CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D4.3
Executable formal semantics of back-end intermediate languages

Version 1.1

Main Authors:
Dominic P. Mulligan and Claudio Sacerdoti Coen

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

Abstract We describe the encoding in the Calculus of Constructions of the semantics of the CerCo compiler's back-end intermediate languages. The CerCo back-end consists of five distinct languages: RTL, RTLntl, ERTL, LTL and LIN. We describe a process of heavy abstraction of the intermediate languages and their semantics. We hope that this process will ease the burden of Deliverable D4.4, the proof of correctness for the compiler.

Contents

1 Task

The Grant Agreement states that Task T4.3, entitled 'Formal semantics of intermediate languages' has associated Deliverable D4.3, consisting of the following:

> Executable Formal Semantics of back-end intermediate languages: This prototype is the formal counterpart of deliverable D2.1 for the back end side of the compiler and validates it.

This report details our implementation of this deliverable.

1.1 Connections with other deliverables

Deliverable D4.3 enjoys a close relationship with three other deliverables, namely deliverables D2.2, D4.3 and D4.4.

Deliverable D2.2, the OCaml implementation of a cost preserving compiler for a large subset of the C programming language, is the basis upon which we have implemented the current deliverable. In particular, the architecture of the compiler, its intermediate languages and their semantics, and the overall implementation of the Matita encodings has been taken from the OCaml compiler. Any variations from the OCaml design are due to bugs identified in the prototype compiler during the Matita implementation, our identification of code that can be abstracted and made generic, or our use of Matita's much stronger type system to enforce invariants through the use of dependent types.

Deliverable D4.2 can be seen as a 'sister' deliverable to the deliverable reported on herein. In particular, where this deliverable reports on the encoding in the Calculus of Constructions of the back-end semantics, D4.2 is the encoding in the Calculus of Constructions of the mutual translations of those languages. As a result, a substantial amount of Matita code is shared between the two deliverables.

Deliverable D4.4, the back-end correctness proofs, is the immediate successor of this deliverable.

2 The back-end intermediate languages' semantics in Matita

2.1 Abstracting related languages

As mentioned in the report for Deliverable D4.2, a systematic process of abstraction, over the OCaml code, has taken place in the Matita encoding. In particular, we have merged many of the syntaxes of the intermediate languages (i.e. RTL, ERTL, LTL and LIN) into a single 'joint' syntax, which is parameterised by various types. Equivalent intermediate languages to those present in the OCaml code can be recovered by specialising this joint structure.

As mentioned in the report for Deliverable D4.2, there are a number of advantages that this process of abstraction brings, from code reuse to allowing us to get a clearer view of the intermediate languages and their structure. However, the semantics of the intermediate languages allow us to concretely demonstrate this improvement in clarity, by noting that the semantics of the LTL and the semantics of the LIN languages are identical. In particular, the semantics of both LTL and LIN are implemented in exactly the same way. The only difference between the two languages is how the next instruction to be interpreted is fetched.

In LTL, this involves looking up in a graph, whereas in LTL, this involves fetching from a list of instructions.

As a result, we see that the semantics of LIN and LTL are both instances of a single, more general language that is parametric in how the next instruction is fetched. Furthermore, any prospective proof that the semantics of LTL and LIN are identical is now almost trivial, saving a deal of work in Deliverable D4.4.

2.2 Type parameters, and their purpose

We mentioned in the Deliverable D4.2 report that all joint languages are parameterised by a number of types, which are later specialised to each distinct intermediate language. As this parameterisation process is also dependent on designs decisions in the language semantics, we have so far held off summarising the role of each parameter.

