CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report
D5.1 Untrusted CerCo Prototype

Version 1.0

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

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

Summary The deliverable D5.1 is composed of the following parts:

1. This summary.
2. The paper [1] and the related software Cost.
3. The paper [2] and the related software.

This document and the softwares above can be downloaded at the page .

`http://cerco.cs.unibo.it/`

References

[1] N. Ayache. Synthesis of certified cost bounds. Université Paris Diderot. Internal report documenting the Cost software, 2012.

[2] P. Tranquilli. Indexed labels for loop iteration dependent costs. Università di Bologna. Internal report documenting the indexed labels software, 2012.

Aim The main aim of WP5 is to develop proof of concept prototypes where the (untrusted) compiler implemented in WP2 is interfaced with existing tools in order to synthesize complexity assertions on the execution time of programs. Eventually, the approach should be adapted to the trusted compiler developed in WP3 and WP4 (cf. deliverable D5.2 at month 36).

Synthesis of certified cost bounds The main planned contribution of deliverable D5.1 is a tool that takes as input an annotated C program produced by the CerCo compiler and tries to synthesize a certified bound on the execution time of the program. The related expected contribution of deliverable D5.3 amounts to apply the developed tool to the C programs generated by a Lustre compiler. This work is described in the first document [1] which accompanies a software distribution called Cost. The development takes the form of a 'Frama − C plug-in'. Frama − C is an open source and well-established platform to reason formally on C programs. The proof obligations generated from Hoare style assertions on C programs are passed to a small number of provers that try to discharge them automatically. The platform has been designed to be extensible by means of so called plug-in's written in ocaml. The Cost software is a Frama − C plug-in which in first approximation takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler to produce a related C program with cost annotations, (3) it applies some heuristics to produce a tentative bound on the cost of executing a C function as a function of the value of its parameters, (4) it calls the provers embedded in the Frama − C tool to discharge the related proof obligations. The current size of the Cost plug-in is 4K lines of ocaml code. More details are available in the the document [1].

Indexed labels for loop iteration dependent costs The first year scientific review report, among other things, contrasts the CerCo approach with the one adopted in tools such as AbsInt which are used by the WCET community and it recommends that the approach to cost annotations described in WP2 is made *coarser*, *i.e.*, that a label covers a larger portion of code. During the second year, most of the work of a post-doc at UNIBO was aimed at addressing this remark [2]. This has resulted in a refinement of the labelling approach into a so called *indexed labelling*. It consists in formally indexing cost labels with the iterations of the containing loops they occur in within the source code. These indexes can be transformed during the compilation, and when lifted back to source code they produce dependent costs. Preliminary experiments suggest that this refinement allows to retain preciseness when the program is subject to loop transformations such as loop peeling and loop unrolling. A prototype implementation has been developed on top of the untrusted CerCo compiler D2.2.

Synthesis of certified cost bounds

Nicolas Ayache

Abstract

The CerCo project aims at building a certified compiler for the C language that can lift in a provably correct way information on the execution cost of the object code to cost annotations on the source code. These annotations are added at specific program points (e.g. inside loops). In this article, we describe a plug-in of the Frama − C platform that, starting from CerCo's cost annotation of a C function, synthesizes a cost bound for the function. We report our experimentations on some C code.

1 Introduction

Estimating the worst case execution time (WCET) of an embedded software is an important task, especially in a critical system. The micro-controller running the system must be efficiently used: money and reaction time depend on it. However, computing the WCET of a program is undecidable in the general case, and static analysis tools dedicated to this task often fail when the program involves complicated loops, living few hopes for the user to obtain a result.

In this article, we present experiments that validate the new approach introduced by the CerCo project[1] for WCET prediction. With CerCo, the user is provided raw and certified cost annotations. We design a tool that uses these annotations to generate WCET bounds. When the tool fails, we show how the user can complete the required information, so as to never be stuck. The tool is able to fully automatically compute and certify a WCET for a C function with loops and whose cost is dependent on its parameters. We briefly recall the goal of CerCo, and we present the platform used both to develop our tool and verify its results, before describing our contributions.

CerCo. The CerCo project aims at building a certified compiler for the C language that lifts in a provably correct way information on the execution cost of the object code to cost annotations on the source code. An untrusted compiler has been developed [2] that targets the 8051, a popular micro-controller typically used in embedded systems. The compiler relies on the *labelling approach* to compute the cost annotations: at the C level, specific program points — called *cost labels* — are identified in the control flow of the program. Each cost label is a symbolic value that represents the cost of the instructions following the label and before the next one. Then, the compilation keeps track of the association between program points and cost labels. In the end, a concrete cost is computed for each cost label from the object code, and the information is sent up to the C level for instrumentation. Figure 1a shows a C code, and figure 1b presents its transformation through CerCo.

As one notices, the result of CerCo is an instrumentation of the input C program:

[1] `http://cerco.cs.unibo.it/`

```
int is_sorted (int *tab, int size) {
  int i, res = 1;

  for (i = 0 ; i < size-1 ; i++) if (tab[i] > tab[i+1]) res = 0;

  return res;
}
```

(a) before CerCo

```
int _cost = 0;

void _cost_incr (int incr) { _cost = _cost + incr; }

int is_sorted (int *tab, int size) {
  int i, res = 1;

  _cost_incr(97);

  for (i = 0; i < size-1; i++) {
    _cost_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

(b) after CerCo

Figure 1: An example of CerCo's action

- a global variable called `_cost` is added. Its role is to hold the cost information during execution;
- a `_cost_incr` function is defined; it will be used to update the cost information;
- finally, update instructions are inserted inside the functions of the program: those are the cost annotations. In the current state of the compiler, they represent the number of processor's cycles that will be spent executing the following instructions before the next annotation. But other kind of information could be computed using the labelling approach, such as stack size for instance.

Frama − C. In order to deduce an upper bound of the WCET of a C function, we need a tool that can analyse C programs and relate the value of the `_cost` variable before and after the function is executed. We chose to use the Frama − C verification tool [4] for the following reasons:

- the platform allows all sorts of analyses in a modular and collaborative way: each analysis is a plug-in that can reuse the results of existing ones. The authors of Frama − C provide

a development guide for writing new plug-ins. Thus, if existing plug-ins experience difficulties in synthesizing the WCET of C functions annotated with CerCo, we can define a new analysis dedicated to this task;

- it supports ACSL, an expressive specification language à la Hoare logic as C comments. Expressing WCET specification using ACSL is very easy;
- the Jessie plug-in builds verification conditions (VCs) from a C program with ACSL annotations. The VCs can be sent to various provers, be they automatic or interactive. When they are discharged, the program is guaranteed to respect its specification.

Figure 2 shows the program of figure 1b with ACSL annotations added manually. The most important is the post-condition attached to the `is_sorted` function:

```
ensures _cost <= \old(_cost) + 101 + (size-1)*195;
```

It means that executing the function yields the value of the `_cost` variable to be incremented by at most `101 + (size-1)*195`: this is the WCET specification of the function. Running the Jessie plug-in on this program creates 8 VCs that an automatic prover such as Alt – Ergo[2] is able to fully discharged, which proves that the WCET specification is indeed correct.

```
int _cost = 0;

/*@ ensures _cost == \old(_cost) + incr; */
void _cost_incr (int incr) { _cost = _cost + incr; }

/*@ requires size >= 1;
  @ ensures _cost <= \old(_cost) + 101 + (size-1)*195; */
int is_sorted (int *tab, int size) {
  int i, res = 1;

  _cost_incr(97);

  /*@ loop invariant i < size;
    @ loop invariant _cost <= \at(_cost, Pre) + 97 + i*195;
    @ loop variant size-i; */
  for (i = 0; i < size-1; i++) {
    _cost_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

Figure 2: Annotations with ACSL

[2] http://ergo.lri.fr/

Contributions. This paper describes a possible back-end for CerCo's framework. It validates the approach with a tool that uses CerCo's results to automatically or semi-automatically compute and verify the WCET of C functions. It is yet one of the many possibilities of using CerCo for WCET validation, and shows its benefit: WCET computation is not a black box as it is usually, and the user can understand and complete manually what the tool failed to compute.

In the remaining of the article, we present a Frama – C plug-in called Cost that adds a WCET specification to the functions of a CerCo-annotated C program. Section 2 briefly details the inner workings of the plug-in and discusses its soundness. Section 3 compares our approach with other WCET tools. Section 4 shows some benchmarks on standard C programs and on C programs for cryptography (typically used in embedded software). Finally, section 5 concludes.

2 The Cost plug-in

The Cost plug-in for the Frama – C platform has been developed in order to automatically synthesize the cost annotations added by the CerCo compiler on a C source program into assertions of the WCET of the functions in the program. The architecture of the plug-in is depicted in figure 3. It accepts a C source file for parameter and creates a new C file that is the former with additional cost annotations (C code) and WCET assertions (ACSL annotations). First, the input file is fed to Frama – C that will in turn send it to the Cost plug-in. The action of the plug-in is then:

1. apply the CerCo compiler to the source file;
2. synthesize an upper bound of the WCET of each function of the source program by reading the cost annotations added by CerCo;
3. add the results in the form of post-conditions in ACSL format, relating the cost of the function before and after its execution.

