CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D4.2
CIC encoding: Back-end

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 intermediate languages for the CerCo compiler. The CerCo backend consists of four distinct intermediate languages—RTL, ERTL, LTL and LIN, respectively—though we also handle the translations from the last frontend intermediate language, RTLabs, into RTL, and the translation of LIN into assembly language. Where feasible, we have made use of dependent types to enforce invariants and to make otherwise partial functions total.

Contents

1 Task

The Grant Agreement states that Task T4.2, entitled 'Functional encoding in the Calculus of Constructions' has associated Deliverable D4.2, consisting of the following:

> CIC encoding: Back-end: Functional Specification in the internal language of the Proof Assistant (the Calculus of Inductive Construction) of the back end of the compiler. This unit is meant to be composable with the front-end of deliverable D3.2, to obtain a full working compiler for Milestone M2. A first validation of the design principles and implementation choices for the Untrusted Cost-annotating OCaml Compiler D2.2 is achieved and reported in the deliverable, possibly triggering updates of the Untrusted Cost-annotating OCaml Compiler sources.

This report details our implementation of this deliverable.

1.1 Connections with other deliverables

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

Deliverable D2.2, the O'Caml 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 the overall implementation of the Matita encodings has been taken from the O'Caml compiler. Any variations from the O'Caml 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 D4.3 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 backend translations, D4.3 is the encoding in the Calculus of Constructions of the semantics of those languages. As a result, a substantial amount of Matita code is shared between the two deliverables.

Deliverable D4.4, the backend correctness proofs, is the immediate successor of this deliverable.

1.2 A brief overview of the backend compilation chain

The Matita compiler's backend consists of five distinct intermediate languages: RTL, RTLntl, ERTL, LTL and LIN. A sixth language, RTLabs, serves as the entry point of the backend and the exit point of the frontend. RTL, RTLntl, ERTL and LTL are 'control flow graph based' languages, whereas LIN is a linearised language, the final language before translation to assembly.

We now briefly discuss the properties of the intermediate languages, and discuss the various transformations that take place during the translation process:

RTLabs ((Abstract) Register Transfer Language) As mentioned, this is the final language of the compiler's frontend and the entry point for the backend. This language uses

pseudoregisters, not hardware registers.[1] Functions still use stackframes, where arguments are passed on the stack and results are stored in addresses. During the pass to RTL instruction selection is carried out.

RTL (Register Transfer Language) This language uses pseudoregisters, not hardware registers. Tailcall elimination is carried out during the translation from RTL to RTLntl.

RTLntl (Register Transfer Language — No Tailcalls) This language is a pseudoregister, graph based language where all tailcalls are eliminated. RTLntl is not present in the O'Caml compiler, the RTL language is reused for this purpose.

ERTL (Explicit Register Transfer Language) This is a language very similar to RTLntl. However, the calling convention is made explicit, in that functions no longer receive and return inputs and outputs via a high-level mechanism, but rather use stack slots or hadware registers. The ERTL to LTL pass performs the following transformations: liveness analysis, register colouring and register/stack slot allocation.

LTL (Linearisable Transfer Language) Another graph based language, but uses hardware registers instead of pseudoregisters. Tunnelling (branch compression) should be implemented here.

LIN (Linearised) This is a linearised form of the LTL language; function graphs have been linearised into lists of statements. All registers have been translated into hardware registers or stack addresses. This is the final stage of compilation before translating directly into assembly language.

2 The backend intermediate languages in Matita

We now discuss the encoding of the compiler backend languages in the Calculus of Constructions proper. We pay particular heed to changes that we made from the O'Caml prototype. In particular, many aspects of the backend languages have been unified into a single 'joint' language. We have also made heavy use of dependent types to reduce 'spurious partiality' and to encode invariants.

2.1 Abstracting related languages

The O'Caml compiler is written in the following manner. Each intermediate language has its own dedicated syntax, notions of internal function, and so on. Here, we make a distinction between 'internal functions'—other functions that are explicitly written by the programmer—and 'external functions', which belong to external library and require explictly linking. In particular, IO can be seen as a special case of the 'external function' mechanism. Internal functions are represented as a record consisting of a sequential structure of statements, entry and exit points to this structure, and other book keeping devices. This sequential structure can either be a control flow graph or a linearised list of statements, depending on the language.

