The labelling approach to precise resource analysis on the source code, revisited*

Mauro Piccolo[1], Claudio Sacerdoti Coen[2], and Paolo Tranquilli[2]

Department of Computer Science, University of Turin[1],
Department of Computer Science and Engineering, University of Bologna[2]

piccolo@di.unibo.it, Claudio.SacerdotiCoen@unibo.it, Paolo.Tranquilli@unibo.it

Abstract. The labelling approach is a technique to lift cost models for non-functional properties of programs from the object code to the source. It allows to perform precise resource analysis of programs directly on the source code, reconciling functional and non functional analysis.
The labelling approach is based on the preservation of the structure of the high level program in every intermediate language used by the compiler. In order to prove the cost model correct, the semantics of programs is described with a labelled transition system that makes the program structure observable.
The original version of the labelling approach does not simply scale to function calls and it may not work properly with source instructions that have several predecessors. In the talk we will present an improved version that take cares of both limitations and that is more modular.
Most of the results presented have being mechanised in the interactive theorem prover Matita.

1 Introduction

The *labelling approach* has been introduced in [3] as a technique to *lift* cost models for non-functional properties of programs from the object code to the source code. Examples of non-functional properties are execution time, amount of stack/heap space consumed and energy required for communication. The basic premise of the approach is that it is impossible to provide a *uniform* cost model for an high level language that is preserved *precisely* by a compiler. For instance, two instances of an assignment $x = y$ in the source code can be compiled very differently according to the place (registers vs stack) where x and y are stored at the moment of execution. Therefore a precise cost model must assign a different cost to every occurrence, and the exact cost can only be known after compilation.

According to the labelling approach, the compiler is free to compile and optimise the source code without any major restriction, but it must keep trace of what happens to basic blocks during the compilation. The cost model is then computed on the object code. It assigns a cost to every basic block. Finally, the compiler propagates back the cost model to the source level, assigning a cost to each basic block of the source code.

Implementing the labelling approach in a certified compiler allows to reason formally on the high level source code of a program to prove non-functional properties that are granted to be preserved by the compiler itself. The trusted code base is then reduced to 1) the interactive theorem prover (or its kernel) used in the certification of the compiler and 2) the software used to certify the property on the source language, that can be itself certified further reducing the trusted code base. In [3] the authors provide an example of a simple certified compiler that implements the labelling approach for the imperative `While` language [7], that does not have pointers and function calls.

The labelling approach has been shown to scale to more interesting scenarios. In particular in [2] it has been applied to a functional language and in [8] it has been shown that the approach can be slightly complicated to handle loop optimisations and, more generally, program optimisations that do not preserve the structure of basic blocks. On-going work also shows that the labelling approach is also compatible with the complex analyses required to obtain a cost model for object code on processors that implement advanced features like pipelining, superscalar architectures and caches.

In the European Project CerCo (Certified Complexity, `http://cerco.cs.unibo.it`) [1] we are certifying a labelling approach based compiler for a large subset of C to 8051 object code. The compiler is moderately optimising and implements a compilation chain that is largely inspired to that of CompCert [5,6]. Compared

* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881

to work done in [3], the main novelties and source of difficulties are 1) the presence of function calls; 2) their interaction with stateful hardware components; 3) the realisation that instructions with multiple predecessors and instructions that need to immediately precede the beginning of a basic block could not be compiled preserving all the invariants required by the labelling approach.

The aforementioned difficulties forced us to rethink the labelling approach. Therefore we proposed in [4] a much more complex solution that accommodates function calls, but does not solve the other problems. In this talk we will describe a third version of the labelling approach which greatly simplifies the second version and solves at once all the issues known at this time. Most of the results presented have also been formalised in the Matita interactive theorem prover, and we expect to complete the formalisation in the near future.

The plan of the abstract is the following. In Section 2 we summarise the original version of the labelling method. In Section 3 we only hint at the issues involved with the original methods and we suggest possible solutions. Instead in the talk we plan to present the technical statements and proofs as formalised in Matita.