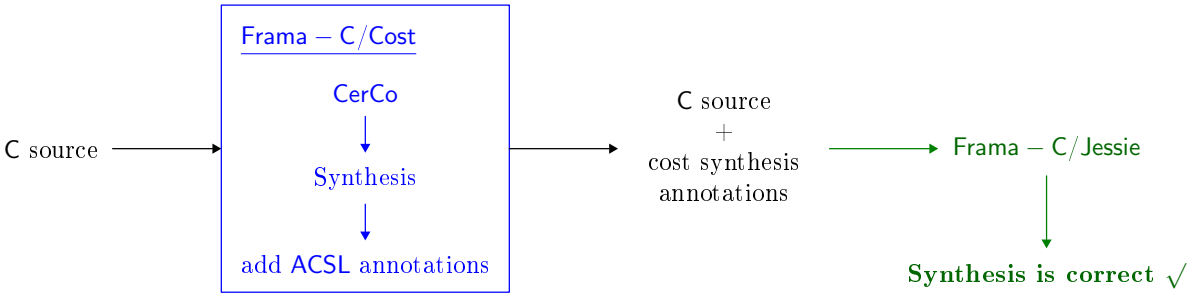

Figure 3: the Cost plug-in

Then, the user can either *trust* the results (the WCET of the functions), or want to *verify* them, in which case he can call Jessie.

We continue our description of the plug-in by discussing the soundness of the framework, because, as we will see, the action of the plug-in is not involved in this issue. Then, the details of the plug-in will be presented.

2.1 Soundness

As figure 3 suggests, the soundness of the whole framework depends on the cost annotations added by CerCo, the synthesis made by the Cost plug-in, the VCs generated by Jessie, and the VCs discharged by external provers. Since the Cost plug-in adds annotations in ACSL format, Jessie (or other verification plug-ins for Frama − C) can be used to verify these annotations. Thus, even if the added annotations are incorrect, the process in its globality is still correct: indeed, Jessie will not validate incorrect annotations and no conclusion can be made about the WCET of the program in this case. This means that the Cost plug-in can add *any* annotation for the WCET of a function, the whole framework will still be correct and thus its soundness does not depend on the action of the Cost plug-in. However, in order to be able to actually prove a WCET of a function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce validity.

2.2 Inner workings

The cost annotations added by the CerCo compiler take the form of C instructions that update by a constant a fresh global variable called the *cost variable*. Synthesizing a WCET of a C function thus consists in statically resolving an upper bound of the difference between the value of the cost variable before and after the execution of the function, i.e. find in the function the instructions that update the cost variable and establish the number of times they are passed through during the flow of execution. This raises two main issues: indecidability caused by loop constructs, and function calls. Indeed, a side effect of function calls is to change the value of the cost variable. When a function calls another one, the cost of the callee is part of the cost of the caller. This means that the computation of a WCET of each function of a C program is subject to the calling dependencies. To cope with the issues of loops and function calls, the Cost plug-in proceeds as follows:

- each function is independently processed and associated a WCET that may depend on the cost of the other functions. This is done with a mix between abstract interpretation [5] and syntactic recognition of specific loops for which we can decide the number of iterations. The abstract domain used is made of expressions whose variables can only be formal parameters of the function;
- a system of inequations is built from the result of the previous step, and is tried to be solved with a fixpoint. At each iteration, the fixpoint replaces in all the inequations the references to the cost of a function by its associated cost if it is independent of the other functions;
- ACSL annotations are added to the program according to the result of the above fixpoint. Note that the two previous steps may fail in finding a concrete WCET for some functions, because of imprecision inherent to abstract interpretation, and recursion in the source program not solved by the fixpoint. At each program point that requires an annotation (function definitions and loops), annotations are added if a solution was found for the program point.

Figure 4 shows the result of the Cost plug-in when fed the program in figure 1a. There are several differences from the manually annotated program, the most noticeable being:

- the manually annotated program had a pre-condition that the `size` formal parameter needed to be greater or equal to 1. The Cost plug-in does not make such an assumption, but instead considers both the case where `size` is greater or equal to 1, and the case where it is not. This results in a ternary expression inside the WCET specification (the post-condition or `ensures` clause), and some new loop invariants;
- the loop invariant specifying the value of the cost variable depending on the iteration number refers to a new local variable named `_cost_tmp0`. It represents the value of the cost variable right before the loop is executed. It allows to express the cost inside the loop with regards to the cost before the loop, instead of the cost at the beginning of the function; it often makes the expression a lot shorter and eases the work for nested loops.

Running Jessie on the program generates VCs that are all proved by Alt − Ergo: the WCET computed by the Cost plug-in is correct.

```
int _cost = 0;

/*@ ensures _cost ≡ \old(_cost) + incr; */
void _cost_incr (int incr) { _cost = _cost + incr; }

/*@ ensures (_cost ≤ \old(_cost)+(101+(0<size-1?(size-1)*195:0))); */
int is_sorted (int *tab, int size) {
  int i, res = 1, _cost_tmp0;

  _cost_incr(97);

  _cost_tmp0 = _cost;
  /*@ loop invariant (0 < size-1) ⇒ (i ≤ size-1);
    @ loop invariant (0 ≥ size-1) ⇒ (i ≡ 0);
    @ loop invariant (_cost ≤ _cost_tmp0+i*195);
    @ loop variant (size-1)-i; */
  for (i = 0; i < size-1; i++) {
    _cost_incr(91);
    if (tab[i] > tab[i+1]) { _cost_incr(104); res = 0; }
    else _cost_incr(84);
  }

  _cost_incr(4);
  return res;
}
```

Figure 4: Result of the Cost plug-in

3 Related work

There exist a lot of tools for WCET analysis. Yet, the framework encompassing the Cost plug-in is the only one, to our knowledge, that enjoys the following features:

- The results of the plug-in have a very high level of trust. First, because the cost annotations added by CerCo are proven correct (this is on-going research in the Matita[3] system). Second, because verification with Jessie is deductive and VCs can be discharged with various provers. The more provers discharge a VC, the more trustful is the result. When automatic provers fail in discharging a VC, the user can still try to verify them manually, with an interactive theorem prover such as Coq[4] that Jessie outputs to.

- While other WCET tools act as black boxes, the Cost plug-in provides the user with as many information as it can. When a WCET tool fails, the user generally have few hopes, if any, of understanding and resolving the issue in order to obtain a result. When the Cost plug-in fails to add an annotation, the user can still try to complete it. And since the results of CerCo is C code, it is much easier to understand the behavior of the annotations.

- The results of the Cost plug-in being added to the source C file, it allows to easily identify the cost of parts of the code and the cost of the functions of the program. The user can modify parts that are too costly and observe their precise influence on the overall cost.

- The framework is modular: the Cost plug-in is yet one possible synthesis, and Jessie is one possible back-end for verification. We can use other synthesis strategies, and choose for each result the one that seems the most precise. The same goes for Jessie: we can use the WP plug-in of Frama − C instead, and even merge the results of both. Similarly, if we were to support more complex architectures, computing the cost of object code instructions could be dedicated to an external tool that is able to provide precise results even in the presence of cache, pipelines, etc [6].

4 Experiments

The Cost plug-in has been developed in order to validate CerCo's framework for modular WCET analysis, by showing the results that could be obtained with this approach. The plug-in allows (semi-)automatic generation and certification of WCET for C programs. Also, we designed a wrapper for supporting Lustre files. Indeed, Lustre is used for developing embedded software for critical systems and the language is distributed with a compiler to C, in such a way that the WCET of the result represents a bound for the reaction time of the system. This section presents results obtained on C programs typically found in embedded software, where WCET is of great importance.

The Cost plug-in is written in 3895 lines of ocaml. They mainly cover an abstract interpretation of C together with a syntactic recognition of specific loops, in a modular fashion: the abstract domains (one for C values and another for cost values) can be changed for better precision. The Lustre wrapper is made of 732 lines of ocaml consisting in executing a command, reading the results and sending them to the next command.

We ran the plug-in and the Lustre wrapper on some files found on the web, from the Lustre distribution or written by hand. For each file, we report its type (either a standard algorithm written in C, a cryptographic protocol for embedded software, or a C program generated from Lustre file), a quick description of the program, the number of lines of the original code and

[3] http://matita.cs.unibo.it/

[4] http://coq.inria.fr/

the number of VCs generated. A WCET is found by the Cost plug-in for everyone of these programs, and Alt − Ergo was able to discharge all VCs.

File	Type	Description	LOC	VCs
`3-way.c`	C	Three way block cipher	144	34
`a5.c`	C	A5 stream cipher, used in GSM cellular	226	18
`array_sum.c`	S	Sums the elements of an integer array	15	9
`fact.c`	S	Factorial function, imperative implementation	12	9
`is_sorted.c`	S	Sorting verification of an array	8	8
`LFSR.c`	C	32-bit linear-feedback shift register	47	3
`minus.c`	L	Two modes button	193	8
`mmb.c`	C	Modular multiplication-based block cipher	124	6
`parity.lus`	L	Parity bit of a boolean array	359	12
`random.c`	C	Random number generator	146	3
S: standard algorithm C: cryptographic protocol L: C generated from a Lustre file				

Programs fully automatically supported. Since the goal of the Cost plug-in is a proof of concept of a full framework with CerCo, we did not put too much effort or time for covering a wide range of programs. CerCo always succeeds, but the Cost plug-in may fail in synthesizing a WCET, and automatic provers may fail in discharging some VCs. We can improve the abstract domains, the form of recognized loops, or the hints that help automatic provers. For now, a typical program that is processed by the Cost plug-in and whose VCs are fully discharged by automatic provers is made of loops with a counter incremented or decremented at the end of the loop, and where the guard condition is a comparison of the counter with some expression. The expressions incrementing or decrementing the counter and used in the guard condition must be so that the abstract interpretation succeeded in relating them to an arithmetic expressions whose variables are parameters of the function. With a flat domain currently used, this means that the values of these expressions may not be modified during a loop.