[1] There are an unbounded number of pseudoregisters. Pseudoregisters are converted to hardware registers or stack positions during register allocation.

Translations between intermediate language map syntaxes to syntaxes, and internal function representations to internal function representations explicitly.

This is a perfectly valid way to write a compiler, where everything is made explicit, but writing a *verified* compiler poses new challenges. In particular, we must look ahead to see how our choice of encodings will affect the size and complexity of the forthcoming proofs of correctness. We now discuss some abstractions, introduced in the Matita code, which we hope will make our proofs shorter, amongst other benefits.

Changes between languages made explicit Due to the bureaucracy inherent in explicating each intermediate language's syntax in the O'Caml compiler, it can often be hard to see exactly what changes between each successive intermediate language. By abstracting the syntax of the RTL, ERTL, LTL and LIN intermediate languages, we make these changes much clearer.

Our abstraction takes the following form:

```
inductive joint_instruction (p: params__) (globals: list ident): Type[0] :=
  | COMMENT: String → joint_instruction p globals
  ...
  | INT: generic_reg p → Byte → joint_instruction p globals
  ...
  | OP1: Op1 ightarrow acc_a_reg p ightarrow acc_a_reg p ightarrow joint_instruction p globals
  ...
  | extension: extend_statements p → joint_instruction p globals.
```

We first note that for the majority of intermediate languages, many instructions are shared. However, these instructions expect different register types (either a pseudoregister or a hardware register) as arguments. We must therefore parameterise the joint syntax with a record of parameters that will be specialised to each intermediate language. In the type above, this parameterisation is realised with the `params__` record. As a result of this parameterisation, we have also added a degree of 'type safety' to the intermediate languages' syntaxes. In particular, we note that the `OP1` constructor expects quite a specific type, in that the two register arguments must both be what passes for the accumulator A in that language. In some languages, for example LIN, this is the hardware accumulator, whilst in others this is any pseudoregister. Contrast this with the `INT` constructor, which expects a `generic_reg`, corresponding to an 'arbitrary' register type.

Further, we note that some intermediate languages have language specific instructions (i.e. the instructions that change between languages). We therefore add a new constructor to the syntax, `extension`, which expects a value of type `extend_statements p`. As `p` varies between intermediate languages, we can provide language specific extensions to the syntax of the joint language. For example, ERTL's extended syntax consists of the following extra statements:

```
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.
```

These are further packaged into an ERTL specific instance of `params__` as follows:

```
definition ertl_params__: params__ :=
  mk_params__ register register ... ertl_statement_extension.
```

Shared code, reduced proofs Many features of individual backend intermediate languages are shared with other intermediate languages. For instance, RTLabs, RTL, ERTL and LTL are all graph based languages, where functions are represented as a control flow graph of statements that form their bodies. Functions for adding statements to a graph, searching the graph, and so on, are remarkably similar across all languages, but are duplicated in the O'Caml code.

As a result, we chose to abstract the representation of internal functions for the RTL, ERTL, LTL and LIN intermediate languages into a 'joint' representation. This representation is parameterised by a record that dictates the layout of the function body for each intermediate language. For instance, in RTL, the layout is graph like, whereas in LIN, the layout is a linearised list of statements. Further, a generalised way of accessing the successor statement to the one currently under consideration is needed, and so forth.

Our joint internal function record looks like so:

```
record joint_internal_function (globals: list ident) (p:params globals) : Type[0] :=
{
  ...
  joint_if_params  : paramsT p;
  joint_if_locals  : localsT p;
  ...
  joint_if_code    : codeT ... p;
  ...
}.
```

In particular, everything that can vary between differing intermediate languages has been parameterised. Here, we see the location where to fetch parameters, the listing of local variables, and the internal code representation has been parameterised. Other particulars are also parameterised, though here omitted.

Hopefully this abstraction process will reduce the number of proofs that need to be written, dealing with internal functions of different languages characterised by parameters.

