CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D3.2
CIC encoding: Front-end

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 describe the translation of the front-end of the CerCo compiler from the OCaml prototype to the Calculus of Inductive Constructions (CIC) in the Matita proof assistant. This transforms programs in the C-like Clight language to the RTLabs language, which is reasonably target-independent and in the form of a control flow graph.

We also report on progress enriching these transformations with dependent types so as to establish key invariants in each intermediate language, which removes potential sources of failure within the compiler and will assist in future work proving correctness properties.

This work was Task 3.2 of the CerCo project, translating the prototype described in Deliverable 2.2 [2] into CIC using the intermediate languages formalized in Deliverables 3.1 [4] and 3.3. It will feed into the front-end correctness proofs in Task 3.4 and is a counterpart to the back-end formalization in Task 4.2.

Contents

1 Introduction

The CerCo compiler has been prototyped in OCaml [1, 2], but the certified compiler will be a program written in the Calculus of Inductive Constructions (CIC), as realised by the Matita proof assistant. This deliverable reports on the translation of the front-end of the compiler into CIC and the subsequent efforts to start exploiting dependent types to maintain invariants and rule out potential sources of failure in the compiler.

The input language for the formalized compiler is the Clight language. This is a C-like language with side-effect free expressions that was adapted from the CompCert project [3][1] and provided with an executable semantics. See [4] for more details on the syntax and semantics.

The front-end of the compiler is summarised in Figure 1. The two intermediate languages involved are

Cminor — a C-like language where local variables are not explicitly allocated memory and control structures are simpler

RTLabs — a language in the form of a control flow graph which retains the values and front-end operations from Cminor

More details on the formalisation of the syntax and semantics of these languages can be found in the accompanying Deliverable 3.4. Development of the formalized front-end was conducted in concert with the development of these intermediate languages to facilitate testing.

[1] We will also use their CIL-based C parser to generate Clight abstract syntax trees, but will not formalize this code.

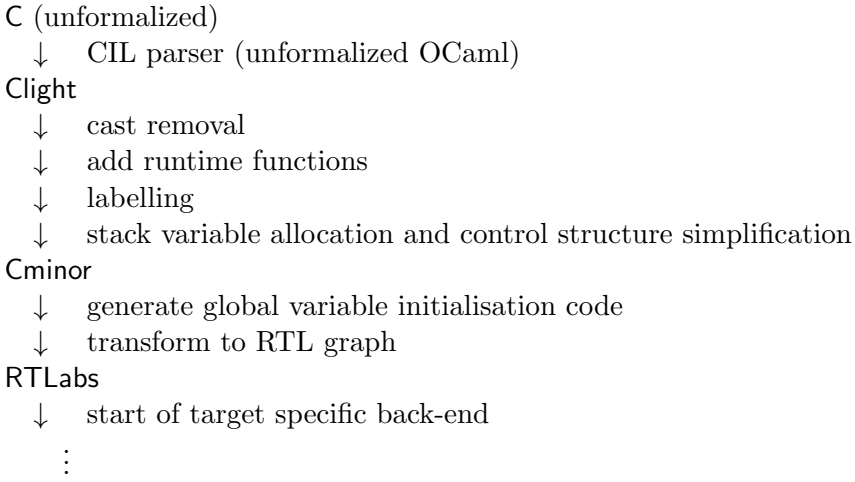

Figure 1: Front-end languages and transformations

1.1 Revisions to the prototype compiler

We have been tracking revisions to the prototype compiler during the development of the formalized version. Most of these changes were minor, but one exception is a major change to the structure of the compiler.

The original plan for the front-end featured a Clight to Clight8 phase near the start which replaced all of the integer values and operations by 8 bit counterparts, while pointers were split into bytes at a later stage. Experience has shown that it would be difficult to produce good code with this approach. Instead, we now have:

- full size integers, pointers and operations until code selection (the first part of the back-end after RTLabs), and
- a cast removal stage which simplifies Clight expressions such as

  ```
  (char)((int)x + (int)y)
  ```

  into equivalent operations on simpler types, `x+y` in this case. The cast removal is important because C requires *arithmetic promotion* of integer types to (at least) `int` before an operation is performed. The Clight semantics do not perform the promotions, instead they are added as casts by the CIL-based parser. However, our targets benefit immensely from performing operations on the smallest possible integer type, so it is important that we remove promotions where possible.

