


CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D2.1
Compiler design and intermediate languages

Version 1.0 with Addendum

Main Authors:
Roberto M. Amadio, Nicolas Ayache, Yann Régis-Gianas

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

Abstract We discuss the problem of building a compiler which can *lift* in a provably correct way pieces of information on the execution cost of the object code to cost annotations on the source code. To this end, we need a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to prove them sound and precise, and (iii) the way such proofs can be composed. We propose a so-called *labelling* approach to these three questions. As a first step, we examine its application to a toy compiler. This formal study suggests that the labelling approach has good compositionality and scalability properties. In order to provide further evidence for this claim, we report our successful experience in implementing and testing the labelling approach on top of a prototype compiler written in ocaml for (a large fragment of) the C language.

Contents

1 Introduction

The formal description and certification of software components is reaching a certain level of maturity with impressing case studies ranging from compilers to kernels of operating systems. A well-documented example is the proof of functional correctness of a moderately optimizing compiler from a large subset of the C language to a typical assembly language of the kind used in embedded systems [9].

In the framework of the *Certified Complexity* (*CerCo*) project [2], we aim to refine this line of work by focusing on the issue of the *execution cost* of the compiled code. Specifically, we aim to build a formally verified C compiler that given a source program produces automatically a functionally equivalent object code plus an annotation of the source code which is a sound and precise description of the execution cost of the object code.

We target in particular the kind of C programs produced for embedded applications; these programs are eventually compiled to binaries executable on specific processors. The current state of the art in commercial products such as Scade [3, 6] is that the *reaction time* of the program is estimated by means of abstract interpretation methods (such as those developed by AbsInt [1, 5]) that operate on the binaries. These methods rely on a specific knowledge of the architecture of the processor and may require explicit annotations of the binaries to determine the number of times a loop is iterated (see, *e.g.*, [13] for a survey of the state of the art).

In this context, our aim is to produce a mechanically verified compiler which can *lift* in a provably correct way the pieces of information on the execution cost of the binary code to cost annotations on the source C code. Eventually, we plan to manipulate the cost annotations with automatic tools such as Frama − C [4] to infer more synthetic cost annotations. In order to carry on our project, we need a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to prove them sound and precise, and (iii) the way such proofs can be composed. Our purpose here is to propose a methodology addressing these three questions and to consider its concrete application to a simple toy compiler and to a moderately optimizing untrusted C compiler.

Meaning of cost annotations The execution cost of the source programs we are interested in depends on their control structure. Typically, the source programs are composed of mutually recursive procedures and loops and their execution cost depends, up to some multiplicative constant, on the number of times procedure calls and loop iterations are performed. Producing a *cost annotation* of a source program amounts to:

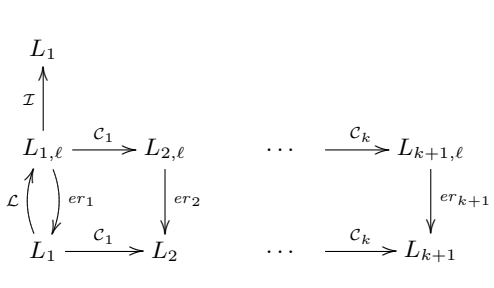
- enrich the program with a collection of *global cost variables* to measure resource consumption (time, stack size, heap size,...)
- inject suitable code at some critical points (procedures, loops,...) to keep track of the execution cost.

Thus producing a cost-annotation of a source program P amounts to build an *annotated program* $An(P)$ which behaves as P while self-monitoring its execution cost. In particular, if we do *not* observe the cost variables then we expect the annotated program $An(P)$ to be functionally equivalent to P. Notice that in the proposed approach an annotated program is a program in the source language. Therefore the meaning of the cost annotations is automatically defined by the semantics of the source language and tools developed to reason on the source programs can be directly applied to the annotated programs too.

Soundness and precision of cost annotations Suppose we have a functionally correct compiler $\mathcal{C}$ that associates with a program P in the source language a program $\mathcal{C}(P)$ in the object language. Further suppose we have some obvious way of defining the execution cost of an object code. For instance, we have a good estimate of the number of cycles needed for the execution of each instruction of the object code. Now the annotation of the source program $An(P)$ is *sound* if its prediction of the execution cost is an upper bound for the 'real' execution cost. Moreover, we say that the annotation is *precise* with respect to the cost model if the *difference* between the predicted and real execution costs is bounded by a constant which depends on the program.

Compositionality In order to master the complexity of the compilation process (and its verification), the compilation function $\mathcal{C}$ must be regarded as the result of the composition of a certain number of program transformations $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$. When building a system of cost annotations on top of an existing compiler a certain number of problems arise. First, the estimated cost of executing a piece of source code is determined only at the *end* of the compilation process. Thus while we are used to define the compilation functions $\mathcal{C}_i$ in increasing order (from left to right), the annotation function An is the result of a progressive abstraction from the object to the source code (from right to left). Second, we must be able to foresee in the source language the looping and branching points of the object code. Missing a loop may lead to unsound cost annotations while missing a branching point may lead to rough cost predictions. This means that we must have a rather good idea of the way the source code will eventually be compiled to object code. Third, the definition of the annotation of the source code depends heavily on *contextual information*. For instance, the cost of the compiled code associated with a simple expression such as $x + 1$ will depend on the place in the memory hierarchy where the variable x is allocated. A previous experience described in appendix D suggests that the process of pushing 'hidden parameters' in the definitions of cost annotations and of manipulating directly numerical cost is error prone and produces complex proofs. For this reason, we advocate next a 'labelling approach' where costs are handled at an abstract level and numerical values are produced at the very end of the construction.

Labelling approach to cost annotations The 'labelling' approach to the problem of building cost annotations is summarized in the following diagram.



$$\begin{array}{rcl} er_{i+1} \circ \mathcal{C}_i & = & \mathcal{C}_i \circ er_i \\ er_1 \circ \mathcal{L} & = & id_{L_1} \\ An & = & \mathcal{I} \circ \mathcal{L} \end{array}$$

For each language L_i considered in the compilation process, we define an extended *labelled* language $L_{i,\ell}$ and an extended operational semantics. The labels are used to mark certain points of the control. The semantics makes sure that whenever we cross a labelled control point a labelled and observable transition is produced.

For each labelled language there is an obvious function er_i erasing all labels and producing a program in the corresponding unlabelled language. The compilation functions $\mathcal{C}_i$ are extended

from the unlabelled to the labelled language so that they enjoy commutation with the erasure functions. Moreover, we lift the soundness properties of the compilation functions from the unlabelled to the labelled languages and transition systems.

A *labelling* $\mathcal{L}$ of the source language L_1 is just a function such that $er_{L_1} \circ \mathcal{L}$ is the identity function. An *instrumentation* $\mathcal{I}$ of the source labelled language $L_{1,\ell}$ is a function replacing the labels with suitable increments of, say, a fresh global *cost* variable. Then an *annotation* An of the source program can be derived simply as the composition of the labelling and the instrumentation functions: $An = \mathcal{I} \circ \mathcal{L}$.

Suppose s is some adequate representation of the state of a program. Let P be a source program and suppose that its annotation satisfies the following property:

$$(An(P), s[c/cost]) \Downarrow s'[c + \delta/cost] \tag{1}$$

where c and δ are some non-negative numbers. Then the definition of the instrumentation and the fact that the soundness proofs of the compilation functions have been lifted to the labelled languages allows to conclude that

$$(\mathcal{C}(\mathcal{L}(P)), s[c/cost]) \Downarrow (s'[c/cost], \lambda) \tag{2}$$

where $\mathcal{C} = \mathcal{C}_k \circ \cdots \circ \mathcal{C}_1$ and λ is a sequence (or a multi-set) of labels whose 'cost' corresponds to the number δ produced by the annotated program. Then the commutation properties of erasure and compilation functions allows to conclude that the *erasure* of the compiled labelled code $er_{k+1}(\mathcal{C}(\mathcal{L}(P)))$ is actually equal to the compiled code $\mathcal{C}(P)$ we are interested in. Given this, the following question arises: under which conditions the sequence λ, *i.e.*, the increment δ, is a sound and possibly precise description of the execution cost of the object code?

To answer this question, we observe that the object code we are interested in is some kind of assembly code and its control flow can be easily represented as a control flow graph. The fact that we have to prove the soundness of the compilation functions means that we have plenty of information on the way the control flows in the compiled code, in particular as far as procedure calls and returns are concerned. These pieces of information allow to build a rather accurate representation of the control flow of the compiled code at run time.

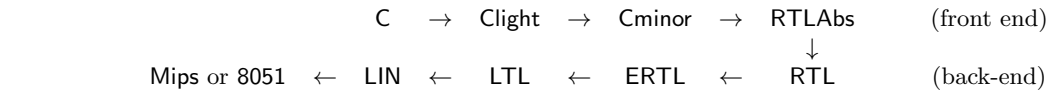
The idea is then to perform two simple checks on the control flow graph. The first check is to verify that all loops go through a labelled node. If this is the case then we can associate a finite cost with every label and prove that the cost annotations are sound. The second check amounts to verify that all paths starting from a label have the same cost. If this check is successful then we can conclude that the cost annotations are precise.

A toy compiler As a first case study for the labelling approach to cost annotations we have sketched, we introduce a *toy compiler* which is summarized by the following diagram.

$$\mathsf{Imp} \xrightarrow{\mathcal{C}} \mathsf{Vm} \xrightarrow{\mathcal{C}'} \mathsf{Mips}$$

The three languages considered can be shortly described as follows: Imp is a very simple imperative language with pure expressions, branching and looping commands, Vm is an assembly-like language enriched with a stack, and Mips is a Mips-like assembly language with registers and main memory. The first compilation function $\mathcal{C}$ relies on the stack of the Vm language to implement expression evaluation while the second compilation function $\mathcal{C}'$ allocates (statically) the base of the stack in the registers and the rest in main memory. This is of course a naive strategy but it suffices to expose some of the problems that arise in defining a compositional approach.

A C compiler As a second, more complex, case study we consider a C compiler. Even if the long-term agenda of the *CerCo* project is to build a mechanically verified compiler using a proof assistant, we only developed an untrusted compiler prototype in ocaml in order to first experiment with the scalability of our approach. The architecture of this prototype is summarized by the following diagram:



The structure follows rather closely the one of the CompCert compiler [9]. Notable differences are that some compilation steps are fusioned, that the front-end goes till RTLAbs (rather than Cminor) and that we target the Mips and Intel 8051 assembly languages (rather than PowerPc). These differences are contingent to the way we built the compiler. The compilation from C to Clight relies on the CIL front-end [11]. The one from Clight to RTL has been programmed from scratch and it is partly based on the Coq definitions available in the CompCert compiler. Finally, the back-end from RTL to Mips is based on a compiler developed in ocaml for pedagogical purposes [12]. We extended this back-end to also target the Intel 8051 [8], an 8bits micro-controller used in embedded systems. [1] The main optimizations the back-end performs are liveness analysis and register allocation and graph compression. We ran some benchmarks to ensure that our prototype implementation is realistic. The results are given in appendix C.9 and the compiler is available from the authors.

Organization The rest of the paper is organized as follows. Section 2 describes the application of the labelling approach to the toy compiler. Section 3 reports our experience in implementing and testing the labelling approach on the C compiler. Section 4 summarizes our contribution and outlines some perspectives for future work. Appendix A gives standard definitions that are skipped in the presentation for the sake of conciseness. Appendix B sketches the proofs that have not been mechanically checked in Coq and appendix C provides some details on the structure of the C compiler we have implemented. Appendix D describes a *direct approach* to cost annotations that in our experience fails to scale to a realistic compiler architecture and appendix E discusses informally some related approaches.

2 Labelling approach for the toy compiler

A first case study for the labelling approach is performed on a toy compiler. The syntax of the source, intermediate and target languages is given in the Figure 1. The semantics of *imp* is defined over configurations (S, K, s) where S is a statement, K is a continuation and s is a state. A *continuation* K is a list of commands which terminates with a special symbol halt. The semantics of Vm is defined over stack-based machine configurations $C \vdash (i, \sigma, s)$ where C is a program, i is a program counter, σ is a stack and s is a state. The semantics of Mips is defined over register-based machine configurations $C \vdash (i, m)$ where C is a program, i is a program counter and m is a machine memory (with registers and main memory). The compilation functions $\mathcal{C}$ from Imp to Vm and $\mathcal{C}'$ from Vm to Mips are standard and thus eluded. (see Appendix A for formal details about semantics and the compilation chain.)

[1] According to [14], in 2008 8051 was the most diffused processor in the microcontroller market with a 19% market share.

