics, and how feasible retrofitting such invariants is.

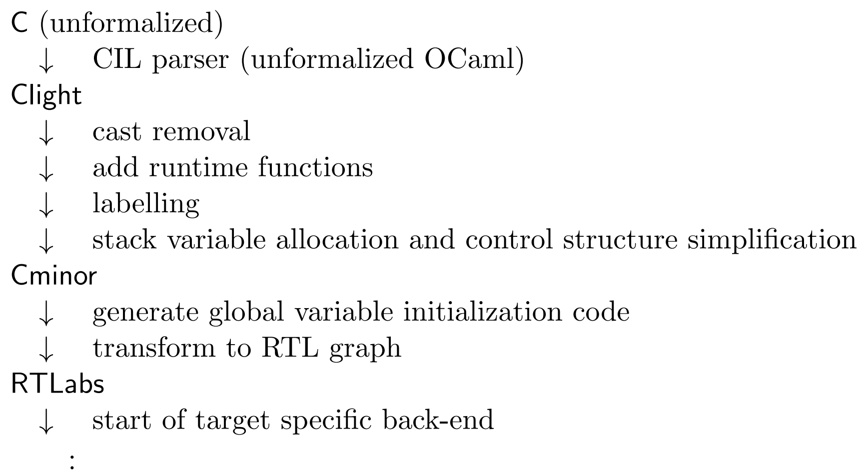


Figure 1: Front-end languages and transformations

### 1.1 Revisions to the prototype compiler

Ongoing work to maintain and improve the prototype compiler has resulted in several changes, mostly minor. The most important change is that the transformations to replace 16 and 32 bit types have been moved from the Clight language to the target-specific stage between RTLabs and RTL to help generate better code, and the addition of a Clight cast removal transformation to reduce the number of 16 and 32 bit operations.

The formalized semantics have tracked these changes, and in this report we describe them as they currently stand.

## 2 Definitions common to several languages

The semantics for many of the languages in the compiler share some core parts: the memory model, environments, runtime values, definitions of operations, a monad for encapsulating failure and I/O (using resumptions), and a common abstraction for small-step executable semantics. The executable memory model was ported from CompCert as part of the work on Clight and was reused for the front-end languages[1]. The failure and I/O monad was introduced in the previous deliverable on Clight [3, §4.2]. In all of the languages except Clight we have a basic form of type, `typ` identifying integers and pointers along with their sizes. The other parts are discussed below, with the only change to the runtime values being the representation of integers.

### 2.1 Identifiers

Each language features one or more kinds of names to represent variables, such as registers, `goto` labels or RTL graph labels. We also need to describe various maps whose domain is a set of identifiers when defining the semantics and compilation.

[1] However, it is likely that we will revise the memory model to make it better suited for describing all of our compiler, not just the front-end.

Previous work on the executable semantics of the target machine code included bit vectors and bit vector tries to define various integers in the semantics, and give a low level view of memory [4]. To keep the size of the development down we have reused these data structures for identifiers and maps respectively.

One difficulty with using fixed size bit vectors for identifiers is that fresh name generation can fail if generate too many. While we use an error monad to deal with failures, we wish to minimize its use in the compiler. Thus we add a flag to detect overflows, and check it after the phase of the compiler is complete to report exhaustion. The rest of the phase can then be written as if name generation always succeeds. In practice this will never occur on normal programs because more identifiers of each sort are available than bytes of code memory on the target.

Given the wide variety of identifiers used in the compiler we also wish to separate the different classes of identifier. Thus we encapsulate the bit vector representing the identifier in a datatype that also carries a tag identifying which class of identifier we are using:

```
inductive identifier (tag:String) : Type[0] :=
  an_identifier : Word → identifier tag.
```

The tries are also tagged in the same manner. These tags have also proved useful during testing by making the resulting terms more readable.