This document describes the formalized front-end after these changes.

2 Clight phases

In addition to the conversion to Cminor, there are several transformations which act directly on the Clight language.

2.1 Cast simplification

We noted above that the arithmetic promotion required by C (and implemented in the CIL-based parser) adds numerous casts, causing arithmetic operations to be performed on 32 bit integers. If left alone, the resulting code will be larger and slower. This phase removes many of the casts so that the operations can be performed more efficiently.

The prototype version worked by recognising fixed patterns in the Clight abstract syntax tree such as

$$(t)((t_1)e_1\ op\ (t_2)e_2),$$

subject to restrictions on the types. These are replaced with a simpler version without the casts. Such 'deep' pattern matching is slightly awkward in Matita and this approach does not capture compositions of operations, such as

```
(char)(((int)a + (int)b) + (int)c)
```

where `a`, `b` and `c` are of type `char`, because the intermediate expression is not cast to and from `char`.

The formalized version uses a different method, recursively examining each expression constructor to see if the expression can be coerced to some 'desired' type. For example, when processing the above expression it reaches each `int` cast with a desired type of `char`, notes that the subexpression is of type `char` and eliminates the cast. Moreover, when the recursive processing is complete the `char` cast is also eliminated because its subexpression is already of the correct type.

This has been implemented in Matita. We have also performed a few proofs that the arithmetic behind these changes is correct to gain confidence in the technique. During Task 3.4 we will extend these proofs to cover more operations and show that the semantics of the expressions are equivalent, not just the underlying arithmetic.

2.2 Labelling

This phase adds cost labels to the Clight program. It is a fairly simple recursive definition, and was straightforward to port to Matita. The generation of cost labels was handled by our generic identifiers code, described in the accompanying Deliverable 3.3 on intermediate languages.

2.3 Runtime functions

Some operations on integers do not have a simple translation to the target machine code. In particular, we need to replace operations for 16 and 32-bit division and most bitwise shifts with calls to runtime functions. These functions need to be added to the program at an early stage because of their impact on execution time: any loops must be available to our labelling mechanism so that we can report on how long the resulting machine code will take to execute.

We follow the prototype in replacing the affected expressions, which requires us to break up expressions into multiple statements because function calls are not permitted in Clight expressions. We may investigate moving these substitutions to a later stage of the compiler if they prove difficult to reason about. However, this would also require adjusting the semantics so that the costs still appear in the evaluation of Clight programs.

The prototype adds the functions themselves by generating C code as text and reparsing the program. This is unsuitable for formalization, so we generate Clight abstract syntax trees directly.

2.4 Conversion to Cminor

The conversion to Cminor performs two essential tasks. First, it determines which local variables need to be stored in memory and generates explicit memory accesses for them. Second, it must translate the control structures (`for`, `while`, ...) into Cminor's more basic structures.

These are both performed by code similar to that in the prototype, although the use of generic fold operations on statements and expressions has been replaced by simpler recursive definitions.

There are two additional pieces of work that the formalized translation must do. The Cminor definition features some mild constraints of the types of expressions, which we can enforce in the translation using some type checking. The error monad is used to dispose of ill-typed Clight programs.

The other difficulty is that we need to generate fresh temporary variables to store function results in before they are written to memory. This is necessary because Clight allows arbitrary *lvalue* expressions as the destination for the returned value, but Cminor only allows local variables. All other variable names in the Cminor program came from the Clight program, but we need to construct a method for generating fresh names for the temporaries.

Our identifiers are based on binary numbers, and generation of fresh names is handled by keeping track of the highest allocated number. Normally this is initialised at zero, but if initialised by the largest existing identifier in the Clight program then the generated names will be fresh. To do this, we extract the maximum identifier by recursively finding the maximum variable name used in every expression, statement and function of the program.

3 Cminor phases

Cminor programs are processed by two passes: one deals with the initialisation of global variables, and the other produces RTLabs code.

3.1 Initialisation code

This replaces the initialisation data with explicit code in the main function. The only remarkable point in the formalization is that we have two slightly different instantiations of the Cminor syntax: one with initialisation data that this pass takes as input, and one with only size information that is the output. In addition to reflecting the purpose of this pass in its type, it also ensures that the pass cannot be accidentally omitted.