2 The (basic) labelling approach

We briefly explain the labelling approach as introduced in [3] on the example in Figure 1. The user wants to analyse the execution time of the program (the black lines in Figure 1a). He compiles the program using a special compiler that first inserts in the code three label emission statements (`EMIT` L_i) to mark the beginning of basic blocks (Figure 1a); then the compiler compiles the code to machine code (Figure 1b), granting that the execution of the source and object code emits the same sequence of labels ($L_1; L_2; L_2; L_3$ in the example). This is achieved by keeping track of basic blocks during compilation, avoiding all optimisations that alter the control flow. The latter can be recovered with a more refined version of the labelling approach [8], but in the present abstract we stick to this simple variant for simplicity. Once the object code is produced, the compiler runs a static code analyser to associate to each label $L_1, \ldots, L_3$ the cost (in clock cycles) of the instructions that belong to the corresponding basic block. For example, the cost k_1 associated to L_1 is the number of cycles required to execute the block I_3 and `COND` l_2, while the cost k_2 associated to L_2 counts the cycles required by the block I_4, `GOTO` l_1 and `COND` l_2. The compiler also guarantees that every executed instruction is in the scope of some code emission label, that each scope does not contain loops (to associate a finite cost), and that both branches of a conditional statement are followed by a code emission statement. Under these assumptions it is true that the total execution cost of the program Δ_t is equal to the sum over the sequence of emitted labels of the cost associated to every label: $\Delta_t = k(L_1; L_2; L_2; L_3) = k_1 + k_2 + k_2 + k_3$. Finally, the compiler emits an instrumented version of the source code (Figure 1c) where label emission statements are replaced by increments of a global variable `cost` that, before every increment, holds the exact number of clock cycles spent by the microprocessor so far: the difference Δ`cost` between the final and initial value of the internal clock is Δ`cost` $= k_1 + k_2 + k_2 + k_3 = \Delta_t$. Finally, the user can employ any available method (e.g. Hoare logic, invariant generators, abstract interpretation and automated provers) to certify that Δ`cost` never exceeds a certain bound [1], which is now a functional property of the code.

```
EMIT $L_1$;
$I_1$;
for (i=0; i<2; i++) {
  EMIT $L_2$;
  $I_2$;
}
EMIT $L_3$;
```

(a) The input program (black lines) with its labelling (red lines).

```
     EMIT $L_1$
     $I_3$
$l_1$: COND $l_2$
     EMIT $L_2$
     $I_4$
     GOTO $l_1$
$l_2$: EMIT $L_3$
```

(b) The output labelled object code.

```
cost += $k_1$;
$I_1$;
for (i=0; i<2; i++) {
  cost += $k_2$;
  $I_2$;
}
cost += $k_3$;
```

(c) The output instrumented code.

Fig. 1: The labelling approach applied to a simple program.. The I_i are sequences of instructions not containing jumps or loops.

3 The labelling approach revisited.

We briefly revise here some hidden assumptions and limitations of the original labelling approach described in Section 2 and called *basic* labelling approach in the rest of this abstract. We also hint at the solutions. The technical details and all the formal statements will be given in the presentation only.

3.1 Conditional statements and multiple predecessors

Assumptions: In order for the cost model to account for every instruction, object code instructions with multiple successors (e.g. conditional branches) must pass control to basic blocks that start with a label emission statement. Moreover, the cost of execution of the branching instruction must be constant whatever branch is taken.

The problem: In order to grant the first previous assumption, label emissions are introduced after every branch in the source code and the compilation is expected to preserve this invariant. The instruction that starts a branch, however, may have multiple predecessors. While this is rarely the case in an high level structured language, it is still be possible in C (due to `goto`s and the fall-back behaviour of `switch` branches) and it becomes very common during intermediate compilation phases (e.g. the instruction that follows a `while` loop may be reached from two different jumps if the loop hoisting optimisation is applied). Moreover, in actual architectures it may be the case that an high level branching instruction needs to be compiled to an instruction that locally branches to a non conditional jump to the actual beginning of the branch. Example: `if(E) { EMIT L; I }` may be compiled to `SJNEQ l1; JUMP l3; l1: JUMP l2; ... l2: EMIT L; I` where the short conditional branch `SJNEQ` can only jump within 256 bytes and the label emission is too far away. Hence the need for the indirect jump through `JUMP l2`. This happens, for instance, in the hardware architecture we picked in CerCo.

Clearly, the example code above violates the requirement because the `JUMP l3` instruction is not in the scope of any label and its cost will be missed. We would like to move the label emission away from `I` and put it just before the jump. This is incorrect as well when the block of `I` can be reached from other predecessors because in this case the label `L` would be emitted only if coming from the `if` instruction. The traces emitted in the source and object code would be different and the cost computed at the source level would be wrong.