We begin the abstraction process with the `params__` record. This holds the types of the representations of the different register varieties in the intermediate languages:

```
record params__: Type[1] :=
{
  acc_a_reg: Type[0];
  acc_b_reg: Type[0];
  dpl_reg: Type[0];
  dph_reg: Type[0];
  pair_reg: Type[0];
  generic_reg: Type[0];
  call_args: Type[0];
  call_dest: Type[0];
  extend_statements: Type[0]
}.
```

We summarise what these types mean, and how they are used in both the semantics and the translation process:

Type	Explanation
`acc_a_reg`	The type of the accumulator A register. In some languages this is implemented as the hardware accumulator, whereas in others this is a pseudoregister.
`acc_b_reg`	Similar to the accumulator A field, but for the processor's auxilliary accumulator, B.
`dpl_reg`	The type of the representation of the low eight bit register of the MCS-51's single 16 bit register, DPL. Can be either a pseudoregister or the hardware DPL register.
`dph_reg`	Similar to the DPL register but for the eight high bits of the 16-bit register.
`pair_reg`	Various different 'move' instructions have been merged into a single move instruction in the joint language. A value can either be moved to or from the accumulator in some languages, or moved to and from an arbitrary pseudoregister in others. This type encodes how we should move data around the registers and accumulators.
`generic_reg`	The representation of generic registers (i.e. those that are not devoted to a specific task).
`call_args`	The actual arguments passed to a function. For some languages this is simply the number of arguments passed to the function.
`call_dest`	The destination of the function call.
`extend_statements`	Instructions that are specific to a particular intermediate language, and which cannot be abstracted into the joint language.

As mentioned in the report for Deliverable D4.2, the record `params__` is enough to be able to specify the instructions of the joint languages:

```
inductive joint_instruction (p: params__) (globals: list ident): Type[0] :=
  | COMMENT: String → joint_instruction p globals
  | COST_LABEL: costlabel → joint_instruction p globals
  ...
  | OP1: Op1 → acc_a_reg p → acc_a_reg p → joint_instruction p globals
  | COND: acc_a_reg p → label → joint_instruction p globals
  ...
```

Here, we see that the instruction `OP1` (a unary operation on the accumulator A) can be given quite a specific type, through the use of the `params__` data structure.

Joint statements can be split into two subclasses: those who simply pass the flow of control onto their successor statement, and those that jump to a potentially remote location in the program. Naturally, as some intermediate languages are graph based, and others linearised, the passing act of passing control on to the 'successor' instruction can either be the act of following a graph edge in a control flow graph, or incrementing an index into a list. We make a distinction between instructions that pass control onto their immediate successors, and those that jump elsewhere in the program, through the use of `succ`, denoting the immediate successor of the current instruction, in the `params_` record described below.

```
record params_: Type[1] :=
{
  pars__ :> params__;
  succ: Type[0]
}.
```

The type `succ` corresponds to labels, in the case of control flow graph based languages, or is instantiated to the unit type for the linearised language, LIN. Using `param_` we can define statements of the joint language:

```
inductive joint_statement (p:params_) (globals: list ident): Type[0] :=
  | sequential: joint_instruction p globals → succ p → joint_statement p globals
  | GOTO: label → joint_statement p globals
  | RETURN: joint_statement p globals.
```

Note that in the joint language, instructions are 'linear', in that they have an immediate successor. Statements, on the other hand, consist of either a linear instruction, or a `GOTO` or `RETURN` statement, both of which can jump to an arbitrary place in the program. The conditional jump instruction COND is 'linear', since it has an immediate successor, but it also takes an arbitrary location (a label) to jump to.

For the semantics, we need further parametererised types. In particular, we parameterise the result and parameter type of an internal function call in `params0`:

```
record params0: Type[1] :=
{
  pars__' :> params__;
  resultT: Type[0];
  paramsT: Type[0]
}.
```

Here, `paramsT` and `resultT` typically are the (pseudo)registers that store the parameters and result of a function.