5 Conclusion

We have described a plug-in for Frama − C that relies on the CerCo compiler to automatically or semi-automatically synthesize a WCET for C programs. The soundness of the overall process is guaranteed through the Jessie plug-in. Finally, we successfully used the plug-in on some C programs, some of which are typically used in embedded software; the result is an automatically computed and certified reaction time for the programs. This experience validates the modular approach of CerCo for WCET computation with a high level of trust.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam,*

The Netherlands, October 24-26, 2007, Revised Lectures, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2008.

[2] R. M. Amadio, N. Ayache, and Y. Régis-Gianas. Deliverable 2.2: Prototype implementation. Technical report, ICT Programme, Feb. 2011. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[3] R. M. Amadio, N. Ayache, Y. Régis-Gianas, K. Memarian, and R. Saillard. Deliverable 2.1: Compiler design and intermediate languages. Technical report, ICT Programme, July 2010. Project FP7-ICT-2009-C-243881 CerCo - Certified Complexity.

[4] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. Frama-C user manual. CEA-LIST, Software Safety Laboratory, Saclay, F-91191. `http://frama-c.com/`.

[5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[6] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society.

Indexed Labels for Loop Iteration Dependent Costs

Paolo Tranquilli

Abstract

We present an extension to the labelling approach to lift resource consumption information from compiled to source code [3]. Such an approach consists in inserting cost labels at key points of the source code and keeping track of them during compilation. However, the plain labelling approach looses preciseness when differences arise as to the cost of the same portion of code, whether due to code transformation such as loop optimisation or advanced architecture features (*e.g.* cache). Our approach addresses this weakness, allowing to retain preciseness even when applying some loop transformations that rearrange the iterations of a loop (namely loop peeling and unrolling). It consists in formally indexing cost labels with the iterations of the containing loops they occur in within the source code. These indexes can be transformed during the compilation, and when lifted back to source code they produce dependent costs.

The proposed changes have been implemented in CerCo's untrusted prototype compiler from a large fragment of C to 8051 assembly [4].

1 Introduction

In [3], Armadio *et al* propose an approach for building a compiler for a large fragment of the C programming language. The novelty of their proposal lies in the fact that their proposed design is capable of lifting execution cost information from the compiled code and presenting it to the user. This idea is foundational for the CerCo project, which strives to produce a mechanically certified version of such a compiler.

To summarise, Armadio's proposal consisted of 'decorations' on the source code, along with the insertion of labels at key points. These labels are preserved as compilation progresses, from one intermediate language to another. Once the final object code is produced, such labels should correspond to the parts of the compiled code that have a constant cost.

Two properties must hold of any cost estimate. The first property, paramount to the correctness of the method, is *soundness*, that is, that the actual execution cost is bounded by the estimate. In the labelling approach, this is guaranteed if every loop in the control flow of the compiled code passes through at least one cost label. The second property, optional but desirable, is *preciseness*: the estimate *is* the actual cost. In the labelling approach, this will be true if, for every label, every possible execution of the compiled code starting from such a label yields the same cost before hitting another one. In simple architectures such as the 8051 micro-controller this can be guaranteed by placing labels at the start of any branch in the control flow, and by ensuring that no labels are duplicated.

The reader should note that the above mentioned requirements must hold when executing the code obtained at the end of the compilation chain. So even if one is careful about injecting the labels at suitable places in the source code, the requirements might still fail because of two main obstacles:

- The compilation process introduces important changes in the control flow, inserting loops or branches. For example, the insertion of functions in the source code replacing instructions that are unavailable in the target architecture. This require loops to be inserted (for example, for multi-word division and generic shift in the 8051 architecture), or effort spent in providing unbranching translations of higher level instructions [4].

- Even when the compiled code *does*—as far as the the syntactic control flow graph is concerned—respect the conditions for soundness and preciseness, the cost of blocks of instructions might not be independent of context, so that different passes through a label might have different costs. This becomes a concern if one wishes to apply the approach to more complex architectures, for example one with caching or pipelining.

The first point unveils a weakness of the current labelling approach when it comes to some common code transformations performed along a compilation chain. In particular, most *loop optimisations* are disruptive, in the sense outlined in the first bulletpoint above. An example optimisation of this kind is *loop peeling*. This optimisation is employed by compilers in order to trigger other optimisations, for example, dead code elimination or invariant code motion. Here, a first iteration of the loop is hoisted out of the body of the loop, possibly being assigned a different cost than later iterations.

The second bulletpoint above highlights another weakness. Different tools allow to predict up to a certain extent the behaviour of cache. For example, the well known tool aiT [1]—based on abstract interpretation—allows the user to estimate the worst-case execution time (WCET) of a piece of source code, taking into account advanced features of the target architecture. While such a tool is not fit for a compositional approach which is central to CerCo's project[1], aiT's ability to produce tight estimates of execution costs would sthill enhance the effectiveness of the CerCo compiler, *e.g.* by integrating such techniques in its development. A typical case where cache analysis yields a difference in the execution cost of a block is in loops: the first iteration will usually stumble upon more cache misses than subsequent iterations.

If one looks closely, the source of the weakness of the labelling approach as presented in [3] is common to both points: the inability to state different costs for different occurrences of labels, where the difference might be originated by labels being duplicated along the compilation, or the costs being sensitive to the current state of execution. The preliminary work we present here addresses this weakness by introducing cost labels that are dependent on which iteration of its containing loops it occurs in. This is achieved by means of *indexed labels*; all cost labels are decorated with formal indices coming from the loops containing such labels. These indices allow us to rebuild, even after multiple loop transformations, which iterations of the original loops in the source code a particular label occurrence belongs to. During the annotation stage of the source code, this information is presented to the user by means of *dependent costs*.

We concentrate on integrating the labelling approach with two loop transformations. For general information on general compiler optimisations (and loop optimisations in particular) we refer the reader to the vast literature on the subject (*e.g.* [8, 7]).

Loop peeling As already mentioned, loop peeling consists in preceding the loop with a copy of its body, appropriately guarded. This is used, in general, to trigger further optimisations, such as those that rely on execution information which can be computed at compile time,

[1] aiT assumes the cache is empty at the start of computation, and treats each procedure call separately, unrolling a great part of the control flow.

but which is erased by further iterations of the loop, or those that use the hoisted code to be more effective at eliminating redundant code. Integrating this transformation in to the labelling approach would also allow the integration of the common case of cache analysis explained above; the analysis of cache hits and misses usually benefits from a form of *virtual* loop peeling [5].

Loop unrolling This optimisation consists of the repetition of several copies of the body of the loop inside the loop itself (inserting appropriate guards, or avoiding them altogether if enough information about the loop's guard is available at compile time). This can limit the number of (conditional or unconditional) jumps executed by the code and trigger further optimisations dealing with pipelining, if appropriate for the architecture.

Whilst we cover only two loop optimisations in this report, we argue that the work presented herein poses a good foundation for extending the labelling approach, in order to cover more and more common optimisations, as well as gaining insight into how to integrate advanced cost estimation techniques, such as cache analysis, into the CerCo compiler. Moreover loop peeling itself has the fortuitous property of enhancing and enabling other optimisations. Experimentation with CerCo's untrusted prototype compiler, which implements constant propagation and partial redundancy elimination [6, 8], show how loop peeling enhances those other optimisations.

Outline We will present our approach on a minimal 'toy' imperative language, Imp with gotos, which we present in Section 2 along with formal definitions of the loop transformations. This language already presents most of the difficulties encountered when dealing with C, so we stick to it for the sake of this presentation. In Section 3 we summarize the labelling approach as presented in [3]. Section 4 presents *indexed labels*, our proposal for dependent labels which are able to describe precise costs even in the presence of the various loop transformations we consider. Finally Section 5 goes into more detail regarding the implementation of indexed labels in CerCo's untrusted compiler and speculates on further work on the subject.

2 Imp with goto

We briefly outline the toy language, Imp with gotos. The language was designed in order to pose problems for the existing labelling approach, and as a testing ground for our new notion of indexed labels.

The syntax and operational semantics of our toy language are presented in 1. Note, we may augment the language further, with break and continue, at no further expense. The precise grammar for expressions is not particularly relevant so we do not give one in full. For the sake of conciseness we also treat boolean and arithmetic expressions together (with the usual C convention of an expression being true iff non-zero). We may omit the else clause of a conditional if it leads to a skip statement.

We will presuppose that all programs are *well-labelled*, *i.e.* every label labels at most one occurrence of a statement in a program, and every goto points to a label actually present in the program. The find helper function has the task of not only finding the labelled statement in the program, but also building the correct continuation. The continuation built by find replaces the current continuation in the case of a jump.

Further down the compilation chain We abstract over the rest of the compilation chain. We posit the existence of a suitable notion of 'sequential instructions', wherein each instruction has a single natural successor to which we can add our own, for every language L further down the compilation chain.