Dependency on instruction selection We note that the backend languages are all essentially 'post instruction selection languages'. The 'joint' syntax makes this especially clear. For instance, in the definition:

```
inductive joint_instruction (p:params__) (globals: list ident): Type[0] :=
  ...
  | INT: generic_reg p → Byte → joint_instruction p globals
  | MOVE: pair_reg p → joint_instruction p globals
  ...
  | PUSH: acc_a_reg p → joint_instruction p globals
  ...
  | extension: extend_statements p → joint_instruction p globals.
```

The capitalised constructors—`INT`, `MOVE`, and so on—are all machine specific instructions. Retargetting the compiler to another microprocessor, improving instruction selection, or simply enlarging the subset of the machine language that the compiler can use, would entail replacing these constructors with constructors that correspond to the instructions of the new target. We feel that this makes which instructions are target dependent, and which are not (i.e. those language specific instructions that fall inside the `extension` constructor) much more explicit.

In the long term, we would really like to try to directly embed the target language in the syntax, in order to reuse the target language's semantics.

Independent development and testing We have essentially modularised the intermediate languages in the compiler backend. As with any form of modularisation, we reap benefits in the ability to independently test and develop each intermediate language separately, with the benefit of fixing bugs just once.

Future reuse for other compiler projects Another advantage of our modularisation scheme is the ability to quickly use and reuse intermediate languages for other compiler projects. For instance, in creating a cost-preserving compiler for a functional language, we may choose to target a linearised version of RTL directly. Adding such an intermediate language would involve the addition of just a few lines of code.

Easy addition of new compiler passes Under our modularisation and abstraction scheme, new compiler passes can easily be injected into the backend. We have a concrete example of this in the RTLntl language, an intermediate language that was not present in the original O'Caml code. To specify a new intermediate language we must simply specify, through the use of the statement extension mechanism, what differs in the new intermediate language from the 'joint' language, and configure a new notion of internal function record, by specialising parameters, to the new language. As generic code for the 'joint' language exists, for example to add statements to control flow graphs, this code can be reused for the new intermediate language.

Possible commutations of translation passes The backend translation passes of the CerCo compiler differ quite a bit from the CompCert compiler. In the CompCert compiler, linearisation occurs much earlier in the compilation chain, and passes such as register colouring and allocation are carried out on a linearised form of program. Contrast this with our own approach, where the code is represented as a graph for much longer. Similarly, in CompCert the calling conventions are enforced after register allocation, whereas we do register allocation before enforcing the calling convention.

However, by abstracting the representation of intermediate functions, we are now much more free to reorder translation passes as we see fit. The linearisation process, for instance, now no longer cares about the specific representation of code in the source and target languages. It just relies on a common interface. We are therefore, in theory, free to pick where we wish to linearise our representation. This adds an unusual flexibility into the compilation process, and allows us to freely experiment with different orderings of translation passes.

2.2 Use of dependent types

We see several potential ways in which a compiler can fail to compile a program:

1. The program is malformed, and there is no hope of making sense of the program.
2. The compiler is buggy, or an invariant in the compiler is invalidated.
3. An incomplete heuristic in the compiler fails.

4. The compiled code exhausts some bounded resource, for instance the processor's code memory.

Standard compilers can fail for all the above reasons. Certified compilers are only required to rule out the second class of failures, but they can still fail for all the remaining reasons. In particular, a compiler that systematically refuses to compile any well formed program is a sound compiler. On the contrary, we would like our certified compiler to fail only in the fourth case. We plan to achieve this with the following strategy.

First, the compiler is abstracted over all incomplete heuristics, seen as total functions. To obtain executable code, the compiler is eventually composed with implementations of the abstracted strategies, with the composition taking care of any potential failure of the heuristics in finding a solution.

Second, we reject all malformed programs using dependent types: only well-formed programs should typecheck and the compiler can be applied only to well-typed programs.

Finally, exhaustion of bounded resources can be checked only at the very end of compilation. Therefore, all intermediate compilation steps are now total functions that cannot diverge, nor fail: these properties are directly guaranteed by the type system of Matita.

Presently, the plan is not yet fulfilled. However, we are improving the implemented code according to our plan. We are doing this by progressively strenghthening the code through the use of dependent types. We detail the different ways in which dependent types have been used so far.