Syntax for Imp

$$
\begin{array}{rcl}
id & ::= & x \mid y \mid \dots \\
n & ::= & 0 \mid -1 \mid +1 \mid \dots \\
v & ::= & n \mid \mathsf{true} \mid \mathsf{false} \\
e & ::= & id \mid n \mid e + e \\
b & ::= & e < e \\
S & ::= & \mathsf{skip} \mid id := e \mid S; S \\
 & \mid & \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \\
 & \mid & \mathsf{while}\ b\ \mathsf{do}\ S \\
P & ::= & \mathsf{prog}\ S
\end{array}
$$

Syntax for Vm

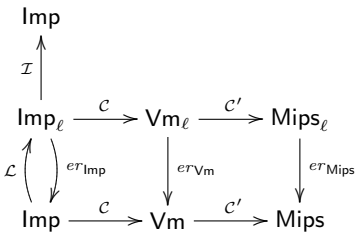
$$
\begin{array}{rcl}
instr_{\mathsf{Vm}} & ::= & \mathtt{cnst}(n) \mid \mathtt{var}(n) \\
 & \mid & \mathtt{setvar}(n) \mid \mathtt{add} \\
 & \mid & \mathtt{branch}(k) \mid \mathtt{bge}(k) \mid \mathtt{halt}
\end{array}
$$

Syntax for Mips

$$
\begin{array}{rcl}
instr_{\mathsf{Mips}} & ::= & \mathtt{loadi}\ R, n \mid \mathtt{load}\ R, l \\
 & \mid & \mathtt{store}\ R, l \mid \mathtt{add}\ R, R, R \\
 & \mid & \mathtt{branch}\ k \mid \mathtt{bge}\ R, R, k \mid \mathtt{halt}
\end{array}
$$

Figure 1: Syntax definitions.

We apply the labelling approach introduced in section 1 to this toy compiler which results in the following diagram.



$$
\begin{array}{rcl}
er_{\mathsf{Vm}} \circ \mathcal{C} & = & \mathcal{C} \circ er_{\mathsf{Imp}} \\
er_{\mathsf{Mips}} \circ \mathcal{C}' & = & \mathcal{C}' \circ er_{\mathsf{Vm}} \\
er_{\mathsf{Imp}} \circ \mathcal{L} & = & id_{\mathsf{Imp}} \\
An_{\mathsf{Imp}} & = & \mathcal{I} \circ \mathcal{L}
\end{array}
$$

2.1 Labelled Imp

We extend the syntax so that statements can be labelled: $S ::= \dots \mid \ell : S$. For instance, $\ell : (\mathsf{while}\ (n < x)\ \mathsf{do}\ \ell : S)$ is a labelled command. The small step semantics of statements is extended as described by the following rule.

$$(\ell : S, K, s) \quad \xrightarrow{\ell} \quad (S, K, s)$$

We denote with $\lambda, \lambda', \dots$ finite sequences of labels. In particular, we denote with ϵ the empty sequence and identify an unlabelled transition with a transition labelled with ϵ. Then the small step reduction relation we have defined on statements becomes a *labelled transition system*. There is an obvious *erasure* function er_{Imp} from the labelled language to the unlabelled one which is the identity on expressions and boolean conditions, and traverses commands removing all labels. We derive a *labelled* big-step semantics as follows: $(S, s) \Downarrow (s', \lambda)$ if $(S, \mathsf{halt}, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (\mathsf{skip}, \mathsf{halt}, s')$ and $\lambda = \lambda_1 \cdots \lambda_n$.

2.2 Labelled Vm

We introduce a new instruction $\mathsf{nop}(\ell)$ whose semantics is defined as follows:

$$C \vdash (i, \sigma, s) \xrightarrow{\ell} (i + 1, \sigma, s) \qquad \text{if } C[i] = \mathsf{nop}(\ell)\ .$$

The erasure function er_{Vm} amounts to remove from a Vm code C all the $\mathsf{nop}(\ell)$ instructions and recompute jumps accordingly. Specifically, let $n(C, i, j)$ be the number of nop instructions

in the interval $[i, j]$. Then, assuming $C[i] = \mathsf{branch}(k)$ we replace the offset k with an offset k' determined as follows:

$$k' = \begin{cases} k - n(C, i, i + k) & \text{if } k \geq 0 \\ k + n(C, i + 1 + k, i) & \text{if } k < 0 \end{cases}$$

The compilation function $\mathcal{C}$ is extended to Imp_ℓ by defining:

$$\mathcal{C}(\ell : b, k) \quad = (\mathsf{nop}(\ell)) \cdot \mathcal{C}(b, k) \quad \mathcal{C}(\ell : S) \quad = (\mathsf{nop}(\ell)) \cdot \mathcal{C}(S) \ .$$

Proposition 1 *For all commands S in Imp_ℓ we have that:*
(1) $er_{\mathsf{Vm}}(\mathcal{C}(S)) = \mathcal{C}(er_{\mathsf{Imp}}(S))$.
(2) *If $(S, s) \Downarrow (s', \lambda)$ then $(\mathcal{C}(S), s) \Downarrow (s', \lambda)$.*

Remark 2 *In the current formulation, a sequence of transitions λ in the source code must be simulated by the same sequence of transitions in the object code. However, in the actual computation of the costs, the order of the labels occurring in the sequence is immaterial. Therefore one may consider a more relaxed notion of simulation where λ is a multi-set of labels.*

2.3 Labelled Mips

The labelled extension of Mips is similar to the one of Vm. We add an instruction $\mathsf{nop}\ \ell$ whose semantics is defined as follows:

$$M \vdash (i, m) \xrightarrow{\ell} (i + 1, m) \quad \text{if } M[i] = (\mathsf{nop}\ \ell) \ .$$

The *erasure function* er_{Mips} is also similar to the one of Vm as it amounts to remove from a Mips code all the $(\mathsf{nop}\ \ell)$ instructions and recompute jumps accordingly. The compilation function $\mathcal{C}'$ is extended to Vm_ℓ by simply translating $\mathsf{nop}(\ell)$ as $(\mathsf{nop}\ \ell)$:

$$\mathcal{C}'(i, C) = (\mathsf{nop}\ \ell) \quad \text{if } C[i] = \mathsf{nop}(\ell)$$

The evaluation predicate for labelled Mips is defined as $(M, m) \Downarrow (m', \lambda)$ if $M \vdash (0, m) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (j, m')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $M[j] = \mathsf{halt}$. The following proposition relates Vm_ℓ code and its compilation and it is similar to proposition 1. $m \Vdash \sigma, s$ means "the low-level Mips memory m realizes the Vm stack σ and state s".

Proposition 3 *Let C be a Vm_ℓ code. Then:*
(1) $er_{\mathsf{Mips}}(\mathcal{C}'(C)) = \mathcal{C}'(er_{\mathsf{Vm}}(C))$.
(2) *If $(C, s) \Downarrow (s', \lambda)$ and $m \Vdash \epsilon, s$ then $(\mathcal{C}'(C), m) \Downarrow (m', \lambda)$ and $m' \Vdash \epsilon, s'$.*

2.4 Labellings and instrumentations

Assuming a function κ which associates an integer number with labels and a distinct variable *cost* which does not occur in the program P under consideration, we abbreviate with $inc(\ell)$ the assignment $cost := cost + \kappa(\ell)$. Then we define the instrumentation $\mathcal{I}$ (relative to κ and *cost*) as follows:

$$\mathcal{I}(\ell : S) = inc(\ell); \mathcal{I}(S) \ .$$

The function $\mathcal{I}$ just distributes over the other operators of the language. We extend the function κ on labels to sequences of labels by defining $\kappa(\ell_1, \ldots, \ell_n) = \kappa(\ell_1) + \cdots + \kappa(\ell_n)$. The instrumented Imp program relates to the labelled one has follows.

Proposition 4 *Let S be an Imp_ℓ command. If $(\mathcal{I}(S), s[c/cost]) \Downarrow s'[c+\delta/cost]$ then $\exists\, \lambda\ \kappa(\lambda) = \delta$ and $(S, s[c/cost]) \Downarrow (s'[c/cost], \lambda)$.*

Definition 5 *A* labelling *is a function $\mathcal{L}$ from an unlabelled language to the corresponding labelled one such that $er_{\mathsf{Imp}} \circ \mathcal{L}$ is the identity function on the* Imp *language.*

Proposition 6 *For any labelling function $\mathcal{L}$, and* Imp *program P, the following holds:*

$$er_{\mathsf{Mips}}(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P)))) = \mathcal{C}'(\mathcal{C}(P))\ . \tag{3}$$

Proposition 7 *Given a function κ for the labels and a labelling function $\mathcal{L}$, for all programs P of the source language if $(\mathcal{I}(\mathcal{L}(P)), s[c/cost]) \Downarrow s'[c + \delta/cost]$ and $m \Vdash\!\!-\epsilon, s[c/cost]$ then $(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))), m) \Downarrow (m', \lambda)$, $m' \Vdash\!\!-\epsilon, s'[c/cost]$ and $\kappa(\lambda) = \delta$.*

2.5 Sound and precise labellings

With any Mips_ℓ code M we can associate a directed and rooted (control flow) graph whose nodes are the instruction positions $\{0, \ldots, |M| - 1\}$, whose root is the node 0, and whose directed edges correspond to the possible transitions between instructions. We say that a node is labelled if it corresponds to an instruction nop ℓ.

Definition 8 *A* simple path *in a Mips_ℓ code M is a directed finite path in the graph associated with M where the first node is labelled, the last node is the predecessor of either a labelled node or a leaf, and all the other nodes are unlabelled.*

Definition 9 *A Mips_ℓ code M is* soundly labelled *if in the associated graph the root node* 0 *is labelled and there are no loops that do not go through a labelled node.*

In a soundly labelled graph there are finitely many simple paths. Thus, given a soundly labelled Mips code M, we can associate with every label ℓ a number $\kappa(\ell)$ which is the maximum (estimated) cost of executing a simple path whose first node is labelled with ℓ. We stress that in the following we assume that the cost of a simple path is proportional to the number of Mips instructions that are crossed in the path.

Proposition 10 *If M is soundly labelled and $(M, m) \Downarrow (m', \lambda)$ then the cost of the computation is bounded by $\kappa(\lambda)$.*

Thus for a soundly labelled Mips code the sequence of labels associated with a computation is a significant information on the execution cost.

Definition 11 *We say that a soundly labelled code is* precise *if for every label ℓ in the code, the simple paths starting from a node labelled with ℓ have the same cost.*

In particular, a code is precise if we can associate at most one simple path with every label.

Proposition 12 *If M is precisely labelled and $(M, m) \Downarrow (m', \lambda)$ then the cost of the computation is $\kappa(\lambda)$.*

$$
\begin{array}{ll}
\mathcal{L}_s(\mathsf{prog}\ S) & = \mathsf{prog}\ \ell : \mathcal{L}_s(S) \\
\mathcal{L}_s(\mathsf{skip}) & = \mathsf{skip} \\
\mathcal{L}_s(x := e) & = x := e \\
\mathcal{L}_s(S; S') & = \mathcal{L}_s(S); \mathcal{L}_s(S') \\
\mathcal{L}_s(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) & = \mathsf{if}\ b\ \mathsf{then}\ \mathcal{L}_s(S_1)\ \mathsf{else}\ \mathcal{L}_s(S_2) \\
\mathcal{L}_s(\mathsf{while}\ b\ \mathsf{do}\ S) & = \mathsf{while}\ b\ \mathsf{do}\ \ell : \mathcal{L}_s(S) \\
 & \\
\mathcal{L}_p(\mathsf{prog}\ S) & = \mathsf{prog}\ \mathcal{L}_p(S) \\
\mathcal{L}_p(S) & = \mathit{let}\ \ell = \mathit{new},\ (S', d) = {\mathcal{L}'}_p(S)\ \mathit{in}\ \ell : S' \\
{\mathcal{L}'}_p(S) & = (S, 0) \quad \text{if } S = \mathsf{skip} \text{ or } S = (x := e) \\
{\mathcal{L}'}_p(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) & = (\mathsf{if}\ b\ \mathsf{then}\ \mathcal{L}_p(S_1)\ \mathsf{else}\ \mathcal{L}_p(S_2), 1) \\
{\mathcal{L}'}_p(\mathsf{while}\ b\ \mathsf{do}\ S) & = (\mathsf{while}\ b\ \mathsf{do}\ \mathcal{L}_p(S), 1) \\
{\mathcal{L}'}_p(S_1; S_2) & = \mathit{let}\ (S'_1, d_1) = {\mathcal{L}'}_p(S_1),\ (S'_2, d_2) = {\mathcal{L}'}_p(S_2)\ \mathit{in} \\
 & \qquad \mathit{case}\ d_1 \\
 & \qquad 0 : (S'_1; S'_2, d_2) \\
 & \qquad 1 : \mathit{let}\ \ell = \mathit{new}\ \mathit{in}\ (S'_1; \ell : S'_2, d_2)
\end{array}
$$

Table 1: Two labellings for the Imp language

The next point we have to check is that there are labelling functions (of the source code) such that the compilation function does produce sound and possibly precise labelled Mips code. To discuss this point, we introduce in table 1 two labelling functions $\mathcal{L}_s$ and $\mathcal{L}_p$ for the Imp language. The first labelling relies on just one label while the second one relies on a function "*new*" which is meant to return fresh labels and on an auxiliary function ${\mathcal{L}'}_p$ which returns a labelled command and a binary directive $d \in \{0, 1\}$. If $d = 1$ then the command that follows (if any) must be labelled.

