CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report
D5.1 Untrusted CerCo Prototype
and
D5.3 Case study: analysis of synchronous code

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-D5.3 is composed of the following parts:

1. This summary.
2. The paper [1] and the related software Cost.
3. The paper [4] and the related prototype compiler IndLabAcc (and the its Cost branch, IndLabCost).
4. The paper [2] and the related software LamCost.
5. The paper [3].

This document and the softwares mentioned 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] R.M. Amadio, Y. Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In Proc. FOPARA, Springer LNCS (to appear), 2012. Also Research Report, Université Paris Diderot, `http://hal.inria.fr/inria-00629473/en/`, 2011.

[3] A. Madet, R.M. Amadio. An Elementary Affine λ-Calculus with Multithreading and Side Effects. In Proc. TLCA, Springer LNCS 6690:138-152, 2011. Also Research Report, Université Paris Diderot, `http://hal.archives-ouvertes.fr/hal-00569095/fr/`, 2011.

[4] 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 first part of the document [1]. The second part of the document (formally, corresponding to deliverable D5.3) tries to delimit the practical applicability of the plug-in. To this end, the tool has been applied to the C code generated by the Lustre compiler and to some other simple C programs.

We pause to recall a redistribution of the workforce of the UPD site. Following the resignation of the doctoral student at the end of year 1, the contract of the post-doc has been extended till month 33. It follows that in the UPD site there has been a shift of manpower from the third to the second year. Because of this shift we decided to anticipate the presentation of deliverable D5.3 at month 24 rather than month 36. Besides this contingent reason, it is clear that the development of the synthesis tool must go hand in hand with its experimentation on larger and larger classes of programs. The UPD post-doc is expected to continue work on D5.1 and D5.3 till the end of its contract.

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 [4]. 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.

Certifying and reasoning on cost annotations of functional programs During the second year some unplanned work related to deliverables D5.1 and D5.3 has taken place at UPD. The main development [2] concerns the extension of the *CerCo* labelling approach described in D2.1 to a standard compilation chain from a higher-order functional language of the ML family to C. This work shows that the approach is sufficiently general to be applied to higher-order programs whose concrete complexity is generally regarded as difficult to estimate. Moreover, the introduction of higher-order functions calls for a higher-order logic to reason on the cost annotations. In this respect, we have built on previous work by one of the authors on a higher-order Hoare logic. Starting from a (higher-order) specification of the expected cost of a function our tool LamCost produces automatically a list of proof obligations. Preliminary experiments suggest that a large part of these proof obligations can be discharged automatically and that the remaining ones can be proved in a proof assistant such as Coq. Ongoing work that should be completed within the third year of the project is extending the compilation chain and the cost analysis to include garbage collection using a *region based* memory management à la Tofte-Talpin.

An Elementary Affine λ-Calculus with Multithreading and Side Effects. A second development at UPD is of a more speculative nature and is concerned with the design of a type system for a functional language with side effects that guarantees complexity bounds. As far as we know, this is the first work that accounts for side effects. The obtained result concerns *elementary time* and ongoing work that should be completed within the third year of the project concerns a similar result for *polynomial time*. We regard this work as a step towards bridging the *CerCo* approach with the work on *Implicit computational complexity* (ICC) in which our universities are also involved (in 2011, the *CerCo* project organised in Paris a joint workshop with an ICC oriented project). As a matter of fact, there is still a large gap to be filled before the results in ICC can have an impact on the practice of programming.

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 plateform that, starting from CerCo's cost annotation of a C function, synthesizes a cost bound for the function. We report our experimentations on standard C code and C code generated from Lustre files.

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 compute and certify fully automatically 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 plate-form 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.

[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

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

- 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.

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 presents a case study for the plug-in on the Lustre synchronous language. Section 5 shows some benchmarks on standard C programs, on C programs for cryptography (typically used in embedded software) and on C programs originated from Lustre files. Finally, section 6 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:

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

```
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

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.

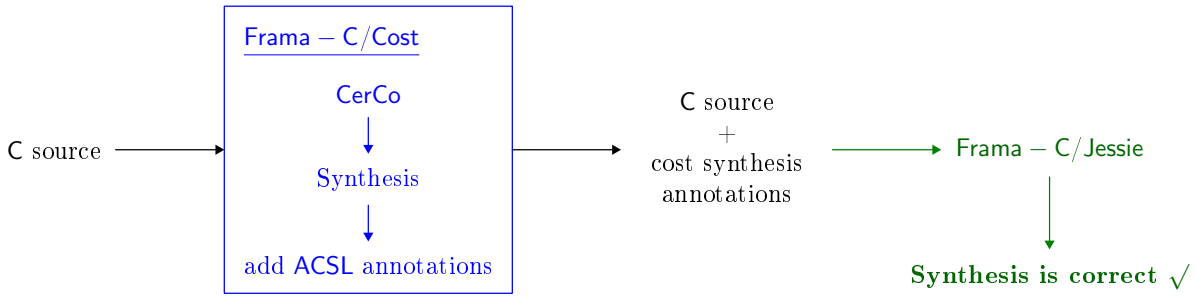

Figure 3: the Cost plug-in

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 Lustre case study

Lustre is a synchronous language where reactive systems are described by flow of values. It comes with a compiler that transforms a Lustre node (any part of or the whole system) into a C *step* function that represents one synchronous cycle of the node. A WCET for the step function is thus a worst case reaction time for the component. The generated C step function neither contains loops nor is recursive, which makes it particularly well suited for a use with the Cost plug-in with a complete support.

We designed a wrapper that has for inputs a Lustre file and a node inside the file, and outputs the cost of the C step function corresponding to the node. Optionally, verification with Jessie or testing can be toggled. The flow of the wrapper is described in figure 5. It simply executes a command line, reads the results, and sends them to the next command.

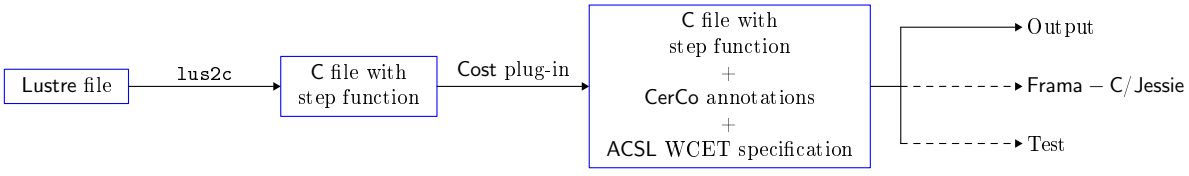

Figure 5: Flow of the Lustre wrapper

[3] http://matita.cs.unibo.it/
[4] http://coq.inria.fr/

A typical run of the wrapper looks as follows (we use the `parity` example from our distribution of Lustre; it computes the parity bit of a boolean array):

```
frama-c_lustre -verify -test parity.lus parity
```

Invoking the above command line produces the following output:

```
WCET of parity_step:  2220+_cost_of_parity_0_parity+_cost_of_parity_0_done
(not verified).
Verifying the result (this may take some time)...
WCET is proven correct.
Testing the result (this may take some time)...
Estimated WCET: 2220
Minimum:  2144
Maximum:  2220
Average:  2151
Estimated WCET is correct for these executions.
```

- All the intermediary results of the wrapper are stored in files. Verbosity can be turned on to show the different commands invoked and the resulting files.
- The step function generated with the Lustre compiler for the node `parity` is called `parity_step`. It might call functions that are not defined but only prototyped, such as `parity_0_parity` or `parity_0_done`. Those are functions that the user of the Lustre compiler can use for debugging, but that are not part of the `parity` system. Therefore, we leave their cost abstract in the expression of the cost of the step function, and we set their cost to 0 when testing (this can be changed by the user).
- Testing consists in adding a `main` function to the C file, that will run the step function on a parameterized number of input states for a parameterized number of cycles. The C file contains information that allows to syntatically distinguish integer variables used as booleans, which helps in generating interesting input states. After each iteration of the step function, the value of the cost variable is fetched in order to compute its overall minimum, maximum and average value for one step. If the maximum were to be greater than the WCET computed by the Cost plug-in, then we could conclude of an error in the plug-in.

5 Experiments

The Cost plug-in and the Lustre wrapper have been developed in order to validate CerCo's framework for modular WCET analysis, by showing the results that could be obtained with this approach. Through CerCo, they allow (semi-)automatic generation and certification of WCET for C and Lustre programs. For the latter, the WCET represents a bound for the reaction time of a component. 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. 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 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.

6 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 C programs generated from Lustre files; the result is an automatically computed and certified reaction time for the Lustre nodes. 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.

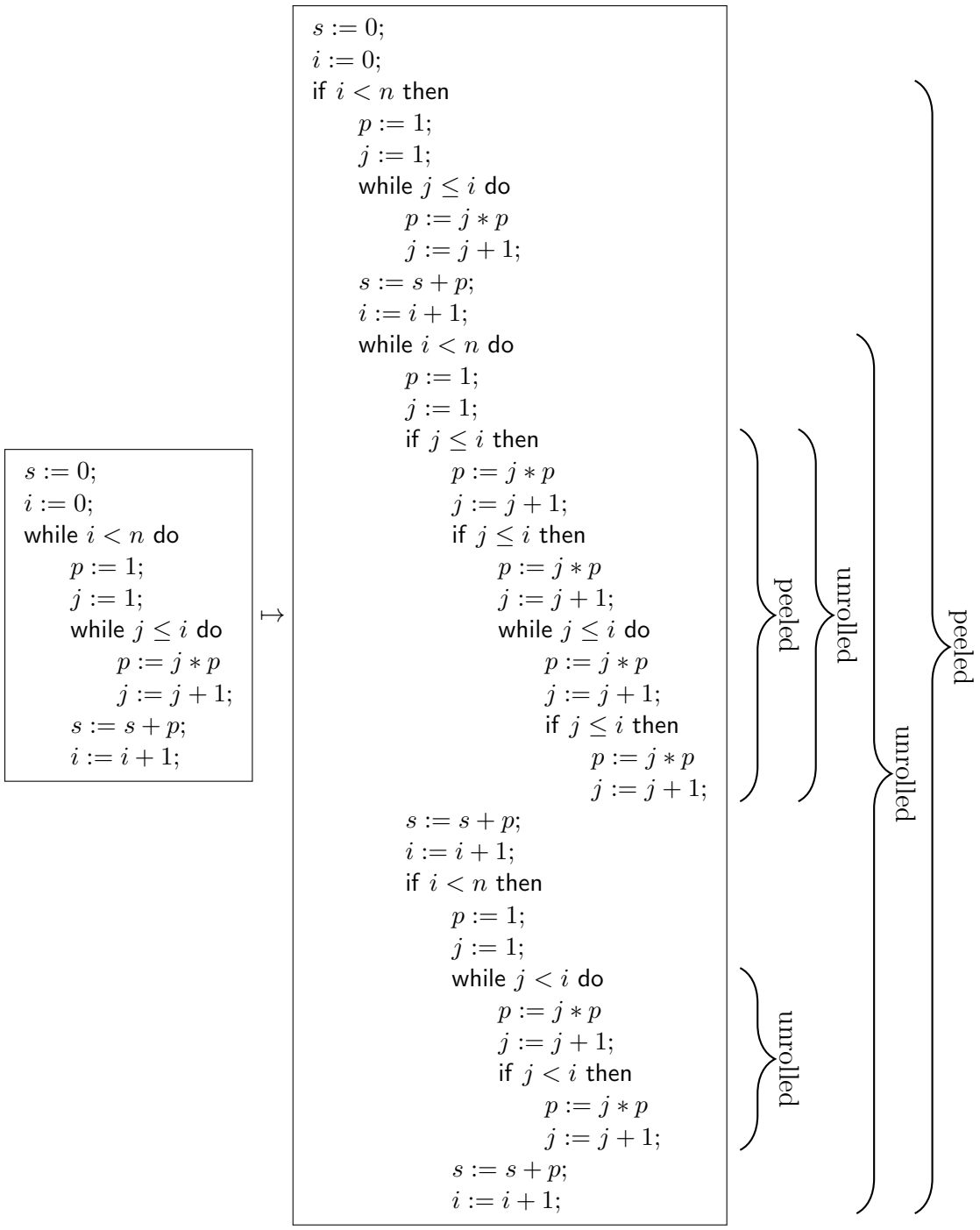

Figure 2: An example of loop transformations done on an Imp program. Parentheses are omitted in favour of blocks by indentation.