3.2 Conversion to RTLabs

This pass breaks down the structure of the Cminor program into a control flow graph, but maintains the same set of operations. The algorithm is stateful in the sense that it builds up the RTLabs function body incrementally, but all of the relevant state is already present in the function record (including the fresh register and graph label name generators) and the prototype passes this around. Thus the formalized code is very similar in nature.

One possible variation would be to explicitly define a state monad to carry the function under construction around, but it is not yet clear if this will make the correctness results easier to prove.

4 Adding and using invariants

The compiler phases described above all use the error monad to deal with inconsistencies in the program being transformed. In particular, lookups in environments may fail, control flow graphs may have missing statements and various structural problems may be present. We would like to show that these failures are absent where possible by establishing that programs are well formed early in the compilation process.

This work overlaps with Deliverable 3.3 (where more details of the additions to the syntax and semantics of the intermediate languages can be found) and Task 3.4 on the correctness of the compiler. Thus this work is experimental in nature, and will evolve during Task 3.4.

The use of the invariants follows a common pattern. Each language embeds invariants in the function record that constrain the function body by other information in the record (such as the list of local variables and types, or the set of labels declared). However, during the transformations they typically need to be refined to constraints on individual statements and expressions with respect to data structures used in the transformation. A similar change in invariants is required between the transformation and the new function.

For example, consider the use of local variables in the Cminor to RTLabs stage. We start with

```
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 the first half of `f_inv` requires every variable in the function body to appear in the parameter or variable list. In the translation to RTLabs, variable lookups are performed in a map to RTLabs pseudo-registers:

```
definition env :=identifier_map SymbolTag register.

let rec add_expr (le:label_env) (env:env) (ty:typ) (e:expr ty)
                 (Env:expr_vars ty e (present ?? env))
                 (dst:register) (f:partial_fn le) on e
    : Σf':partial_fn le. fn_graph_included le f f' :=
match e return λty,e.expr_vars ty e (present ?? env) →
               Σf':partial_fn le. fn_graph_included le f f' with
[ Id _ i ⇒ λEnv.
   let r :=lookup_reg env i Env in
   ...
```

Note that `lookup_reg` returns a register without any possibility of error. The reason this works is that the pattern match on `e` refines the type of the invariant `Env` to a proof that the variable `i` is present. We then pass this proof to the lookup function to rule out failure.

When this map `env` is constructed at the start of the phase, we prove that the proof `f_inv` from the function implies the invariant on variables needed by `add_expr` and its equivalent on statements:

```
lemma populates_env : ∀l,e,u,l',e',u'.
  populate_env e u l = ⟨l',e',u'⟩ →
  ∀i. Exists ? (λx.\fst x = i) l →
      present ?? e' i.
```

A similar mechanism is used to show that `goto` labels are always declared in the function.

Also note the return type of `add_expr` is a dependent pair. We build the resulting RTLabs function incrementally, using a type `partial_fn` that does not contain the final invariant for functions. We always require the `fn_graph_included` property for partially built functions to show that the graph only gets larger, a key part of the proof that the resulting control flow graph is closed. Dependent pairs are used in a similar manner in the Clight to Cminor phase too.

This work does not currently cover all of the possible sources of failure; in particular some structural constraints on functions are not yet covered and some properties of RTLabs programs that may be useful for later stages or the correctness proofs are not produced. Moreover, we may experiment with variations to try to make the proof obligations and syntax simpler to deal with. However, it does show that retrofitting these properties using dependent types in Matita is feasible.

5 Testing

To provide some early testing and bug fixing of this code we constructed it in concert with the executable semantics described in Deliverable 3.3, and Matita term pretty printers in the prototype compiler. Using these, we were able to test the phases individually and together by running programs within the proof assistant itself, and comparing the results with the expected output.

6 Conclusion

We have formalized the front-end of the CerCo compiler in the Matita proof assistant, and shown that invariants can be added to the intermediate languages to help show properties of it. This work provides a solid basis for the compiler correctness proofs in Task 3.4.

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] Sandrine Blazy and Zaynah Dargaye andXavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

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