## 2.2 Machine integers and arithmetic

The bit vectors in [4] also came equipped with some basic arithmetic for the target semantics. The front-end required these operations to be generalized and extended. In particular, we required operations such as zero and sign extension and translation between bit vectors and full integers. It also became apparent that while the original definitions worked reasonably on 8-bit vectors, they did not scale up to 32-bit integers. The definitions were then reworked to make them efficient enough to animate programs in the front-end semantics.

## 2.3 Front-end operations

The two front-end intermediate languages, Cminor and RTLabs, share the same set of operations on values. They differ from Clight's operations by incorporating casts and by having a separate operation for each type of data operated upon. For example, subtraction of pointers is treated as a different operation from subtraction of integers.

A common semantics is given for these operations in the form of simple CIC functions on the operation and runtime values.

## 2.4 Presentation of small-step executable semantics

Each language's semantics is described by an instantiation of two records for defining transition systems. We already use these to animate the semantics of the languages for testing, but they will also be used to state the simulation properties we will prove in Tasks 3.4 and 4.4.

First we describe suitable transition systems,

```
record trans_system (outty:Type[0]) (inty:outty → Type[0]) : Type[2] :=
{ global : Type[1]
; state : global → Type[0]
; is_final : ∀g. state g → option int
```

```
; step : ∀g. state g → IO outty inty (trace×(state g))
}.
```

where we have some type of `global` data that remains fixed throughout evaluation (typically used for the global environment), a type for states, a function to detect a final successful state and a step function which can also return an error or request for I/O. The type of states may depend on the fixed data so that we will be able to add invariants asserting that (for example) all the global variables referenced by the program state exist.

We also define a coinductive description of executions and a cofixpoint to produce them. The second record extends the transition system with a type of programs and functions to initialise the transition system:

```
record fullexec (outty:Type[0]) (inty:outty → Type[0]) : Type[2] :=
{ program : Type[0]
; es1 :> trans_system outty inty
; make_global : program → global ?? es1
; make_initial_state : ∀p:program. res (state ?? es1 (make_global p))
}.
```

Finally another function is given which uses them to produce a full execution starting from the program alone.

## 3 Clight modifications

The Clight input language remained largely the same as in the previous deliverable [3]. The principal changes were to use the identifiers and arithmetic described above in place of the arbitrarily large integers used before. For the identifiers, this relieved us of the burden of adding an efficient datatype for maps by reusing the bit vector tries instead.

The arithmetic replaced a dependent pair of an arbitrary integer and a proof that it was in range of 32 bit integers by the exact bit vector for each size of integer. This direct approach is closer to the implementation and more obviously correct — no extra precision can be left in by accident.

## 4 Cminor

The Cminor language does not store local variables in memory, and has simpler control structures than Clight. It is similar in nature to the Cminor language in CompCert, although the semantics have been based on the CerCo prototype rather than ported from CompCert. The syntax is similar to the prototype, except that the types attached to expressions are restricted so that some corner cases are ruled out in the Cminor to RTLabs stage (see the accompanying Deliverable 3.2 for details):

```
inductive expr : typ → Type[0] :=
| Id : ∀t. ident → expr t
| Cst : ∀t. constant → expr t
| Op1 : ∀t,t'. unary_operation → expr t → expr t'
| Op2 : ∀t1,t2,t'. binary_operation → expr t1 → expr t2 → expr t'
| Mem : ∀t,r. memory_chunk → expr (ASTptr r) → expr t
| Cond : ∀sz,sg,t. expr (ASTint sz sg) → expr t → expr t → expr t
| Ecost : ∀t. costlabel → expr t → expr t.
```

For example, note that conditional expressions only switch on integer expressions. In principle we could extend this to statically ensure that only well-typed Cminor expressions are considered, and we will consider this as part of the work on correctness in Task 3.4.

We also provide a variant of the syntax where the only initialization data is the size of each global variable, for use after the initialization code has been generated.

The definition of the semantics is routine: a functional definition of a single small-step of the machine is given, reusing the memory model, environments, arithmetic and operations mentioned above.