```
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.

Certifying and reasoning on cost annotations of functional programs

Roberto M. Amadio[1] and Yann Régis-Gianas[1,2]

[1] Université Paris Diderot (UMR-CNRS 7126)
[2] INRIA (Team πr^2)

Abstract We present a so-called labelling method to insert cost annotations in a higher-order functional program, to certify their correctness with respect to a standard compilation chain to assembly code, and to reason on them in a higher-order Hoare logic.

1 Introduction

In [1] we have discussed the problem of building a C compiler which can *lift* in a provably correct way pieces of information on the execution cost of the object code to cost annotations on the source code. To this end, we have introduced a so called *labelling* approach and presented its application to a prototype compiler written in Ocaml from a large fragment of the C language to the assembly languages of Mips and 8051, a 32 bits and 8 bits processor, respectively.

In the following, we are interested in extending the approach to (higher-order) functional languages. On this issue, a common belief is well summarized by the following epigram [9]: *A Lisp programmer knows the value of everything, but the cost of nothing.* However, we shall show that, with some ingenuity, the methodology developed for the C language can be lifted to functional languages. Specifically, we shall focus on a rather standard compilation chain from a call-by-value λ-calculus to a register transfer level (RTL) language. Similar compilation chains have been explored from a formal viewpoint in [8] (with an emphasis on typing but with no simulation proofs) and in [4] (for type-free languages but with machine certified simulation proofs).

The compilation chain is described in the lower part of table 1. Starting from a standard call-by-value λ-calculus with pairs, one performs first a CPS translation, then a transformation into administrative form, followed by a closure conversion, and a hoisting transformation. All languages considered are subsets of the initial one though their evaluation mechanism is refined along the way. In particular, one moves from an ordinary substitution to a specialized one where variables can only be replaced by other variables. Notable differences with respect to [4] is a different choice of the intermediate languages and the fact that we rely on a small-step operational semantics. We also diverge from [4] in that our proofs, following the usual mathematical tradition, are written to explain to a human why a certain formula is valid rather than to provide a machine with a compact witness of the validity of the formula.

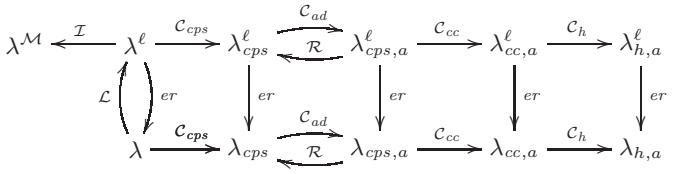

Table1. The compilation chain with its labelling and instrumentation.

The final language of this compilation chain can be directly mapped to an RTL language: functions correspond to assembly level routines and the functions' bodies correspond to sequences of assignments on pseudo-registers ended by a tail recursive call.

While the *extensional* properties of the compilation chain have been well studied, we are not aware of previous work focusing on more *intensional* properties relating to the way the compilation preserves the complexity of the programs. Specifically, in the following we will apply to this compilation chain the 'labelling approach' to building certified cost annotations. In a nutshell the approach consists in identifying, by means of labels, points in the source program whose cost is constant and then determining the value of the constants by propagating the labels along the compilation chain and analysing small pieces of object code with respect to a target architecture.

Technically the approach is decomposed in several steps. First, for each language considered in the compilation chain, we define an extended *labelled* language and an extended operational semantics (upper part of Table 1). The labels are used to mark certain points of the control. The semantics makes sure that whenever we cross a labelled control point a labelled and observable transition is produced.

Second, for each labelled language there is an obvious function *er* erasing all labels and producing a program in the corresponding unlabelled language. The compilation functions are extended from the unlabelled to the labelled language so that they commute with the respective erasure functions. Moreover, the simulation properties of the compilation functions are lifted from the unlabelled to the labelled languages and transition systems.

Third, assume a *labelling* $\mathcal{L}$ of the source language is a right inverse of the respective erasure function. The evaluation of a labelled source program produces both a value and a sequence of labels, say Λ, which intuitively stands for the sequence of labels crossed during the program's execution. The central question we are interested in is whether there is a way of labelling the source programs so that the sequence Λ is a sound and possibly precise representation of the execution cost of the program.

To answer this question, we observe that the object code is some kind of RTL code and that its control flow can be easily represented as a control flow graph. The fact that we have to prove the soundness of the compilation function means that we have plenty of information on the way the control flows in the

compiled code, in particular as far as procedure calls and returns are concerned. These pieces of information allow to build a rather accurate representation of the control flow of the compiled code at run time.

The idea is then to perform some simple checks on the control flow graph. The main check consists in verifying that all 'loops' go through a labelled node. If this is the case then we can associate a 'cost' with every label which overapproximates the actual cost of running a sequence of instructions. An optional check amounts to verify that all paths starting from a label have the same abstract cost. If this check is successful then we can conclude that the cost annotations are 'precise' in an abstract sense (and possibly concrete too depending on the processor considered).

If the check described above succeeds every label has a cost which in general can be taken as an element of a 'cost' monoid. Then an *instrumentation* of the source labelled language is a monadic transformation $\mathcal{I}$ (left upper part of Table 1) in the sense of [6] that replaces labels with the associated elements of the cost monoid. Following this monadic transformation we are back into the source language (possibly enriched with a 'cost monoid' such as integers with addition). As a result, the source program is instrumented so as to monitor its execution cost with respect to the associated object code. In the end, general logics developed to reason about functional programs such as higher-order Hoare logic [11] can be employed to reason about the concrete complexity of source programs by proving properties on their instrumented versions.

We stress that previous work on building cost annotations for higher-order functional programs we are aware of does not take formally into account the compilation process. For instance, in an early work D. Sands [12] proposes an instrumentation of call-by-value λ-calculus in order to describe its execution cost. However the notion of cost adopted is essentially the number of function calls in the source code. In a standard implementation such as the one considered in this work, different function calls may have different costs and moreover there are 'hidden' function calls which are not immediately apparent in the source code. In a more recent work, [3] addresses the problem of determining the worst case execution time of a a specialised functional language called Hume. The compilation chain considered consists in compiling first Hume to the code of an intermediate abstract machine, then to C, and finally to generate the assembly code of the Resenas M32C/85 processor using standard C compilers. Then for each instruction of the abstract machine, one computes an upper bound on the worst-case execution time (WCET) of the instruction relying on a well-known aiT tool [2] that uses abstract interpretation to determine the WCET of sequences of binary instructions. While we share common motivations with this work, we differ significantly in the technical approach. In particular, (i) [3] does not address at all the proof of correctness of the cost annotations as we do, and (ii) the granularity of the cost annotations is fixed in [3] (the instructions of the Hume abstract machine) while it can vary in our approach.

In [1] we have showed that it is possible to produce a sound and precise (in an abstract sense) labelling for a large class of C programs with respect

to a moderately optimising compiler. In the following we show that a similar result can be obtained for a higher-order functional language with respect to the standard compilation chain described above. Specifically we show that there is a simple labelling of the source program that guarantees that the generated object code is sound and precise. The labelling of the source program can be informally described as follows: it associates a distinct label with every abstraction and with every application which is not 'immediately surrounded' by an abstraction.

In this paper our analysis will stop at the level of an abstract RTL language, however our previously quoted work [1] shows that the approach extends to the back-end of a typical moderately optimising compiler including, *e.g.*, dead-code elimination and register allocation. Concerning the source language, preliminary experiments suggest that the approach scales to a larger functional language such as the one considered in [4] including sums, exceptions, and side effects. Finally, we mention that the approach has also been implemented for a simpler compilation chain that bypasses the CPS translation. In this case, the function calls are not necessarily tail-recursive and the compiler generates a Cminor program.[1]

In the following, section 2 describes the certification of the cost-annotations and section 3 a method to reason on them. Examples and proofs are available in appendices A and B, respectively.

2 The compilation chain: commutation and simulation

This section describes the intermediate languages and the compilation functions from an ordinary λ-calculus to a hoisted, administrative λ-calculus. For each step we check that: (i) the compilation function commutes with the function that erases labels and (ii) the object code simulates the source code.

2.1 Conventions

The reader is supposed to be acquainted with the λ-calculus and its evaluation strategies and continuation passing style translations. In the following calculi, all terms are manipulated up to α-renaming of bound names. We denote with $\equiv$ syntactic identity up to α-renaming. Whenever a reduction rule is applied, it is assumed that terms have been renamed so that all binders use distinct variables and these variables are distinct from the free ones. Similar conventions are applied when performing a substitution, say $[T/x]T'$, of a term T for a variable x in a term T'. We denote with $\mathsf{fv}(T)$ the set of variables occurring free in a term T.

Let $C, C_1, C_2, \ldots$ be one hole contexts and T a term. Then $C[T]$ is the term resulting from the replacement in the context C of the hole by the term T and $C_1[C_2]$ is the one hole context resulting from the replacement in the context C_1 of the hole by the context C_2.

For each calculus, we assume a syntactic category id of identifiers with generic elements $x, y, \ldots$ and a syntactic category ℓ of labels with generic elements

[1] Cminor is a type-free, memory aware fragment of C defined in [7].

Syntax

$$
\begin{array}{lll}
V & ::= id \mid \lambda id^{+}.M \mid (V^{+}) & \text{(values)} \\
M & ::= V \mid @(M, M^{+}) \mid \mathsf{let}\ id = M\ \mathsf{in}\ M \mid (M^{+}) \mid \pi_i(M) \mid \ell > M \mid M > \ell & \text{(terms)} \\
E & ::= [\,] \mid @(V^{*}, E, M^{*}) \mid \mathsf{let}\ id = E\ \mathsf{in}\ M \mid (V^{*}, E, M^{*}) \mid \pi_i(E) \mid E > \ell & \text{(eval. cxts.)}
\end{array}
$$

Reduction Rules

$$
\begin{array}{ll}
E[@(\lambda x_1 \ldots x_n.M, V_1, \ldots, V_n)] & \to E[[V_1/x_1, \ldots, V_n/x_n]M] \\
E[\mathsf{let}\ x = V\ \mathsf{in}\ M] & \to E[[V/x]M] \\
E[\pi_i(V_1, \ldots, V_n)] & \to E[V_i] \qquad 1 \le i \le n \\
E[\ell > M] & \xrightarrow{\ell} E[M] \\
E[V > \ell] & \xrightarrow{\ell} E[V]
\end{array}
$$

Label erasure

$$er(\ell > M) = er(M > \ell) = er(M)\ .$$

Table2. An ordinary call-by-value λ-calculus: λ^{ℓ}

$\ell, \ell_1, \ldots$ For each calculus, we specify the syntactic categories and the reduction rules. We let α range over labels and the empty word. We write $M \xrightarrow{\alpha} N$ if M rewrites to N with a transition labelled by α. We abbreviate $M \xrightarrow{\epsilon} N$ with $M \to N$. We also define $M \stackrel{\alpha}{\Rightarrow} N$ as $M \xrightarrow{*} N$ if $\alpha = \epsilon$ and as $M \xrightarrow{*}\xrightarrow{\alpha}\xrightarrow{*} N$ otherwise.