Proposition 13 *For all* Imp *programs* P*:*

(1) $\mathcal{C}'(\mathcal{C}(\mathcal{L}_s(P)))$ *is a soundly labelled* Mips *code.*

(2) $\mathcal{C}'(\mathcal{C}(\mathcal{L}_p(P)))$ *is a soundly and precisely labelled* Mips *code.*

For an example of command which is not soundly labelled, consider $\ell : \mathsf{while}\ 0 < x\ \mathsf{do}\ x := x+1$, which when compiled, produces a loop that does not go through any label. On the other hand, for an example of a program which is not precisely labelled consider $\ell : (\mathsf{if}\ 0 < x\ \mathsf{then}\ x := x + 1\ \mathsf{else}\ \mathsf{skip})$. In the compiled code, we find two simple paths associated with the label ℓ whose cost will be quite different in general.

Once a sound and possibly precise labelling $\mathcal{L}$ has been designed, we can determine the cost of each label and define an instrumentation $\mathcal{I}$ whose composition with $\mathcal{L}$ will produce the desired cost annotation.

Definition 14 *Given a labelling function* $\mathcal{L}$ *for the source language* Imp *and a program* P *in the* Imp *language, we define an annotation for the source program as follows:*

$$An_{\mathsf{Imp}}(P) = \mathcal{I}(\mathcal{L}(P)) \ .$$

Proposition 15 *If* P *is a program and* $\mathcal{C}'(\mathcal{C}(\mathcal{L}(P)))$ *is a sound (sound and precise) labelling then* $(An_{\mathsf{Imp}}(P), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *and* $m \Vdash \epsilon, s[c/cost]$ *entails that* $(\mathcal{C}'(\mathcal{C}(P)), m) \Downarrow m'$*,* $m' \Vdash \epsilon, s'[c/cost]$ *and the cost of the execution is bound (is exactly)* δ*.*

To summarize, producing sound and precise labellings is mainly a matter of designing the labelled source language so that the labelling is sufficiently *fine grained*. For instance, in

the toy compiler, it is enough to label commands while it is not necessary to label boolean conditions and expressions.

Besides soundness and precision, a third criterion to evaluate labellings is that they do not introduce too many unnecessary labels. We call this property *economy*. In practice, it seems that one can produce first a sound and possibly precise labelling and then apply heuristics to eliminate unnecessary labels.

We stress that our approach to labelling is based on the hypothesis that we can obtain precise informations on the execution time of each instruction of the generated binary code. This hypothesis is indeed realised in the processors of the 8051 family we are considering. On the other hand, the execution time of instructions running on more complex architectures including, *e.g.*, cache memory or pipelines are much less predictable. This lack of precision may somehow be compensated by analysing the worst-case execution time of (long) sequences of instructions. This point is further developed in appendix E.

3 Labelling approach for the C compiler

This section informally describes the labelled extensions of the languages in the compilation chain (see Appendix C for details), the way the labels are propagated by the compilation functions, the labelling of the source code, the hypotheses on the control flow of the labelled assembly code and the verification that we perform on it, the way we build the instrumentation, and finally the way the labelling approach has been tested. The process of annotating a Clight program using the labelling approach is detailed in the following sections. It is summarized by the following steps.

1. Label the input Clight program.
2. Compile the labelled Clight program in the labelled world. This produces a labelled assembly code.
3. For each label of the labelled assembly code, compute the cost of the instructions under its scope and generate a *label-cost mapping*. An unlabelled assembly code — the result of the compilation — is obtained by removing the labels from the labelled assembly code.
4. Add a fresh *cost variable* to the labelled Clight program and replace the labels by an increment of this cost variable according to the label-cost mapping. The result is an *annotated* Clight program with no label.

3.1 Labelled languages

Both the Clight and Cminor languages are extended in the same way by labelling both statements and expressions (by comparison, in the toy language Imp we just labelled statements). The labelling of expressions aims to capture precisely their execution cost. Indeed, Clight and Cminor include expressions such as $a_1?a_2;a_3$ whose evaluation cost depends on the boolean value a_1. As both languages are extended in the same way, the extended compilation does nothing more than sending Clight labelled statements and expressions to those of Cminor.

The labelled versions of RTLAbs and the languages in the back-end simply consist in adding a new instruction whose semantics is to emit a label without modifying the state. For the CFG based languages (RTLAbs to LTL), this new instruction is emit *label* $\rightarrow$ *node*. For LIN, Mips and 8051, it is emit *label*. The translation of these label instructions is immediate. In Mips and 8051, we also rely on a reserved label begin_function to pinpoint the beginning of a function code (cf. section 3.2).

3.2 Labelling of the source language

As for the toy compiler (cf. end of section 2), the goals of a labelling are soundness, precision, and possibly economy. We explain our labelling by considering the constructions of Clight and their compilation to assembly code.

Sequential instructions A sequence of Clight instructions that compile to sequential assembly code, such as a sequence of assignments, can be handled by a single label which covers the unique execution path.

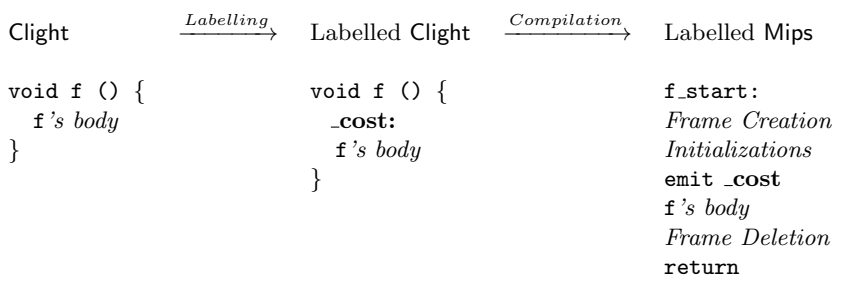
Ternary expressions and conditionals Most Clight expressions compile to sequential assembly code. *Ternary expressions*, that introduce a branching in the control flow, are one exception. In this case, we achieve precision by associating a label with each branch. This is similar to the treatment of the conditional we have already discussed in section 2. As for the Clight operations `&&` and `||` which have a lazy semantics, they are transformed to ternary expressions *before* computing the labelling.

Loops Loops in Clight are guarded by a condition. Following the arguments for the previous cases, we add two labels when encountering a loop construct: one label to start the loop's body, and one label when exiting the loop. This is similar to the treatment of while loops discussed in section 2 and it is enough to guarantee that the loop in the compiled code goes through a label.

Program Labels and Gotos In Clight, program labels and gotos are intraprocedural. Their only effect on the control flow of the resulting assembly code is to potentially introduce an unguarded loop. This loop must contain at least one cost label in order to satisfy the soundness condition, which we ensure by adding a cost label right after a program label.

Clight	$\xrightarrow{Labelling}$	Labelled Clight	$\xrightarrow{Compilation}$	Labelled Mips
`lbl:`		`lbl:`		`lbl:`
`i++;`		`_cost:`		`emit _cost`
`...`		`i++;`		`li $v0, 1`
`goto lbl;`		`...`		`add $a0, $a0, $v0`
		`goto lbl;`		`...`
				`j lbl`

Function calls Function calls in assembly code are performed by indirect jumps, the address of the callee being in the memory. In the general case, this address cannot be inferred statically. Even though the destination point of a function call is unknown, when the considered assembly code has been produced by our compiler, we know for a fact that this function ends with a return statement that transfers the control back to the instruction following the function call in the caller. As a result, we treat function calls according to the following global invariants of the compilation: (1) the instructions of a function are covered by the labels inside this function, (2) we assume a function call always returns and runs the instruction following the call. Invariant (1) entails in particular that each function must contain at least one label. To ensure this, we simply add a starting label in every function definition. The example below illustrates this point:



We notice that some instructions in assembly code will be inserted *before* the first label is emitted. These instructions relate to the frame creation and/or variable initializations, and are composed of sequential instructions (no branching). To deal with this issue, we take the convention that the instructions that precede the first label in a function code are actually under the scope of the first label. Invariant (2) is of course an over-approximation of the program behavior as a function might fail to return because of an infinite loop. In this case, the proposed labelling remains correct: it just assumes that the instructions following the function call will be executed, and takes their cost into consideration. The final computed cost is still an over-approximation of the actual cost.

3.3 Verifications on the object code

The labelling previously described has been designed so that the compiled assembly code satisfies the soundness and precision conditions. However, we do not need to prove this, instead we have to devise an algorithm that checks the conditions on the compiled code. The algorithm assumes a correct management of function calls in the compiled code. In particular, when we call a function we always jump to the first instruction of the corresponding code segment and when we return we always jump to an an instruction that follows a call. We stress that this is a reasonable hypothesis that is essentially subsumed by the proof that the object code *simulates* the source code.

In our current implementation, we check the soundness and the precision conditions while building at the same time the label-cost mapping. To this end, the algorithm takes the following main steps.

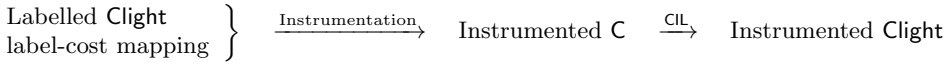
- First, for each function a control flow graph is built.
- For each graph, we check whether there is a unique label that is reachable from the root by a unique path. This unique path corresponds to the instructions generated by the calling conventions as discussed in section 3.2. We shift the occurrence of the label to the root of the graph.
- By a strongly connected components algorithm, we check whether every loop in the graphs goes through at least one label.
- We perform a (depth-first) search of the graph. Whenever we reach a labelled node, we perform a second (depth-first) search that stops at labelled nodes and computes an upper bound on the cost of the occurrence of the label. Of course, when crossing a branching instruction, we take the maximum cost of the branches. When the second search stops we update the current cost of the label-cost mapping (by taking a maximum) and we continue the first search.
- Warning messages are emitted whenever the maximum is taken between two different values as in this case the precision condition may be violated.

In the particular case of the 8051, which is a very basic micro-controller, the number of cycles required by each instruction is fully specified. Therefore, if the labelling is precise, the cost associated to a label is the exact number of cycles necessary to execute every simple execution path starting from that label.

3.4 Building the cost annotation

Once the label-cost mapping is computed, instrumenting the labelled source code is an easy task. A fresh global variable which we call *cost variable* is added to the source program with the purpose of holding the cost value and it is initialized at the very beginning of the `main` program. Then, every label is replaced by an increment of the cost variable according to the label-cost mapping. Following this replacement, the cost labels disappear and the result is a Clight program with annotations in the form of assignments.

There is one final problem: labels inside expressions. As we already mentioned, Clight does not allow writing side-effect instructions — such as cost increments — inside expressions. To cope with this restriction, we produce first an instrumented C program — with side-effects in expressions — that we translate back to Clight using CIL. This process is summarized below.



3.5 Testing

It is desirable to test the coherence of the labelling from Clight to assembly code. To this end, each labelled language comes with an interpreter that produces the trace of the labels encountered during the computation.

Then, one naive approach is to test the equality of the traces produced by the program at the different stages of the compilation. Our current implementation passes this kind of tests. For some optimizations that may re-order computations, the weaker condition mentioned in remark 2 could be considered.

Furthermore, in the case of the 8051 back-end, we can run our compiled code into a dedicated simulator and we check that the statically computed number of cycles for each simple path is the same as the real one.

4 Conclusion and future work

We have discussed the problem of building a compiler which can *lift* in a provably correct way pieces of information on the execution cost of the object code to cost annotations on the source code. To this end, we have introduced the so called *labelling* approach and discussed its formal application to a toy compiler. Based on this experience, we have argued that the approach has good scalability properties, and to substantiate this claim, we have reported on our successful experience in implementing and testing the labelling approach on top of a prototype compiler written in ocaml for a large fragment of the C language which can be shortly described as Clight without floating point.

We discuss next a few directions for future work. First, we are currently testing the current compiler on the kind of C code produced for embedded applications by a Lustre compiler. Starting from the annotated C code, we are relying on the Frama − C tool to produce automatically meaningful information on, say, the reaction time of a given synchronous program. Second, we plan to formalize and validate in the *Calculus of Inductive Constructions* the prototype implementation of the labelling approach for the C compiler described in section 3. This requires a major implementation effort which will be carried on in collaboration with our partners of the *CerCo* project [2].

References

[1] AbsInt Angewandte Informatik. `http://www.absint.com/`.

[2] Certified Complexity (Project description). ICT-2007.8.0 FET Open, Grant 243881. `http://cerco.cs.unibo.it`.

[3] Esterel Technologies. `http://www.esterel-technologies.com`.

[4] Frama − C software analysers. `http://frama-c.com/`.

[5] C. Ferdinand, R. Heckmann, T. Le Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In *Embedded Real Time Software (ERTS)*, 2008.

[6] X. Fornari. Understanding how SCADE suite KCG generates safe C code. White paper, Esterel Technologies, 2010.

[7] J. Larus. Assemblers, linkers, and the SPIM simulator. Appendix of *Computer Organization and Design: the hw/sw interface*, by Hennessy and Patterson, 2005.