## 5 RTLabs

The RTLabs language provides a target independent Register Transfer Language, where programs are represented as control flow graphs. We use the identifiers described above for the graph labels and the maps for the graph itself. The tagging mechanism ensures that labels cannot be mixed up with other identifiers in the program (in particular, we cannot accidentally reuse a `goto` label from Cminor where a graph label should appear).

Otherwise, the syntax and semantics of RTLabs mirrors that of the prototype compiler. Some of the syntax is shown below, including the type of the control flow graphs. The same common elements are used as for Cminor, including the front-end operations mentioned above.

```
inductive statement : Type[0] :=
| St_skip : label → statement
| St_cost : costlabel → label → statement
| St_const : register → constant → label → statement
| St_op1 : unary_operation → register → register → label → statement
...
| St_return : statement
.

definition graph : Type[0] → Type[0] :=identifier_map LabelTag.

record internal_function : Type[0] :=
{ f_labgen   : universe LabelTag
; f_reggen   : universe RegisterTag
; f_result   : option (register × typ)
; f_params   : list (register × typ)
; f_locals   : list (register × typ)
; f_stacksize : nat
; f_graph    : graph statement
}.
```

## 6 Testing

To provide some assurance that the semantics were properly implemented, and to support the testing described in the accompanying Deliverable 3.2, we have adapted the pretty printers in the prototype compiler to produce Matita terms for the syntax of each language described above.

A few common definitions were added for animating the small-step semantics of any of the front-end languages in Matita, given a bound on the number of steps to execute and any input

required. These use the definitions described in Section 2.4 to perform the actual execution. We then used a small selection of test cases to ensure basic functionality. However, this is still a time consuming process, so more testing will carried out once the extraction of CIC terms to OCaml programs is implemented in Matita.

# 7 Embedding invariants

Each phase of the prototype compiler can fail in a number of places if the input language permits programs that are badly structured in some sense: a missing label in a `goto` statement or CFG, an undefined variable name, a `break` statement outside of a loop or `switch`, and so on. We wish to restrict our intermediate languages using dependent types to remove as many 'junk' programs as possible to rule out such failures. We also hope that such restrictions will help in other correctness proofs.

This goal lies in the overlap between several tasks in the project: it involves manipulating the syntax and semantics of the intermediate languages (the present work), the encoding of the front-end compiler phases in Matita (Task 3.2) and the correctness of the front-end (Task 3.4). Thus this work is rather experimental; it is being carried out on branches in our source code repository and the final form will be decided and merged in during Task 3.4.

So far we have tried adding two forms of invariant — one using dependent types to index statements in Cminor by their block depth, and the other asserts that variables and labels are present in the appropriate environments by adding a separate invariant to each function. Note that these do not yet cover all of the properties that a program in these languages is expected to enjoy; for example, there are currently no checks that references to globals are well-defined.

## 7.1 Cminor block depth

The Cminor language has relatively simple control structures. Statements are provided for infinite loops, non-looping blocks and exiting an arbitrary number of blocks (for `break`, failing loop guards and the switch statement[2]).

However, this means that there are badly-formed Cminor programs such as

```
int main() {
  block {
    loop {
      exit 5
    }
  }
}
```

where we attempt to exit more blocks than exist. To rule these out (including demonstrating that the previous phase of the compiler does not generate them) we can index the statements of the language by the depth of the enclosing blocks.

The adaption of the syntax adds the depth to every statement, and uses bounded integers in the exit and switch statements:

```
inductive stmt : ∀blockdepth:nat. Type[0] :=
| St_skip : ∀n. stmt n
```

[2]We are considering replacing the Cminor switch statement with one that uses `goto`-like labels in both the prototype and the formalized compilers, but for now we stick with this CompCert-style arrangement.

```
...
| St_loop : ∀n. stmt n → stmt n
| St_block : ∀n. stmt (S n) → stmt n
| St_exit : ∀n. Fin n → stmt n
(* expr to switch on, table of <switch value, #blocks to exit>, default *)
| St_switch : expr → ∀n. list (int × (Fin n)) → Fin n → stmt n
...
```

where `stmt n` is a statement enclosed in `n` blocks, and `Fin n` is a standard construction for a natural number which is at most `n`.