First, we encode informal invariants, or uses of `assert false` in the O'Caml code, with dependent types, converting partial functions into total functions. There are numerous examples of this throughout the backend. For example, in the `RTLabs` to `RTL` transformation pass, many functions only 'make sense' when lists of registers passed to them as arguments conform to some specific length. For instance, the `translate_negint` function, which translates a negative integer constant:

```
definition translate_negint :=
  λglobals: list ident.
  λdestrs: list register.
  λsrcrs: list register.
  λstart_lbl: label.
  λdest_lbl: label.
  λdef: rtl_internal_function globals.
  λprf: |destrs| = |srcrs|. (* assert here *)
    ...
```

The last argument to the function, `prf`, is a proof that the lengths of the lists of source and destination registers are the same. This was an assertion in the O'Caml code.

Secondly, we make use of dependent types to make the Matita code correct by construction, and eventually the proofs of correctness for the compiler easier to write. For instance, many intermediate languages in the backend of the compiler, from RTLabs to LTL, are graph based languages. Here, function definitions consist of a graph (i.e. a map from labels to statements) and a pair of labels denoting the entry and exit points of this graph. Practically, we would always like to ensure that the entry and exit labels are present in the statement graph. We ensure that this is so with a dependent sum type in the `joint_internal_function` record, which all graph based languages specialise to obtain their own internal function representation:

```
record joint_internal_function (globals: list ident) (p: params globals): Type[0] :=
```

```
{
  ...
  joint_if_code   : codeT ... p;
  joint_if_entry  : Σl: label. lookup ... joint_if_code l ≠ None ...;
  ...
}.
```

Here, `codeT` is a parameterised type representing the 'structure' of the function's body (a graph in graph based languages, and a list in the linearised LIN language). Specifically, the `joint_if_entry` is a dependent pair consisting of a label and a proof that the label in question is a vertex in the function's graph. A similar device exists for the exit label.

We make use of dependent types also for another reason: experimentation. Namely, CompCert makes little use of dependent types to encode invariants. In contrast, we wish to make as much use of dependent types as possible, both to experiment with different ways of encoding compilers in a proof assistant, but also as a way of 'stress testing' Matita's support for dependent types.

Moreover, at the moment we make practically no use of inductive predicates to specify compiler invariants and to describe the semantics of intermediate languages. On the contrary, all predicates are computable functions. Therefore, the proof style that we will adopt will be necessarily significantly different from, say, CompCert's one. At the moment, in Matita 'Russell-'-style reasoning (in the sense of [Soz06]) seems to be the best solution for working with computable functions. This style is heavily based on the idea that all computable functions should be specified using dependent types to describe their pre- and post-conditions. As a consequence, it is natural to add dependent types almost everywhere in the Matita compiler's codebase.

2.3 What we do not implement

There are several classes of functionality that we have chosen not to implement in the backend languages:

- **Datatypes and functions over these datatypes that are not supported by the compiler.** In particular, the compiler does not support the floating point datatype, nor accompanying functions over that datatype. At the moment, frontend languages within the compiler possess constructors corresponding to floating point code. These are removed during instruction selection (in the RTLabs to RTL transformation) using a daemon.[2] However, at some point, we would like the front end of the compiler to recognise programs that use floating point code and reject them as being invalid.

- **Axiomatised components that will be implemented using external oracles.** Several large, complex pieces of compiler infrastructure, most noticably register colouring and fixed point calculation during liveness analysis have been axiomatised. This was already agreed upon before the start of the project, and is clearly marked in the project proposal, following comments by those involved with the CompCert project about the difficulty in formalising register colouring in that project. Instead, these components are axiomatised, along with the properties that they need to satisfy in order for the rest of the compilation chain to be correct. These axiomatised components are found in the ERTL to LTL pass.

[2] A Girardism. An axiom of type `False`, from which we can prove anything.

It should be noted that these axiomatised components fall into the following pattern: whilst their implementation is complex, and their proof of correctness is difficult, we are able to quickly and easily verify that any answer that they provide is correct. Therefore, we plan to provide implementations in OCaml only, and to provide certified verifiers in Matita. At the moment, the implementation of the certified verifiers is left as future work.