2.1 Loop transformations

We call a loop L *single-entry* in P if there is no goto to P outside of L which jumps into L.[2] Many loop optimisations do not preserve the semantics of multi-entry loops in general, or are otherwise rendered ineffective. Usually compilers implement a single-entry loop detection which avoids the multi-entry ones from being targeted by optimisations [8, 7]. The loop transformations we present are local, *i.e.* they target a single loop and transform it. Which loops are targeted may be decided by some *ad hoc* heuristic. However, the precise details of which loops are targetted and how is not important here.

Loop peeling

$$\mathsf{while}\ b\ \mathsf{do}\ S \mapsto \mathsf{if}\ b\ \mathsf{then}\ S; \mathsf{while}\ b\ \mathsf{do}\ S[\ell_i'/\ell_i]$$

where ℓ_i' is a fresh label for any ℓ_i labelling a statement in S. This relabelling is safe for gotos occurring outside the loop because of the single-entry condition. Note that for break and continue statements, those should be replaced with gotos in the peeled body S.

Loop unrolling

$$\mathsf{while}\ b\ \mathsf{do}\ S \mapsto \mathsf{while}\ b\ \mathsf{do}\ (S; \mathsf{if}\ b\ \mathsf{then}\ (S[\ell_i^1/\ell_i]; \cdots \mathsf{if}\ b\ \mathsf{then}\ S[\ell_i^n/\ell_i]) \cdots)$$

where ℓ_i^j are again fresh labels for any ℓ_i labelling a statement in S. This is a wilfully naïve version of loop unrolling, which usually targets less general loops. The problem this transformation poses to CerCo's labelling approach are independent of the sophistication of the actual transformation.

Example 1 *In Figure 2 we show a program (a wilfully inefficient computation of of the sum of the first n factorials) and a possible transformation of it, combining loop peeling and loop unrolling.*

3 Labelling: a quick sketch of the previous approach

Plainly labelled Imp is obtained by adding to the code *cost labels* (with metavariables $\alpha, \beta, \ldots$), and cost-labelled statements:

$$S, T ::= \cdots \mid \alpha : S$$

Cost labels allow us to track some program points along the compilation chain. For further details we refer to [3].

[2] This is a reasonable aproximation: it defines a loop as multi-entry if it has an external but unreachable goto jumping into it.

With labels the small step semantics turns into a labelled transition system along with a natural notion of trace (*i.e.* lists of labels) arises. The evaluation of statements is enriched with traces, so that rules follow a pattern similar to the following:

$$\begin{gathered}(\alpha : S, K, s) \quad \xrightarrow{\alpha}_P (S, K, s) \\ (\mathsf{skip}, S \cdot K, s) \xrightarrow{\varepsilon}_P (S, K, s) \\ \text{etc.}\end{gathered}$$

Here, we identify cost labels α with singleton traces and we use ε for the empty trace. Cost labels are emitted by cost-labelled statements only[3]. We then write $\xrightarrow{\lambda}{}^*$ for the transitive closure of the small step semantics which produces by concatenation the trace λ.

Labelling Given an Imp program P its *labelling* $\alpha : \mathcal{L}(P)$ in $\ell - \mathsf{Imp}$ is defined by putting cost labels after every branching statement, at the start of both branches, and a cost label at the beginning of the program. Also, every labelled statement gets a cost label, which is a conservative approach to ensuring that all loops have labels inside them, as a loop might be done with gotos. The relevant cases are

$$\begin{aligned}\mathcal{L}(\mathsf{if}\ e\ \mathsf{then}\ S\ \mathsf{else}\ T) &= \mathsf{if}\ e\ \mathsf{then}\ \alpha : \mathcal{L}(S)\ \mathsf{else}\ \beta : \mathcal{L}(T) \\ \mathcal{L}(\mathsf{while}\ e\ \mathsf{do}\ S) &= (\mathsf{while}\ e\ \mathsf{do}\ \alpha : \mathcal{L}(S)); \beta : \mathsf{skip} \\ \mathcal{L}(\ell : S) &= (\ell : \alpha : \mathcal{L}(S))\end{aligned}$$

where α, β are fresh cost labels. In all other cases the definition just passes to substatements.

Labels in the rest of the compilation chain All languages further down the chain get a new sequential statement $\mathsf{emit}\ \alpha$ whose effect is to be consumed in a labelled transition while keeping the same state. All other instructions guard their operational semantics and do not emit cost labels.

Preservation of semantics throughout the compilation process is restated, in rough terms, as:

$$\text{starting state of } P \xrightarrow{\lambda}{}^* \text{ halting state} \iff \text{starting state of } \mathcal{C}(P) \xrightarrow{\lambda}{}^* \text{ halting state}$$

Here P is a program of a language along the compilation chain, starting and halting states depend on the language, and $\mathcal{C}$ is the compilation function[4].

Instrumentations Let $\mathcal{C}$ be the whole compilation from $\ell\mathsf{Imp}$ to the labelled version of some low-level language L. Supposing such compilation has not introduced any new loop or branching, we have that:

- Every loop contains at least a cost label (*soundness condition*)
- Every branching has different labels for the two branches (*preciseness condition*).

[3] In the general case the evaluation of expressions can emit cost labels too (see 5).

[4] The case of divergent computations needs to be addressed too. Also, the requirement can be weakened by demanding some sort weaker form of equivalence of the traces than equality. Both of these issues are beyond the scope of this presentation.

With these two conditions, we have that each and every cost label in $\mathcal{C}(P)$ for any P corresponds to a block of sequential instructions, to which we can assign a constant *cost*[5] We therefore may assume the existence of a *cost mapping* κ_P from cost labels to natural numbers, assigning to each cost label α the cost of the block containing the single occurrance of α.

Given any cost mapping κ, we can enrich a labelled program so that a particular fresh variable (the *cost variable* c) keeps track of the summation of costs during the execution. We call this procedure *instrumentation* of the program, and it is defined recursively by:

$$\mathcal{I}(\alpha : S) = c := c + \kappa(\alpha); \mathcal{I}(S)$$

In all other cases the definition passes to substatements.

The problem with loop optimisations Let us take loop peeling, and apply it to the labelling of a program without any prior adjustment:

$$(\mathsf{while}\ e\ \mathsf{do}\ \alpha : S); \beta : \mathsf{skip} \mapsto (\mathsf{if}\ b\ \mathsf{then}\ \alpha : S; \mathsf{while}\ b\ \mathsf{do}\ \alpha : S[\ell'_i/\ell_i]); \beta : \mathsf{skip}$$

What happens is that the cost label α is duplicated with two distinct occurrences. If these two occurrences correspond to different costs in the compiled code, the best the cost mapping can do is to take the maximum of the two, preserving soundness (*i.e.* the cost estimate still bounds the actual one) but losing preciseness (*i.e.* the actual cost could be strictly less than its estimate).

4 Indexed labels

This section presents the core of the new approach. In brief points it amounts to the following:

4.1. Enrich cost labels with formal indices corresponding, at the beginning of the process, to which iteration of the loop they belong to.

4.2. Each time a loop transformation is applied and a cost labels is split in different occurrences, each of these will be reindexed so that every time they are emitted their position in the original loop will be reconstructed.

4.3. Along the compilation chain, alongside the emit instruction we add other instructions updating the indices, so that iterations of the original loops can be rebuilt at the operational semantics level.

4.4. The machinery computing the cost mapping will still work, but assigning costs to indexed cost labels, rather than to cost labels as we wish. However, *dependent costs* can be calculated, where dependency is on which iteration of the containing loops we are in.

4.1 Indexing the cost labels

Formal indices and $\iota\ell$mp Let $i_0, i_1, \ldots$ be a sequence of distinguished fresh identifiers that will be used as loop indices. A *simple expression* is an affine arithmetical expression in one of these indices, that is $a * i_k + b$ with $a, b, k \in \mathbb{N}$. Simple expressions $e_1 = a_1 * i_k + b_1$,

[5] This in fact requires the machine architecture to be 'simple enough', or for some form of execution analysis to take place.

$e_2 = a2 * i_k + b_2$ in the same index can be composed, yielding $e_1 \circ e_2 := (a_1 a_2) * i_k + (a_1 b2 + b_1)$, and this operation has an identity element in $id_k := 1 * i_k + 0$. Constants can be expressed as simple expressions, so that we identify a natural c with $0 * i_k + c$.

An *indexing* (with metavariables I, J, ...) is a list of transformations of successive formal indices dictated by simple expressions, that is a mapping[6]

$$i_0 \mapsto a_0 * i_0 + b_0, \ldots, i_{k-1} \mapsto a_{k-1} * i_{k-1} + b_{k-1}$$

An *indexed cost label* (metavariables $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, ...) is the combination of a cost label α and an indexing I, written $\alpha\langle I \rangle$. The cost label underlying an indexed one is called its *atom*. All plain labels can be considered as indexed ones by taking an empty indexing.

Imp with indexed labels ($\iota\ell$Imp) is defined by adding to Imp statements with indexed labels, and by having loops with formal indices attached to them:

$$S, T, \ldots ::= \cdots i_k : \mathsf{while}\ e\ \mathsf{do}\ S \mid \boldsymbol{\alpha} : S$$

Note than unindexed loops still exist in the language: they will correspond to multi-entry loops which are ignored by indexing and optimisations. We will discuss the semantics later.

Indexed labelling Given an Imp program P, in order to index loops and assign indexed labels, we must first distinguish single-entry loops. We sketch how this can be computed in the sequel.