We shall write X^{+} (resp. X^{*}) for a non-empty (possibly empty) finite sequence $X_1, \ldots, X_n$ of symbols. By extension, $\lambda x^{+}.M$ stands for $\lambda x_1 \ldots \lambda x_n.M$, $[V^{+}/x^{+}]M$ stands for $[V_1/x_1](\cdots[V_n/x_n]M \cdots)$, and $\mathsf{let}\ (x = V)^{+}\ \mathsf{in}\ M$ stands for $\mathsf{let}\ x_1 = V_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n = V_n\ \mathsf{in}\ M$.

2.2 The source language

Table 2 introduces a type-free, call-by-value λ-calculus. The calculus includes *let-definitions* and *polyadic abstraction* and *pairing* with the related application and projection operators. Any term M can be *pre-labelled* by writing $\ell > M$ or *post-labelled* by writing $M > \ell$. In the pre-labelling, the label ℓ is emitted immediately while in the post-labelling it is emitted after M has reduced to a value. It is tempting to reduce the post-labelling to the pre-labelling by writing $M > \ell$ as $@(\lambda x.\ell > x, M)$, however the second notation introduces an additional abstraction and a related reduction step which is not actually present in the original code. Table 2 also introduces an *erasure* function er from the λ^{ℓ}-calculus to the λ-calculus. This function simply traverses the term and erases all pre and post labellings. Similar definitions arise in the following calculi of the compilation chain and are omitted.

2.3 Compilation to CPS form

Table 3 introduces a fragment of the λ^ℓ-calculus described in Table 2 and a related CPS translation. We recall that in a CPS translation each function takes its evaluation context as an additional parameter. Then the evaluation context is always trivial. Notice that the reduction rules are essentially those of the λ^ℓ-calculus modulo the fact that we drop the rule to reduce $V > \ell$ since post-labelling does not occur in CPS terms and the fact that we optimize the rule for the projection to guarantee that CPS terms are closed under reduction. For instance, the term $\mathsf{let}\ x = \pi_1(V_1, V_2)\ \mathsf{in}\ M$ reduces directly to $[V_1/x]M$ rather than going through the intermediate term $\mathsf{let}\ x = V_1\ \mathsf{in}\ M$ which does not belong to the CPS terms.

We study next the properties enjoyed by the CPS translation. In general, the commutation of the compilation function with the erasure function only holds up to call-by-value η-conversion, namely $\lambda x.@(V, x) =_\eta V$ if $x \notin \mathsf{fv}(V)$. This is due to the fact that post-labelling introduces an η-expansion of the continuation if and only if the continuation is a variable. To cope with this problem, we introduce next the notion of *well-labelled* term. We will see later (section 3.1) that terms generated by the initial labelling are well-labelled.

Definition 1 (well-labelling). *We define two predicates W_i, $i = 0, 1$ on the terms of the λ^ℓ-calculus as the least sets such that W_1 is contained in W_0 and the following conditions hold:*

$$\frac{}{x \in W_1} \qquad \frac{M \in W_0}{M > \ell \in W_0} \qquad \frac{M \in W_1}{\lambda x^+.M \in W_1}$$

$$\frac{M \in W_i \quad i \in \{0,1\}}{\ell > M \in W_i} \qquad \frac{N \in W_0, M \in W_i \quad i \in \{0,1\}}{\mathsf{let}\ x = N\ \mathsf{in}\ M \in W_i}$$

$$\frac{M_i \in W_0 \quad i = 1, \ldots, n}{@(M_1, \ldots, M_n) \in W_1} \qquad \frac{M_i \in W_0 \quad i = 1, \ldots, n}{(M_1, \ldots, M_n) \in W_1} \qquad \frac{M \in W_0}{\pi_i(M) \in W_1}\ .$$

The intuition is that we want to avoid the situation where a post-labelling receives as continuation the continuation variable generated by the translation of a λ-abstraction.

Proposition 1 (CPS commutation). *Let $M \in W_0$ be a term of the λ^ℓ-calculus (Table 2). Then: $er(\mathcal{C}_{cps}(M)) \equiv \mathcal{C}_{cps}(er(M))$.*

The proof of the CPS simulation is non-trivial but rather standard since [10]. The general idea is that the CPS translation pre-computes many 'administrative' reductions so that the translation of a term, say $E[@(\lambda x.M, V)]$ is a term of the shape $@(\psi(\lambda x.M), \psi(V), K_E)$ for a suitable continuation K_E depending on the evaluation context E.

Proposition 2 (CPS simulation). *Let M be a term of the λ^ℓ-calculus. If $M \xrightarrow{\alpha} N$ then $\mathcal{C}_{cps}(M) \overset{\alpha}{\Rightarrow} \mathcal{C}_{cps}(N)$.*

Syntax CPS terms

$$
\begin{array}{lll}
V & ::= id \mid \lambda id^{+}.M \mid (V^{+}) & \text{(values)} \\
M & ::= @(V, V^{+}) \mid \mathsf{let}\ id = \pi_i(V)\ \mathsf{in}\ M \mid \ell > M & \text{(CPS terms)} \\
K & ::= id \mid \lambda id.M & \text{(continuations)}
\end{array}
$$

Reduction rules

$$
\begin{array}{lll}
@(\lambda x_1, \ldots, x_n.M, V_1, \ldots, V_n) & \rightarrow [V_1/x_1, \ldots, V_n/x_n]M & \\
\mathsf{let}\ x = \pi_i(V_1, \ldots, V_n)\ \mathsf{in}\ M & \rightarrow [V_i/x]M & 1 \leq i \leq n \\
\ell > M & \xrightarrow{\ell} M &
\end{array}
$$

CPS translation

$$
\begin{array}{ll}
\psi(x) & = x \\
\psi(\lambda x^{+}.M) & = \lambda x^{+}, k.(M : k) \\
\psi(V_1, \ldots, V_n) & = (\psi(V_1), \ldots, \psi(V_n)) \\
 & \\
V : k & = @(k, \psi(V)) \\
V : (\lambda x.M) & = [\psi(V)/x]M \\
@(M_0, \ldots, M_n) : K & = M_0 : \lambda x_0. \ldots (M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\
\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : K & = M_1 : \lambda x.(M_2 : K) \\
(M_1, \ldots, M_n) : K & = M_1 : \lambda x_1. \ldots (M_n : \lambda x_n.(x_1, \ldots, x_n) : K\ ) \\
\pi_i(M) : K & = M : \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K \\
(\ell > M) : K & = \ell > (M : K) \\
(M > \ell) : K & = M : (\lambda x.\ell > (x : K)) \\
 & \\
\mathcal{C}_{cps}(M) & = M : \lambda x.@(\mathit{halt}, x), \qquad \mathit{halt}\ \text{fresh}
\end{array}
$$

Table3. CPS λ-calculus (λ^{ℓ}_{cps}) and CPS translation

2.4 Transformation in administrative CPS form

Table 4 introduces an *administrative* λ-calculus in CPS form: $\lambda^{\ell}_{cps,a}$. In the ordinary λ-calculus, the application of a λ-abstraction to an argument (which is value) may produce the duplication of the argument as in: $@(\lambda x.M, V) \rightarrow [V/x]M$. In the administrative λ-calculus, all values are named and when we apply the name of a λ-abstraction to the name of a value we create a new copy of the body of the function and replace its formal parameter name with the name of the argument as in:

$$\mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ @(f, y)\ \rightarrow\ \mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ f = \lambda x.M\ \mathsf{in}\ [y/x]M\ .$$

We also remark that in the administrative λ-calculus the evaluation contexts are a sequence of let definitions associating values to names. Thus, apart for the fact that the values are not necessarily closed, the evaluation contexts are similar to the environments of abstract machines for functional languages.

Table 5 defines the compilation into administrative form along with a read-back translation. The latter is useful to state the simulation property. Indeed,

SYNTAX

$$
\begin{array}{lll}
V & ::= \lambda id^{+}.M \mid (id^{+}) & \text{(values)} \\
B & ::= V \mid \pi_i(id) & \text{(let-bindable terms)} \\
M & ::= @(id, id^{+}) \mid \mathsf{let}\ id = B\ \mathsf{in}\ M \mid \ell > M & \text{(CPS terms)} \\
E & ::= [\,] \mid \mathsf{let}\ id = V\ \mathsf{in}\ E & \text{(evaluation contexts)}
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{rcll}
E[@(x, z_1, \ldots, z_n)] & \rightarrow & E[[z_1/y_1, \ldots, z_n/y_n]M] & \text{if } E(x) = \lambda y_1, \ldots, y_n.M \\
E[\mathsf{let}\ z = \pi_i(x)\ \mathsf{in}\ M] & \rightarrow & E[[y_i/z]M]] & \text{if } E(x) = (y_1, \ldots, y_n), 1 \leq i \leq n \\
E[\ell > M] & \stackrel{\ell}{\rightarrow} & E[M] &
\end{array}
$$

$$
\text{where: } E(x) = \begin{cases} V & \text{if } E = E'[\mathsf{let}\ x = V\ \mathsf{in}\ [\,]] \\ E'(x) & \text{if } E = E'[\mathsf{let}\ y = V\ \mathsf{in}\ [\,]], x \neq y \\ \text{undefined otherwise} & \end{cases}
$$

Table4. An administrative CPS λ-calculus: $\lambda^{\ell}_{cps,a}$

it is not true that if $M \rightarrow M'$ in λ^{ℓ}_{cps} then $\mathcal{C}_{ad}(M) \stackrel{*}{\rightarrow} \mathcal{C}_{ad}(M')$ in $\lambda^{\ell}_{cps,a}$. For instance, consider $M \equiv (\lambda x.xx)I$ where $I \equiv (\lambda y.y)$. Then $M \rightarrow II$ but $\mathcal{C}_{ad}(M)$ does not reduce to $\mathcal{C}_{ad}(II)$ but rather to a term where the 'sharing' of the duplicated value I is explicitly represented.

Proposition 3 (AD commutation). *Let M be a λ-term in CPS form. Then:*

(1) $\mathcal{R}(\mathcal{C}_{ad}(M)) \equiv M$.

(2) $er(\mathcal{C}_{ad}(M)) \equiv \mathcal{C}_{ad}(er(M))$.

Proposition 4 (AD simulation). *Let N be a λ-term in CPS administrative form. If $\mathcal{R}(N) \equiv M$ and $M \stackrel{\alpha}{\rightarrow} M'$ then $N \stackrel{\alpha}{\rightarrow} N'$ and $\mathcal{R}(N') \equiv M'$.*

2.5 Closure conversion

The next step is called *closure conversion*, it consists in providing each functional value with an additional parameter that accounts for the names free in the body of the function. Following this transformation which is described in Table 6, all functional values are closed. In our opinion, this is the only compilation step where the proofs are rather straightforward.