- **A few non-computational proof obligations.** A few difficult-to-close, but non-computational (i.e. they do not prevent us from executing the compiler inside Matita), proof obligations have been closed using daemons in the backend. These proof obligations originate with our use of dependent types for expressing invariants in the compiler. However, here, it should be mentioned that many open proof obligations are simply impossible to close until we start to obtain stronger invariants from the proof of correctness for the compiler proper. In particular, in the RTLabs to RTL pass, several proof obligations relating to lists of registers stored in a 'local environment' appear to fall into this pattern.
- **Branch compression (tunnelling).** This was a feature of the O'Caml compiler. It is not yet currently implemented in the Matita compiler. This feature is only an optimisation, and will not affect the correctness of the compiler.
- **'Real' tailcalls** For the time being, tailcalls in the backend are translated to 'vanilla' function calls during the ERTL to LTL pass. This follows the O'Caml compiler, which did not implement tailcalls, and did this simplification step. 'Real' tailcalls are being implemented in the O'Caml compiler, and when this implementation is complete, we aim to port this code to the Matita compiler.

3 Associated changes to O'Caml compiler

At the moment, only bugfixes, but no architectural changes we have made in the Matita backend have made their way back into the O'Caml compiler. We do not see the heavy process of modularisation and abstraction as making its way back into the O'Caml codebase, as this is a significant rewrite of the backend code that is supposed to yield the same code after instantiating parameters, anyway.

4 Future work

As mentioned in Section 2.3, there are several unimplemented features in the compiler, and several aspects of the Matita code that can be improved in order to make currently partial functions total. We summarise this future work here:

- We plan to make use of dependent types to identify 'floating point' free programs and make all functions total over such programs. This will remove a swathe of uses of daemons. This should be routine.
- We plan to move expansion of integer modulus, and other related functions, into the instruction selection (RTLabs to RTL) phase. This will also help to remove a swathe of uses of daemons, as well as potentially introduce new opportunities for optimisations that we currently miss in expanding these instructions at the C-light level.

- We plan to close all existing proof obligations that are closed using daemons, arising from our use of dependent types in the backend. However, many may not be closable until we have completed Deliverable D4.4, the certification of the whole compiler, as we may not have invariants strong enough at the present time.
- We plan to port the O'Caml compiler's implementation of tailcalls when this is completed, and eventually port the branch compression code currently in the O'Caml compiler to the Matita implementation. This should not cause any major problems.
- We plan to validate the backend translations, removing any obvious bugs, by executing the translation inside Matita on small C programs. This is not critical, as the certification process will find all bugs anyway.
- We plan to provide certified validators for all results provided by external oracles written in OCaml. At the moment, we have axiomatized oracles for computing least fixpoints during liveness analysis, for colouring registers and for branch displacement in the assembler code.

5 Code listing

5.1 Listing of files

Translation specific files (files relating to language semantics have been omitted) are presented in Table 1. Here, the O'Caml column denotes the O'Caml source file 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 O'Caml file(s). 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 O'Caml file to the abstracted Matita code that has been spread across multiple files. The ratio between total Matita code lines and total O'Caml code lines is more reflective of the compressed and abstracted state of the Matita translation.

Translations and utilities are presented in Table 2.

Given that Matita code is much more verbose than O'Caml code, with explicit typing and inline proofs, we have achieved respectable line count ratios in the translation. Some of these ratio, however, need explanation. In particular, the RTLabs to RTL translation stands out. Note that RTLabs is not subject to our abstraction process, and the language's syntax is fully explicated. Further, the RTLabs to RTL translation is quite involved, including instruction selection.

Other, longer translations include the file `ERTLtoLTL.ma`. This is actually the concatenation of the files `ERTLtoLTLI.ml` and `ERTLtoLTL.ml` in the O'Caml source, hence its length.

Further, many translations are actually significantly shorter than their O'Caml counterparts due to axiomatisation, and the lack of structure and functor declarations in Matita.

We note that the O'Caml backend codebase consists of 6770 lines of O'Caml code (including comments). The Matita codebase consists of 6447 lines of Matita code (including comments). This is a ratio of 0.95.