A first pass of the program P can easily compute two maps: loopof_P from each label ℓ to the occurrence (*i.e.* the path) of a while loop containing ℓ, or the empty path if none exists; and $\mathsf{gotosof}_P$ from a label ℓ to the occurrences of gotos pointing to it. Then the set $\mathsf{multientry}_P$ of multi-entry loops of P can be computed by

$$\mathsf{multientry}_P := \{\, p \mid \exists \ell, q.p = \mathsf{loopof}_P(\ell), q \in \mathsf{gotosof}_P(\ell), q \not\leq p \,\}$$

Here $\leq$ is the prefix relation[7].

Let Id_k be the indexing of length k made from identity simple expressions, *i.e.* the sequence $i_0 \mapsto id_0, \ldots, i_{k-1} \mapsto id_{k-1}$. We define the tiered indexed labelling $\mathcal{L}^\iota_P(S,k)$ in program P for occurrence S of a statement in P and a natural k by recursion, setting:

$$\mathcal{L}^\iota_P(S,k) := \begin{cases} (i_k : \mathsf{while}\ b\ \mathsf{do}\ \alpha\langle Id_{k+1}\rangle : \mathcal{L}^\iota_P(T,k+1)); \beta\langle Id_k\rangle : \mathsf{skip} & \\ \qquad \text{if } S = \mathsf{while}\ b\ \mathsf{do}\ T \text{ and } S \notin \mathsf{multientry}_P, & \\ (\mathsf{while}\ b\ \mathsf{do}\ \alpha\langle Id_k\rangle : \mathcal{L}^\iota_P(T,k)); \beta\langle Id_k\rangle : \mathsf{skip} & \\ \qquad \text{otherwise, if } S = \mathsf{while}\ b\ \mathsf{do}\ T, & \\ \mathsf{if}\ b\ \mathsf{then}\ \alpha\langle Id_k\rangle : \mathcal{L}^\iota_P(T_1,k)\ \mathsf{else}\ \beta\langle Id_k\rangle : \mathcal{L}^\iota_P(T_2,k) & \\ \qquad \text{if } S = \mathsf{if}\ b\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2, & \\ \ell : \alpha\langle Id_k\rangle : \mathcal{L}^\iota_P(T,k) \qquad \text{if } S = \ell : T, & \\ \ldots & \end{cases}$$

[6]Here we restrict each mapping to be a simple expression *on the same index*. This might not be the case if more loop optimisations are accounted for (for example, interchanging two nested loops).

[7]Possible simplifications to this procedure include keeping track of just the while loops containing labels and gotos (rather than paths in the syntactic tree of the program), and making two passes while avoiding building the map to sets gotosof

Here, as usual, α and β are fresh cost labels, and other cases just keep making the recursive calls on the substatements. The *indexed labelling* of a program P is then defined as $\alpha\langle\rangle : \mathcal{L}^{\iota}_{P}(P, 0)$, *i.e.* a further fresh unindexed cost label is added at the start, and we start from level 0.

In plainer words: each single-entry loop is indexed by i_k where k is the number of other single-entry loops containing this one, and all cost labels under the scope of a single-entry loop indexed by i_k are indexed by all indices $i_0, \ldots, i_k$, without any transformation.

4.2 Indexed labels and loop transformations

We define the *reindexing* $I \circ (i_k \mapsto a * i_k + b)$ as an operator on indexings by setting:

$$(i_0 \mapsto e_0, \ldots, i_k \mapsto e_k, \ldots, i_n \mapsto e_n) \circ (i_k \mapsto a * i_k + b) := \\ i_0 \mapsto e_0, \ldots, i_k \mapsto e_k \circ (a * i_k + b), \ldots, i_n \mapsto e_n,$$

We further extend to indexed labels (by $\alpha\langle I\rangle \circ (i_k \mapsto e) := \alpha\langle I \circ (i_k \mapsto e)\rangle$) and also to statements in $\iota\ell$Imp (by applying the above transformation to all indexed labels).

We can then redefine loop peeling and loop unrolling, taking into account indexed labels. It will only be possible to apply the transformation to indexed loops, that is loops that are single-entry. The attentive reader will notice that no assumptions are made on the labelling of the statements that are involved. In particular the transformation can be repeated and composed at will. Also, note that after erasing all labelling information (*i.e.* indexed cost labels and loop indices) we recover exactly the same transformations presented in 2.

Indexed loop peeling

$$i_k : \mathsf{while}\ b\ \mathsf{do}\ S \mapsto \mathsf{if}\ b\ \mathsf{then}\ S \circ (i_k \mapsto 0); i_k : \mathsf{while}\ b\ \mathsf{do}\ S[\ell'_i/\ell_i] \circ (i_k \mapsto i_k + 1)$$

As can be expected, the peeled iteration of the loop gets reindexed, always being the first iteration of the loop, while the iterations of the remaining loop are shifted by 1. Notice that this transformation can lower the actual depth of some loops, however their index is left untouched.

Indexed loop unrolling

$$\begin{array}{l} i_k : \mathsf{while}\ b\ \mathsf{do}\ S \\ \qquad \downarrow \\ i_k : \mathsf{while}\ b\ \mathsf{do} \\ \quad (S \circ (i_k \mapsto n * i_k); \\ \quad \mathsf{if}\ b\ \mathsf{then} \\ \qquad (S[\ell^1_i/\ell_i] \circ (i_k \mapsto n * i_k + 1); \\ \qquad\ \vdots \\ \qquad \mathsf{if}\ b\ \mathsf{then} \\ \qquad\quad S[\ell^n_i/\ell_i] \circ (i_k \mapsto n * i_k + n - 1)) \cdots) \end{array}$$

Again, the reindexing is as expected: each copy of the unrolled body has its indices remapped so that when they are executed, the original iteration of the loop to which they correspond can be recovered.

4.3 Semantics and compilation of indexed labels

In order to make sense of loop indices, one must keep track of their values in the state. A *constant indexing* (metavariables $C, \ldots$) is an indexing which employs only constant simple expressions. The evaluation of an indexing I in a constant indexing C, noted $I|_C$, is defined by:

$$I \circ (i_0 \mapsto c_0, \ldots, i_{k-1} \mapsto c_{k-1}) := \boldsymbol{\alpha} \circ (i_0 \mapsto c_0) \circ \cdots \circ (i_{k-1} \mapsto c_{k-1})$$

Here, we are using the definition of $- \circ -$ given in 4.1. We consider the above defined only if the the resulting indexing is a constant one too[8]. The definition is extended to indexed labels by $\alpha\langle I\rangle|_C := \alpha\langle I|_C\rangle$.

Constant indexings will be used to keep track of the exact iterations of the original code that the emitted labels belong to. We thus define two basic actions to update constant indexings: $C[i_k\uparrow]$ increments the value of i_k by one, and $C[i_k\downarrow 0]$ resets it to 0.

We are ready to update the definition of the operational semantics of indexed labelled Imp. The emitted cost labels will now be ones indexed by constant indexings. We add a special indexed loop construct for continuations that keeps track of active indexed loop indices:

$$K, \ldots ::= \cdots \mid i_k : \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{then}\ K$$

The difference between the regular stack concatenation $i_k : \mathsf{while}\ b\ \mathsf{do}\ S \cdot K$ and the new constructor is that the latter indicates the loop is the active one in which we already are, while the former is a loop that still needs to be started[9]. The find function is updated accordingly with the case

$$\mathsf{find}(\ell, i_k : \mathsf{while}\ b\ \mathsf{do}\ S, K) := \mathsf{find}(\ell, S, i_k : \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{then}\ K)$$

The state will now be a 4-tuple (S, K, s, C) which adds a constant indexing to the triple of the regular semantics. The small-step rules for all statements remain the same, without touching the C parameter (in particular unindexed loops behave the same as usual), apart from the ones regarding cost-labels and indexed loops. The remaining cases are:

$$(\boldsymbol{\alpha} : S, K, s, C) \overset{\boldsymbol{\alpha}|_C}{\to}_P (S, K, s, C)$$

$$(i_k : \mathsf{while}\ b\ \mathsf{do}\ S, K, C) \overset{\varepsilon}{\to}_P \begin{cases} (S, i_k : \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{then}\ K, s, C[i_k\downarrow 0]) & \\ & \text{if } (b, s) \Downarrow v \neq 0, \\ (\mathsf{skip},\ \mathrm{K},\ \mathrm{s},\ \mathrm{C}) & \text{otherwise} \end{cases}$$

$$(\mathsf{skip}, i_k : \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{then}\ K, C) \overset{\varepsilon}{\to}_P \begin{cases} (S, i_k : \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{then}\ K, s, C[i_k\uparrow]) & \\ & \text{if } (b, s) \Downarrow v \neq 0, \\ (\mathsf{skip},\ \mathrm{K},\ \mathrm{s},\ \mathrm{C}) & \text{otherwise} \end{cases}$$

Some explanations are in order:

- Emitting a label always instantiates it with the current indexing.
- Hitting an indexed loop the first time initializes the corresponding index to 0; continuing the same loop increments the index as expected.

[8] For example $(i_0 \mapsto 2 * i_0, i_1 \mapsto i_1 + 1)|_{i_0 \mapsto 2}$ is undefined, but $(i_0 \mapsto 2 * i_0, i_1 \mapsto 0)|_{i_0 \mapsto 2} = i_0 \mapsto 4, i_1 \mapsto 2$, is indeed a constant indexing, even if the domain of the original indexing is not covered by the constant one.