We further extend `params0` with a type for local variables in internal function calls:

```
record params1 : Type[1] :=
{
  pars0 :> params0;
  localsT: Type[0]
}.
```

Again, we expand our parameters with types corresponding to the code representation (either a control flow graph or a list of statements). Further, we hypothesise a generic method for looking up the next instruction in the graph, called `lookup`. Note that `lookup` may fail, and returns an `option` type:

```
record params (globals: list ident): Type[1] :=
{
  succ_ : Type[0];
  pars1 :> params1;
  codeT : Type[0];
  lookup: codeT → label → option (joint_statement (mk_params_ pars1 succ_) globals)
}.
```

We now have what we need to define internal functions for the joint language. The first two 'universe' fields are only used in the compilation process, for generating fresh names, and do not affect the semantics. The rest of the fields affect both compilation and semantics. In particular, we have a description of the result, parameters and the local variables of a function. Note also that we have lifted the hypothesised `lookup` function from `params` into a dependent sigma type, which combines a label (the entry and exit points of the control flow graph or list) combined with a proof that the label is in the graph structure:

```
record joint_internal_function (globals: list ident) (p:params globals) : Type[0] :=
{
  joint_if_luniverse: universe LabelTag;
  joint_if_runiverse: universe RegisterTag;
  joint_if_result : resultT p;
  joint_if_params : paramsT p;
  joint_if_locals : localsT p;
  joint_if_stacksize: nat;
  joint_if_code   : codeT ... p;
  joint_if_entry  : Σl: label. lookup ... joint_if_code l ≠ None ?;
  joint_if_exit   : Σl: label. lookup ... joint_if_code l ≠ None ?
}.
```

Naturally, a question arises as to why we have chosen to split up the parameterisation into so many intermediate records, each slightly extending earlier ones. The reason is because some intermediate languages share a host of parameters, and only differ on some others. For instance, in instantiating the ERTL language, certain parameters are shared with RTL, whilst others are ERTL specific:

```
...
definition ertl_params__: params__ :=
 mk_params__ register register register register (move_registers × move_registers)
  register nat unit ertl_statement_extension.
...
definition ertl_params1: params1 := rtl_ertl_params1 ertl_params0.
definition ertl_params: ∀globals. params globals := rtl_ertl_params ertl_params0.
...
definition ertl_statement := joint_statement ertl_params_.

definition ertl_internal_function :=
  λglobals.joint_internal_function ... (ertl_params globals).
```

Here, `rtl_ertl_params1` are the common parameters of the ERTL and RTL languages:

```
definition rtl_ertl_params1 := λpars0. mk_params1 pars0 (list register).
```

The record `more_sem_params` bundles together functions that store and retrieve values in various forms of register:

```
record more_sem_params (p:params_): Type[1] :=
{
  framesT: Type[0];
  empty_framesT: framesT;

  regsT: Type[0];
  empty_regsT: regsT;

  call_args_for_main: call_args p;
  call_dest_for_main: call_dest p;

  greg_store_: generic_reg p → beval → regsT → res regsT;
  greg_retrieve_: regsT → generic_reg p → res beval;
  acca_store_: acc_a_reg p → beval → regsT → res regsT;
  acca_retrieve_: regsT → acc_a_reg p → res beval;
  ...
```

```
  dpl_store_: dpl_reg p → beval → regsT → res regsT;
  dpl_retrieve_: regsT → dpl_reg p → res beval;
  ...
  pair_reg_move_: regsT → pair_reg p → res regsT;
}.
```

Here, the fields `empty_framesT`, `empty_regsT`, `call_args_for_main` and `call_dest_for_main` are used for state initialisation.

The fields `greg_store_` and `greg_retrieve_` store and retrieve values from a generic register, respectively. Similarly, `pair_reg_move` implements the generic move instruction of the joint language. Here `framesT` is the type of stack frames, with `empty_framesT` an empty stack frame.

