CerCo Work Package 4

Dominic Mulligan
Postdoc, University of Bologna

CerCo project review meeting
March 2012

Overview of progress

Status at end of first period:

Executable semantics of MCS-51 in OCaml and Matita

Goals for the end of second period:

Executable semantics of back-end intermediate languages

Encoding of compiler back-end in CIC

Contents

Outline

SEVENTH FRAMEWORK PROGRAMME

Backend intermediate languages

RTLabs
- ↓ copy propagation ×
- ↓ instruction selection ✓
- ↓ change of memory models in compiler ✓

RTL
- ↓ constant propagation ×
- ↓ calling convention made explicit ✓
- ↓ layout of activation records ✓

ERTL
- ↓ register allocation and spilling ✓
- ↓ dead code elimination ✓

LTL
- ↓ function linearisation ✓
- ↓ branch compression ×

LIN
- ↓ relabeling ✓

ASM

Joint: a new approach I

- Consecutive languages in backend must be similar
- Transformations between languages translate away some small specific set of features
- But looking at OCaml code, not clear precisely what differences between languages are, as code is repeated
- Not clear if translation passes can commute, for instance
- CerCo passes are in a different order to CompCert (calling convention and register allocation done in different places)
- Instruction selection done early: changing subset of instructions used would require instructions to be duplicated everywhere in backend

`Joint`: a new approach II

- Idea: all of these languages are just instances of a single language
- This language `Joint` is parameterised by a type of registers to be used in instructions, and so forth
- Each language after RTLabs is now just defined as the `Joint` language instantiated with some concrete types
- Similarly for semantics: common definitions that take e.g. type representing program counters as parameters

Joint: a new approach III

Joint instructions allow us to embed language-specific instructions:

```
inductive joint_instruction (p: params__) (globals: list ident): Type[0] :=
 | COMMENT: String → joint_instruction p globals
 | COST_LABEL: costlabel → joint_instruction p globals
 ...
 | COND: acc_a_reg p → label → joint_instruction p globals
 | extension: extend_statements p → joint_instruction p globals.

inductive ertl_statement_extension: Type[0] :=
 | ertl_st_ext_new_frame: ertl_statement_extension
 | ertl_st_ext_del_frame: ertl_statement_extension
 | ertl_st_ext_frame_size: register → ertl_statement_extension.
```

Joint: a new approach IV

- Languages that provide extensions need to provide translations and semantics for those extensions
- Everything else can be handled at the Joint-level
- This modularises the handling of these languages

`Joint`: advantages I

- We can recover the concrete OCaml languages by instantiating parameterized types
- Why use `Joint`?
- Reduces repeated code (fewer bugs, or places to change)
- Unify some proofs, making correctness proof easier
- Generic optimizations (e.g. constant propagation)

`Joint`: advantages II

- Easier to add new intermediate languages as needed
- Easier to see relationship between consecutive languages at a glance
- MCS-51 instruction set embedded in `Joint` syntax
- Simplifies instruction selection
- We can investigate which translation passes commute much more easily

Semantics of `Joint` I

- As mentioned, use of `Joint` also unifies semantics of these languages
- We use several sets of records, which represent the state that a program is in
- These records are parametric in representations for e.g. frames

A new intermediate language

- Matita backend includes a new intermediate language: RTLntc
- Sits between RTL and ERTL
- RTLntc is the RTL language where all tailcalls have been eliminated
- This language is 'implicit' in the OCaml compiler
- There, the RTL to ERTL transformation eliminates tailcalls as part of translation
- But including an extra, explicit intermediate language is 'almost free' using the `Joint` language approach

The LTL to LIN transform I

- `Joint` clearly separates fetching from program execution
- We can vary how one works whilst fixing the other
- Linearisation is moving from fetching from a graph-based language to fetching from a list-based program representation
- The order of transformations in OCaml prototype is fixed
- Linearisation takes place at a fixed place, in the translation between LTL and LIN
- The Matita compiler is different: linearisation is a generic process
- Any graph-based language can now be linearised