[9] In the presence of continue and break statements active loops need to be kept track of in any case.

- The find function ignores the current indexing: this is correct under the assumption that all indexed loops are single entry, so that when we land inside an indexed loop with a goto, we are sure that its current index is right.
- The starting state with store s for a program P is $(P, \mathsf{halt}, s, (i_0 \mapsto 0, \ldots, i_{n-1} \mapsto 0)$ where $i_0, \ldots, i_{n-1}$ cover all loop indices of P[10].

Compilation Further down the compilation chain the loop structure is usually partially or completely lost. We cannot rely on it anymore to keep track of the original source code iterations. We therefore add, alongside the emit instruction, two other sequential instructions $\mathsf{ind_reset}\ k$ and $\mathsf{ind_inc}\ k$ whose sole effect is to reset to 0 (resp. increment by 1) the loop index i_k, as kept track of in a constant indexing accompanying the state.

The first step of compilation from $\iota\ell$Imp consists of prefixing the translation of an indexed loop $i_k : \mathsf{while}\ b\ \mathsf{do}\ S$ with $\mathsf{ind_reset}\ k$ and postfixing the translation of its body S with $\mathsf{ind_inc}\ k$. Later in the compilation chain we must propagate the instructions dealing with cost labels.

We would like to stress the fact that this machinery is only needed to give a suitable semantics of observables on which preservation proofs can be done. By no means are the added instructions and the constant indexing in the state meant to change the actual (let us say denotational) semantics of the programs. In this regard the two new instruction have a similar role as the emit one. A forgetful mapping of everything (syntax, states, operational semantics rules) can be defined erasing all occurrences of cost labels and loop indices, and the result will always be a regular version of the language considered.

Stating the preservation of semantics In fact, the statement of preservation of semantics does not change at all, if not for considering traces of evaluated indexed cost labels rather than traces of plain ones.

4.4 Dependent costs in the source code

The task of producing dependent costs from constant costs induced by indexed labels is quite technical. Before presenting it here, we would like to point out that the annotations produced by the procedure described in this Subsection, even if correct, can be enormous and unreadable. In Section 5, where we detail the actual implementation, we will also sketch how we mitigated this problem.

Having the result of compiling the indexed labelling $\mathcal{L}^\iota(P)$ of an Imp program P, we may still suppose that a cost mapping can be computed, but from indexed labels to naturals. We want to annotate the source code, so we need a way to express and compute the costs of cost labels, *i.e.* group the costs of indexed labels to ones of their atoms. In order to do so we introduce *dependent costs*. Let us suppose that for the sole purpose of annotation, we have available in the language ternary expressions of the form

$$e\ ?\ f_1 : f_2,$$

and that we have access to common operators on integers such as equality, order and modulus.

[10] For a program which is the indexed labelling of an Imp one this corresponds to the maximum nesting of single-entry loops. We can also avoid computing this value in advance if we define $C[i{\downarrow}0]$ to extend C's domain as needed, so that the starting constant indexing can be the empty one.

Simple conditions First, we need to shift from *transformations* of loop indices to *conditions* on them. We identify a set of conditions on natural numbers which are able to express the image of any composition of simple expressions. *Simple conditions* are of three possible forms:

- Equality $i_k = n$ for some natural n.
- Inequality $i_k \geq n$ for some natural n.
- Modular equality together with inequality $i_k \bmod a = b \wedge i_k \geq n$ for naturals a, b, n.

The 'always true' simple condition is given by $i_k \geq 0$. We write $i_k \bmod a = b$ as a simple condition for $i_k \bmod a = b \wedge i_k \geq 0$.

Given a simple condition p and a constant indexing C we can easily define when p holds for C (written $p \circ C$). A *dependent cost expression* is an expression built solely out of integer constants and ternary expressions with simple conditions at their head. Given a dependent cost expression e where all of the loop indices appearing in it are in the domain of a constant indexing C, we can define the value $e \circ C \in \mathbb{N}$ by:

$$n \circ C := n, \qquad (p \ ? \ e : f) \circ C := \begin{cases} e \circ C & \text{if } p \circ C, \\ f \circ C & \text{otherwise.} \end{cases}$$

From indexed costs to dependent ones Every simple expression e corresponds to a simple condition $p(e)$ which expresses the set of values that e can take. Following is the definition of such a relation. We recall that in this development, loop indices are always mapped to simple expressions over the same index. If it was not the case, the condition obtained from an expression should be on the mapped index, not the indeterminate of the simple expression. We leave all generalisations of what we present here for further work:

$$p(a * i_k + b) := \begin{cases} i_k = b & \text{if } a = 0, \\ i_k \geq b & \text{if } a = 1, \\ i_k \bmod a = b' \wedge i_k \geq b & \text{otherwise, where } b' = b \bmod a. \end{cases}$$

Now, suppose we are given a mapping κ from indexed labels to natural numbers. We will transform it in a mapping (identified, via abuse of notation, with the same symbol κ) from atoms to Imp expressions built with ternary expressions which depend solely on loop indices. To that end we define an auxiliary function κ^{α}_{L}, parameterized by atoms and words of simple expressions, and defined on *sets* of n-uples of simple expressions (with n constant across each such set, *i.e.* each set is made of words all with the same length).

We will employ a bijection between words of simple expressions and indexings, given by:[11]

$$i_0 \mapsto e_0, \ldots, i_{k-1} \mapsto e_{k-1} \cong e_0 \cdots e_{k-1}.$$

As usual, ε denotes the empty word/indexing, and juxtaposition is used to denote word concatenation.

For every set s of n-uples of simple expressions, we are in one of the following three exclusive cases:

[11] Lists of simple expressions are in fact how indexings are -represented in CerCo's current implementation of the compiler.

- $S = \emptyset$.
- $S = \{\varepsilon\}$.
- There is a simple expression e such that S can be decomposed in $eS' + S''$, with $S' \neq \emptyset$ and none of the words in S'' starting with e.

Here eS' denotes prepending e to all elements of S' and $+$ is disjoint union. This classification can serve as the basis of a definition by recursion on $n + \sharp S$ where n is the size of tuples in S and $\sharp S$ is its cardinality. Indeed in the third case in S' the size of tuples decreases strictly (and cardinality does not increase) while for S'' the size of tuples remains the same but cardinality strictly decreases. The expression e of the third case will be chosen as minimal for some total order[12].

Following is the definition of the auxiliary function κ^α_L, which follows the recursion scheme presented above:

$$\kappa^\alpha_L(\emptyset) := 0$$
$$\kappa^\alpha_L(\{\varepsilon\}) := \kappa(\alpha\langle L\rangle)$$
$$\kappa^\alpha_L(eS' + S'') := p(e)\ ?\ \kappa^\alpha_{Le}(S') : \kappa^\alpha_L(S'')$$

Finally, the wanted dependent cost mapping is defined by

$$\kappa(\alpha) := \kappa^\alpha_\varepsilon(\{\, L \mid \alpha\langle L\rangle \text{ appears in the compiled code}\,\})$$

Indexed instrumentation The *indexed instrumentation* generalises the instrumentation presented in 3. We described above how cost atoms can be mapped to dependent costs. The instrumentation must also insert code dealing with the loop indices. As instrumentation is done on the code produced by the labelling phase, all cost labels are indexed by identity indexings. The relevant cases of the recursive definition (supposing c is the cost variable) are then:

$$\mathcal{I}^\iota(\alpha\langle Id_k\rangle : S) = c := c + \kappa(\alpha); \mathcal{I}^\iota(S)$$
$$\mathcal{I}^\iota(i_k : \mathsf{while}\ b\ \mathsf{do}\ S) = i_k := 0; \mathsf{while}\ b\ \mathsf{do}\ (\mathcal{I}^\iota(S); i_k := i_k + 1)$$

4.5 A detailed example

Take the program in Figure 2. Its initial labelling will be:

$\alpha\langle\rangle : s := 0;$
$i := 0;$
$i_0 : \mathsf{while}\ i < n\ \mathsf{do}$
  $\beta\langle i_0\rangle : p := 1;$
  $j := 1;$
  $i_1 : \mathsf{while}\ j \leq i\ \mathsf{do}$
    $\gamma\langle i_0, i_1\rangle : p := j * p$
    $j := j + 1;$
  $\delta\langle i_0\rangle : s := s + p;$
  $i := i + 1;$
$\epsilon\langle\rangle : \mathsf{skip}$

[12]The specific order used does not change the correctness of the procedure, but different orders can give more or less readable results. A "good" order is the lexicographic one, with $a * i_k + b \leq a' * i_k + b'$ if $a < a'$ or $a = a'$ and $b \leq b'$.

(a single skip after the δ label has been suppressed, and we are using the identification between indexings and tuples of simple expressions explained in subsection 4.4). Supposing for example, $n = 3$ the trace of the program will be

$$\alpha\langle\rangle\,\beta\langle 0\rangle\,\delta\langle 0\rangle\,\beta\langle 1\rangle\,\gamma\langle 1,0\rangle\,\delta\langle 1\rangle\,\beta\langle 2\rangle\,\gamma\langle 2,0\rangle\,\gamma\langle 2,1\rangle\,\delta\langle 2\rangle\,\epsilon\langle\rangle$$

Now let as apply the transformations of Figure 2 with the additional information detailed in subsection 4.2. The result is shown in Figure 3. One can check that the transformed code leaves the same trace when executed.