[8] MCS 51 Microcontroller Family User's Manual. Publication number 121517, by Intel Corporation, 1994,

[9] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107-115, 2009.

[10] X. Leroy. Mechanized semantics, with applications to program proof and compiler verification. *Marktoberdorf summer school*, 2009.

[11] G. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compiler Construction*, Springer LNCS 2304:213–228, 2002.

[12] F. Pottier. Compilation (INF 564), École Polytechnique, 2009-2010. `http://www.enseignement.polytechnique.fr/informatique/INF564/`.

[13] R. Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[14] Microcontroller market and Technology. Analysis report 2008. EMITT solutions.

$$
\begin{array}{lcl}
(x := e, K, s) & \to & (\mathsf{skip}, K, s[v/x]) \quad \text{if } (e, s) \Downarrow v \\
(S; S', K, s) & \to & (S, S' \cdot K, s) \\
(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S', K, s) & \to & \begin{cases} (S, K, s) & \text{if } (b, s) \Downarrow \mathsf{true} \\ (S', K, s) & \text{if } (b, s) \Downarrow \mathsf{false} \end{cases} \\
(\mathsf{while}\ b\ \mathsf{do}\ S, K, s) & \to & \begin{cases} (S, (\mathsf{while}\ b\ \mathsf{do}\ S) \cdot K, s) & \text{if } (b, s) \Downarrow \mathsf{true} \\ (\mathsf{skip}, K, s) & \text{if } (b, s) \Downarrow \mathsf{false} \end{cases} \\
(\mathsf{skip}, S \cdot K, s) & \to & (S, K, s)
\end{array}
$$

Table 2: Small-step operational semantics of Imp commands

A A toy compiler

We formalize the toy compiler introduced in section 1 [2].

A.1 Imp: language and semantics

The syntax of the Imp language is described below. This is a rather standard imperative language with while loops and if-then-else.

id	$::= x \mid y \mid \ldots$	(identifiers)
n	$::= 0 \mid -1 \mid +1 \mid \ldots$	(integers)
v	$::= n \mid \mathsf{true} \mid \mathsf{false}$	(values)
e	$::= id \mid n \mid e + e$	(numerical expressions)
b	$::= e < e$	(boolean conditions)
S	$::= \mathsf{skip} \mid id := e \mid S; S \mid \mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S \mid \mathsf{while}\ b\ \mathsf{do}\ S$	(commands)
P	$::= \mathsf{prog}\ S$	(programs)

Let s be a total function from identifiers to integers representing the **state**. If s is a state, x an identifier, and n an integer then $s[n/x]$ is the 'updated' state such that $s[n/x](x) = n$ and $s[n/x](y) = s(y)$ if $x \neq y$. The *big-step* operational semantics of Imp expressions and boolean conditions is defined as follows:

$$
\frac{}{(v, s) \Downarrow v} \qquad \frac{}{(x, s) \Downarrow s(x)} \qquad \frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e + e', s) \Downarrow (v +_{\mathbf{Z}} v')} \qquad \frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e < e', s) \Downarrow (v <_{\mathbf{Z}} v')}
$$

A *continuation* K is a list of commands which terminates with a special symbol halt: $K ::= \mathsf{halt} \mid S \cdot K$. Table 2 defines a small-step semantics of Imp commands whose basic judgement has the shape: $(S, K, s) \to (S', K', s')$. We define the semantics of a program prog S as the semantics of the command S with continuation halt. We derive a big step semantics from the small step one as follows: $(S, s) \Downarrow s'$ if $(S, \mathsf{halt}, s) \to \cdots \to (\mathsf{skip}, \mathsf{halt}, s')$.

A.2 Vm: language and semantics

Following [10], we define a virtual machine Vm and its programming language. The machine includes the following elements: (1) a fixed code C (a possibly empty sequence of instructions), (2) a program counter pc, (3) a store s (as for the source program), (4) a stack of integers σ.

[2] A mechanized proof in the Coq proof assistant can be found in K. Memarian's *Travail d'étude et de recherche* entitled *Complexité Certifiée* which can be downloaded at `http://www.pps.jussieu.fr/~yrg/miniCerCo/`.

Rule	$C[i] =$
$C \vdash (i, \sigma, s) \to (i+1, n \cdot \sigma, s)$	$\texttt{cnst}(n)$
$C \vdash (i, \sigma, s) \to (i+1, s(x) \cdot \sigma, s)$	$\texttt{var}(x)$
$C \vdash (i, n \cdot \sigma, s) \to (i+1, \sigma, s[n/x])$	$\texttt{setvar}(x)$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+1, (n +_{\mathbf{Z}} n') \cdot \sigma, s)$	$\texttt{add}$
$C \vdash (i, \sigma, s) \to (i+k+1, \sigma, s)$	$\texttt{branch(k)}$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+1, \sigma, s)$	$\texttt{bge(k)}$ and $n <_{\mathbf{Z}} n'$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \to (i+k+1, \sigma, s)$	$\texttt{bge(k)}$ and $n \geq_{\mathbf{Z}} n'$

Table 3: Operational semantics Vm programs

$C[i] =$	Conditions for $C : h$
$\mathsf{cnst}(n)$ or $\mathsf{var}(x)$	$h(i+1) = h(i) + 1$
add	$h(i) \geq 2, \quad h(i+1) = h(i) - 1$
$\mathsf{setvar}(x)$	$h(i) = 1, \quad h(i+1) = 0$
$\mathsf{branch}(k)$	$0 \leq i+k+1 \leq \lvert C\rvert, \quad h(i) = h(i+1) = h(i+k+1) = 0$
$\mathsf{bge}(k)$	$0 \leq i+k+1 \leq \lvert C\rvert, \quad h(i) = 2, \quad h(i+1) = h(i+k+1) = 0$
halt	$i = \lvert C\rvert - 1, \quad h(i) = h(i+1) = 0$

Table 4: Conditions for well-formed code

Given a sequence C, we denote with $|C|$ its length and with $C[i]$ its i^{th} element (the leftmost element being the 0^{th} element). The operational semantics of the instructions is formalized by rules of the shape $C \vdash (i, \sigma, s) \to (j, \sigma', s')$ and it is fully described in table 3. Notice that Imp and Vm semantics share the same notion of store. We write, *e.g.*, $n \cdot \sigma$ to stress that the top element of the stack exists and is n. We will also write $(C, s) \Downarrow s'$ if $C \vdash (0, \epsilon, s) \xrightarrow{*} (i, \epsilon, s')$ and $C[i] = \mathsf{halt}$.

Code coming from the compilation of Imp programs has specific properties that are used in the following compilation step when values on the stack are allocated either in registers or in main memory. In particular, it turns out that for every instruction of the compiled code it is possible to predict statically the *height of the stack* whenever the instruction is executed. We now proceed to define a simple notion of *well-formed* code and show that it enjoys this property. In the following section, we will define the compilation function from Imp to Vm and show that it produces well-formed code.

Definition 16 *We say that a sequence of instructions C is well formed if there is a function $h : \{0, \ldots, |C|\} \to \mathbf{N}$ which satisfies the conditions listed in table 4 for $0 \leq i \leq |C| - 1$. In this case we write $C : h$.*

The conditions defining the predicate $C : h$ are strong enough to entail that h correctly predicts the stack height and to guarantee the uniqueness of h up to the initial condition.

Proposition 17 *(1) If $C : h$, $C \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma', s')$, and $h(i) = |\sigma|$ then $h(j) = |\sigma'|$. (2) If $C : h$, $C : h'$ and $h(0) = h'(0)$ then $h = h'$.*

A.3 Compilation from Imp to Vm

In table 5, we define compilation functions $\mathcal{C}$ from Imp to Vm which operate on expressions, boolean conditions, statements, and programs. We write $sz(e)$, $sz(b)$, $sz(S)$ for the number of

$$\mathcal{C}(x) = \texttt{var}(x) \qquad \mathcal{C}(n) = \texttt{cnst}(n) \qquad \mathcal{C}(e + e') = \mathcal{C}(e) \cdot \mathcal{C}(e') \cdot \texttt{add}$$

$$\mathcal{C}(e < e', k) = \mathcal{C}(e') \cdot \mathcal{C}(e) \cdot \texttt{bge(k)}$$

$$\mathcal{C}(x := e) = \mathcal{C}(e) \cdot \texttt{setvar(x)} \qquad \mathcal{C}(S; S') = \mathcal{C}(S) \cdot \mathcal{C}(S')$$

$$\mathcal{C}(\textsf{if } b \textsf{ then } S \textsf{ else } S') = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot (\textsf{branch}(k')) \cdot \mathcal{C}(S')$$
$$\text{where: } k = sz(S) + 1, \quad k' = sz(S')$$

$$\mathcal{C}(\textsf{while } b \textsf{ do } S) = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot \textsf{branch}(k')$$
$$\text{where: } k = sz(S) + 1, \quad k' = -(sz(b) + sz(S) + 1)$$

$$\mathcal{C}(\textsf{prog } S) = \mathcal{C}(S) \cdot \textsf{halt}$$

Table 5: Compilation from Imp to Vm

instructions the compilation function associates with the expression e, the boolean condition b, and the statement S, respectively.

We follow [10] for the proof of soundness of the compilation function for expressions and boolean conditions.

Proposition 18 *The following properties hold:*

(1) *If* $(e, s) \Downarrow v$ *then* $C \cdot \mathcal{C}(e) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j, v \cdot \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(e)|$.

(2) *If* $(b, s) \Downarrow \textsf{true}$ *then* $C \cdot \mathcal{C}(b, k) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j + k, \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(b, k)|$.

(3) *If* $(b, s) \Downarrow \textsf{false}$ *then* $C \cdot \mathcal{C}(b, k) \cdot C' \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma, s)$ *where* $i = |C|$ *and* $j = |C \cdot \mathcal{C}(b, k)|$.

Next we focus on the compilation of statements. We introduce a ternary relation $R(C, i, K)$ which relates a Vm code C, a number $i \in \{0, \ldots, |C| - 1\}$ and a continuation K. The intuition is that relative to the code C, the instruction i can be regarded as having continuation K. (A formal definition is available in Appendix 19.) We can then state the correctness of the compilation function as follows.

Proposition 19 *If* $(S, K, s) \to (S', K', s')$ *and* $R(C, i, S \cdot K)$ *then* $C \vdash (i, \sigma, s) \xrightarrow{*} (j, \sigma, s')$ *and* $R(C, j, S' \cdot K')$.

As announced, we can prove that the result of the compilation is a well-formed code.

Proposition 20 *For any program* P *there is a unique* h *such that* $\mathcal{C}(P) : h$.

A.4 Mips: language and semantics

We consider a Mips-like machine [7] which includes the following elements: (1) a fixed code M (a sequence of instructions), (2) a program counter *pc*, (3) a finite set of registers including the registers A, B, and $R_0, \ldots, R_{b-1}$, and (4) an (infinite) main memory which maps locations to integers.

We denote with $R, R', \ldots$ registers, with $l, l', \ldots$ locations and with $m, m', \ldots$ memories which are total functions from registers and locations to (unbounded) integers. We denote with M a list of instructions. The operational semantics is formalized in table 6 by rules of the shape $M \vdash (i, m) \to (j, m')$, where M is a list of Mips instructions, i, j are natural numbers and m, m' are memories. We write $(M, m) \Downarrow m'$ if $M \vdash (0, m) \xrightarrow{*} (j, m')$ and $M[j] = \textsf{halt}$.

Rule	$M[i] =$
$M \vdash (i,m) \to (i+1, m[n/R])$	loadi R, n
$M \vdash (i,m) \to (i+1, m[m(l)/R])$	load R, l
$M \vdash (i,m) \to (i+1, m[m(R)/l]))$	store R, l
$M \vdash (i,m) \to (i+1, m[m(R') + m(R'')/R])$	add R, R', R''
$M \vdash (i,m) \to (i+k+1, m)$	branch k
$M \vdash (i,m) \to (i+1, m)$	bge R, R', k and $m(R) <_{\mathbf{Z}} m(R')$
$M \vdash (i,m) \to (i+k+1, m)$	bge R, R', k and $m(R) \geq_{\mathbf{Z}} m(R')$

Table 6: Operational semantics Mips programs

$C[i] =$	$\mathcal{C}'(i, C) =$
cnst(n)	$\begin{cases} (\text{loadi } R_h, n) & \text{if } h = h(i) < b \\ (\text{loadi } A, n) \cdot (\text{store } A, l_h) & \text{otherwise} \end{cases}$
var(x)	$\begin{cases} (\text{load } R_h, l_x) & \text{if } h = h(i) < b \\ (\text{load } A, l_x) \cdot (\text{store } A, l_h) & \text{otherwise} \end{cases}$
add	$\begin{cases} (\text{add } R_{h-2}, R_{h-2}, R_{h-1}) & \text{if } h = h(i) < (b-1) \\ (\text{load } A, l_{h-1}) \cdot (\text{add } R_{h-2}, R_{h-2}, A) & \text{if } h = h(i) = (b-1) \\ (\text{load } A, l_{h-1}) \cdot (\text{load } B, l_{h-2}) & \text{if } h = h(i) > (b-1) \\ \quad (\text{add } A, B, A) \cdot (\text{store } A, l_{h-2}) & \end{cases}$
setvar(x)	$\begin{cases} (\text{store } R_{h-1}\ l_x) & \text{if } h = h(i) < b \\ (\text{load } A, l_{h-1}) \cdot (\text{store } A, l_x) & \text{if } h = h(i) \geq b \end{cases}$
branch(k)	$(\text{branch } k')$ if $k' = p(i+k+1, C) - p(i+1, C)$
bge(k)	$\begin{cases} (\text{bge } R_{h-2}, R_{h-1}, k') & \text{if } h = h(i) < (b-1) \\ (\text{load } A, l_{h-1}) \cdot (\text{bge } R_{h-2}, A, k') & \text{if } h = h(i) = (b-1) \\ (\text{load } A, l_{h-2}) \cdot (\text{load } B, l_{h-1}) \cdot (\text{bge } A, B, k') & \text{if } h = h(i) > (b-1),\ k' = p(i+k+1, C) - p(i+1, C) \end{cases}$
halt	halt

Table 7: Compilation from Vm to Mips

A.5 Compilation from Vm to Mips

In order to compile Vm programs to Mips programs we make the following hypotheses: (1) for every Vm program variable x we reserve an address l_x, (2) for every natural number $h \geq b$, we reserve an address l_h (the addresses $l_x, l_h, \ldots$ are all distinct), and (3) we store the first b elements of the stack σ in the registers $R_0, \ldots, R_{b-1}$ and the remaining (if any) at the addresses $l_b, l_{b+1}, \ldots$.