Proposition 5 (CC commutation). *Let M be a CPS term in administrative form. Then $er(\mathcal{C}_{cc}(M)) \equiv \mathcal{C}_{cc}(er(M))$.*

Proposition 6 (CC simulation). *Let M be a CPS term in administrative form. If $M \stackrel{\alpha}{\rightarrow} M'$ then $\mathcal{C}_{cc}(M) \stackrel{\alpha}{\Rightarrow} \mathcal{C}_{cc}(M')$.*

Transformation in administrative form (from λ^{ℓ}_{cps} to $\lambda^{\ell}_{cps,a}$)

$$
\begin{array}{ll}
\mathcal{C}_{ad}(@(x_0,\ldots,x_n)) & = @(x_0,\ldots,x_n) \\
\mathcal{C}_{ad}(@(x^*,V,V^*)) & = \mathcal{E}_{ad}(V,y)[\mathcal{C}_{ad}(@(x^*,y,V^*))] \quad V \neq id, y \text{ fresh} \\
\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M) & = \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{ad}(M) \\
\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M) & = \mathcal{E}_{ad}(y,V)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{ad}(M)] \quad V \neq id, y \text{ fresh} \\
\mathcal{C}_{ad}(\ell > M) & = \ell > \mathcal{C}_{ad}(M)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}_{ad}(\lambda x^+.M, y) & = \mathsf{let}\ y = \lambda x^+.\mathcal{C}_{ad}(M)\ \mathsf{in}\ [\,] \\
\mathcal{E}_{ad}((x^+), y) & = \mathsf{let}\ y = (x^+)\ \mathsf{in}\ [\,] \\
\mathcal{E}_{ad}((x^*,V,V^*), y) & = \mathcal{E}_{ad}(V,z)[\mathcal{E}_{ad}((x^*,z,V^*), y)] \quad V \neq id, z \text{ fresh}
\end{array}
$$

Readback translation (from $\lambda^{\ell}_{cps,a}$ to λ^{ℓ}_{cps})

$$
\begin{array}{ll}
\mathcal{R}(\lambda x^+.M) & = \lambda x^+.\mathcal{R}(M) \\
\mathcal{R}(x^+) & = (x^+) \\
\mathcal{R}(@(x,x_1,\ldots,x_n)) & = @(x,x_1,\ldots,x_n) \\
\mathcal{R}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M) & = \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{R}(M) \\
\mathcal{R}(\mathsf{let}\ x = V\ \mathsf{in}\ M) & = [\mathcal{R}(V)/x]\mathcal{R}(M) \\
\mathcal{R}(\ell > M) & = \ell > \mathcal{R}(M)
\end{array}
$$

Table5. Transformations in administrative CPS form and readback

2.6 Hoisting

The last compilation step consists in moving all functions definitions at top level. In Table 7, we formalise this compilation step as the iteration of a set of program transformations that commute with the erasure function and the reduction relation. Denote with $\lambda z^+.T$ a function that does *not* contain function definitions. The transformations consist in hoisting (moving up) the definition of a function $\lambda z^+.T$ with respect to either a definition of a pair or a projection, or another including function, or a labelling. Note that the hoisting transformations do not preserve the property that all functions are closed. Therefore the hoisting transformations are defined on the terms of the $\lambda^{\ell}_{cps,a}$-calculus. As a first step, we analyse the hoisting transformations.

Proposition 7 (on hoisting transformations). *The iteration of the hoisting transformation on a term in $\lambda^{\ell}_{cc,a}$ (all function are closed) terminates and produces a term satisfying the syntactic restrictions specified in table 7.*

Next we check that the hoisting transformations commute with the erasure function.

Proposition 8 (hoisting commutation). *Let M be a term of the $\lambda^{\ell}_{cps,a}$-calculus.*

(1) *If $M \rightsquigarrow N$ then $er(M) \rightsquigarrow er(N)$ or $er(M) \equiv er(N)$.*

(2) *If $M \not\rightsquigarrow \cdot$ then $er(M) \not\rightsquigarrow \cdot$.*

(3) *$er(\mathcal{C}_h(M)) \equiv \mathcal{C}_h(er(M))$.*

Syntactic restrictions on $\lambda^{\ell}_{cps,a}$ after closure conversion
All functional values are closed.

Closure Conversion

$$\mathcal{C}_{cc}(@(x, y^{+})) = \mathsf{let}\ z = \pi_1(x)\ \mathsf{in}\ @(z, x, y^{+})$$

$$\mathcal{C}_{cc}(\mathsf{let}\ x = B\ \mathsf{in}\ M) = \begin{array}{l} \mathsf{let}\ y = \lambda z, x^{+}.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\ \mathsf{let}\ x = (y, z_1, \ldots, z_k)\ \mathsf{in} \\ \mathcal{C}_{cc}(M) \qquad (\text{if } B = \lambda x^{+}.N, \mathsf{fv}(B) = \{z_1, \ldots, z_k\}) \end{array}$$

$$\mathcal{C}_{cc}(\mathsf{let}\ x = B\ \mathsf{in}\ M) = \mathsf{let}\ x = B\ \mathsf{in}\ \mathcal{C}_{cc}(M) \qquad (\text{if } B \text{ not a function})$$

$$\mathcal{C}_{cc}(\ell > M) = \ell > \mathcal{C}_{cc}(M)$$

Table6. Closure conversion on administrative CPS terms

The proof of the simulation property requires some work because to close the diagram we need to collapse repeated definitions. We proceed as follows. First we introduce a relation S_h that collapses repeated definitions and show that it is a simulation. Second, we show that the hoisting transformations induce a 'simulation up to S_h'. Namely if $M \xrightarrow{\ell} M'$ and $M \rightsquigarrow N$ then there is a N' such that $N \xrightarrow{\ell} N'$ and M' $(\rightsquigarrow^* \circ S_h)$ N'. Third, we iterate the previous property to derive the following one.

Proposition 9 (hoisting simulation). *There is a simulation relation $\mathcal{T}_h$ on the terms of the $\lambda^{\ell}_{cps,a}$-calculus such that for all terms M of the $\lambda^{\ell}_{cc,a}$-calculus we have $M\ \mathcal{T}_h\ \mathcal{C}_h(M)$.*

2.7 Composed commutation and simulation properties

Let $\mathcal{C}$ be the composition of the compilation steps we have considered:

$$\mathcal{C} = \mathcal{C}_h \circ \mathcal{C}_{cc} \circ \mathcal{C}_{ad} \circ \mathcal{C}_{cps}\ .$$

We also define a relation $\mathcal{R}_C$ between terms in λ^{ℓ} and terms in λ^{ℓ}_h as:

$$M \mathcal{R}_C P \text{ if } \exists N\ \ \mathcal{C}_{cps}(M) \equiv \mathcal{R}(N) \text{ and } \mathcal{C}_{cc}(N)\ \mathcal{T}_h\ P$$

Note that for all M, $M\ \mathcal{R}_C\ \mathcal{C}(M)$.

Theorem 1 (commutation and simulation). *Let $M \in W_0$ be a term of the λ^{ℓ}-calculus. Then:*

(1) $er(\mathcal{C}(M)) \equiv \mathcal{C}(er(M))$.

(2) *If $M\ \mathcal{R}_C\ N$ and $M \xrightarrow{\alpha} M'$ then $N \stackrel{\alpha}{\Rightarrow} N'$ and $M'\ \mathcal{R}_C\ N'$.*

Syntactic restrictions on $\lambda^{\ell}_{cps,a}$ after hoisting

All function definitions are at top level.

$$
\begin{array}{lll}
C ::= (id^{+}) \mid \pi_i(id) & & \text{(restricted let-bindable terms)} \\
T ::= @(id, id^{+}) \mid \mathsf{let}\ id = C\ \mathsf{in}\ T \mid \ell > T & & \text{(restricted terms)} \\
P ::= T \mid \mathsf{let}\ id = \lambda id^{+}.T\ \mathsf{in}\ P & & \text{(programs)}
\end{array}
$$

Specification of the hoisting transformation

$$\mathcal{C}_h(M) = N \text{ if } M \rightsquigarrow \cdots \rightsquigarrow N \not\rightsquigarrow, \quad \text{where:}$$

$$D ::= [\,] \mid \mathsf{let}\ id = B\ \mathsf{in}\ D \mid \mathsf{let}\ id = \lambda id^{+}.D\ \mathsf{in}\ M \mid \ell > D \qquad \text{(hoisting contexts)}$$

$$
\begin{array}{lll}
(h_1)\ D[\mathsf{let}\ x = C\ \mathsf{in}\ \mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ M] \rightsquigarrow & & \\
\quad D[\mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ \mathsf{let}\ x = C\ \mathsf{in}\ M] & \text{if } x \notin \mathsf{fv}(\lambda z^{+}.T) & \\
(h_2)\ D[\mathsf{let}\ x = \lambda w^{+}.\mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ M\ \mathsf{in}\ N] \rightsquigarrow & & \\
\quad D[\mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ \mathsf{let}\ x = \lambda w^{+}.M\ \mathsf{in}\ N] & \text{if } \{w^{+}\} \cap \mathsf{fv}(\lambda z^{+}.T) = \emptyset & \\
(h_3)\ D[\ell > \mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ M] \rightsquigarrow & & \\
\quad D[\mathsf{let}\ y = \lambda z^{+}.T\ \mathsf{in}\ \ell > M] & &
\end{array}
$$

Table7. Hoisting transformation

3 Reasoning on the cost annotations

We describe an initial labelling of the source code leading to a sound and precise labelling of the object code and an instrumentation of the labelled source program which produces a source program monitoring its own execution cost. Then, we explain how to obtain static guarantees on this execution cost by means of a Hoare logic for purely functional programs.

3.1 Initial labelling

We define a labelling function $\mathcal{L}$ of the source code (terms of the λ-calculus) which guarantees that the associated RTL code satisfies the conditions necessary for associating a cost with each label. We set $\mathcal{L}(M) = \mathcal{L}_0(M)$, where the functions $\mathcal{L}_i$ are specified in Table 8.