Now let us compute the dependent cost of γ, supposing no other loop transformations are done. Ordering its indexings we have the following list:

$$\begin{array}{l} 0, i_1 \\ 2 * i_0 + 1, 0 \\ 2 * i_0 + 1, 1 \\ 2 * i_0 + 1, 2 * i_1 + 2 \\ 2 * i_0 + 1, 2 * i_1 + 3 \\ 2 * i_0 + 2, 2 * i_1 \\ 2 * i_0 + 2, 2 * i_1 + 1 \end{array} \tag{1}$$

The resulting dependent cost will then be

$$\begin{array}{l} \kappa^\iota(\gamma) = (i_0 = 0)\ ? \\ \quad (i_1 \geq 0)\ ?\ a : 0 : \\ \quad (i_0 \bmod 2 = 1 \wedge i_0 \geq 1)\ ? \\ \quad\quad (i_1 = 0)\ ? \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \\ \quad\quad\quad b : \\ \quad\quad\quad (i_1 = 1)\ ? \\ \quad\quad\quad\quad c : \\ \quad\quad\quad\quad (i_1 \bmod 2 = 0 \wedge i_1 \geq 2)\ ? \\ \quad\quad\quad\quad\quad d : \\ \quad\quad\quad\quad\quad (i_1 \bmod 2 = 1 \wedge i_1 \geq 3)\ ?\ e : 0 \\ \quad\quad (i_0 \bmod 2 = 0 \wedge i_0 \geq 2)\ ? \\ \quad\quad\quad (i_1 \bmod 2 = 0 \wedge i_1 \geq 0)\ ? \qquad\qquad : \\ \quad\quad\quad\quad f : \\ \quad\quad\quad\quad (i_1 \bmod 2 = 1 \wedge i_1 \geq 1)\ ?\ g : 0 \\ \quad\quad\quad 0 \end{array} \tag{2}$$

We will see later on page 15 how such an expression can be simplified.

5 Notes on the implementation and further work

Implementing the indexed label approach in CerCo's untrusted Ocaml prototype does not introduce many new challenges beyond what has already been presented for the toy language, Imp with gotos. Clight, the C fragment source language of CerCo's compilation chain [3], has several more fetaures, but few demand changes in the indexed labelled approach.

Indexed loops *vs.* index update instructions In our presentation we have indexed loops in $\iota\ell$Imp, while we hinted that later languages in the compilation chain would have specific index update instructions. In CerCo's actual compilation chain from Clight to 8051 assembly, indexed loops are only in Clight, while from Cminor onward all languages have the same three cost-involving instructions: label emitting, index resetting and index incrementing.

Loop transformations in the front end We decided to implement the two loop transformations in the front end, namely in Clight. This decision is due to user readability concerns: if costs are to be presented to the programmer, they should depend on structures written by the programmer himself. If loop transformation were performed later it would be harder to create a correspondence between loops in the control flow graph and actual loops written in the source code. However, another solution would be to index loops in the source code and then use these indices later in the compilation chain to pinpoint explicit loops of the source code: loop indices can be used to preserve such information, just like cost labels.

Break and continue statements Clight's loop flow control statements for breaking and continuing a loop are equivalent to appropriate goto statements. The only difference is that we are assured that they cannot cause loops to be multi-entry, and that when a transformation such as loop peeling is complete, they need to be replaced by actual gotos (which happens further down the compilation chain anyway).

Function calls Every internal function definition has its own space of loop indices. Executable semantics must thus take into account saving and resetting the constant indexing of current loops upon hitting a function call, and restoring it upon return of control. A peculiarity is that this cannot be attached to actions that save and restore frames: namely in the case of tail calls the constant indexing needs to be saved whereas the frame does not.

Cost-labelled expressions In labelled Clight, expressions also get cost labels, due to the presence of ternary conditional expressions (and lazy logical operators, which get translated to ternary expressions too). Adapting the indexed labelled approach to cost-labelled expressions does not pose any particular problems.

Simplification of dependent costs As previously mentioned, the naïve application of the procedure described in 4.4 produces unwieldy cost annotations. In our implementation several transformations are used to simplify such complex dependent costs.

Disjunctions of simple conditions are closed under all logical operations, and it can be computed whether such a disjunction implies a simple condition or its negation. This can be used to eliminate useless branches of dependent costs, to merge branches that share the same value, and possibly to simplify the third case of simple condition. Examples of the three transformations are respectively:

- `(_i_0 == 0)?`x`:(_i_0 >= 1)?`y`:`z $\mapsto$ `(_i_0 == 0)?`x`:`y,
- c`?`x`:(`d`?`x`:`y`)` $\mapsto$ `(`c `||` d`)?`x`:`y,
- `(_i_0 == 0)?`x`:(_i_0 % 2 == 0 && _i_0 >= 2)?`y`:`z $\mapsto$ `(_i_0 == 0)?`x`:(_i_0 % 2 == 0)?`y`:`z.

The second transformation tends to accumulate disjunctions, to the detriment of readability. A further transformation swaps two branches of the ternary expression if the negation of the condition can be expressed with fewer clauses. For example:

$$\texttt{(\_i\_0 \% 3 == 0 || \_i\_0 \% 3 == 1)?}x\texttt{:}y \mapsto \texttt{(\_i\_0 \% 3 == 2)?}y\texttt{:}x.$$

Picking up again the example depicted in subsection 4.5, we can see that the cost in (2) can be simplified to the following, using some of the transformation described above:

$$\begin{array}{l}
\kappa^{\iota}(\gamma) = (i_0 = 0)\ ? \\
\qquad a : \\
\qquad (i_0 \bmod 2 = 1)\ ? \\
\qquad\quad (i_1 = 0)\ ? \qquad\qquad\qquad\qquad : \\
\qquad\qquad b : \\
\qquad\qquad (i_1 = 1)\ ? \\
\qquad\qquad\quad c : \\
\qquad\qquad\quad (i_1 \bmod 2 = 0)\ ? \\
\qquad\qquad\qquad d : \\
\qquad\qquad\qquad e \\
\qquad\quad (i_1 \bmod 2 = 0)\ ? \\
\qquad\qquad f : \\
\qquad\qquad g
\end{array}$$

One should keep in mind that the example was wilfully complicated, in practice the cost expressions produced have rarely more clauses than the number of nested loops containing the annotation.

Updates to the frama-C cost plugin Cerco's frama-C [2] cost plugin has been updated to take into account our new notion of dependent costs. The frama-c framework expands ternary expressions to branch statements, introducing temporaries along the way. This makes the task of analyzing ternary cost expressions rather daunting. It was deemed necessary to provide an option in the compiler to use actual branch statements for cost annotations rather than ternary expressions, so that at least frama-C's use of temporaries in cost annotation could be avoided. The cost analysis carried out by the plugin now takes into account such dependent costs.

The only limitation (which actually simplifies the code) is that, within a dependent cost, simple conditions with modulus on the same loop index should not be modulo different numbers. This corresponds to a reasonable limitation on the number of times loop unrolling may be applied to the same loop: at most once.

Further work For the time being, indexed labels are only implemented in the untrusted Ocaml compiler, while they are not present yet in the Matita code. Porting them should pose no significant problem. Once ported, the task of proving properties about them in Matita can begin.

Because most of the executable operational semantics of the languages across the frontend and the backend are oblivious to cost labels, it should be expected that the bulk of the semantic preservation proofs that still needs to be done will not get any harder because of

indexed labels. The only trickier point that we foresee would be in the translation of Clight to Cminor, where we pass from structured indexed loops to atomic instructions on loop indices.

An invariant which should probably be proved and provably preserved along the compilation chain is the non-overlap of indexings for the same atom. Then, supposing cost correctness for the unindexed approach, the indexed one will just need to amend the proof that

$$\forall C \text{ constant indexing}.\forall \alpha\langle I\rangle \text{ appearing in the compiled code}.\kappa(\alpha)\circ(I\circ C)=\kappa(\alpha\langle I\rangle).$$

Here, C represents a snapshot of loop indices in the compiled code, while $I \circ C$ is the corresponding snapshot in the source code. Semantics preservation will ensure that when, with snapshot C, we emit $\alpha\langle I\rangle$ (that is, we have $\alpha\langle I\circ C\rangle$ in the trace), α must also be emitted in the source code with indexing $I \circ C$, so the cost $\kappa(\alpha)\circ(I\circ C)$ applies.

Aside from carrying over the proofs, we would like to extend the approach to more loop transformations. Important examples are loop inversion (where a for loop is reversed, usually to make iterations appear to be truly independent) or loop interchange (where two nested loops are swapped, usually to have more loop invariants or to enhance strength reduction). This introduces interesting changes to the approach, where we would have indexings such as:

$$i_0 \mapsto n - i_0 \quad \text{or} \quad i_0 \mapsto i_1, i_1 \mapsto i_0.$$

In particular dependency over actual variables of the code would enter the frame, as indexings would depend on the number of iterations of a well-behaving guarded loop (the n in the first example).