The two hypothesised values `call_args_for_main` and `call_dest_for_main` deal with problems with the `main` function of the program, and how it is handled. In particular, we need to know when the `main` function has finished executing. But this is complicated, in C, by the fact that the `main` function is explicitly allowed to be recursive (disallowed in C++). Therefore, to understand whether the exiting `main` function is really exiting, or just recursively calling itself, we need to remember the address to which `main` will return control once the initial call to `main` has finished executing. This is done with `call_dest_for_main`, whereas `call_args_for_main` holds the `main` function's arguments.

We extend `more_sem_params` with yet more parameters via `more_sem_params2`:

```
record more_sem_params1 (globals: list ident) (p: params globals) : Type[1] :=
{
  more_sparams1 :> more_sem_params p;

  succ_pc: succ p → address → res address;
  pointer_of_label: genv ... p → pointer →
    label → res (Σp:pointer. ptype p = Code);
  ...
  fetch_statement:
    genv ... p → state (mk_sem_params ... more_sparams1) →
    res (joint_statement (mk_sem_params ... more_sparams1) globals);
  ...
  save_frame:
    address → nat → paramsT ... p → call_args p → call_dest p →
    state (mk_sem_params ... more_sparams1) →
    res (state (mk_sem_params ... more_sparams1));
  pop_frame:
    genv globals p → state (mk_sem_params ... more_sparams1) →
    res ((state (mk_sem_params ... more_sparams1)));
  ...
  set_result:
    list val → state (mk_sem_params ... more_sparams1) →
    res (state (mk_sem_params ... more_sparams1));
  exec_extended:
    genv globals p → extend_statements (mk_sem_params ... more_sparams1) →
    succ p → state (mk_sem_params ... more_sparams1) →
    IO io_out io_in (trace × (state (mk_sem_params ... more_sparams1)))
 }.
```

The field `succ_pc` takes an address, and a 'successor' label, and returns the address of the

instruction immediately succeeding the one at hand.

Here, `fetch_statement` fetches the next statement to be executed. The fields `save_frame` and `pop_frame` manipulate stack frames. In particular, `save_frame` creates a new stack frame on the top of the stack, saving the destination and parameters of a function, and returning an updated state. The field `pop_frame` destructively pops a stack frame from the stack, returning an updated state. Further, `set_result` saves the result of the function computation, and `exec_extended` is a function that executes the extended statements, peculiar to each individual intermediate language.

We bundle `params` and `sem_params` together into a single record. This will be used in the function `eval_statement` which executes a single statement of the joint language:

```
record sem_params2 (globals: list ident): Type[1] :=
{
  p2 :> params globals;
  more_sparams2 :> more_sem_params2 globals p2
}.
```

The `state` record holds the current state of the interpreter:

```
record state (p: sem_params): Type[0] :=
{
  st_frms: framesT ? p;
  pc: address;
  sp: pointer;
  isp: pointer;
  carry: beval;
  regs: regsT ? p;
  m: bemem
}.
```

Here `st_frms` represent stack frames, `pc` the program counter, `sp` the stack pointer, `isp` the internal stack pointer, `carry` the carry flag, `regs` the registers (hardware and pseudoregisters) and `m` external RAM. Note that we have two stack pointers, as we have two stacks: the physical stack of the MCS-51 microprocessor, and an emulated stack in external RAM. The MCS-51's own stack is minuscule, therefore it is usual to emulate a much larger, more useful stack in external RAM. We require two stack pointers as the MCS-51's `PUSH` and `POP` instructions manipulate the physical stack, and not the emulated one.

We use the function `eval_statement` to evaluate a single joint statement:

```
definition eval_statement:
  ∀globals: list ident.∀p:sem_params2 globals.
    genv globals p → state p → IO io_out io_in (trace × (state p)) :=
...
```

We examine the type of this function. Note that it returns a monadic action, `IO`, denoting that it may have an IO *side effect*, where the program reads or writes to some external device or memory address. Monads and their use are further discussed in Subsection 2.3. Further, the function returns a new state, updated by the single step of execution of the program. Finally, a *trace* is also returned, which records externally observable 'events', such as the calling of external functions and the emission of cost labels.