The LTL to LIN transform II

- CompCert backend linearises much sooner than CerCo's
- Can now experiment with linearising much earlier
- Many transformations and optimisations can work fine on a linearised form
- Only place in the (current) backend that requires a graph-based language is in the ERTL pass, where we do a dataflow analysis

SEVENTH FRAMEWORK PROGRAMME

Outline

Time not reported

- We had six months of time which is not reported on in any deliverable
- We invested this time working on:
  - The global proof sketch
  - The setup of 'proof infrastructure', common definitions, lemmas, invariants etc. required for main body of proof
  - The proof of correctness for the assembler
  - A notion of 'structured traces', used throughout the compiler formalisation, as a means of eventually proving that the compiler correctly preserves costs
  - Structured traces were defined in collaboration with the team at UEDIN

Assembler

- After LIN, compiler spits out assembly language for MCS-51
- Assembler has pseudoinstructions similar to many commercial assembly languages
- For instance, instead of computed jumps (e.g. `SJMP` to a specific address), compiler can simply spit out a generic jump instruction to a label
- Simplifies the compiler, at the expense of introducing more proof obligations
- Now need a formalized assembler (a step further than CompCert)

A problem: jump expansion

- 'Jump expansion' is our name for the standard 'branch displacement' problem
- Given a pseudojump to a label *l*, how best can this be expanded into an assembly instruction `SJMP`, `AJMP` or `LJMP` to a concrete address?
- Problem also applies to conditional jumps
- Problem especially relevant for MCS-51 as it has a small code memory, therefore aggressive expansion of jumps into smallest possible concrete jump instruction needed
- But a known hard problem (NP-complete depending on architecture), and easy to imagine knotty configurations where size of jumps are interdependent

Jump expansion I

- We employed the following tactic: split the decision over how any particular pseudoinstruction is expanded from pseudoinstruction expansion
- Call the decision maker a 'policy'
- We started the proof of correctness for the assembler based on the premise that a correct policy exists
- Further, we know that the assembler only fails to assemble a program if a good policy does not exist (a side-effect of using dependent types)
- A bad policy is a function that expands a given pseudojump into a concrete jump instruction that is 'too small' for the distance to be jumped, or makes the program consume too much memory

Jump expansion II

- Jaap Boender at UNIBO has been working on a verified implementation of a good jump expansion policy for the MCS-51
- The strategy initially translates all pseudojumps as `SJMP` and then increases their size if necessary
- Termination of the procedure is proved, as well as a safety property, stating that jumps are not expanded into jumps that are too long
- His strategy is not optimal (though the computed solution is optimal for the strategy employed)
- Jaap's work is the first formal treatment of the 'jump expansion problem'

Assembler correctness proof

- Assuming the existence of a good jump expansion property, we completed about 75% of the correctness proof for the assembler
- Jaap's work has just been completed (modulo a few missing lemmas)
- Postponed the remainder of main assembler proof to start work on other tasks (and for Jaap to finish)
- We intend to return to proof, and publish an account of the work (possibly) as a journal paper

Outline

Who pays? I

In C:

```
int main(int argc, char** argv) {
  cost_label1:
    ...
    some_function();
  cost_label2:
    ...
}
```

In ASM:

```
  ...
main:
  ...
cost_label1:
  ...
  LCALL some_function
cost_label2:
  ...
```

- Where do we put cost labels to capture execution costs?
- Proof obligations complicated by panoply of labels
- Doesn't work well with `g(h() + 2 + f())`
- Is `cost_label2` ever reached?
- `some_function()` may not return correctly

Who pays? II

- Solution: omit `cost_label2` and just keep `cost_label1`
- We pay for everything 'up front' when entering a function
- No need to prove `some_function()` terminates
- But now execution of functions in CerCo takes a particular form
- Functions begin with a label, call other functions that begin with a label, eventually return, but *return* to the correct place
- 'Recursive structure'