Proposition 10 (labelling properties). *Let M be a term of the λ-calculus and let $P \equiv \mathcal{C}(M)$ be its compilation.*

(1) *The function $\mathcal{L}$ is a labelling and produces well-labelled terms, namely:*

$$er(\mathcal{L}_i(M)) \equiv M \text{ and } \mathcal{L}_i(M) \in W_i \text{ for } i = 0, 1.$$

(2) *We have: $P \equiv er(\mathcal{C}(\mathcal{L}(M)))$.*

$$
\begin{array}{ll}
\mathcal{L}(M) & = \mathcal{L}_0(M) \quad \text{where:} \\
 & \\
\mathcal{L}_i(x) & = x \\
\mathcal{L}_i(\lambda id^+.M) & = \lambda id^+.\ell > \mathcal{L}_1(M) \quad \ell \text{ fresh} \\
\mathcal{L}_i((M_1,\ldots,M_n)) & = (\mathcal{L}_0(M_1),\ldots,\mathcal{L}_0(M_n)) \\
\mathcal{L}_i(\pi_i(M)) & = \pi_i(\mathcal{L}_0(M)) \\
\mathcal{L}_i(@(M,M^+)) & = \begin{cases} @(\mathcal{L}_0(M),(\mathcal{L}_0(M))^+) > \ell & i = 0, \ell \text{ fresh} \\ @(\mathcal{L}_0(M),(\mathcal{L}_0(M))^+) & i = 1 \end{cases} \\
\mathcal{L}_i(\mathsf{let}\ x = M\ \mathsf{in}\ N) & = \mathsf{let}\ x = \mathcal{L}_0(M)\ \mathsf{in}\ \mathcal{L}_i(N)
\end{array}
$$

Table8. A sound and precise labelling of the source code

(3) *Labels occur exactly once in the body of each function definition and nowhere else, namely, with reference to Table 7, P is generated by the following grammar:*

$$
\begin{array}{lll}
P & ::= & T \parallel \mathsf{let}\ id = \lambda id^+.Tlab\ \mathsf{in}\ P \\
Tlab & ::= & \ell > T \parallel \mathsf{let}\ id = C\ \mathsf{in}\ Tlab \\
T & ::= & @(id, id^+) \parallel \mathsf{let}\ id = C\ \mathsf{in}\ T
\end{array}
$$

The associated RTL program is composed of a set of routines which in turn is composed of a sequence of assignments on pseudo-registers and a terminal call to another routine. For such programs the back end of the moderately optimising compiler described in [1] produces assembly code which satisfies the checks outlined in the introduction.

3.2 Instrumentation

Given a cost monoid $\mathcal{M}$ with identity $\mathbf{1}$, we assume the analysis of the RTL code associates with each label ℓ an element m_ℓ of the cost monoid. This element is an upper bound on the cost of running the code starting from a control point labelled by ℓ and leading either to a control point without successors or to another labelled control point. Table 9 describes a monadic transformation which has been extensively analysed in [6] which instruments a program (in our case λ^ℓ) with the cost of executing its instructions. We are then back to a standard λ-calculus (without labels) which includes a basic data type to represent the cost monoid.

3.3 Higher-order Hoare Logic

Many proof systems can be used to obtain static guarantees on the evaluation of a purely functional program. In our setting, such systems can also be used to obtain static guarantees on the execution cost of a functional program by reasoning on its instrumentation.

We illustrate this point using an Hoare logic dedicated to call-by-value purely functional programs [11]. Given a well-typed program annotated by logic assertions, this system computes a set of proof obligations, whose validity ensures the

$$
\begin{array}{ll}
[\![x]\!] & = (\mathbf{1}, x) \\
[\![\lambda x^{+}.M]\!] & = (\mathbf{1}, \lambda x^{+}.[\![M]\!]) \\
[\![@(M_0, \ldots, M_n)]\!] & = \mathsf{let}\ (m_0, x_0) = [\![M_0]\!] \cdots (m_n, x_n) = [\![M_n]\!], \\
 & \quad (m_{n+1}, x_{n+1}) = @(x_0, \ldots, x_n)\ \mathsf{in} \\
 & \quad (m_{n+1} \cdot m_n \cdots m_0, x_{n+1}) \\
[\![(M_1, \ldots, M_n)]\!] & = \mathsf{let}\ (m_1, x_1) = [\![M_1]\!] \cdots (m_n, x_n) = [\![M_n]\!]\ \mathsf{in} \\
 & \quad (m_n \cdots m_1, (x_1, \ldots, x_n)) \\
[\![\pi_i(M)]\!] & = \mathsf{let}\ (m, x) = [\![M]\!]\ \mathsf{in}\ (m, \pi_i(x)) \\
[\![\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2]\!] & = \mathsf{let}\ (m_1, x) = [\![M_1]\!]\ \mathsf{in}\ (m_2, x_2) = [\![M_2]\!]\ \mathsf{in} \\
 & \quad (m_2 \cdot m_1, x_2) \\
[\![\ell > M]\!] & = \mathsf{let}\ (m, x) = [\![M]\!]\ \mathsf{in}\ (m \cdot m_\ell, x) \\
[\![M > \ell]\!] & = \mathsf{let}\ (m, x) = [\![M]\!]\ \mathsf{in}\ (m_\ell \cdot m, x)
\end{array}
$$

Table9. Instrumentation of labelled λ-calculus.

correctness of the logic assertions with respect to the evaluation of the functional program.

Logic assertions are written in a typed higher-order logic whose syntax is given in Table 10. From now on, we assume that our source language is also typed. The metavariable τ ranges over simple types, whose syntax is $\tau ::= \iota \mid \mid \tau \times \tau \mid \tau \to \tau$ where ι are the basic types including a data type cm for the values of the cost monoid. Types are lifted to the logical level through a logical reflection $\lceil \bullet \rceil$ defined in Table 10.

We write "$\mathsf{let}\ x : \tau / F = M\ \mathsf{in}\ M$" to annotate a let definition by a postcondition F of type $\lceil \tau \rceil \to \mathsf{prop}$. We write "$\lambda(x_1 : \tau_1)/F_1 : (x_2 : \tau_2)/F_2.\ M$" to ascribe to a λ-abstraction a precondition F_1 of type $\lceil \tau_1 \rceil \to \mathsf{prop}$ and a postcondition F_2 of type $\lceil \tau_1 \rceil \times \lceil \tau_2 \rceil \to \mathsf{prop}$. Computational values are lifted to the logical level using the reflection function defined in Table 10. The key idea of this definition is to reflect a computational function as a pair of predicates consisting in its precondition and its postcondition. Given a computational function f, a formula can refer to the precondition (resp. the postcondition) of f using the predicate $\mathsf{pre}\ f$ (resp. $\mathsf{post}\ f$). Thus, pre (resp. post) is a synonymous for π_1 (resp. π_2).

To improve the usability of our tool, we define in Table 10 a surface language by extending λ with several practical facilities. First, terms are explicitly typed. Therefore, the labelling $\mathcal{L}$ must be extended to convey type annotations in an explicitly typed version of λ^{ℓ}. The instrumentation $\mathcal{I}$ defined in Table 9 is extended to types by replacing each type annotation τ by its monadic interpretation $[\![\tau]\!]$ defined by $[\![\tau]\!] = \mathsf{cm} \times \overline{\tau}, \overline{\iota} = \iota, \overline{\tau_1 \times \tau_2} = ([\![\tau_1]\!] \times [\![\tau_2]\!])$ and $\overline{\tau_1 \to \tau_2} = \overline{\tau_1} \to [\![\tau_2]\!]$.

Second, since the instrumented version of a source program would be cumbersome to reason about because of the explicit threading of the cost value, we keep the program in its initial form while allowing logic assertions to implicitly refer to the instrumented version of the program. Thus, in the surface language, in the term "$\mathsf{let}\ x : \tau / F = M\ \mathsf{in}\ M$", F has type $\lceil [\![\tau]\!] \rceil \to \mathsf{prop}$, that is to say a predicate over pairs of which the first component is the execution cost.

Syntax

$$
\begin{array}{rcll}
F & ::= & \mathsf{True} \mid \mathsf{False} \mid x \mid F \wedge F \mid F = F \mid (F, F) & \text{(formulae)} \\
 & & \mid \pi_1 \mid \pi_2 \mid \lambda(x:\theta).F \mid F\,F \mid F \Rightarrow F \mid \forall(x:\theta).F & \\
\theta & ::= & \mathsf{prop} \mid \iota \mid \theta \times \theta \mid \theta \to \theta & \text{(types)} \\
V & ::= & id \mid \lambda(id:\tau)^+/F : (id:\tau)/F.M \mid (V^+) & \text{(values)} \\
M & ::= & V \mid @(M, M^+) \mid \mathsf{let}\ id : \tau/F = M\ \mathsf{in}\ M \mid (M^+) \mid \pi_i(M) & \text{(terms)}
\end{array}
$$

Logical reflection of types

$$
\begin{aligned}
\lceil \iota \rceil &= \iota \\
\lceil \tau_1 \times \ldots \times \tau_n \rceil &= \lceil \tau_1 \rceil \times \ldots \lceil \tau_n \rceil \\
\lceil \tau_1 \to \tau_2 \rceil &= (\lceil \tau_1 \rceil \to \mathsf{prop}) \times (\lceil \tau_1 \rceil \times \lceil \tau_2 \rceil \to \mathsf{prop})
\end{aligned}
$$

Logical reflection of values

$$
\begin{aligned}
\lceil id \rceil &= id \\
\lceil (V_1, \ldots, V_n) \rceil &= (\lceil V_1 \rceil, \ldots, \lceil V_n \rceil) \\
\lceil \lambda(x_1:\tau_1)/F_1 : (x_2:\tau_2)/F_2.\ M \rceil &= (F_1, F_2)
\end{aligned}
$$

Table10. The surface language.

Third, we allow labels to be written in source terms as a practical way of giving names to the labels introduced by the labelling $\mathcal{L}$. By that means, the constant cost assigned to a label ℓ can be symbolically used in specifications by writing $\mathsf{costof}(\ell)$.

Finally, as a convenience, we write "$x : \tau/F$" for "$x : \tau/\lambda(\mathrm{cost} : \mathsf{cm}, x : \lceil [\![\tau]\!] \rceil).F$". This improves the conciseness of specifications by automatically allowing reference to the cost variable in logic assertions without having to introduce it explicitly.

3.4 Prototype implementation

We implemented a prototype compiler [13] in OCaml ($\sim$ 3.5Kloc). This compiler accepts a program P written in the surface language extended with fixpoint and algebraic datatypes. Specifications are written in the Coq proof assistant [5]. A logic keyword is used to include logical definitions written in Coq to the source program.

Type checking is performed on P and, upon success, it produces a type annotated program P_t. Then, the labelled program $P_\ell = \mathcal{L}(P_t)$ is generated. Following the same treatment of branching as in our previous work on imperative programs [1], the labelling introduces a label at the beginning of each pattern matching branch.

By erasure of specifications and type annotations, we obtain a program P_λ of λ (Table 2). Using the compilation chain presented earlier, P_λ is compiled into a program P_h of $\lambda_{h,a}$ (Table 7) . The annotating compiler uses the cost model that consists in counting for each label ℓ the number of primitive operations that

belong to execution paths starting from ℓ (and ending in another label or in an instruction without successor).

Finally, the instrumented version of P_ℓ as well as the actual cost of each label is given as input to a verification condition generator to produce a set of proof obligations. These proof obligations are either proved automatically using first order theorem provers or manually in Coq.

3.5 Example

Let us consider an higher-order function *pexists* that looks for an integer x in a list l such that x validates a predicate p. In addition to the functional specification, we want to prove that the cost of this function is linear in the length n of the list l. The corresponding program written in the surface language can be found in Table 11.

A prelude declares the type and logical definitions used by the specifications. On lines 1 and 2, two type definitions introduce data constructors for lists and booleans. Between lines 4 and 5, a Coq definition introduces a predicate *bound* over the reflection of computational functions from nat to $nat \times bool$ that ensures that the cost of a computational function p is uniformly bounded by a constant k.

On line 9, the precondition of function *pexists* requires the function p to be total. Between lines 10 and 11, the postcondition first states a functional specification for *pexists*: the boolean result witnesses the existence of an element x of the input list l that is related to $BTrue$ by the postcondition of p. The second part of the postcondition characterizes the cost of *pexists* in case of a negative result: assuming that the cost of p is bounded by a constant k, the cost of *pexists* is proportional to $k.n$.