We say that the memory m represents the stack σ and the store s, and write $m \Vdash \sigma, s$, if the following conditions are satisfied: (1) $s(x) = m(l_x)$, and (2) if $0 \leq i < |\sigma|$ then $\sigma[i] = m(R_i)$ if $i < b$, and $\sigma[i] = m(l_i)$ if $i \geq b$.

The compilation function $\mathcal{C}'$ from Vm to Mips is described in table 7. It operates on a well-formed Vm code C whose last instruction is halt. Hence, by proposition 20(3), there is a unique h such that $C : h$. We denote with $\mathcal{C}'(C)$ the concatenation $\mathcal{C}'(0, C) \cdots \mathcal{C}'(|C| - 1, C)$. Given a well formed Vm code C with $i < |C|$ we denote with $p(i, C)$ the position of the first instruction in $\mathcal{C}'(C)$ which corresponds to the compilation of the instruction with position i in C. This is defined as[3] $p(i, C) = \Sigma_{0 \leq j < i} d(i, C)$, where the function $d(i, C)$ is defined as $d(i, C) = |\mathcal{C}'(i, C)|$. Hence $d(i, C)$ is the number of Mips instructions associated with the i^{th} instruction of the (well-formed) C code. The functional correctness of the compilation function

[3]There is an obvious circularity in this definition that can be easily eliminated by defining first the function d following the case analysis in table 7, then the function p, and finally the function $\mathcal{C}'$ as in table 7.

can then be stated as follows.

Proposition 21 *Let $C : h$ be a well formed code. If $C \vdash (i, \sigma, s) \to (j, \sigma', s')$ with $h(i) = |\sigma|$ and $m \Vdash \sigma, s$ then $\mathcal{C}'(C) \vdash (p(i, C), m) \xrightarrow{*} (p(j, C), m')$ and $m' \Vdash \sigma', s'$.*

B Proofs

We omit the proofs that have been mechanically checked by K. Memarian and R. Saillard with the Coq proof assistant [4].

B.1 Notation

Let $\xrightarrow{t}$ be a family of reduction relations where t ranges over the set of labels and ϵ. Then we define:

$$\overset{t}{\Rightarrow} = \begin{cases} (\xrightarrow{\epsilon})^* & \text{if } t = \epsilon \\ (\xrightarrow{\epsilon})^* \circ \xrightarrow{t} \circ (\xrightarrow{\epsilon})^* & \text{otherwise} \end{cases}$$

where as usual R^* denote the reflexive and transitive closure of the relation R and $\circ$ denotes the composition of relations.

B.2 Proof of proposition 1

(1) By induction on the structure of the command S.

(2) By iterating the following proposition.

Proposition 22 *If* $(S, K, s) \xrightarrow{t} (S', K', s')$ *and* $R(C, i, S \cdot K)$ *with* $t = \ell$ *or* $t = \epsilon$ *then* $C \vdash (i, \sigma, s) \overset{t}{\Rightarrow} (j, \sigma, s')$ *and* $R(C, j, S' \cdot K')$.

This is an extension of proposition 19 and it is proven in the same way with an additional case for labelled commands. □

B.3 Proof of proposition 3

(1) The compilation of the Vm instruction $\mathsf{nop}(\ell)$ is the Mips instruction (nop ℓ).

(2) By iterating the following proposition.

Proposition 23 *Let* $C : h$ *be a well formed code. If* $C \vdash (i, \sigma, s) \xrightarrow{t} (j, \sigma', s')$ *with* $t = \ell$ *or* $t = \epsilon$, $h(i) = |\sigma|$ *and* $m \Vdash \sigma, s$ *then* $\mathcal{C}'(C) \vdash (p(i, C), m) \overset{t}{\Rightarrow} (p(j, C), m')$ *and* $m' \Vdash \sigma', s'$.

B.4 Proof of proposition 4

We extend the instrumentation to the continuations by defining:

$$\mathcal{I}(S \cdot K) = \mathcal{I}(S) \cdot \mathcal{I}(K) \qquad \mathcal{I}(\mathsf{halt}) = \mathsf{halt}\ .$$

Then we examine the possible reductions of a configuration $(\mathcal{I}(S), \mathcal{I}(K), s[c/cost])$.

- If S is an unlabelled statement such as while b do S' then $\mathcal{I}(S) =$ while b do $\mathcal{I}(S')$ and assuming $(b, s) \Downarrow$ true the reduction step is:

$$(\mathcal{I}(S), \mathcal{I}(K), s[c/cost]) \to (\mathcal{I}(S'), \mathcal{I}(S) \cdot \mathcal{I}(K), s[c/cost])\ .$$

[4] These proofs can be downloaded at `http://www.pps.jussieu.fr/~yrg/miniCerCo/` and at `http://www.pps.jussieu.fr/~saillard`.

Noticing that $\mathcal{I}(S) \cdot \mathcal{I}(K) = \mathcal{I}(S \cdot K)$, this step is matched in the labelled language as follows:

$$(S, K, s[c/cost]) \to (S', S \cdot K, s[c/cost]) \ .$$

- On the other hand, if $S = \ell : S'$ is a labelled statement then $\mathcal{I}(S) = inc(\ell); \mathcal{I}(S')$ and, by a sequence of reductions steps, we have:

$$(\mathcal{I}(S), \mathcal{I}(K), s[c/cost]) \xrightarrow{*} (\mathcal{I}(S'), \mathcal{I}(K), s[c + \kappa(\ell)/cost]) \ .$$

This step is matched by the labelled reduction:

$$(S, K, s[c/cost]) \xrightarrow{\ell} (S', K, s[c/cost]) \ .$$

□

B.5 Proof of proposition 6

By diagram chasing using propositions 1(1), 3(1), and the definition 5 of labelling. □

B.6 Proof of proposition 7

Suppose that:

$$(\mathcal{I}(\mathcal{L}(P)), s[c/cost]) \Downarrow s'[c + \delta/cost] \text{ and } m \Vdash s[c/cost] \ .$$

Then, by proposition 4, for some λ:

$$(\mathcal{L}(P), s[c/cost]) \Downarrow (s'[c/cost], \lambda) \text{ and } \kappa(\lambda) = \delta \ .$$

Finally, by propositions 1(2) and 3(2) :

$$(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))), m) \Downarrow (m', \lambda) \text{ and } m' \Vdash s'[c/cost] \ .$$

□

B.7 Proof of proposition 10

If $\lambda = \ell_1 \cdots \ell_n$ then the computation is the concatenation of simple paths labelled with $\ell_1, \ldots, \ell_n$. Since $\kappa(\ell_i)$ bounds the cost of a simple path labelled with ℓ_i, the cost of the overall computation is bounded by $\kappa(\lambda) = \kappa(\ell_1) + \cdots \kappa(\ell_n)$. □

B.8 Proof of proposition 12

Same proof as proposition 10, by replacing the word *bounds* by *is exactly* and the words *bounded by* by *exactly*. □

B.9 Proof of proposition 13

In both labellings under consideration the root node is labelled. An obvious observation is that only commands of the shape $\mathsf{while}\ b\ \mathsf{do}\ S$ introduce loops in the compiled code. We notice that both labelling introduce a label in the loop (though at different places). Thus all loops go through a label and the compiled code is always sound.

To show the precision of the second labelling $\mathcal{L}_p$, we note the following property.

Lemma 24 *A soundly labelled graph is precise if each label occurs at most once in the graph and if the immediate successors of the* bge *nodes are either* halt *(no successor) or labelled nodes.*

Indeed, in a such a graph starting from a labelled node we can follow a unique path up to a leaf, another labelled node, or a bge node. In the last case, the hypotheses in the lemma 24 guarantee that the two simple paths one can follow from the bge node have the same length/cost. □

B.10 Proof of proposition 15

By applying consecutively proposition 7 and propositions 10 or 12. □

B.11 Proof of proposition 19

Given a Vm code C, we define an 'accessibility relation' $\overset{C}{\rightsquigarrow}$ as the least binary relation on $\{0,\dots,|C|-1\}$ such that:

$$\frac{}{i \overset{C}{\rightsquigarrow} i} \qquad \frac{C[i] = \mathsf{branch}(k) \quad (i+k+1) \overset{C}{\rightsquigarrow} j}{i \overset{C}{\rightsquigarrow} j}$$

We also introduce a ternary relation $R(C,i,K)$ which relates a Vm code C, a number $i \in \{0,\dots,|C|-1\}$ and a continuation K. The relation is defined as the least one that satisfies the following conditions.

$$\frac{i \overset{C}{\rightsquigarrow} j \qquad C[j] = \mathsf{halt}}{R(C,i,\mathsf{halt})} \qquad \frac{\begin{array}{c} i \overset{C}{\rightsquigarrow} i' \quad C = C_1 \cdot \mathcal{C}(S) \cdot C_2 \\ i' = |C_1| \quad j = |C_1 \cdot \mathcal{C}(S)| \quad R(C,j,K) \end{array}}{R(C,i,S \cdot K)}\ .$$

The following properties are useful.

Lemma 25 (1) *The relation* $\overset{C}{\rightsquigarrow}$ *is transitive.*

(2) *If* $i \overset{C}{\rightsquigarrow} j$ *and* $R(C,j,K)$ *then* $R(C,i,K)$.

The first property can be proven by induction on the definition of $\overset{C}{\rightsquigarrow}$ and the second by induction on the structure of K.

Next we can focus on the proposition. The notation $C \overset{i}{\cdot} C'$ means that $i = |C|$. Suppose that:

$$(S,K,s) \to (S',K',s') \quad (1) \quad \text{and} \quad R(C,i,S \cdot K) \quad (2)\ .$$

From (2), we know that there exist i' and i'' such that:

$$i \overset{C}{\rightsquigarrow} i' \quad (3), \quad C = C_1 \overset{i'}{\cdot} \mathcal{C}(S) \overset{i''}{\cdot} C_2 \quad (4), \quad \text{and} \quad R(C, i'', K) \quad (5)$$

and from (3) it follows that:

$$C \vdash (i, \sigma, s) \overset{*}{\to} (i', \sigma, s) \quad (3') \ .$$

We are looking for j such that:

$$C \vdash (i, \sigma, s) \overset{*}{\to} (j, \sigma, s') \quad (6), \quad \text{and} \quad R(C, j, S' \cdot K') \quad (7) \ .$$

We proceed by case analysis on S. We just detail the case of the conditional command as the the remaining cases have similar proofs. If $S = \mathsf{if}\ e_1 < e_2\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ then (4) is rewritten as follows:

$$C = C_1 \overset{i'}{\cdot} \mathcal{C}(e_1) \cdot \mathcal{C}(e_2).\mathsf{bge}(k_1) \overset{a}{\cdot} \mathcal{C}(S_1) \overset{b}{\cdot} \mathsf{branch}(k_2) \overset{c}{\cdot} \mathcal{C}(S_2) \overset{i''}{\cdot} C_2$$

where $c = a + k_1$ and $i'' = c + k_2$. We distinguish two cases according to the evaluation of the boolean condition. We describe the case $(e_1 < e_2) \Downarrow \mathsf{true}$. We set $j = a$.