2.3 Use of monads

Monads are a categorical notion that have recently gained an amount of traction in functional programming circles. In particular, it was noted by Moggi that monads could be used to sequence *effectful* computations in a pure manner. Here, 'effectful computations' cover a lot of ground, from writing to files, generating fresh names, or updating an ambient notion of state.

A monad can be characterised by the following:

- A data type, M. For instance, the `option` type in OCaml or Matita.
- A way to 'inject' or 'lift' pure values into this data type (usually called `return`). We call this function `return` and say that it must have type $\alpha \to M\alpha$, where M is the name of the monad. In our example, the 'lifting' function for the `option` monad can be implemented as:

```
let return x = Some x
```

- A way to 'sequence' monadic functions together, to form another monadic function, usually called `bind`. Bind has type $M\alpha \to (\alpha \to M\beta) \to M\beta$. We can see that bind 'unpacks' a monadic value, applies a function after unpacking, and 'repacks' the new value in the monad. In our example, the sequencing function for the `option` monad can be implemented as:

```
let bind o f =
  match o with
    None -> None
    Some s -> f s
```

- A series of algebraic laws that relate `return` and `bind`, ensuring that the sequencing operation 'does the right thing' by retaining the order of effects. These *monad laws* should also be useful in reasoning about monadic computations in the proof of correctness of the compiler.

In the semantics of both front and back-end intermediate languages, we make use of monads. This monadic infrastructure is shared between the front-end and back-end languages.

In particular, an 'IO' monad, signalling the emission of a cost label, or the calling of an external function, is heavily used in the semantics of the intermediate languages. Here, the monad's sequencing operation ensures that cost label emissions and function calls are maintained in the correct order. We have already seen how the `eval_statement` function of the joint language is monadic, with type:

```
definition eval_statement:
  ∀globals: list ident.∀p:sem_params2 globals.
    genv globals p → state p → IO io_out io_in (trace × (state p)) :=
...
```

If we examine the body of `eval_statement`, we may also see how the monad sequences effects. For instance, in the case for the `LOAD` statement, we have the following:

```
definition eval_statement:
  ∀globals: list ident. ∀p:sem_params2 globals.
```

```
  genv globals p → state p → IO io_out io_in (trace × (state p)) :=
λglobals, p, ge, st.
...
match s with
| LOAD dst addrl addrh ⇒
  ! vaddrh ← dph_retrieve ... st addrh;
  ! vaddrl ← dpl_retrieve ... st addrl;
  ! vaddr ← pointer_of_address ⟨vaddrl,vaddrh⟩;
  ! v ← opt_to_res ... (msg FailedLoad) (beloadv (m ... st) vaddr);
  ! st ← acca_store p ... dst v st;
  ! st ← next ... l st ;
    ret ? ⟨E0, st⟩
```

Here, we employ a certain degree of syntactic sugaring. The syntax

```
  ...
! vaddrh ← dph_retrieve ... st addrh;
! vaddrl ← dpl_retrieve ... st addrl;
  ...
```

is sugaring for the `IO` monad's binding operation. We can expand this sugaring to the following much more verbose code:

```
  ...
  bind (dph_retrieve ... st addrh) (λvaddrh. bind (dpl_retrieve ... st addrl)
    (λvaddrl. ...))
```

Note also that the function `ret` is implementing the 'lifting', or return function of the `IO` monad.

We believe the sugaring for the monadic bind operation makes the program much more readable, and therefore easier to reason about. In particular, note that the functions `dph_retrieve`, `pointer_of_address`, `acca_store` and `next` are all monadic.

Note, however, that inside this monadic code, there is also another monad hiding. The `res` monad signals failure, along with an error message. The monad's sequencing operation ensures the order of error messages does not get rearranged. The function `opt_to_res` lifts an option type into this monad, with an error message to be used in case of failure. The `res` monad is then coerced into the `IO` monad, ensuring the whole code snippet typechecks.