The verification condition generator produces 53 proof obligations out of this annotated program; 46 of these proof obligations are automatically discharged and 7 of them are manually proved in Coq.

4 Conclusion

We have shown that the so-called 'labelling' approach can be used to obtain certified execution costs on functional programs. In a realistic implementation of a functional programming language though, the runtime environment usually includes a garbage collector. The execution cost of such an automatic memory deallocation algorithm is *a priori* proportional to the size of the heap, which is not a sufficiently precise bound for practical use. An accurate static tracking of memory allocation, following region based or linear logic approaches, would be necessary to get relevant worst-case execution costs for memory deallocation.

Acknowledgements We are indebted to our Master students Guillaume Claret and David Giron for their implementation effort which provided valuable feedback. This work was supported by the *Information and Communication Technologies (ICT) Programme* as Project FP7-ICT-2009-C-243881 CerCo.

```
01 type list = Nil | Cons (nat, list)
02 type bool = BTrue | BFalse
03 logic {
04  Definition bound (p : nat ⟶ (nat × bool)) (k : nat) : Prop : =
05      ∀ x m : nat, ∀ r : bool, post p x (m, r) ⇒ m ≤ k.
06  Definition k0 : = costof(ℓ_m) + costof(ℓ_nil).
07  Definition k1 : = costof(ℓ_m) + costof(ℓ_p) + costof(ℓ_c) + costof(ℓ_f) + costof(ℓ_r).
08 }
09 let rec pexists (p : nat → bool, l : list) { ∀ x, pre p x } : bool {
10  ((result = BTrue) ⇔ (∃ x c : nat, mem x l ∧ post p x (c, BTrue))) ∧
11  (∀ k : nat, bound p k ∧ (result = BFalse) ⇒ cost ≤ k0 + (k + k1) × length (l))
12 } = ℓ_m> match l with
13    | Nil → ℓ_nil> BFalse
14    | Cons (x, xs) → ℓ_c> match p (x) > ℓ_p with
15                          | BTrue → BTrue
16                          | BFalse → ℓ_f> (pexists (p, xs) > ℓ_r)
```

Table11. An higher-order function and its specification.

References

1. R.M. Amadio, N. Ayache, Y. Régis-Gianas, R. Saillard. Certifying cost annotations in compilers. Université Paris Diderot, Research Report, `http://hal.archives-ouvertes.fr/hal-00524715/fr/`, 2010.
2. AbsInt Angewandte Informatik. `http://www.absint.com/`.
3. A. Bonenfant, C. Ferdinand, K. Hammond, R. Heckmann. Worst-case execution times for a purely functional language. In Proc. IFL, Springer LNCS 4449:235-252, 2006.
4. A. Chlipala. A verified compiler for an impure functional language. In Proc. ACM-POPL:93-106, 2010.
5. The Coq Development Team. The Coq Proof Assistant. INRIA-Rocquencourt, December 2001. `http://coq.inria.fr`.
6. D. Gurr. Semantic frameworks for complexity. PhD thesis, University of Edinburgh, 1991.
7. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107-115, 2009.
8. J. Morrisett, D. Walker, K. Crary, N. Glew. From system F to typed assembly language. ACM Trans. Program. Lang. Syst. 21(3): 527-568, 1999.
9. A. Perlis. Epigrams on programming. SIGPLAN Notices Vol. 17(9):7-13, 1982.
10. G. Plotkin. Call-by-name, Call-by-value and the lambda-Calculus. Theor. Comput. Sci. 1(2):125-159, 1975.
11. Y. Régis-Gianas, F. Pottier. A Hoare logic for call-by-value functional programs. In Proc. Mathematics of Program Construction, pp 305-335, 2008.
12. D. Sands. Complexity analysis for a lazy higher-order language. In Proc. ESOP, Springer LNCS 432:361-376, 1990.
13. Y. Régis-Gianas. An annotating compiler for MiniML. `http://www.pps.jussieu.fr/~yrg/fun-cca`.

A Examples

This section collects some examples.

Example 1 (labelling and commutation). Let $M \equiv \lambda x.xx > \ell$. Then $M \notin W_0$ because the rule for abstraction requires $xx > \ell \in W_1$ while we can only show $xx > \ell \in W_0$. Notice that we have:

$$\begin{array}{l} er(\mathcal{C}_{cps}(M)) \equiv @(halt, \lambda x, k.@(x, x, \lambda x.@(k, x))) \\ \mathcal{C}_{cps}(er(M)) \equiv @(halt, \lambda x, k.@(x, x, k)) \ . \end{array}$$

So for M the commutation of the cps-compilation and the erasure function only holds up to η.

Example 2 (CPS). Let $M \equiv @(\lambda x.@(x, @(x, x)), I)$, where $I \equiv \lambda x.x$. Then

$$\mathcal{C}_{cps}(M) \equiv @(\lambda x, k.@(x, x, \lambda y.@(x, y, k)), I', H))$$

where: $I' \equiv \lambda x, k.@(k, x)$ and $H \equiv \lambda x.@(halt, x)$. The term M is simulated by $\mathcal{C}_{cps}(M)$ as follows:

$$\begin{array}{ccccccc} M & \to & @(I, @(I, I)) & \to & @(I, I) & \to & I \\ \mathcal{C}_{cps}(M) & \to & @(I', I', \lambda y.@(I', y, H)) & \to^+ & @(I', I', H) & \to^+ & @(halt, I') \ . \end{array}$$

Example 3 (administrative form). Suppose $N \equiv @(\lambda x, k.@(x, x, \lambda y.@(x, y, k)), I', H))$ where: $I' \equiv \lambda x, k.@(k, x)$ and $H \equiv \lambda x.@(halt, x)$ (this is the term resulting from the CPS translation in example 2). The corresponding term in administrative form is:

$$\begin{array}{l} \mathsf{let}\ z_1 = \lambda x, k.\mathsf{let}\ z_2 = \lambda y.@(x, y, k)\ \mathsf{in}\ @(x, x, z_2)\ \mathsf{in} \\ \mathsf{let}\ z_3 = I'\ \mathsf{in} \\ \mathsf{let}\ z_4 = H\ \mathsf{in} \\ @(z_1, z_3, z_4) \ . \end{array}$$

Example 4 (closure conversion). Let $M \equiv \mathcal{C}_{ad}(\mathcal{C}_{cps}(\lambda x.y))$, namely

$$M \equiv \mathsf{let}\ z_1 = \lambda x, k.@(k, y)\ \mathsf{in}\ @(halt, z_1) \ .$$

Then $\mathcal{C}_{cc}(M)$ is the following term:

$$\begin{array}{l} \mathsf{let}\ z_2 = \lambda z, x, k.\mathsf{let}\ y = \pi_2(z)\ \mathsf{in}\ \mathsf{let}\ z = \pi_1(k)\ \mathsf{in}\ @(z, k, y)\ \mathsf{in} \\ \mathsf{let}\ z_1 = (z_2, y)\ \mathsf{in} \\ \mathsf{let}\ z = \pi_1(halt)\ \mathsf{in}\ @(z, halt, z_1) \ . \end{array}$$

Example 5 (hosting transformations and transitions). Let $M \equiv \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ @(x_1, z)$ where $N \equiv \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ T_1$ and $y_1 \notin \mathsf{fv}(\lambda y_2.T_2)$. Then we either reduce and then hoist:

$$\begin{array}{l} M \to \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ [z/y_1]N \\ \quad \equiv \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ [z/y_1]T_1 \\ \quad \rightsquigarrow \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ [z/y_1]T_1 \not\rightsquigarrow \end{array}$$

or hoist and then reduce:

$$\begin{aligned} M &\rightsquigarrow \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ @(x_1, z) \\ &\to \mathsf{let}\ x_2 = \lambda y_2.T_2\ \mathsf{in}\ \mathsf{let}\ x_1 = \lambda y_1.T_1\ \mathsf{in}\ [z/y_1]T_1 \quad \not\rightsquigarrow \end{aligned}$$

In the first case, we end up duplicating the definition of x_2.

Example 6 (labelling application). Let $M \equiv \lambda x.@(x, @(x, x))$. Then $\mathcal{L}(M) \equiv \lambda x.\ell_0 > @(x, @(x, x) > \ell_1)$. Notice that only the inner application is post-labelled.

B Proofs

This section collects the proofs of the results we have stated.

B.1 Proof of proposition 1 [CPS commutation]

The proof takes the following steps:

1. We remark that if V is a value in λ^ℓ and K a continuation in λ^ℓ_{cps} then so are $er(V)$ and $er(K)$. The proof is a direct induction on the structure of V and K, respectively.
2. For all values V and terms M of the λ^ℓ-calculus, we check that:

   $$er([V/x]M) \equiv [er(V)/x]er(M)\ .$$

   The proof proceeds by induction on the structure of M.
3. We notice that for all continuations K such that K is an abstraction, $\lambda x.(x : K) \equiv K$.
4. For all terms M and continuations K such that either $M \in W_0$ and K is an abstraction or $M \in W_1$ the following holds:

   $$er(M : K) \equiv er(M) : er(K)\ .$$

   We proceed by induction on M.

   x We expand the definition of $x : K$ depending on whether K is a variable or a function and we rely on step 2.

   $\lambda x^+.M$ We have $\lambda x^+.M \in W_1$ and $M \in W_1$. We analyse $\lambda x^+.M : K$ depending on whether K is a variable or a function and we apply the inductive hypothesis on M and step 2. Notice that it is essential that $M \in W_1$ to apply the inductive hypothesis.

   $@(M_0, \ldots, M_n)$ We know $M_0, \ldots, M_n \in W_0$. We apply the inductive hypothesis on $M_n, \ldots, M_0$ to conclude that:

   $$\begin{aligned} & er(@(M_0, \ldots, M_n)) : er(K) \\ &\equiv er(M_0) : \lambda x_0. \ldots er(M_n) : \lambda x_n.@(x_0, \ldots, x_n, er(K)) \\ &\equiv er(M_0) : \lambda x_0. \ldots er(M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\ &\equiv \cdots \\ &\equiv er(M_0 : \lambda x_0. \ldots M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\ &\equiv er(@(M_0, \ldots, M_N) : K)\ . \end{aligned}$$

$\ell > M$ We know that if $\ell > M \in W_i$ then $M \in W_i$ and we apply the inductive hypothesis on M.

$M > \ell$ By definition, we must have $M > \ell \in W_0$. Hence K is a function and $M \in W_0$. Then we apply the inductive hypothesis on M and step 3.