The solution: in order to solve the problem, we change the labelling approach modifying the syntax (and semantics) of all the languages involved. In place of (or in addition to) having label emission statements, we incorporate label emissions into all branching instructions. For example, an `if` statement would now have syntax `if(E) L:{I}` meaning that the label `L` is emitted when the block `I` is reached coming from the `if` branch. If `I` has another predecessor (e.g. a `GOTO` statement), this other predecessor will be label emitting too (e.g. `GOTO L:l` to mean jump to location `l` emitting the label `L`). Instrumentation of the source code is now more complex, requiring a memory cell to remember the last block visited and thus pay the correct cost to be emitted at the beginning of a basic block.

The issue seen above is now solved: the `SJNEQ` instruction will emit the label `L` that will pay for both `JUMP l3` and `I`. Similarly, the other assembly level predecessor of `I` will emit another label that will pay for `I`. This new schema also allows to cope with architectures where a branching instruction has a different cost according to the branch taken. It suffices to make the labels emitted by the instruction pay the cost of the branching instruction. This is not possible in the basic labelling approach because the label emissions can have multiple predecessors.

3.2 Function calls

Let us now consider a simple program written in C that contains a function pointer call inside the scope of the cost label L_1, in Figure 2a. The labelling method works exactly as before, inserting code emission statements/`cost` variable increments at the beginning of every basic block and at the beginning of every function. The compiler still grants that the sequence of labels observed on the two programs are the same. A new difficulty appears when the compiler needs to statically analyse the object code to assign a cost to every label. What should the scope of the L_1 label be? After executing the I_4 block, the `CALL` statement passes control to a function that cannot be determined statically. Therefore the cost of executing the body must be paid by some other label (hence the requirement that every function starts with a code emission statement). What label should pay for the cost for the block I_5? The only reasonable answer is L_1, i.e. *the scope of labels should extend to the next label emission statement or the end of the function, stepping over function calls.*

The latter definition of scope is adequate on the source level because C is a structured language that guarantees that every function call, if it returns, passes control to the first instruction that follows the call. However, this is not guaranteed for object code, the backend languages of a compiler and, more generally, for unstructured languages that use a writable control stack to store the return address of calls. For example, I_6 could increment by 1 the return address on the stack so that the next `RETURN` would start at the second

```
void main() {
  EMIT $L_1$;
  $I_1$;
  (*f)();
  $I_2$;
}

void g() {
  EMIT $L_2$;
  $I_3$;
}
```

(a) The input labelled C program.

```
main:
  EMIT $L_1$
  $I_4$
  CALL
  $I_5$
  RETURN

g:
  EMIT $L_2$
  $I_6$
  RETURN
```

(b) The output labelled object code.

Fig. 2: An example compilation of a simple program with a function pointer call.

instruction of I_5. The compiler would still be perfectly correct if a random, dead code instruction was added after the `CALL` as the first instruction of I_5. More generally, *there is no guarantee that a correct compiler that respects the functional behaviour of a program also respects the calling structure of the source code.* Without such an assumption, however, it may not be true that the execution cost of the program is the sum of the costs associated to the labels emitted. In our example, the cost of I_5 is paid by L_1, but in place of I_5 the processor could execute any other code after g returns. We are therefore in the following situation.

Assumptions: to statically associate a cost to every block/label, the compiler needs to statically predict that all instructions in the block will be reached during computation. Moreover, if the sequence of instructions in the block is not sequentially executed, the function that computes the cost of the block given the cost of the instructions need to be commutative to allow reordering of instructions.

The problem: in the intermediate languages and the object code it is not possible to statically predict which instruction will be executed after a function returns. Thus, in order to extend the scope of blocks after function calls, we need to grant an additional property on the object code that is not granted by the basic labelling approach. In the latter case, moreover, the body of the function called is executed before the instructions after the function returns, requiring the function that computes the cost to be commutative. Because of stateful hardware components (e.g. caches and pipelines), the cost of execution of an instruction depends on the execution history. Therefore the function that given the steps in history computes the total cost is not commutative. Thus extending blocks after function calls does not scale to modern hardware.

We investigated two possible solutions.

First solution: we add more structure to execution traces. An execution trace is no longer a flat stream of observables. Instead, it becomes a recursively defined datatype where observables that correspond to function calls carry a nested copy of the datatype that describes the instructions executed in the function body. Moreover, it is now possible to observe the address of instructions in memory. Finally, we define the requirement that the observed function call and the instruction that follows the sub-datatype must be consecutive in memory. The only programs whose traces satisfy the requirement are those that do not tamper with the stack and such that every function call terminates returning just after the call.