2.4 Memory models

Currently, the semantics of the front and back-end intermediate languages are built around two distinct memory models. The front-end languages reuse the CompCert 1.6 memory model, whereas the back-end languages employ a version tailored to their needs. This split between the memory models reflects the fact that the front-end and back-end languages have different requirements from their memory models.

In particular, the CompCert 1.6 memory model places quite heavy restrictions on where in memory one can read from. To read a value in this memory model, you must supply an address, complete with the size of 'chunk' to read following that address. The read is only successful if you attempt to read at a genuine 'value boundary', and read the appropriate amount of memory for that value. As a result, with that memory model you are unable to read the third byte of a 32-bit integer value directly from memory, for instance. This has some consequences for the compiler, namely an inability to write a `memcpy` routine.

However, the CerCo memory model operates differently, as we need to move data 'piecemeal' between stacks in the back-end of the compiler. As a result, the back-end memory model allows one to read data at any memory location, not just on value boundaries. This has the advantage that we can successfully give a semantics to a `memcpy` routine in the back-end of the CerCo compiler (remembering that `memcpy` is nothing more than 'read a byte, copy a byte' repeated in a loop), an advantage over CompCert. However, the front-end of CerCo cannot because its memory model and values are the similar to CompCert 1.6.

More recent versions of CompCert's memory model have evolved in a similar direction, with a byte-by-byte representation of memory blocks. However, there remains an important difference in the handling of pointer values in the rest of the formalisation. In particular, in CompCert 1.10 only complete pointer values can be loaded in all of the languages in the compiler, whereas in CerCo we need to represent individual bytes of a pointer in the back-end to support our 8-bit target architecture.

Right now, the two memory models are interfaced during the translation from RTLabs to RTL. It is an open question whether we will unify the two memory models, using only the back-end, bespoke memory model throughout the compiler, as the CompCert memory model seems to work fine for the front-end, where such byte-by-byte copying is not needed. However, should we decide to port the front-end to the new memory model, it has been written in such an abstract way that doing so would be relatively straightforward.

3 Future work

Most things related to external function calls are currently axiomatised. This is due to there being a difficulty with how stackframes are handled with external function calls. We leave this for further work, due to there being no pressing need to implement this feature at the present time.

There is also, as mentioned, an open problem as to whether the front-end languages should use the same memory model as the back-end languages, as opposed to reusing the CompCert memory model. Should this decision be taken, this will likely be straightforward but potentially time consuming[1].

4 Code listing

4.1 Listing of files

Syntax specific files are presented in Table 1 (files relating to language translations omitted). Here, the OCaml column denotes the OCaml source file(s) in the prototype compiler's implementation that corresponds to the Matita script in question. The ratios are the linecounts of the Matita file divided by the line counts of the corresponding OCaml file. These are computed with `wc -l`, a standard Unix tool.

Individual file's ratios are an over approximation, due to the fact that it's hard to relate an individual OCaml file to the abstracted Matita code that has been spread across multiple

[1] After the original version of this deliverable was written we ported the front-end languages' semantics to the back-end memory model. This turned out not to be time consuming, and moreover used definitions linking front-end and back-end values that are required for the correctness proofs anyway. However, the front-end still cannot give a semantics to `memcpy`, for the same reason as CompCert 1.10; the language currently has no representation for a single byte of a pointer.

files. The ratio between total Matita code lines and total OCaml code lines is more reflective of the compressed and abstracted state of the Matita translation.

Semantics specific files are presented in Table 2.