- The instance of (1) is $(S, K, s) \to (S_1, K, s)$.
- The reduction required in (6) takes the form $C \vdash (i, \sigma, s) \overset{*}{\to} (i', \sigma, s) \overset{*}{\to} (a, \sigma, s')$, and it follows from $(3')$, the fact that $(e_1 < e_2) \Downarrow \mathsf{true}$, and proposition 18(2).
- Property (7), follows from lemma 25(2), fact (5), and the following proof tree:

$$\dfrac{j \overset{C}{\rightsquigarrow} j \quad \dfrac{b \overset{C}{\rightsquigarrow} i'' \quad R(C, i'', K)}{R(C, b, K)}}{R(C, j, S_1 \cdot K)} \ .$$

□

C A C compiler

This section gives an informal overview of the compiler, in particular it highlights the main features of the intermediate languages, the purpose of the compilation steps, and the optimisations.

C.1 Clight

Clight is a large subset of the C language that we adopt as the source language of our compiler. It features most of the types and operators of C. It includes pointer arithmetic, pointers to functions, and `struct` and `union` types, as well as all C control structures. The main difference with the C language is that Clight expressions are side-effect free, which means that side-effect operators (=,+=,++,...) and function calls within expressions are not supported. Given a C program, we rely on the CIL tool [11] to deal with the idiosyncrasy of C concrete syntax and to produce an equivalent program in Clight abstract syntax. We refer to the CompCert project [9] for a formal definition of the Clight language. Here we just recall in figure C.1 its syntax which is classically structured in expressions, statements, functions, and whole programs. In order to limit the implementation effort, our current compiler for Clight does *not* cover the operators relating to the floating point type `float`. So, in a nutshell, the fragment of C we have implemented is Clight without floating point.

There is a notable difficulty to compile the C language into 8051 assembly code due to the fact that 8051's machine word are 8bits long and C has 32bits primitive datatypes. To deal with the dissimilarity in that case, we first translate the Clight input program into a Clight input program that only uses 8bits primitive datatypes.

C.2 Cminor

Cminor is a simple, low-level imperative language, comparable to a stripped-down, typeless variant of C. Again we refer to the CompCert project for its formal definition and we just recall in figure C.2 its syntax which as for Clight is structured in expressions, statements, functions, and whole programs.

Translation of Clight to Cminor As in Cminor stack operations are made explicit, one has to know which variables are stored in the stack. This information is produced by a static analysis that determines the variables whose address may be 'taken'. Also space is reserved for local arrays and structures. In a second step, the proper compilation is performed: it consists mainly in translating Clight control structures to the basic ones available in Cminor.

C.3 RTLAbs

RTLAbs is the last architecture independent language in the compilation process. It is a rather straightforward *abstraction* of the *architecture-dependent* RTL intermediate language available in the CompCert project and it is intended to factorize some work common to the various target assembly languages (e.g. optimizations) and thus to make retargeting of the compiler a simpler matter.

We stress that in RTLAbs the structure of Cminor expressions is lost and that this may have a negative impact on the following instruction selection step. Still, the subtleties of instruction

Expressions:	$a ::=$	id	variable identifier
		$\mid n$	integer constant
		$\mid$ `sizeof`(τ)	size of a type
		$\mid op_1\ a$	unary arithmetic operation
		$\mid a\ op_2\ a$	binary arithmetic operation
		$\mid *a$	pointer dereferencing
		$\mid a.id$	field access
		$\mid \&a$	taking the address of
		$\mid (\tau)a$	type cast
		$\mid a?a : a$	conditional expression
Statements:	$s ::=$	`skip`	empty statement
		$\mid a = a$	assignment
		$\mid a = a(a^*)$	function call
		$\mid a(a^*)$	procedure call
		$\mid s; s$	sequence
		$\mid$ `if` a `then` s `else` s	conditional
		$\mid$ `switch` $a\ sw$	multi-way branch
		$\mid$ `while` a `do` s	"while" loop
		$\mid$ `do` s `while` a	"do" loop
		$\mid$ `for`$(s,a,s)\ s$	"for" loop
		$\mid$ `break`	exit from current loop
		$\mid$ `continue`	next iteration of the current loop
		$\mid$ `return` $a^?$	return from current function
		$\mid$ `goto` lbl	branching
		$\mid lbl : s$	labelled statement
Switch cases:	$sw ::=$	`default` $: s$	default case
		$\mid$ `case` $n : s; sw$	labelled case
Variable declarations:	$dcl ::=$	$(\tau \quad id)^*$	type and name
Functions:	$Fd ::=$	$\tau\ id(dcl)\{dcl; s\}$	internal function
		$\mid$ `extern` $\tau\ id(dcl)$	external function
Programs:	$P ::=$	$dcl; Fd^*;$ `main` $= id$	global variables, functions, entry point

Figure 2: Syntax of the Clight language

Signatures:	sig ::=	`sig` $\vec{\texttt{int}}$ (`int`|`void`)	arguments and result
Expressions:	a ::=	id	local variable
		| n	integer constant
		| `addrsymbol`(id)	address of global symbol
		| `addrstack`(δ)	address within stack data
		| $op_1\ a$	unary arithmetic operation
		| $op_2\ a\ a$	binary arithmetic operation
		| $\kappa[a]$	memory read
		| $a?a:a$	conditional expression
Statements:	s ::=	`skip`	empty statement
		| $id = a$	assignment
		| $\kappa[a] = a$	memory write
		| $id^? = a(\vec{a}) : sig$	function call
		| `tailcall` $a(\vec{a}) : sig$	function tail call
		| `return`($a^?$)	function return
		| $s; s$	sequence
		| `if` a `then` s `else` s	conditional
		| `loop` s	infinite loop
		| `block` s	block delimiting `exit` constructs
		| `exit` n	terminate the $(n+1)^{th}$ enclosing block
		| `switch` a tbl	multi-way test and exit
		| $lbl : s$	labelled statement
		| `goto` lbl	jump to a label
Switch tables:	tbl ::=	`default:exit`(n)	
		| `case` i: `exit`(n);tbl	
Functions:	Fd ::=	`internal` sig $\vec{id}$ $\vec{id}$ n s	internal function: signature, parameters, local variables, stack size and body
		| `external` id sig	external function
Programs:	P ::=	`prog` $(id = data)^*$ $(id = Fd)^*$ id	global variables, functions and entry point

Figure 3: Syntax of the Cminor language

$$\begin{array}{rcl}
\mathit{return_type} & ::= & \mathsf{int} \mid \mathsf{void} \\
\mathit{signature} & ::= & (\mathsf{int} \rightarrow)^* \ \mathit{return_type} \\
\mathit{memq} & ::= & \mathsf{int8s} \mid \mathsf{int8u} \mid \mathsf{int16s} \mid \mathsf{int16u} \mid \mathsf{int32} \\
\mathit{fun_ref} & ::= & \mathit{fun_name} \mid \mathit{psd_reg} \\
\mathit{instruction} & ::= & \mid \mathsf{skip} \rightarrow \mathit{node} \qquad \text{(no instruction)} \\
 & & \mid \mathit{psd_reg} := \mathit{op}(\mathit{psd_reg}^*) \rightarrow \mathit{node} \qquad \text{(operation)} \\
 & & \mid \mathit{psd_reg} := \&\mathit{var_name} \rightarrow \mathit{node} \qquad \text{(address of a global)} \\
 & & \mid \mathit{psd_reg} := \&\mathsf{locals}[n] \rightarrow \mathit{node} \qquad \text{(address of a local)} \\
 & & \mid \mathit{psd_reg} := \mathit{fun_name} \rightarrow \mathit{node} \qquad \text{(address of a function)} \\
 & & \mid \mathit{psd_reg} := \mathit{memq}(\mathit{psd_reg}[\mathit{psd_reg}]) \rightarrow \mathit{node} \qquad \text{(memory load)} \\
 & & \mid \mathit{memq}(\mathit{psd_reg}[\mathit{psd_reg}]) := \mathit{psd_reg} \rightarrow \mathit{node} \qquad \text{(memory store)} \\
 & & \mid \mathit{psd_reg} := \mathit{fun_ref}(\mathit{psd_reg}^*) : \mathit{signature} \rightarrow \mathit{node} \qquad \text{(function call)} \\
 & & \mid \mathit{fun_ref}(\mathit{psd_reg}^*) : \mathit{signature} \qquad \text{(function tail call)} \\
 & & \mid \mathsf{test}\ \mathit{op}(\mathit{psd_reg}^*) \rightarrow \mathit{node}, \mathit{node} \qquad \text{(branch)} \\
 & & \mid \mathsf{return}\ \mathit{psd_reg}? \qquad \text{(return)} \\
\mathit{fun_def} & ::= & \mathit{fun_name}(\mathit{psd_reg}^*) : \mathit{signature} \\
 & & \mathsf{result} : \mathit{psd_reg}? \\
 & & \mathsf{locals} : \mathit{psd_reg}^* \\
 & & \mathsf{stack} : n \\
 & & \mathsf{entry} : \mathit{node} \\
 & & \mathsf{exit} : \mathit{node} \\
 & & (\mathit{node} : \mathit{instruction})^* \\
\mathit{init_datum} & ::= & \mathsf{reserve}(n) \mid \mathsf{int8}(n) \mid \mathsf{int16}(n) \mid \mathsf{int32}(n) \\
\mathit{init_data} & ::= & \mathit{init_datum}^+ \\
\mathit{global_decl} & ::= & \mathsf{var}\ \mathit{var_name}\{\mathit{init_data}\} \\
\mathit{fun_decl} & ::= & \mathsf{extern}\ \mathit{fun_name}(\mathit{signature}) \mid \mathit{fun_def} \\
\mathit{program} & ::= & \mathit{global_decl}^* \\
 & & \mathit{fun_decl}^*
\end{array}$$

Table 8: Syntax of the RTLAbs language

selection seem rather orthogonal to our goals and we deem the possibility of retargeting easily the compiler more important than the efficiency of the generated code.

Syntax. In RTLAbs, programs are represented as *control flow graphs* (CFGs for short). We associate with the nodes of the graphs instructions reflecting the Cminor commands. As usual, commands that change the control flow of the program (e.g. loops, conditionals) are translated by inserting suitable branching instructions in the CFG. The syntax of the language is depicted in table 8. Local variables are now represented by *pseudo registers* that are available in unbounded number. The grammar rule *op* that is not detailed in table 8 defines usual arithmetic and boolean operations (`+`, `xor`, $\leq$, etc.) as well as constants and conversions between sized integers.

Translation of Cminor to RTLAbs. Translating Cminor programs to RTLAbs programs mainly consists in transforming Cminor commands in CFGs. Most commands are sequential and have a rather straightforward linear translation. A conditional is translated in a branch instruction; a loop is translated using a back edge in the CFG.

$$size ::= \mathsf{Byte} \mid \mathsf{HalfWord} \mid \mathsf{Word} \qquad fun\_ref ::= fun\_name \mid psd\_reg$$

instruction ::=	$\mid \mathsf{skip} \to node$	(no instruction)
	$\mid psd\_reg := n \to node$	(constant)
	$\mid psd\_reg := unop(psd\_reg) \to node$	(unary operation)
	$\mid psd\_reg := binop(psd\_reg, psd\_reg) \to node$	(binary operation)
	$\mid psd\_reg := \&\mathsf{globals}[n] \to node$	(address of a global)
	$\mid psd\_reg := \&\mathsf{locals}[n] \to node$	(address of a local)
	$\mid psd\_reg := fun\_name \to node$	(address of a function)
	$\mid psd\_reg := size(psd\_reg[n]) \to node$	(memory load)
	$\mid size(psd\_reg[n]) := psd\_reg \to node$	(memory store)
	$\mid psd\_reg := fun\_ref(psd\_reg^*) \to node$	(function call)
	$\mid fun\_ref(psd\_reg^*)$	(function tail call)
	$\mid \mathsf{test}\ uncon(psd\_reg) \to node, node$	(branch unary condition)
	$\mid \mathsf{test}\ bincon(psd\_reg, psd\_reg) \to node, node$	(branch binary condition)
	$\mid \mathsf{return}\ psd\_reg?$	(return)

$$fun\_def ::= fun\_name(psd\_reg^*)\ \mathsf{result}:psd\_reg?\ \mathsf{locals}:psd\_reg^*\ \mathsf{stack}:n\ \mathsf{entry}:node\ \mathsf{exit}:node\ (node:instruction)^*$$

$$program ::= \mathsf{globals}:n\ fun\_def^*$$

Table 9: Syntax of the RTL language

C.4 RTL

As in RTLAbs, the structure of RTL programs is based on CFGs. RTL is the first architecture-dependant intermediate language of our compiler which, in its current version, targets the Mips and 8051 assembly languages.

Syntax. RTL is very close to RTLAbs. It is based on CFGs and explicits the Mips or the 8051 instructions corresponding to the RTLAbs instructions. Type information disappears: everything is represented using machine words. Moreover, each global of the program is associated to an offset. The syntax of the language can be found in table 9. The grammar rules *unop*, *binop*, *uncon*, and *bincon*, respectively, represent the sets of unary operations, binary operations, unary conditions and binary conditions of the target assembly language.

Translation of RTLAbs to RTL. This translation is mostly straightforward. A RTLAbs instruction is often directly translated to a corresponding assembly instruction. There are a few exceptions: some RTLAbs instructions are expanded in two or more assembly instructions. When the translation of a RTLAbs instruction requires more than a few simple assembly instruction, it is translated into a call to a function defined in the preamble of the compilation result.