The next step consists in defining local simulation conditions that have the following property: if every step in the source program is simulated by a structured trace fragment that satisfy the local conditions, then every trace in the source code is simulated by a structured trace in the object code that satisfy the requirement.

The main drawbacks of this solutions are: 1) the local simulation conditions are quite technical and the beauty of the simulation argument is lost in the details. We will present them and the simulation argument during the talk; 2) it allows for block scopes that extend after function calls, but it does not solve the problem with stateful hardware components since it still requires the cost function to be commutative.

Second solution: intuitively, we just ask every function call to be immediately followed by a label emission statement. Thus the scope of blocks no longer extends after function calls, solving both issues. More technically, we require return instructions to emit the cost label associated to the predecessor of the instruction control is returned to. No additional structure on execution traces is required and the simulation argument is essentially the same used with standard LTS and the basic labelling approach. Moreover, stateful hardware components are seamlessly supported.

The main drawbacks of this solution are: 1) the solution requires many more labels to be inserted in the source code, since blocks are now smaller. In turn, this makes the computation of cost invariants for source programs more difficult, the proof that the invariants hold harder and the computed cost less readable; 2) in order to reason on the cost of the source program, the user now needs to know that the value of the `cost` variable at the end of a block that contains function calls is the sum of the increments that occur after every call. This is not trivial because it requires a proof of termination of every call involved. In the first solution, instead, just a global assumption of termination of the whole program was required.

Amendment to the second solution: in order to mitigate the problems of the second solution, we studied a preliminary compilation step that turns a program labelled according to the first solution into one labelled according to the second solution. In practice, more observables are introduced just after function calls. The transformation also builds a sort of inverse function that maps traces of the compiled program back to traces of the source program (removing the new observables) and costs of blocks of the compiled program to costs of blocks of the source program (by consolidating the blocks together).

The benefits of this pass is that the user can still reason with smaller blocks and simpler costs like in the first solution, and the remaining parts of the compiler do not need to care about extra invariants like in the second solution. The proof of simulation is also as simple as in the second solution. The drawback is that the consolidation is not always possible in case of non commutative costs (e.g. in presence of stateful hardware components).

The additional pass was suggested as an obvious improvement of the second solution by an anonymous reviewer of a previous paper. Surprisingly at first, the proof of correctness of the pass is actually quite complex and its mechanisation in Matita is not finished yet. We will describe in the talk the reasons for the complexity. Intuitively, however, it is clear that the pass is correct only under non trivial assumptions like termination of every initiated function call. Moreover, following the literature we described the pass on a simple programming language equipped with an SOS semantics. In SOS it happens that parts of the source program are duplicated during execution (e.g. when entering the body of a `while` loop). In order for the proof to go through, we need to establish global invariants of the machine state that relate all copies of the same statements. This is required to keep track of labels that, being attached to instructions, are duplicated as well. In retrospect, it is likely that the proof could be greatly simplified abandoning the SOS semantics in favour of some alternative description where the code is read-only. This will be discussed in the talk, but left as future work.

References

1. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Régis-Gianas, Y., Sacerdoti Coen, C., Stark, I.: Certified complexity. Procedia Computer Science 7, 175–177 (2011)
2. Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of functional programs. In: Pea, R., Eekelen, M., Shkaravska, O. (eds.) Foundational and Practical Aspects of Resource Analysis, Lecture Notes in Computer Science, vol. 7177, pp. 72–89. Springer Berlin Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-32495-6_5`, extended version to appear in Higher Order and Symbolic Computation, 2013
3. Ayache, N., Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations in C programs. In: Stoelinga, M., Pinger, R. (eds.) Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 7437, pp. 32–46. Springer Berlin Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-32469-7_3`
4. Boender, J., Mulligan, D.P., Piccolo, M., Coen, C.S., Tranquilli, P.: CerCo Report n. D4.4, Back-end Correctness Proof. Technical report of the eu project cerco, University of Bologna (2013), `http://cerco.cs.unibo.it/raw-attachment/wiki/WikiStart/D4_4.pdf`
5. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009), `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`
6. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning 41(1), 1–31 (2008)
7. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
8. Tranquilli, P.: Indexed labels for loop iteration dependent costs. In: Proceedings 11th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2013. pp. 19–33. Electronic Proceedings in Theoretical Computer Science (EPTCS) (2013)