In the semantics the number of blocks is also added to the continuations and state, and the function to find the continuation from an exit statement can be made failure-free. We note in passing that adding this parameter detected a small mistake in the semantics concerning continuations and tail calls, although the mistake itself was benign.

### 7.2 Identifier invariants

To show that the variables and labels occurring in the body of a function are present in the relevant structures we add an additional invariant to the function records.

For Cminor we use a higher-order predicate which recursively applies a predicate to all substatements:

```
let rec stmt_P (P:stmt → Prop) (s:stmt) on s : Prop :=
match s with
[ St_seq s1 s2 ⇒ P s ∧ stmt_P P s1 ∧ stmt_P P s2
| St_ifthenelse _ _ _ s1 s2 ⇒ P s ∧ stmt_P P s1 ∧ stmt_P P s2
| St_loop s' ⇒ P s ∧ stmt_P P s'
| St_block s' ⇒ P s ∧ stmt_P P s'
| St_label _ s' ⇒ P s ∧ stmt_P P s'
| St_cost _ s' ⇒ P s ∧ stmt_P P s'
| _ ⇒ P s
].
```

Dependent pattern matching on statements thus allows an accompanying `stmt_P` fact to be unfold to the predicate on the current statement and the predicate applied to all substatements.

We require two properties to hold in Cminor functions:

1. All variables in the body are present in the list of parameters or the list of variables for the function (this also uses a similar recursive predicate on expressions).

2. All labels in `goto` statements appear in a label statement.

The function definition thus becomes:

```
record internal_function : Type[0] :=
{ f_return   : option typ
; f_params   : list (ident × typ)
; f_vars     : list (ident × typ)
; f_stacksize : nat
; f_body     : stmt
; f_inv      : stmt_P (λs.stmt_vars (λi.Exists ? (λx.\fst x = i) (f_params @ f_vars)) s ∧
                         stmt_labels (λl.Exists ? (λl'.l' = l) (labels_of f_body)) s) f_body
}.
```

where `stmt_vars` and `stmt_labels` constrain the variables and labels that appear directly in a statement (but not substatements) to appear in the given list, and `labels_of` returns a list of all the labels defined in a statement.

The Clight semantics can be amended to use these invariants, although the main benefit is for the compiler stages (see the accompanying Deliverable 3.2 for details). The semantics require the invariants to be added to the state and continuations. It was convenient to split the continuations between the local continuation representing the rest of the code to be executed within the function, and the stack of function calls because it becomes easier to state the property on the local continuation alone. The invariant for variables is slightly different — we require that every variable appear in the local environment. We use `f_inv` from the function to establish this invariant when the environment is set up on function entry.

It is unclear whether changing the semantics is really worthwhile. It witnesses that the invariants are those we wanted, but makes no difference to the actual execution of the program, especially as the execution can still fail due to genuine runtime errors. Moreover, it is unclear what effect the presence of proof terms and more dependent pattern matching in the semantics will have on the complexity of future correctness proofs. We plan to examine this issue during Task 3.4.

We use a similar method to specify the invariant that the RTLabs graph is closed — that is, any successor labels in a statement in the graph are present in the graph. The definition is simpler in RTLabs because the flat representation of the graph does not require recursive definitions like `stmt_P` above.

## 8 Conclusion

We have developed executable semantics for each of the front-end languages of the CerCo compiler. These will form the basis of the correctness statements for each stage of the compiler in Task 3.4. We have also shown that useful invariants can be added as dependent types, and intend to use these in subsequent work.

## 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] Nicolas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. Prototype implementation. Deliverable 2.2, Project FP7-ICT-2009-C-243881 CerCo.

[3] Brian Campbell and Randy Pollack. Executable formal semantics of C. Deliverable 3.1, Project FP7-ICT-2009-C-243881 CerCo.

[4] Dominic P. Mulligan and Claudio Sacerdoti Coen. Executable formal semantics of machine code. Deliverable 4.1, Project FP7-ICT-2009-C-243881 CerCo.