Description	Matita	Lines	O'Caml	Lines	Ratio
Abstracted syntax for backend 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 O'Caml code for the above files:	616
Ratio of total lines:	0.56

Table 1: Syntax files

Description	Matita	Lines	O'Caml	Lines	Ratio
Generic translation utilities	`joint/TranslateUtils.ma`	59	N/A	N/A	N/A
Translation from RTLabs to RTL	`RTLabs/RTLabsToRTL.ma`	1251	`RTLabs/RTLabsToRTL.ml`	778	1.61
Translation from RTL to ERTL	`RTL/RTLToERTL.ma`	417	`RTL/RTLToERTL.ml`	472	1.01[a]
Elimination of tailcalls	`RTL/RTLTailcall.ma`	52	`RTL/RTLtailcall.ml`	25	2.02
Translation from ERTL to LTL	`ERTL/ERTLToLTL.ma`	464	`ERTL/ERTLToLTL.ml`	429[b]	1.22[c]
Axiomatised graph colouring component	`ERTL/Interference.ma`	26	`common/interference.ml`	861	0.03[a]
Liveness analysis	`ERTL/liveness.ma`	298	`ERTL/liveness.ml`	323	0.92
Translation from LTL to LIN	`LTL/LTLToLIN.ma`	92	`LTL/LTLToLIN.ml`	302[d]	0.53[e]
Generic code for LTL and LIN languages	`LIN/joint_LTL_LIN.ma`	10	N/A	N/A	N/A
Translation from LIN to assembly	`LIN/LINToASM.ma`	335	`LIN/LINToASM.ml`	142	2.85[f]

[a] After inlining of `joint/TranslateUtils.ma`.
[b] After inlining of `ERTL/ERTLToLTLI.ml`.
[c] After inlining of `joint/TranslateUtils.ma` and `ERTL/ERTLToLTLI.ml`.
[d] After inlining of `LTL/LTLToLINI.ml`.
[e] After inlining of `joint/TranslateUtils.ma`, `LTL/LTLToLINI.ml` and `joint/joint_LTL_LIN.ma`.
[f] After inlining of `joint/TranslateUtils.ma`, and `joint/joint_LTL_LIN.ma`.

Total lines of Matita code for the above files:	3032.
Total lines of O'Caml code for the above files:	3332.
Ratio of total lines:	0.91.

Table 2: Translation files.

5.2 Listing of important functions and axioms

We list some important functions and axioms in the backend compilation:

From RTLabs/RTLabsToRTL.ma

Title	Description
`translate_stmt`	Translation of an RTLabs statement to an RTL statement
`translate_internal`	Translation of an RTLabs internal function to an RTL internal function
`rtlabs_to_rtl`	Translation of an RTLabs program to an RTL program

From joint/TranslateUtils.ma

Title	Description
`fresh_regs`	Generic fresh pseudoregister generation, for any intermediate language
`adds_graph`	Generic means of adding a statement to a graph, for any intermediate language

From RTL/RTLTailcall.ma

Title	Description
`simplify_statement`	Remove a single tailcall
`simplify_graph`	Remove all tailcalls in the function graph
`tailcall_simplify`	Simplify an RTL program by removing tailcalls

From RTL/RTLToERTL.ma

Title	Description
`translate_stmt`	Translation of an RTL statement to an ERTL statement
`translate_funct_internal`	Translation of an RTL internal function to an ERTL internal function
`translate`	Translation of an RTL program to an ERTL program

From ERTL/liveness.ma

Title	Description
`analyse`	Dead code analysis

From ERTL/Interference.ma

Title	Description
`build`	The (axiomatised) graph colouring for register and stack slot allocation

From ERTL/ERTLToLTL.ma

Title	Description
`translate_statement`	Translation of an ERTL statement into multiple LTL statements
`translate_internal`	Translation of an ERTL internal function into an LTL internal function
`ertl_to_ltl`	Translation of an ERTL program into an LTL program

From LTL/LTLToLIN.ma

Title	Description
`visit`	Visits, in order, every node in the statement graph for linearisation
`translate_stmt`	Translation of an LTL statement to a LIN statement
`branch_compress`	Place holder (currently identity) function for branch compression
`translate_internal`	Translation of an LTL internal function into a LIN internal function
`ltl_to_lin`	Translation of an LTL program to a LIN program

From LIN/LINToASM.ma

Title	Description
`translate_statements`	Translation of a LIN statement to mutliple assembly instructions
`translate_fun_def`	Translation of a LIN internal function definition into assembly
`translate`	Translation of a LIN program into assembly

References

[Soz06] M. Sozeau. Subset coercions in Coq. In *TYPES*, pages 237–252, 2006.