Structured traces I

- We introduced a notion of 'structured traces'
- These are intended to statically capture the (good) execution traces of a program
- To borrow a slogan: they are the 'computational content of a well-formed program's execution'
- Come in two variants: inductive and coinductive
- Inductive captures program execution traces that eventually halt, coinductive ones that diverge

Structured traces II

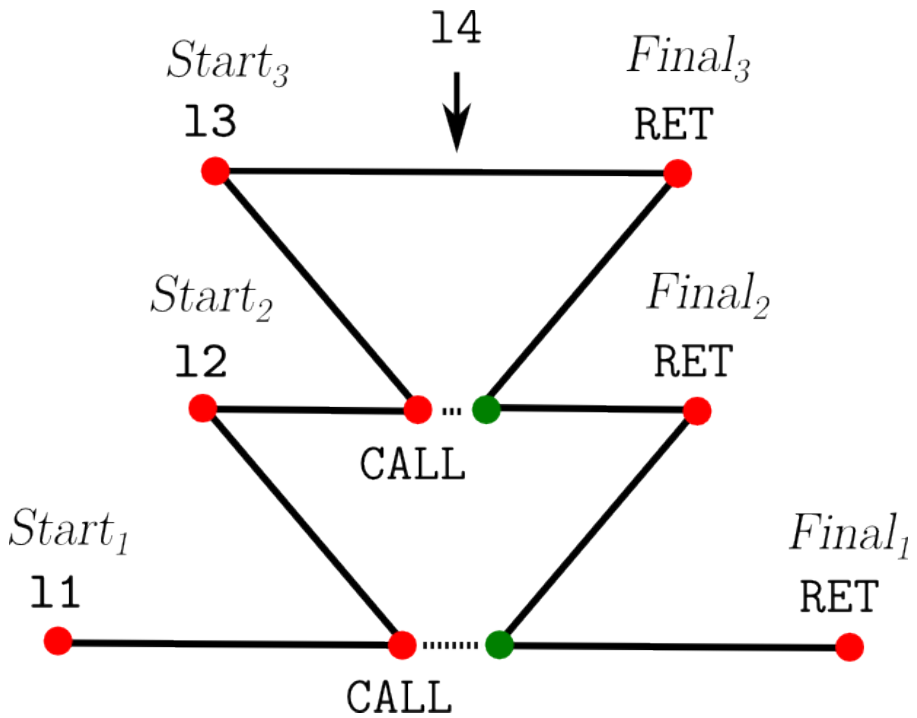
- I focus on the inductive variety, as used the most (for now) in the backend
- In particular, used in the proof that static and dynamic cost computations coincide
- Traces preserved by backend compilation, initially created at RTL
- This will be explained later

Structured traces III

- Central insight is that program execution is always in the body of some function (from `main` onwards)
- A well formed program must have labels appearing at certain spots
- Similarly, the final instruction executed when executing a function must be a `RET`
- Execution must then continue in body of calling function, at correct place
- These invariants, and others, are crystalised in the specific syntactic form of a structured trace

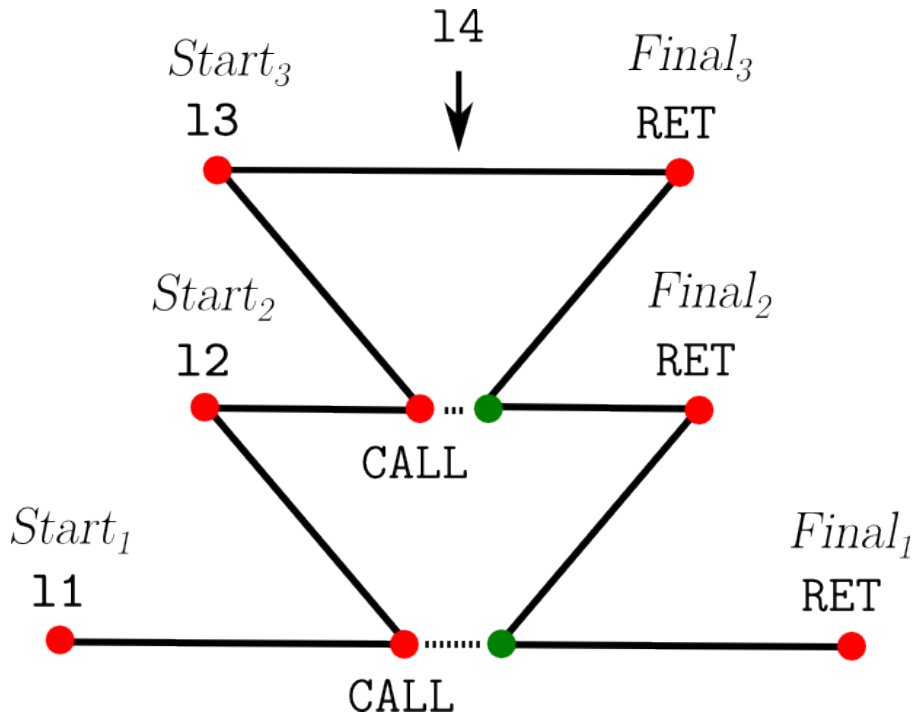
Recursive structure of 'good' execution I