Description	Matita	Lines	OCaml	Lines	Ratio
Abstracted syntax for back-end languages	`joint/Joint.ma`	173	N/A	N/A	N/A
The syntax of RTLabs	`RTLabs/syntax.ma`	73	`RTLabs/RTLabs.mli`	113	0.65
The syntax of RTL	`RTL/RTL.ma`	49	`RTL/RTL.mli`	120	1.85[a]
The syntax of ERTL	`ERTL/ERTL.ma`	25	`ERTL/ERTL.mli`	191	1.04[a]
The syntax of the abstracted combined LTL and LIN language	`LIN/joint_LTL_LIN.ma`	10	N/A	N/A	N/A
The specialisation of the above file to the syntax of LTL	`LTL/LTL.ma`	10	`LTL/LTL.mli`	104	1.86[b]
The specialisation of the above file to the syntax of LIN	`LIN/LIN.ma`	17	`LIN/LIN.mli`	88	2.27[b]

[a] After inlining of `joint/Joint.ma`.
[b] After inlining of `joint/Joint_LTL_LIN.ma` and `joint/Joint.ma`.

Total lines of Matita code for the above files:	347
Total lines of OCaml code for the above files:	616
Ratio of total lines:	0.56

Table 1: Syntax specific files in the intermediate language semantics

Description	Matita	Lines	OCaml	Lines	Ratio
Semantics of the abstracted languages	`joint/semantics.ma`	434	N/A	N/A	N/A
Generic utilities used in semantics 'joint' languages	`joint/SemanticUtils.ma`	70	N/A	N/A	N/A
Semantics of RTLabs	`RTLabs/semantics.ma`	223	`RTLabs/RTLabsInterpret.ml`	355	0.63
Semantics of RTL	`RTL/semantics.ma`	173	`RTL/RTLInterpret.ml`	324	2.01[a]
Semantics of ERTL	`ERTL/semantics.ma`	130	`ERTL/ERTLInterpret.ml`	504	1.26[a]
Semantics of the joint LTL-LIN language	`LIN/joint_LTL_LIN_semantics.ma`	67	N/A	N/A	N/A
Semantics of LTL	`LTL/semantics.ma`	5	`LTL/LTLInterpret.ml`	416	1.38[b]
Semantics of LIN	`LIN/semantics.ma`	43	`LIN/LINInterpret.ml`	379	1.62[b]

a Includes `joint/semantics.ma` and `joint/SemanticUtils.ma`.
b Includes `joint/semantics.ma`, `joint/SemanticUtils.ma` and `joint/joint_LTL_LIN_semantics.ma`.

Total lines of Matita code for the above files:	1145
Total lines of OCaml code for the above files:	1978
Ration of total lines:	0.58

Table 2: Semantics specific files in the intermediate language semantics

4.2 Listing of important functions and axioms

We list some important functions and axioms in the back-end semantics:

From RTLabs/semantics.ma

Title	Description
`make_initial_state`	Build an initial state
`eval_statement`	Evaluate a single RTLabs statement
`is_final`	Check whether a state is in a 'final' configuration
`RTLabs_exec`	Execute an RTLabs program

From RTL/semantics.ma

Title	Description
`rtl_exec_extended`	Execute a single step of the RTL language's extended instructions
`rtl_fullexec`	Execute an RTL program

From ERTL/semantics.ma

Title	Description
`ertl_exec_extended`	Execute a single step of the ERTL language's extended instructions
`ertl_fullexec`	Execute an ERTL program

From LTL/semantics.ma

Title	Description
`ltl_fullexec`	Execute an LTL program

From LIN/semantics.ma

Title	Description
`lin_fullexec`	Execute a LIN program

From LIN/joint_LTL_LIN_semantics.ma

Title	Description
`ltl_lin_exec_extended`	Execute a single step of the joint LTL-LIN language's extended instructions
`ltl_lin_fullexec`	Execute a joint LTL-LIN language program

From joint/semantics.ma

Title	Description
`eval_statement`	Evaluate a single joint language statement
`is_final`	Check whether a state is in a 'final' configuration
`joint_fullexec`	Execute a joint language program

From joint/SemanticUtils.ma

Title	Description
`graph_fetch_statement`	Fetch a statement from a control flow graph