Finally, as stated in the introduction, the approach should allow some integration of techniques for cache analysis, a possibility that for now has been put aside as the standard 8051 target architecture for the CerCo project lacks a cache. Two possible developments for this line of work present themselves:

1. One could extend the development to some 8051 variants, of which some have been produced with a cache.

2. One could make the compiler implement its own cache: this cannot apply to RAM accesses of the standard 8051 architecture, as the difference in cost of accessing the two types of RAM is only one clock cycle, which makes any implementation of cache counterproductive. So for this proposal, we could either artificially change the accessing cost of RAM of the model just for the sake of possible future adaptations to other architectures, or otherwise model access to an external memory by means of the serial port.

References

[1] Absint angewandte informatik. `http://www.absint.com/`.

[2] Frama-c software analyzers. `http://frama-c.com/`.

[3] R. M. Amadio, N. Ayache, Y. Régis-Gianas, and R. Saillard. Certifying cost annotations in compilers. Deliverable 2.1 of Project FP7-ICT-2009-C-243881 CerCo, Available at `http://hal.archives-ouvertes.fr/hal-00524715`.

[4] R. M. Amadio, N. Ayache, Y. Régis-Gianas, and R. Saillard. Prototype implementation. Deliverable 2.2 of Project FP7-ICT-2009-C-243881 CerCo, Available at `http://cerco.cs.unibo.it/`.

[5] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17:131–181, December 1999.

[6] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22:96–103, February 1979.

[7] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

Syntax

$\ell, \ldots$ (labels) $x, y, \ldots$ (identifiers) $e, f, \ldots$ (expression)

$P, S, T, \ldots ::= \mathsf{skip} \mid s; t \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ t \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid x := e \mid \ell : s \mid \mathsf{goto}\ \ell$ (statements)

Semantics

$K, \ldots ::= \mathsf{halt} \mid S \cdot K$ (continuations)

$$\mathsf{find}(\ell, S, K) := \begin{cases} \bot & \text{if } S = \mathsf{skip}, \mathsf{goto}\ \ell' \text{ or } x := e, \\ (T, K) & \text{if } S = \ell : T, \\ \mathsf{find}(\ell, T, K) & \text{otherwise, if } S = \ell' : T, \\ \mathsf{find}(\ell, T_1, T_2 \cdot K) & \text{if defined and } S = T_1; T_2, \\ \mathsf{find}(\ell, T_1, K) & \text{if defined and } S = \mathsf{if}\ b\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2, \\ \mathsf{find}(\ell, T_2, K) & \text{otherwise, if } S = T_1; T_2 \text{ or } \mathsf{if}\ b\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2, \\ \mathsf{find}(\ell, T, S \cdot K) & \text{if } S = \mathsf{while}\ b\ \mathsf{do}\ T. \end{cases}$$

$$(x := e, K, s) \to_P (\mathsf{skip}, K, s[v/x]) \quad \text{if } (e, s) \Downarrow v$$

$$(S; T, K, s) \to_P (S, T \cdot K, s)$$

$$(\mathsf{if}\ b\ \mathsf{then}\ S\ \mathsf{else}\ T, K, s) \to_P \begin{cases} (S, K, s) & \text{if } (b, s) \Downarrow v \neq 0 \\ (T, K, s) & \text{if } (b, s) \Downarrow 0 \end{cases}$$

$$(\mathsf{while}\ b\ \mathsf{do}\ S, K, s) \to_P \begin{cases} (S, \mathsf{while}\ b\ \mathsf{do}\ S \cdot K, s) & \text{if } (b, s) \Downarrow v \neq 0 \\ (\mathsf{skip}, K, s) & \text{if } (b, s) \Downarrow 0 \end{cases}$$

$$(\mathsf{skip}, S \cdot K, s) \to_P (S, K, s)$$

$$(\ell : S, K, s) \to_P (S, K, s)$$

$$(\mathsf{goto}\ \ell, K, s) \to_P (\mathsf{find}(\ell, P, \mathsf{halt}), s)$$

Figure 1: The syntax and operational semantics of Imp.

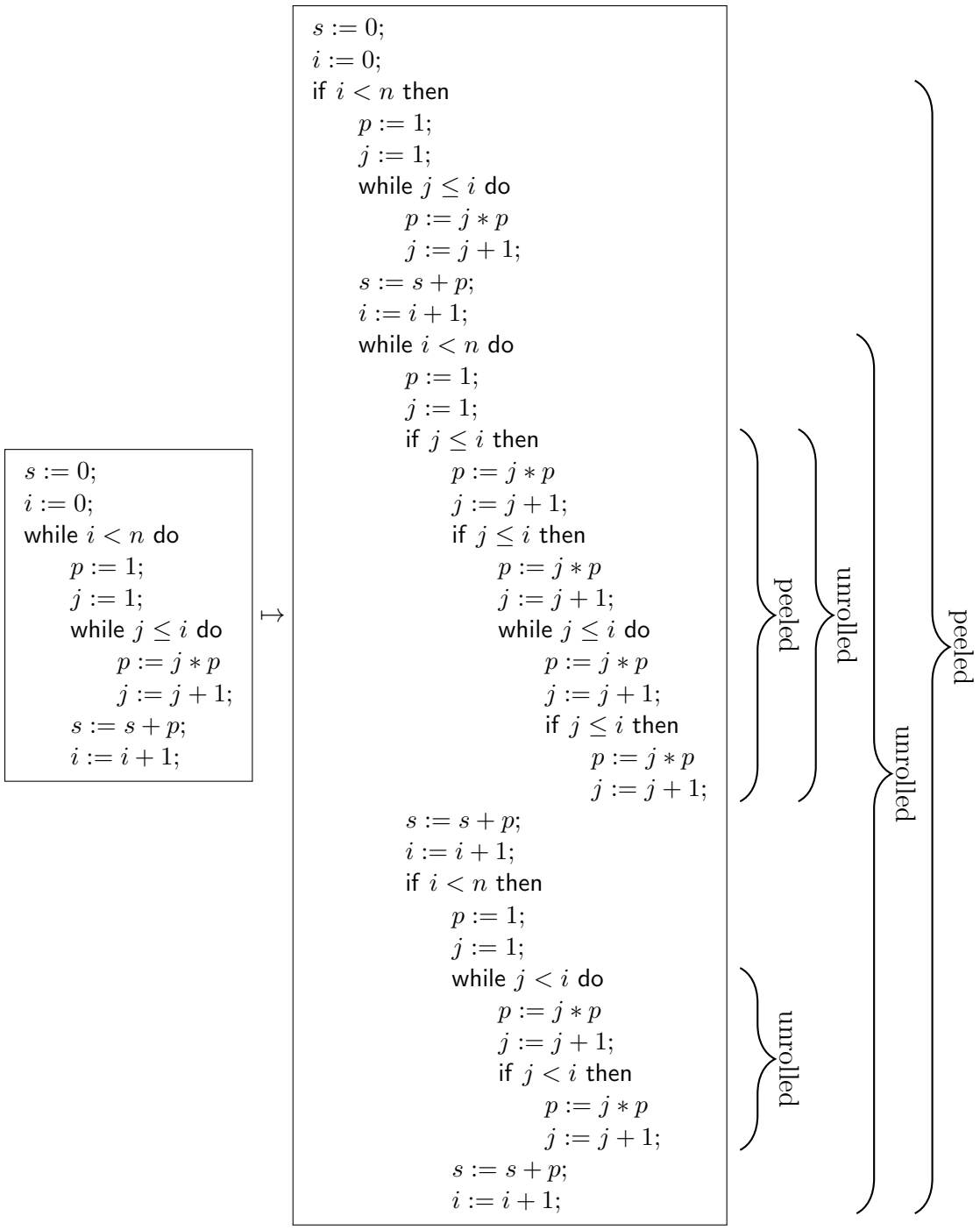

Figure 2: An example of loop transformations done on an Imp program. Parentheses are omitted in favour of blocks by indentation.

```
α⟨⟩ : s := 0;
i := 0;
if i < n then
    β⟨0⟩ : p := 1;
    j := 1;
    i1 : while j ≤ i do
        γ⟨0, i1⟩ : p := j ∗ p
        j := j + 1;
    δ⟨0⟩ : s := s + p;
    i := i + 1;
    i0 : while i < n do
        β⟨2 ∗ i0 + 1⟩ : p := 1;
        j := 1;
        if j ≤ i then
            γ⟨2 ∗ i0 + 1, 0⟩ : p := j ∗ p
            j := j + 1;
            if j ≤ i then
                γ⟨2 ∗ i0 + 1, 1⟩ : p := j ∗ p
                j := j + 1;
                i1 : while j ≤ i do
                    γ⟨2 ∗ i0 + 1, 2 ∗ i1 + 2⟩ : p := j ∗ p
                    j := j + 1;
                    if j ≤ i then
                        γ⟨2 ∗ i0 + 1, 2 ∗ i1 + 3⟩ : p := j ∗ p
                        j := j + 1;
        δ⟨2 ∗ i0 + 1⟩ : s := s + p;
        i := i + 1;
        if i < n then
            β⟨2 ∗ i0 + 2⟩ : p := 1;
            j := 1;
            i1 : while j < i do
                γ⟨2 ∗ i0 + 2, 2 ∗ i1⟩ : p := j ∗ p
                j := j + 1;
                if j < i then
                    γ⟨2 ∗ i0 + 2, 2 ∗ i1 + 1⟩ : p := j ∗ p
                    j := j + 1;
            δ⟨2 ∗ i0 + 2⟩ : s := s + p;
            i := i + 1;
ϵ⟨⟩ : skip
```

Figure 3: The result of applying reindexing loop transformations on the program in Figure 2.