- All calls return to the correct place
- `CALL`s and `JMP`s only to labels



SEVENTH FRAMEWORK PROGRAMME

Static and dynamic costs




```
emit(I1)
MOV r1 0
ADD r1 r2
CALL f
ADD r2 r2
MOV r2 0
RET
```

$k(I_1) = k(\text{MOV}) + k(\text{ADD}) + \ldots + k(\text{RET})$

Static-cost(trace) $= k(I_1) + \ldots + k(I_4)$

Dynamic-cost(trace) $=$ clock(Final_1) - clock(Start_1)

Theorem: Static-cost(trace) $=$ Dynamic-cost(trace)

SEVENTH FRAMEWORK PROGRAMME

Static and dynamic costs II

This proof is surprisingly tricky to complete ($\approx$ 3 man months)

- Issues
  - Finite memory
    the PC can overflow during fetching and/or execution
  - Variable length instructions
    not all addresses corresponds to instruction boundaries
- Requires predicates defining 'good program' and 'good program counter'
- Is now about 95% complete

Outline

Changes ported to OCaml prototype

- Bug fixes spotted in the formalisation so far have been merged back into the OCaml compiler
- Larger changes like the `Joint` machinery have so far not
- It is unclear whether they will be
- Just a generalisation of what is already there
- Supposed to make formalisation easier
- Further, we want to ensure that the untrusted compiler is as correct as possible, for experiments in e.g. Frama-C
- Porting a large change to the untrusted compiler would jeopardise these experiments

Improvements in Matita

- Part of the motivation for using Matita was for CerCo to act a 'stress test'
- The proofs talked about in this talk have done this
- Many improvements to Matita have been made since the last project meeting
- These include major speed-ups of e.g. printing large goals, bug fixes, the porting of CerCo code to standard library, and more options for passing to tactics

SEVENTH FRAMEWORK PROGRAMME

The next period

UNIBO has following pool of remaining manpower (postdocs):

Person	Man months remaining
Boender	10 months
Mulligan	6 months
Tranquilli	10 months

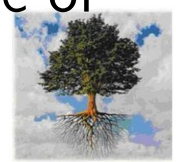
- Boender finishing assembler correctness proof
- Mulligan proofs of correctness for 1 intermediate language
- Tranquilli proofs of correctness for 2 intermediate languages
- Sacerdoti Coen 'floating'
- Believe we have enough manpower to complete backend (required 21 man months)

SEVENTH FRAMEWORK PROGRAMME

Summary

We have:

- Translated the OCaml prototype's backend intermediate languages into Matita
- Implemented the translations between languages, and given the intermediate languages a semantics
- Refactored many of the backend intermediate languages into a common, parametric 'joint' language, that is later specialised
- Spotted opportunities for possibly commuting backend translation passes
- Used six months to define structured traces and start the proof of correctness for the assembler
- Distinguished our proof from CompCert's by heavy use of dependent types throughout whole compiler



SEVENTH FRAMEWORK PROGRAMME