CerCo Work Package 3
Verified Compiler — Front-end

Brian Campbell

Brian.Campbell@ed.ac.uk

Laboratory for Foundations of Computer Science
The University of Edinburgh

March 16, 2011

SEVENTH FRAMEWORK PROGRAMME

Achievements in period 2

T3.2 Completed: Matita encoding of compiler front-end

T3.3 Completed: Executable semantics for intermediate languages

T3.4 In progress: Correctness proofs

Deliverables D3.2 and D3.3 submitted.

This talk is an interleaved presentation of these results.

Outline

- Shared definitions
- C (unformalized)
  - ↓ CIL parser (unformalized OCaml)

  Clight
  - ↓ cast removal
  - ↓ add runtime functions
  - ↓ cost labelling
  - ↓ stack variable allocation and control structure simplification

  Cminor
  - ↓ generate global variable initialisation code
  - ↓ transform to RTL graph

  RTLabs
  - ⋮ start of target specific back-end
- Structured traces

SEVENTH FRAMEWORK PROGRAMME

Shared definitions

Identifiers: variables, cost labels, CFG labels, ...

- Represented by arbitrarily large binary numbers and trees
  - D3.3 'lazy failure' approach reusing existing structures
  - Invariant's types may depend on success of name generation
- Tags for some type safety:

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

Memory, global environments from D3.1
Bit vector based arithmetic from D4.1

- added extra operations, increased efficiency

Cminor and RTLabs share operations on values.

Clight: syntax and semantics

Modest evolution from D3.1:

- use precise bit vectors instead of integer and range proof
- adapting to changes in common definitions
- added function to produce fresh name generator
  - used for adding temporary variables
  - works by finding largest existing identifier
  - have shown necessary freshness proof

Clight: cast simplification

C insists on arithmetic promotion

- CIL-based OCaml parser adds suitable casts
- bad for our target (32-bit ops instead of 8-bit)

Prototype recognises fixed patterns to simplify:

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

- Deep pattern matching is awkward in Matita
- Misses (e.g.) `(char)((int)a+(int)b+(int)c)`

Instead: recursive 'coerce to desired type' approach.
Have done some preliminary proofs that this works.

Clight: cost labelling

Adds cost labels to Clight program.

- a simple recursive function, like prototype
- uses common identifiers definition to produce fresh cost labels

The cost labelling is sound and precise, so we will be able to prove some syntactic properties required later:

1. every function starts with a cost label
2. every branching instruction is followed by a cost label
3. the head of every loop (including `gotos`) is a cost label

Cminor syntax and semantics

Similar to CompCert's — but developed from prototype.

- Some type enforcement for expressions:

```
inductive expr : typ → Type[0] :=
| Id : ∀t. ident → expr t
| Cst : ∀t. constant → expr t
| Op1 : ∀t,t'. unary_operation t t' → 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.
```

- indirectly forces checking that temporaries are fresh
- ⋆ reduces failure cases in translation to RTLabs

Semantics unexciting small-step function.

Adding invariants

Lots of sources of failure in front-end due to badly formed programs:

- missing variables, graph nodes, ...
- badly structured programs

Have been refining the intermediate languages to rule these out:

1. Indexing Cminor syntax by nesting depth
2. adding type constraints
3. adding invariants asserting presence of variables, etc.

Invariants for identifiers

Invariants change between languages and compilation:

syntax constraints between parts of function body and other information about the function

compilation constraints between statements and expressions and data structures in the translation

When using or creating function definitions have proved that we can switch between invariants:

```
lemma populates_env : ∀l,e,u,l',e',u'.
  distinct_env ?? l →                          (* distinct names in l *)
  populate_env e u l = ⟨l',e',u'⟩ → (* build register mapping *)
  ∀i,t. Exists ? (λx.x = ⟨i,t⟩) l → (* Anything in the environment... *
      Env_has e' i t.    (* maps to something of the correct type *)
```

SEVENTH FRAMEWORK PROGRAMME

Invariants in Cminor

Embedded invariant in the function record:

```
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,t.Exists ? (λx. x = ⟨i,t⟩) (f_params @ f_vars)) s ∧
    stmt_labels (λl.Exists ? (λl'.l' = l) (labels_of f_body)) s)
  f_body
}.
```

1. All variables are in the parameters or locals lists
   — with the correct type
2. All `goto` labels mentioned are defined in the body

Clight to Cminor

Two main jobs:

1. make memory allocation of variables explicit
2. use simpler control structures

Again, based on prototype rather than CompCert.