C.5 ERTL

As in RTL, the structure of ERTL programs is based on CFGs. ERTL explicits the calling conventions of the Mips assembly language. In the back-end for 8051, we defined our own

$$size ::= \mathsf{Byte} \mid \mathsf{HalfWord} \mid \mathsf{Word} \qquad fun\_ref ::= fun\_name \mid psd\_reg$$

instruction	::=	$\mid \mathsf{skip} \to node$	(no instruction)
		$\mid \mathsf{NewFrame} \to node$	(frame creation)
		$\mid \mathsf{DelFrame} \to node$	(frame deletion)
		$\mid psd\_reg := \mathsf{stack}[slot, n] \to node$	(stack load)
		$\mid \mathsf{stack}[slot, n] := psd\_reg \to node$	(stack store)
		$\mid hdw\_reg := psd\_reg \to node$	(pseudo to hardware)
		$\mid psd\_reg := hdw\_reg \to node$	(hardware to pseudo)
		$\mid psd\_reg := n \to node$	(constant)
		$\mid psd\_reg := unop(psd\_reg) \to node$	(unary operation)
		$\mid psd\_reg := binop(psd\_reg, psd\_reg) \to node$	(binary operation)
		$\mid psd\_reg := fun\_name \to node$	(address of a function)
		$\mid psd\_reg := size(psd\_reg[n]) \to node$	(memory load)
		$\mid size(psd\_reg[n]) := psd\_reg \to node$	(memory store)
		$\mid fun\_ref(n) \to node$	(function call)
		$\mid fun\_ref(n)$	(function tail call)
		$\mid \mathsf{test}\ uncon(psd\_reg) \to node, node$	(branch unary condition)
		$\mid \mathsf{test}\ bincon(psd\_reg, psd\_reg) \to node, node$	(branch binary condition)
		$\mid \mathsf{return}\ b$	(return)

$$fun\_def ::= fun\_name(n)\ \mathsf{locals}:psd\_reg^*\ \mathsf{stack}:n\ \mathsf{entry}:node\ (node:instruction)^* \qquad program ::= \mathsf{globals}:n\ fun\_def^*$$

Table 10: Syntax of the ERTL language

calling convention since there were none.

Syntax. The syntax of the language is given in table 10. The main difference between RTL and ERTL is the use of hardware registers. Parameters are passed in specific hardware registers; if there are too many parameters, the remaining are stored in the stack. Other conventionally specific hardware registers are used: a register that holds the result of a function, a register that holds the base address of the globals, a register that holds the address of the top of the stack, and some registers that need to be saved when entering a function and whose values are restored when leaving a function. Following these conventions, function calls do not list their parameters anymore; they only mention their number. Two new instructions appear to allocate and deallocate on the stack some space needed by a function to execute. Along with these two instructions come two instructions to fetch or assign a value in the parameter sections of the stack; these instructions cannot yet be translated using regular load and store instructions because we do not know the final size of the stack area of each function. At last, the return instruction has a boolean argument that tells whether the result of the function may later be used or not (this is exploited for optimizations).

Translation of RTL to ERTL. The work consists in expliciting the conventions previously mentioned. These conventions appear when entering, calling and leaving a function, and when referencing a global variable or the address of a local variable.

$$size ::= \mathsf{Byte} \mid \mathsf{HalfWord} \mid \mathsf{Word} \qquad fun\_ref ::= fun\_name \mid hdw\_reg$$

instruction	::=	$\mid \mathsf{skip} \to node$	(no instruction)
		$\mid \mathsf{NewFrame} \to node$	(frame creation)
		$\mid \mathsf{DelFrame} \to node$	(frame deletion)
		$\mid hdw\_reg := n \to node$	(constant)
		$\mid hdw\_reg := unop(hdw\_reg) \to node$	(unary operation)
		$\mid hdw\_reg := binop(hdw\_reg, hdw\_reg) \to node$	(binary operation)
		$\mid hdw\_reg := fun\_name \to node$	(address of a function)
		$\mid hdw\_reg := size(hdw\_reg[n]) \to node$	(memory load)
		$\mid size(hdw\_reg[n]) := hdw\_reg \to node$	(memory store)
		$\mid fun\_ref() \to node$	(function call)
		$\mid fun\_ref()$	(function tail call)
		$\mid \mathsf{test}\ uncon(hdw\_reg) \to node, node$	(branch unary condition)
		$\mid \mathsf{test}\ bincon(hdw\_reg, hdw\_reg) \to node, node$	(branch binary condition)
		$\mid \mathsf{return}$	(return)

$$fun\_def ::= fun\_name(n)\ \mathsf{locals}:n\ \mathsf{stack}:n\ \mathsf{entry}:node\ (node:instruction)^* \qquad program ::= \mathsf{globals}:n\ fun\_def^*$$

Table 11: Syntax of the LTL language

Optimizations. A *liveness analysis* is performed on ERTL to replace unused instructions by a skip. An instruction is tagged as unused when it performs an assignment on a register that will not be read afterwards. Also, the result of the liveness analysis is exploited by a *register allocation* algorithm whose result is to efficiently associate a physical location (a hardware register or an address in the stack) to each pseudo register of the program.

C.6 LTL

As in ERTL, the structure of LTL programs is based on CFGs. Pseudo registers are not used anymore; instead, they are replaced by physical locations (a hardware register or an address in the stack).

Syntax. Except for a few exceptions, the instructions of the language are those of ERTL with hardware registers replacing pseudo registers. Calling and returning conventions were explicited in ERTL; thus, function calls and returns do not need parameters in LTL. The syntax is defined in table 11.

Translation of ERTL to LTL. The translation relies on the results of the liveness analysis and of the register allocation. Unused instructions are eliminated and each pseudo register is replaced by a physical location. In LTL, the size of the stack frame of a function is known; instructions intended to load or store values in the stack are translated using regular load and store instructions.

Optimizations. A *graph compression* algorithm removes empty instructions generated by previous compilation passes and by the liveness analysis.

size	::=	Byte | HalfWord | Word	*fun_ref*	::=	*fun_name* | *hdw_reg*

instruction	::=	| NewFrame	(frame creation)
		| DelFrame	(frame deletion)
		| $hdw\_reg := n$	(constant)
		| $hdw\_reg := unop(hdw\_reg)$	(unary operation)
		| $hdw\_reg := binop(hdw\_reg, hdw\_reg)$	(binary operation)
		| $hdw\_reg := fun\_name$	(address of a function)
		| $hdw\_reg := size(hdw\_reg[n])$	(memory load)
		| $size(hdw\_reg[n]) := hdw\_reg$	(memory store)
		| call *fun_ref*	(function call)
		| tailcall *fun_ref*	(function tail call)
		| $uncon(hdw\_reg) \rightarrow node$	(branch unary condition)
		| $bincon(hdw\_reg, hdw\_reg) \rightarrow node$	(branch binary condition)
		| *asm_label* :	(assembly label)
		| goto *mips_label*	(goto)
		| return	(return)

fun_def	::=	*fun_name*(n) locals :n *instruction**	*program*	::=	globals : n *fun_def**

Table 12: Syntax of the LIN language

C.7 LIN

In LIN, the structure of a program is no longer based on CFGs. Every function is represented as a sequence of instructions.

Syntax. The instructions of LIN are very close to those of LTL. *Program labels*, *gotos* and branch instructions handle the changes in the control flow. The syntax of LIN programs is shown in table 12.

Translation of LTL to LIN. This translation amounts to transform in an efficient way the graph structure of functions into a linear structure of sequential instructions.

C.8 Assembly

Mips is a rather simple assembly language whereas 8051's is rather complex [8]. We only describe Mips. As for other assembly languages, a program in Mips is a sequence of instructions. The Mips code produced by the compilation of a Clight program starts with a preamble in which some useful and non-primitive functions are predefined (e.g. conversion from 8 bits unsigned integers to 32 bits integers). The subset of the Mips assembly language that the compilation produces is defined in table 13.

Translation of LIN to Mips. This final translation is simple enough. Stack allocation and deallocation are explicited and the function definitions are sequentialized.

load ::= lb | lhw | lw *store* ::= sb | shw | sw *fun_ref* ::= *fun_name* | *hdw_reg*

instruction ::=		
	nop	(empty instruction)
	li *hdw_reg*, *n*	(constant)
	unop hdw_reg, *hdw_reg*	(unary operation)
	binop hdw_reg, *hdw_reg*, *hdw_reg*	(binary operation)
	la *hdw_reg*, *fun_name*	(address of a function)
	load hdw_reg, *n*(*hdw_reg*)	(memory load)
	store hdw_reg, *n*(*hdw_reg*)	(memory store)
	call *fun_ref*	(function call)
	uncon hdw_reg, *node*	(branch unary condition)
	bincon hdw_reg, *hdw_reg*, *node*	(branch binary condition)
	mips_label :	(Mips label)
	j *mips_label*	(goto)
	return	(return)

program ::= globals : *n*
entry : *mips_label**
instruction*

Table 13: Syntax of the Mips language

	gcc -O0	acc	gcc -O1
badsort	55.93	34.51	12.96
fib	76.24	34.28	45.68
mat_det	163.42	156.20	54.76
min	12.21	16.25	3.95
quicksort	27.46	17.95	9.41
search	463.19	623.79	155.38

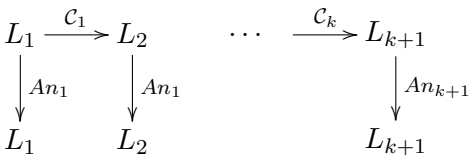
Figure 4: Benchmarks results (execution time is given in seconds).

C.9 Benchmarks

To ensure that our prototype compiler is realistic, we performed some preliminary benchmarks on a 183MHz MIPS 4KEc processor, running a linux based distribution. We compared the wall clock execution time of several simple C programs compiled with our compiler against the ones produced by Gcc set up with optimization levels 0 and 1. As shown by Figure 4, our prototype compiler produces executable programs that are on average faster than Gcc's without optimizations.

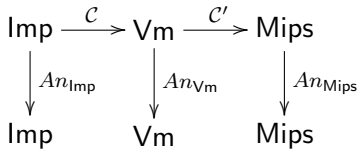
D A direct approach

Our first attempt at building cost annotations followed a 'direct' approach which is summarised by the following diagram.



With respect to the introductive discussion in section 1, L_1 is the source language with the related annotation function An_1 while L_{k+1} is the object language with a related annotation An_{k+1}. This annotation of the object code is supposed to be truly straightforward and it is taken as an 'axiomatic' definition of the 'real' execution cost of the program. The languages L_i, for $2 \leq i \leq k$, are intermediate languages which are also equipped with increasingly 'realistic' annotation functions. Suppose we denote with S the source program, with $\mathcal{C}(S)$ the compiled program, and that we write $(P, s) \Downarrow s'$ to mean that the (source or object) program P in the state s terminates successfully in the state s'. The soundness proof of the compilation function guarantees that if $(S, s) \Downarrow s'$ then $(\mathcal{C}(S), s) \Downarrow s'$. In the direct approach, the *proof of soundness* of the cost annotations amounts to lift the proof of functional equivalence of the source program and the object code to a proof of 'quasi-equivalence' of the respective instrumented codes. Suppose we write $s[c/cost]$ for a state that associates c with the cost variable *cost*. Then what we want to show is that whenever $(An_1(S), s[c/cost]) \Downarrow s'[c'/cost]$ we have that $(An_{k+1}(\mathcal{C}(S)), s[d/cost]) \Downarrow s'[d'/cost]$ and $|d' - d| \leq |c' - c| + k$. This means that the increment in the annotated source program bounds up to an additive constant the increment in the annotated object program. We will also say that the cost annotation is precise if we can also prove that $|c' - c| \leq |d' - d|$, *i.e.*, the 'estimated' cost is not too far away from the 'real' one. We will see that while in theory one can build sound and precise annotation functions, in practice definitions and proofs become unwieldy.

Applying the direct approach to the toy compiler, results in the following diagram.



D.1 Mips and Vm cost annotations

The definition of the cost annotation $An_{\mathsf{Mips}}(M)$ for a Mips code M goes as follows assuming that all the Mips instructions have cost 1.

1. We select fresh locations l_{cost}, l_A, l_B not used in M.
2. Before each instruction in M we insert the following list of 8 instructions whose effect is to increase by 1 the contents of location l_{cost}:

$$(\mathsf{store}\ A, l_A) \cdot (\mathsf{store}\ B, l_B) \cdot (\mathsf{loadi}\ A, 1) \cdot (\mathsf{load}\ B, l_{cost}) \cdot \\ (\mathsf{add}\ A, A, B) \cdot (\mathsf{store}\ A, l_{cost}) \cdot (\mathsf{load}\ A, l_A) \cdot (\mathsf{load}\ B, l_B)\ .$$

$C[i] =$	$\mathsf{cnst}(n)$ or $\mathsf{var}(x)$	add	$\mathsf{setvar}(x)$	$\mathsf{branch}(k)$	$\mathsf{bge}(k)$	halt
$\kappa(C[i]) =$	2	4	2	1	3	1

Table 14: Upper bounds on the cost of Vm instructions

3. We update the offset of the branching instructions according to the map $k \mapsto (9 \cdot k)$.

The definition of the cost annotation $An_{\mathsf{Vm}}(C)$ for a well-formed Vm code C such that $C : h$ goes as follows.