$(M_1, \ldots, M_n)$ We know that $M_i \in W_0$ for $i = 1, \ldots, n$. First we notice that:

$$er(\lambda x_n.(x_1, \ldots, x_n) : K) \equiv \lambda x_n.(x_1, \ldots, x_n) : er(K) \ .$$

Then we apply the inductive hypothesis on $M_n, \ldots, M_0$ to conclude that:

$$\begin{array}{l} er((M_1, \ldots, M_n)) : er(K) \\ \equiv er(M_1) : \lambda x_1 \ldots er(M_n) : \lambda x_n.(x_1, \ldots, x_n) : er(K) \\ \equiv er(M_1) : \lambda x_1 \ldots er(M_n) : er(\lambda x_n.(x_1, \ldots, x_n) : K) \\ \equiv er(M_1) : \lambda x_1 \ldots er(M_n : \lambda x_n.(x_1, \ldots, x_n) : K) \\ \equiv \cdots \\ \equiv er(M_1 : \lambda x_1 \ldots M_n : \lambda x_n.(x_1, \ldots, x_n) : K) \\ \equiv er((M_1, \ldots, M_n) : K) \ . \end{array}$$

$\pi_i(M)$ We know $M \in W_0$. We observe that $er(y : K) \equiv y : er(K)$. Then we apply the inductive hypothesis on M to conclude that:

$$\begin{array}{l} er(\pi_i(M)) : er(K) \\ \equiv \pi_i(er(M)) : er(K) \\ \equiv er(M) : \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : er(K) \\ \equiv er(M) : er(\lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K) \\ \equiv er(M : \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K) \\ \equiv er(\pi_i(M) : K) \ . \end{array}$$

$\mathsf{let}\ x = N\ \mathsf{in}\ M$ If $\mathsf{let}\ x = N\ \mathsf{in}\ M \in W_i$ then we know $N \in W_0$ and $M \in W_i$. We apply the inductive hypothesis on N and M to conclude that:

$$\begin{array}{l} er(\mathsf{let}\ x = N\ \mathsf{in}\ M : K) \\ \equiv er(N : \lambda x.(M : K)) \\ \equiv er(N) : \lambda x.er(M : K) \\ \equiv er(N) : \lambda x.er(M) : er(K) \\ \equiv er(\mathsf{let}\ x = N\ \mathsf{in}\ M) : er(K) \ . \end{array}$$

5. Then we prove the assertion for $M \in W_0$ as follows:

$$\begin{array}{rll} er(\mathcal{C}_{cps}(M)) & \equiv er(M : \lambda x.@(halt, x)) & \text{(by definition)} \\ & \equiv er(M) : \lambda x.@(halt, x) & \text{(by point 4)} \\ & \equiv \mathcal{C}_{cps}(er(M)) & \text{(by definition).} \end{array}$$

□

B.2 Proof of proposition 2 [CPS simulation]

The proof takes the following steps.

1. We show that for all values V, terms M, and continuations $K \neq x$:

$$[V/x]M : [\psi(V)/x]K \equiv [\psi(V)/x](M : K) \ .$$

We proceed by induction on M.

variable By case analysis: $M \equiv x$ or $M \equiv y \neq x$.

$\lambda z^+.M$ By case analysis on K which is either a variable or a function. We develop the second case with $K = \lambda y.N$. We observe:

$$\begin{array}{l} [V/x](\lambda z^+.M) : [\psi(V)/x]K \\ \equiv [\lambda z^+, k.([V/x]M : k)/y][\psi(V)/x]N \\ \equiv [\lambda z^+, k.[\psi(V)/x](M : k)/y][\psi(V)/x]N \\ \equiv [\psi(V)/x][\lambda z^+, k.(M : k)/y]N \\ \equiv [\psi(V)/x]((\lambda z^+.M) : K) \ . \end{array}$$

$@(M_0, \ldots, M_n)$ We apply the inductive hypothesis on $M_0, \ldots, M_n$ as follows:

$$\begin{array}{l} [\psi(V)/x](@(M_0, \ldots, M_n) : K) \\ \equiv [\psi(V)/x](M_0 : \lambda x_0 \ldots M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\ \ldots \\ \equiv [V/x]M_0 : \lambda x_0 \ldots [\psi(V)/x](M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\ \equiv [V/x]M_0 : \lambda x_0 \ldots [V/x]M_n : \lambda x_n.@(x_0, \ldots, x_n, [\psi(V)/x]K) \\ \equiv [V/x]@(M_0, \ldots, M_n) : [\psi(V)/x]K \ . \end{array}$$

Note that in this case the substitution $[\psi(V)/x]$ may operate on the continuation. The remaining cases (pairing, projection, let, pre and post labelling) follow a similar pattern and are omitted.

2. The evaluation contexts for the λ^ℓ-calculus described in table 2 can also be specified 'bottom up' as follows:

$$\begin{array}{rl} E ::= & [\ ] \mid E[@(V^*, [\ ], M^*)] \mid E[\mathsf{let}\ id = [\ ]\ \mathsf{in}\ M] \mid E[(V^*, [\ ], M^*)] \mid \\ & E[\pi_i([\ ])] \mid E[[\ ] > \ell] \ . \end{array}$$

Following this specification, we associate a continuation K_E with an evalution context as follows:

$$\begin{array}{ll} K_{[\ ]} & = \lambda x.@(halt, x) \\ K_{E[@(V^*,[\ ],M^*)]} & = \lambda x.M^* : \lambda y^*.@(\psi(V)^*, x, y^*, K_E) \\ K_{E[\mathsf{let}\ x=[\ ]\ \mathsf{in}\ N]} & = \lambda x.N : K_E \\ K_{E[(V^*,[\ ],M^*)]} & = \lambda x.M^* : \lambda y^*.(\psi(V)^*, x, y^*) : K_E \\ K_{E[\pi_i([\ ])} & = \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K_E \\ K_{E[[\ ]>\ell]} & = \lambda x.\ell > x : K_E \end{array}$$

where $M^* : \lambda x^*.N$ stands for $M_0 : \lambda x_0 \ldots M_n : \lambda x_n.N$ with $n \geq 0$.

3. For all terms M and evaluation contexts E, E' we prove by induction on the evaluation context E that the following holds:

$$E[M] : K_{E'} \equiv M : K_{E'[E]} \ .$$

For instance we detail the case the context has the shape $E[@(V^*, [\,], M^*)$.

$$\begin{array}{ll} & E[@(V^*, [M], M^*) : K_{E'} \\ \equiv & @(V^*, [M], M^*) : K_{E'[E]} \qquad \text{(by inductive hypothesis)} \\ \equiv & M : \lambda x.M^* : \lambda x^*.@(\psi(V)^*, x, x^*, K_{E'[E]}) \\ \equiv & M : K_{E'[E[@(V^*,[\,],M^*)]]} \ . \end{array}$$

4. For all terms M, continuations K, K', and variable $x \notin \mathsf{fv}(M)$ we prove by induction on M and case analysis that the following holds:

$$[K/x](M : K') \begin{cases} \to M : K' & \text{if } K \text{ abstraction}, M \text{ value}, K' = x \\ \equiv (M : [K/x]K') & \text{otherwise.} \end{cases}$$

5. Finally, we prove the assertion by proceeding by case analysis on the reduction rule.
   - $E[@(\lambda x^+.M, V^+)] \to E[[V^+/x^+]M]$. We have:

$$\begin{array}{ll} & E[@(\lambda x^+.M, V^+)] : K_{[\,]} \\ \equiv & @(\lambda x^+.M, V^+) : K_E \\ \equiv & @(\lambda x^+, k.M : k, \psi(V)^+, K_E) \\ \to & [K_E/k, \psi(V)^+/x^+](M : k) \\ \equiv & [K_E/k]([V/x]M : k) \\ \overset{*}{\to} & [V/x]M : K_E \\ \equiv & E[[V/x]M] : K_{[\,]} \ . \end{array}$$

   - $E[\mathsf{let}\ x = V\ \mathsf{in}\ M] \to E[[V/x]M]$. We have:

$$\begin{array}{ll} & E[\mathsf{let}\ x = V\ \mathsf{in}\ M] : K_{[\,]} \\ \equiv & \mathsf{let}\ x = V\ \mathsf{in}\ M : K_E \\ \equiv & V : \lambda x.(M : K_E) \\ \equiv & [\psi(V)/x](M : K_E) \\ \equiv & [V/x]M : K_E \\ \equiv & E[[V/x]M] : K_{[\,]} \ . \end{array}$$

   - $E[\pi_i(V)] \to E[V_i]$, where $V \equiv (V_1, \ldots, V_n)$ and $1 \le i \le n$. We have:

$$\begin{array}{ll} & E[\pi_i(V)] : K_{[\,]} \\ \equiv & \pi_i(V) : K_E \\ \equiv & V : \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K_E \\ \equiv & \mathsf{let}\ y = \pi_i(\psi(V_1), \ldots, \psi(V_n))\ \mathsf{in}\ y : K_E \\ \to & [\psi(V_i)/y](y : K_E) \\ \equiv & V_i : K_E \\ \equiv & E[V_i] : K_{[\,]} \ . \end{array}$$

   - $E[\ell > M] \xrightarrow{\ell} E[M]$. We have:

$$\begin{array}{ll} & E[\ell > M] : K_{[\,]} \\ \equiv & \ell > M : K_E \\ \equiv & \ell > (M : K_E) \\ \xrightarrow{\ell} & (M : K_E) \\ \equiv & E[M] : K_{[\,]} \ . \end{array}$$

– $E[V > \ell] \xrightarrow{\ell} E[V]$. We have:

$$\begin{array}{l} E[V > \ell] : K_{[\,]} \\ \equiv V > \ell : K_E \\ \equiv V : \lambda x.\ell > x : K_E \\ \equiv \ell > (V : K_E) \\ \xrightarrow{\ell} V : K_E \\ \equiv E[V] : K_{[\,]} \ . \end{array}$$

□

B.3 Proof of proposition 3 [AD commutation]

(1) We show that for every P which is either a term or a value of the λ^{ℓ}_{cps}-calculus the following properties hold:

A If P is a term then $\mathcal{R}(\mathcal{C}_{ad}(P)) \equiv P$.
B If P is a value then for any term N, $\mathcal{R}(\mathcal{E}_{ad}(P, x)[N]) \equiv [P/x]\mathcal{R}(N)$.

We prove the two properties at once by induction on the structure of P.