Added type checking:

- satisfies the restrictions for Cminor expressions
- could separate out, have a 'nice Clight' language

Similarly, code checks

- variable environments are well-formed
- all `goto` labels are translated

Clight to Cminor proofs

Beyond the invariants already shown, we will prove:

1. a simulation relation
   - between statement and continuation pairs
   - using memory injection (similar to CompCert)
2. syntactic cost labelling properties are preserved
   - structural induction on function body

Cminor: initialization

Replace initialization data by code in the initial function.

- straightforward to define
- instantiate a slightly different version of the semantics for it
  - Can't accidentally forget the pass

Correctness should follow because the state after the initialisation will be the same as the initial state of the original.

RTLabs: syntax and semantics

Register Transfer Language with front-end operations.
Control flow graph implemented by generic identifiers map:

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

inductive statement : Type[0] :=
| St_skip : label → statement
| St_const : register → constant → label → statement
| St_op1 : ∀t,t'. unary_operation t' t → register
   → register → label → statement
...
```

- Shares basic operations (including semantics) with Cminor.
- Tags prevent confusion of labels (graph vs. goto vs. cost).

Semantics straightforward interpretation of statements.

RTLabs: syntax and semantics

```
record internal_function : Type[0] :=
{ f_labgen   : universe LabelTag
; f_reggen   : universe RegisterTag
...
; f_graph    : graph statement
; f_closed   : graph_closed f_graph
; f_typed    : graph_typed (f_locals @ f_params) f_graph
...
```

Enforce that

- every statement's successor labels are present in the CFG
- every statement should be well-typed (for limited type system)

Cminor to RTLabs

Break down statements and expressions into RTL graph.

- Incrementally build function backwards
- all state is in function record, like prototype

Showing graph closure requires

- monotonicity of graph construction
- eventual insertion of all `goto` destinations

Carry these along in the partially built function record.

Establishing RTLabs invariants

Use dependent pairs to show invariant along with results.

```
let rec add_expr ... (e:expr ty)
   (Env:expr_vars ty e (present ?? env))    (* invariant *)
   (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) → ... with
[ ...
| Cst _ c ⇒ λ_. ≪add_fresh_to_graph ? (St_const dst c) f ?, ?≫
...
```

Continually appeal to monotonicity (`fn_graph_included`).
Use unification hints to simplify stepping back.

Cminor to RTLabs: cost labels

Two cost label properties are the same, the third will require more work:

1. cost label at head of function
2. cost label after branching instructions
3. cost labels at the head of each loop / `goto` destination

No simple notion of the head of a loop or `goto` any more.

Instead: will prove in Cminor that after following a finite number of instructions we reach either

- a cost label, or
- the end of the function

RTLabs structured traces

Front-end only uses flat traces consisting of single steps.

The back-end will need the function call structure and the labelling properties in order to show that the generated costs are correct.

- Traces are structured in sections from cost label to cost label,
- the entire execution of function calls nested as a single ‘step’,
- a coinductive definition presents non-terminating traces, using the inductive definition for all terminating function calls

RTLabs chosen because it is the first languages where statements:

- take one step each (modulo function calls)
- have individual ‘addresses’

SEVENTH FRAMEWORK PROGRAMME

RTLabs structured traces

Have already established the existence of such traces

- termination decided classically
- syntactic labelling properties used to build semantic structure
- show stack preservation to ensure that function calls `return` to the correct location
- tricky to establish guardedness of definitions

Next, prove that flattening these traces yields the original.

Conclusion

Syntax, semantics and translations of prototype now implemented in Matita.

Have defined and established

- invariants regarding variables, typing and program structure
- a rich form of execution trace to pass to the back-end

Work in progress:

- showing functional correctness of the front-end
- proving that cost labelling is appropriate, and preserved

Finally, end-to-end functional and cost correctness results.

SEVENTH FRAMEWORK PROGRAMME