1. We select a fresh *cost* variable not used in C.
2. Before the instruction i^{th} in C, we insert the following list of 4 instructions whose effect is to increase the *cost* variable by the maximum number $\kappa(C[i])$ of instructions associated with $C[i]$ (as specified in table 14):

$$(\mathsf{cnst}(\kappa(C[i]))) \cdot (\mathsf{var}(cost)) \cdot (\mathsf{add}) \cdot (\mathsf{setvar}(cost)) \ .$$

3. We update the offset of the branching instructions according to the map $k \mapsto (5 \cdot k)$.

To summarise, the situation is that the instruction i of the Vm code C corresponds to: the instruction $5 \cdot i + 4$ of the annotated Vm code $An_{\mathsf{Vm}}(C)$, the instruction $p(i, C)$ of the Mips code $\mathcal{C}'(C)$, and the instruction $9 \cdot p(i, C) + 8$ of the instrumented Mips code $An_{\mathsf{Mips}}(\mathcal{C}'(C))$.

The following lemma describes the effect of the injected sequences of instructions.

Lemma 26 *The following properties hold:*
(1) *If M is a* Mips *instruction then* $C_1 \cdot An_{\mathsf{Mips}}(M) \cdot C_2 \vdash (|C_1|, m[c/l_{cost}]) \xrightarrow{*} (|C_1| + 8, m[c + 1/l_{cost}, m(A)/l_A, m(B)/l_B])$.
(2) *If C is a* Vm *instruction then* $C_1 \cdot An_{\mathsf{Vm}}(C) \cdot C_2 \vdash (|C_1|, \sigma, s[c/cost]) \xrightarrow{*} (|C_1| + 4, \sigma, s[c + \kappa(C[i])/cost])$, *where* $i = |C_1|$.

The simulation proposition 21 is extended to the annotated codes as follows where we write $\rightarrow^k$ for the relation obtained by composing k times the reduction relation $\rightarrow$.

Proposition 27 *Let $C : h$ be a well-formed code. If* $An_{\mathsf{Vm}}(C) \vdash (5 \cdot i, \sigma, s) \rightarrow^5 (5 \cdot j, \sigma', s')$, $m \Vdash \sigma, s$, *and* $h(i) = |\sigma|$ *then* $An_{\mathsf{Mips}}(\mathcal{C}'(C)) \vdash (9 \cdot p(i, C), m) \xrightarrow{*} (9 \cdot p(j, C), m')$, *with* $m'[s'(cost)/l_{cost}] \Vdash \sigma', s'$ *and*

$$m'(l_{cost}) - m(l_{cost}) \leq s'(cost) - s(cost) \ .$$

D.2 Imp cost annotation

The definition of the cost annotation $An_{\mathsf{Imp}}(P)$ for an Imp program P is defined in table 15 and it relies on an auxiliary function κ which provides an upper bound on the cost of executing the various operators of the language. For the sake of simplicity, this annotation introduces two main *approximations*:

$$
\begin{array}{ll}
An_{\mathsf{Imp}}(\mathsf{prog}\ S) & = cost := cost + \kappa(S); An_{\mathsf{Imp}}(S) \\
An_{\mathsf{Imp}}(\mathsf{skip}) & = \mathsf{skip} \\
An_{\mathsf{Imp}}(x := e) & = x := e \\
An_{\mathsf{Imp}}(S; S') & = An_{\mathsf{Imp}}(S); An_{\mathsf{Imp}}(S') \\
An_{\mathsf{Imp}}(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S') & = (\mathsf{if}\ b\ \mathsf{then}\ An_{\mathsf{Imp}}(S) \\
 & \qquad \mathsf{else}\ An_{\mathsf{Imp}}(S')) \\
An_{\mathsf{Imp}}(\mathsf{while}\ b\ \mathsf{do}\ S) & = (\mathsf{while}\ b\ \mathsf{do}\ cost := cost + \kappa(b) + \kappa(S) + 1; An_{\mathsf{Imp}}(S))
\end{array}
$$

$$
\begin{array}{ll}
\kappa(\mathsf{skip}) = 0 & \kappa(x := e) = \kappa(e) + \kappa(\mathsf{setvar}) \\
\kappa(S; S') = \kappa(S) + \kappa(S') & \kappa(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ S') = \kappa(b) + max(\kappa(S) + \kappa(\mathsf{branch}), \kappa(S')) \\
\kappa(\mathsf{while}\ b\ \mathsf{do}\ S) = \kappa(b) & \\
\kappa(e < e') = \kappa(e) + \kappa(e') + \kappa(\mathsf{bge}) & \kappa(e + e') = \kappa(e) + \kappa(e') + \kappa(\mathsf{add}) \\
\kappa(n) = \kappa(\mathsf{cnst}) & \kappa(x) = \kappa(\mathsf{var})
\end{array}
$$

Table 15: Annotation for Imp programs

- It over-approximates the cost of an if_then_else by taking the maximum cost of the cost of the two branches.
- It always considers the worst case where the data in the stack resides in the main memory.

We will discuss in section D.5 to what extent these approximations can be removed.

Next we formulate the soundness of the Imp annotation with respect to the Vm annotation along the pattern of proposition 18.

Proposition 28 *The following properties hold.*

(1) *If* $(e, s) \Downarrow v$ *then*

$$C \cdot An_{\mathsf{Vm}}(\mathcal{C}(e)) \cdot C' \vdash (i, \sigma, s[c/cost]) \xrightarrow{*} (j, v \cdot \sigma, s[c + \kappa(e)/cost])$$

where $i = |C|$, $j = |C \cdot An_{\mathsf{Vm}}(\mathcal{C}(e))|$.

(2) *If* $(b, s) \Downarrow \mathsf{true}$ *then*

$$C \cdot An_{\mathsf{Vm}}(\mathcal{C}(b, k)) \cdot C' \vdash (i, \sigma, s[c/cost]) \xrightarrow{*} (5 \cdot k + j, \sigma, s[c + \kappa(b)/cost])$$

where $i = |C|$, $j = |C \cdot An_{\mathsf{Vm}}(\mathcal{C}(b, k))|$.

(3) *If* $(b, s) \Downarrow \mathsf{false}$ *then*

$$C \cdot An_{\mathsf{Vm}}(\mathcal{C}(b, k)) \cdot C' \vdash (i, \sigma, s[c/cost]) \xrightarrow{*} (j, \sigma, s[c + \kappa(b)/cost])$$

where $i = |C|$, $j = |C \cdot An_{\mathsf{Vm}}(\mathcal{C}(b, k))|$.

(4) *If* $(An_{\mathsf{Imp}}(S), s[c/cost]) \Downarrow s'[c'/cost]$ *then*

$$C \cdot An_{\mathsf{Vm}}(\mathcal{C}(S)) \cdot C' \vdash (i, \sigma, s[d/cost]) \xrightarrow{*} (j, \sigma, s'[d'/cost])$$

where $i = |C|$, $j = |C \cdot An_{\mathsf{Vm}}(\mathcal{C}(S))|$, *and* $(d' - d) \leq (c' - c) + \kappa(S)$.

D.3 Composition

The soundness of the cost annotations can be composed so as to obtain the soundness of the cost annotations of the source program relatively to the one of the object code.

Proposition 29 *If* $(An_{\mathsf{Imp}}(P), s[0/cost]) \Downarrow s'[c'/cost]$ *and* $m \Vdash \epsilon, s[0/l_{cost}]$ *then*

$$(An_{\mathsf{Mips}}(\mathcal{C}'(\mathcal{C}(P))), m) \Downarrow m'$$

where $m'(l_{cost}) \leq c'$ *and* $m'[c'/l_{cost}] \Vdash \epsilon, s'[c'/cost]$.

D.4 Coq development

We have formalised and mechanically checked in Coq the application of the direct approach to the toy compiler (but for propositions 27 and 29 for which we produced 'paper proofs'). By current standards, this is a small size development including 1000 lines of specifications and 4000 lines of proofs. Still there are a couple of points that deserve to be mentioned. First, we did not find a way of re-using the soundness proof of the compiler in a modular way. As a matter of fact, the soundness proof of the annotations is intertwined with the soundness proof of the compiler. Second, the manipulation of the cost variables in the annotated programs entails a significant increase in the size of the proofs. In particular, the soundness proof for the compilation function $\mathcal{C}$ from Imp to Vm available in [10] is roughly 7 times smaller than the soundness proof of the annotation function of the Imp code relatively to the one of the Vm code.

D.5 Limitations of the direct approach

As mentioned in section D.2, the annotation function for the source language introduces some over-approximation thus *failing to be precise*. On one hand, it is easy to modify the definitions in table 15 so that they compute the cost of each branch of an `if_then_else` separately rather than taking the maximum cost of the two branches. On the other hand, it is rather difficult to refine the annotation function so that it accounts for the memory hierarchy in the Mips machine; one needs to pass to the function κ an 'hidden parameter' which corresponds to the stack height. This process of pushing hidden parameters into the definition of the annotation is error prone and it seems unlikely to work in practice for a realistic compiler. We programmed sound (but not precise) cost annotations for the C compiler introduced in section 1 and found that the approach is difficult to test because an over-approximation of the cost at some point may easily compensate an under-approximation somewhere else. By contrast, in the labelling approach introduced in section 1, we manipulate costs at an abstract level as labels and produce numerical values only at the very end of the construction.

E Related approaches

Our fine-grained approach to labelling is based on the hypothesis that we can obtain precise information on the execution time of each instruction of the generated binary code. This hypothesis is indeed realised in the processors of the 8051 family we are considering. On the other hand, the execution time of instructions running on more complex architectures including, *e.g.*, cache memory or pipelines are much less predictable. This lack of precision may somehow be compensated by analysing the worst-case execution time of sequences of instructions. As mentioned in the introduction, tools such as those developped by AbsInt require explicit annotations of the binaries to determine the number of times a loop is iterated and apply abstract interpretation methods in order to provide an estimation of the execution time of the code. From the point of view of a *formal* certification this approach rises at least two questions.

First, one needs a formal certification of the fact that the abstract interpretation method does indeed produce correct results for a given processor. This in turn supposes a formal modelling of the processor and a proof that the abstract interpretation method does indeed abstract the processor's behaviour. Taking a pragmatic (and informal) approach, one could argue that the results of the tool could be trusted, in the same way we trust the execution times given by the manufacturers.

Second, one needs a formal certification of the fact that the annotations on the loops are indeed correct. For instance, suppose we can identify in the source code *for-loops* that iterate a *constant* number of times some sequence of statements. In our approach, we must insert a label in the body of the loop so as to avoid that the control flow graph associated with the binary code contains a loop that does not cross any label. Having determined the cost of one iteration of the body of the loop, we can then reason at the level of the source code to obtain the cost of several iterations.

In an alternative approach, we could try to propagate to the control flow graph the information that the body of the loop is iterated a constant number of times and then invoke tools such as those mentioned above to get an estimation of the cost of running the whole iteration. As mentioned above, the rationale for this approach is to get a more precise estimation of the cost of executing the iteration on a complex architecture. While the approach seems easily implementable, it is not clear how one should proceed in order to formally certify its correctness. Specifically, one needs to propagate enough information at the level of the binary code to be able to verify automatically that the predicted number of iterations of the body of the loop is indeed correct. In turn, this means to be able to track down the initial value of a certain number of memory locations (registers and/or stack locations) and to guarantee that their contents is modified at each iteration leading eventually to the exit of the loop. While this seems within the reach of a *proof-carrying code* approach, the challenge is to devise a more lightweight approach which targets the specific properties of interest.

F Assessment of the deliverable within the *CerCo* project, with hindsight

The port of the compiler to the 8051 processor has required a major and rather unrewarding implementation effort which is described in deliverable D2.2.

One problem that has emerged is that the compilation of general purpose C programs with recursion and/or 16-32 integers bits produces binaries whose size approaches quickly the limits of the architecture (program memory is limited to 64Kbytes).

This is a problem for instance for the case study which is planned in deliverable D5.3 (Case study: analysis of synchronous code) whose purpose is to produce automatically cost annotations for the C programs generated by a compiler of a synchronous language such as Lustre. The generated C programs are rather large (though their control flow is simple) and assume 32 bits integers.

At the time being, two options are being considered as far as deliverable D5.3 is concerned.

1. Improve the efficiency of the compilation of 16/32 bits integers for the 8051 compiler.
2. Conduct the case study relying on a compiler to a 32 bits processor.

If the second option is chosen, one possibility is to target the Mips processor for which a prototype compiler has already been developed. More generally, one advantage of the second option is that it allows to conduct realistic case studies on a larger class of programming constructs. For instance, we are interested in studying the extension of the labelling approach to a compilation chain from a functional language of the ML family to C.