$@(x, x^+)$ We are in case A and by definition we have:

$$\mathcal{R}(\mathcal{C}_{ad}(@(x, x^+))) \equiv \mathcal{R}(@(x, x^+)) \equiv @(x, x^+) \ .$$

$@(x^*, V, V^*), V \neq id$ Again in case A. We have:

$$\begin{array}{ll} \mathcal{R}(\mathcal{C}_{ad}(@(x^*, V, V^*))) & \\ \equiv \mathcal{R}(\mathcal{E}_{ad}(V, y)[\mathcal{C}_{ad}(@(x^*, y, V^*))]) & \\ \equiv [V/y]\mathcal{R}(\mathcal{C}_{ad}(@(x^*, y, V^*))) & \text{(by ind. hyp. on B)} \\ \equiv [V/y]@(x^*, y, V^*) & \text{(by ind. hyp. on A)} \\ \equiv @(x^*, V, V^*) \ . & \end{array}$$

$\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M$ Again in case A. We have:

$$\begin{array}{ll} \mathcal{R}(\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M)) & \\ \equiv \mathcal{R}(\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{C}_{ad}(M)) & \\ \equiv \mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{R}(\mathcal{C}_{ad}(M)) & \\ \equiv \mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M & \text{(by ind. hyp. on A)} \ . \end{array}$$

$\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M, V \neq id$ Again in case A. We have:

$$\begin{array}{ll} \mathcal{R}(\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M)) & \\ \equiv \mathcal{R}(\mathcal{E}_{ad}(V, y)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{ad}(M)]) & \\ \equiv [V/y]\mathcal{R}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{ad}(M)) & \text{(by ind. hyp. on B)} \\ \equiv [V/y]\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{R}(\mathcal{C}_{ad}(M)) & \\ \equiv [V/y]\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M & \text{(by ind. hyp. on A)} \\ \equiv \mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M \ . & \end{array}$$

$\ell > M$ Last case for A. We have:

$$\begin{array}{ll} & \mathcal{R}(\mathcal{C}_{ad}(\ell > M)) \\ \equiv & \mathcal{R}(\ell > \mathcal{C}_{ad}(M)) \\ \equiv & \ell > \mathcal{R}(\mathcal{C}_{ad}(M)) \\ \equiv & \ell > M \qquad \text{(by ind. hyp. on A)} \ . \end{array}$$

$\lambda y^+.M$ We now turn to case B. We have:

$$\begin{array}{ll} & \mathcal{R}(\mathcal{E}_{ad}(\lambda y^+.M, x)[N]) \\ \equiv & \mathcal{R}(\mathsf{let}\ x = \lambda y^+.\mathcal{C}_{ad}(M)\ \mathsf{in}\ N) \\ \equiv & [\mathcal{R}(\lambda y^+.\mathcal{C}_{ad}(M))/x]\mathcal{R}(N) \\ \equiv & [\lambda y^+.\mathcal{R}(\mathcal{C}_{ad}(M))/x]\mathcal{R}(N) \\ \equiv & [\lambda y^+.M/x]\mathcal{R}(N) \qquad \text{(by ind. hyp. on A)} \ . \end{array}$$

(y^+) Again in case B. We have:

$$\begin{array}{ll} & \mathcal{R}(\mathcal{E}_{ad}((y^+), x)[N]) \\ \equiv & \mathcal{R}(\mathsf{let}\ x = (y^+)\ \mathsf{in}\ N) \\ \equiv & [(y^+)/x]\mathcal{R}(N) \ . \end{array}$$

$(y^*, V, V^*), V \neq id$ Last case for B. We have:

$$\begin{array}{ll} & \mathcal{R}(\mathcal{E}_{ad}((y^*, V, V^*), x)[N]) \\ \equiv & \mathcal{R}(\mathcal{E}_{ad}(V, z)[\mathcal{E}_{ad}((y^*, z, V^*), x)[N]]) \\ \equiv & [V/z]\mathcal{R}(\mathcal{E}_{ad}((y^*, z, V^*), x)[N]) \qquad \text{(by ind. hyp. on B)} \\ \equiv & [V/z]([(y^*, z, V^*)/x]\mathcal{R}(N)) \qquad \text{(by ind. hyp. on B)} \\ \equiv & [(y^*, V, V^*)/x]\mathcal{R}(N) \ . \end{array}$$

(2) The proof is similar to the previous one. We show that for every P which is either a term or a value of the λ^{ℓ}_{cps}-calculus the following properties hold:

A If P is a term then $er(\mathcal{C}_{ad}(P)) \equiv \mathcal{C}_{ad}(er(P))$.
B If P is a value then for any term N, $er(\mathcal{E}_{ad}(P, x)[N]) \equiv \mathcal{E}_{ad}(er(P), x)[er(N)]$.

We prove the two properties at once by induction on the structure of P.

$@(x, x^+)$ We are in case A and by definition we have:

$$er(\mathcal{C}_{ad}(@(x, x^+))) \equiv er(@(x, x^+)) \equiv @(x, x^+) \equiv \mathcal{C}_{ad}(er(@(x, x^+))) \ .$$

$@(x^*, V, V^*), V \neq id$ Again in case A. We have:

$$\begin{array}{ll} & er(\mathcal{C}_{ad}(@(x^*, V, V^*))) \\ \equiv & er(\mathcal{E}_{ad}(V, y)[\mathcal{C}_{ad}(@(x^*, y, V^*))]) \\ \equiv & \mathcal{E}_{ad}(er(V), y)[er(\mathcal{C}_{ad}(@(x^*, y, V^*)))] \text{ (by ind. hyp. on B)} \\ \equiv & \mathcal{E}_{ad}(er(V), y)[\mathcal{C}_{ad}(er(@(x^*, y, V^*)))] \text{ (by ind. hyp. on A)} \\ \equiv & \mathcal{C}_{ad}(er(@(x^*, V, V^*))) \ . \end{array}$$

$\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M$ Again in case A. We have:

$$\begin{array}{l} er(\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M)) \\ \equiv er(\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{C}_{ad}(M)) \\ \equiv \mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ er(\mathcal{C}_{ad}(M)) \\ \equiv \mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{C}_{ad}(er(M)) \quad \text{(by ind. hyp. on A)} \\ \equiv \mathcal{C}_{ad}(er(\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ M))\ . \end{array}$$

$\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M, V \neq id$ Again in case A. We have:

$$\begin{array}{l} er(\mathcal{C}_{ad}(\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M)) \\ \equiv er(\mathcal{E}_{ad}(V, z)[\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{C}_{ad}(M)]) \\ \equiv \mathcal{E}_{ad}(er(V), z)[\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ er(\mathcal{C}_{ad}(M))] \text{ (by ind. hyp. on B)} \\ \equiv \mathcal{E}_{ad}(er(V), z)[\mathsf{let}\ x = \pi_i(z)\ \mathsf{in}\ \mathcal{C}_{ad}(er(M))] \text{ (by ind. hyp. on A)} \\ \equiv \mathcal{C}_{ad}(er(\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M))\ . \end{array}$$

$\ell > M$ Last case for A. We have:

$$\begin{array}{l} er(\mathcal{C}_{ad}(\ell > M)) \\ \equiv er(\ell > \mathcal{C}_{ad}(M)) \\ \equiv er(\mathcal{C}_{ad}(M)) \\ \equiv \mathcal{C}_{ad}(er(M)) \qquad \text{(by ind. hyp. on A)} \\ \equiv \mathcal{C}_{ad}(er(\ell > M))\ . \end{array}$$

$\lambda y^+.M$ We now turn to case B. We have:

$$\begin{array}{l} er(\mathcal{E}_{ad}(\lambda y^+.M, x)[N]) \\ \equiv er(\mathsf{let}\ x = \lambda y^+.\mathcal{C}_{ad}(M)\ \mathsf{in}\ N) \\ \equiv \mathsf{let}\ x = \lambda y^+.er(\mathcal{C}_{ad}(M))\ \mathsf{in}\ er(N) \\ \equiv \mathsf{let}\ x = \lambda y^+.\mathcal{C}_{ad}(er(M))\ \mathsf{in}\ er(N) \text{ (by ind. hyp. on A)} \\ \equiv \mathcal{E}_{ad}(er(\lambda y^+.M), x)[er(N)]\ . \end{array}$$

(y^+) Again in case B. We have:

$$\begin{array}{l} er(\mathcal{E}_{ad}((y^+), x)[N]) \\ \equiv er(\mathsf{let}\ x = (y^+)\ \mathsf{in}\ N) \\ \equiv \mathsf{let}\ x = (y^+)\ \mathsf{in}\ er(N) \\ \equiv \mathcal{E}_{ad}(er((y^+)), x)[er(N)]\ . \end{array}$$

$(y^*, V, V^*), V \neq id$ Last case for B. We have:

$$\begin{array}{l} er(\mathcal{E}_{ad}((y^*, V, V^*), x)[N]) \\ \equiv er(\mathcal{E}_{ad}(V, z)[\mathcal{E}_{ad}((y^*, z, V^*), x)[N]]) \\ \equiv \mathcal{E}_{ad}(er(V), x)[er(\mathcal{E}_{ad}((y^*, z, V^*), x)[N])] \qquad \text{(by ind. hyp. on B)} \\ \equiv \mathcal{E}_{ad}(er(V), x)[\mathcal{E}_{ad}(er((y^*, z, V^*)), x)[er(N)]] \text{ (by ind. hyp. on B)} \\ \equiv \mathcal{E}_{ad}(er((y^*, V, V^*)), x)[er(N)]\ . \end{array}$$

□

B.4 Proof of proposition 4 [AD simulation]

First we fix some notation. We associate a substitution σ_E with an evaluation context E of the $\lambda^{\ell}_{cps,a}$-calculus as follows:

$$\sigma_{[\ ]} = Id \ \sigma_{\mathsf{let}\ x=V\ \mathsf{in}\ E} = [\mathcal{R}(V)/x] \circ \sigma_E \ .$$

Then we prove the property by case analysis.

- If $\mathcal{R}(N) \equiv @(\lambda y^+.M, V^+) \to [V^+/y^+]M$ then $N \equiv E[@(x, x^+)]$, $\sigma_E(x) \equiv \lambda y^+.M$, and $\sigma_E(x^+) \equiv V^+$.
  Moreover, $E \equiv E_1[\mathsf{let}\ x = \lambda y^+.M'\ \mathsf{in}\ E_2]$ and $\sigma_{E_1}(\lambda y^+.M') \equiv \lambda y^+.M$.
  Therefore, $N \to E[[x^+/y^+]M'] \equiv N'$ and we check that $\mathcal{R}(N') \equiv \sigma_E([x^+/y^+]M') \equiv [V^+/y^+]M$.
- If $\mathcal{R}(N) \equiv \mathsf{let}\ x = \pi_i((V_1, \ldots, V_n))\ \mathsf{in}\ M \to [V_i/x]M$ then $N \equiv E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ N'']$, $\sigma_E(y) \equiv (V_1, \ldots, V_n)$, and $\sigma_E(N'') \equiv M$.
  Moreover, $E \equiv E_1[\mathsf{let}\ y = (z_1, \ldots, z_n)\ \mathsf{in}\ E_2]$ and $\sigma_{E_1}(z_1, \ldots, z_n) \equiv (V_1, \ldots, V_n)$.
  Therefore, $N \to E[[z_i/x]N''] \equiv N'$ and we check that $\mathcal{R}(N') \equiv \sigma_E([z_i/x]N'') \equiv [V_i/x]M$.
- If $\mathcal{R}(N) \equiv \ell > M \xrightarrow{\ell} M$ then $N \equiv E[\ell > N'']$ and $\sigma_E(N'') \equiv M$. We conclude by observing that $N \xrightarrow{\ell} E[N'']$. □

B.5 Proof of proposition 5 [CC commutation]

This is a simple induction on the structure of the term M.

$@(x, y^+)$ We have:

$$\begin{aligned}
&er(\mathcal{C}_{cc}(@(x, y^+))) \\
&\equiv er(\mathsf{let}\ z = \pi_1(x)\ \mathsf{in}\ @(z, x, y^+)) \\
&\equiv \mathsf{let}\ z = \pi_1(x)\ \mathsf{in}\ @(z, x, y^+) \\
&\equiv \mathcal{C}_{cc}(@(x, y^+)) \\
&\equiv er(\mathcal{C}_{cc}(@(x, y^+))) \ .
\end{aligned}$$

$\mathsf{let}\ x = B\ \mathsf{in}\ M$**,** B **not a function** We have:

$$\begin{aligned}
&er(\mathcal{C}_{cc}(\mathsf{let}\ x = B\ \mathsf{in}\ M)) \\
&\equiv er(\mathsf{let}\ x = B\ \mathsf{in}\ \mathcal{C}_{cc}(M)) \\
&\equiv \mathsf{let}\ x = B\ \mathsf{in}\ er(\mathcal{C}_{cc}(M)) \\
&\equiv \mathsf{let}\ x = B\ \mathsf{in}\ \mathcal{C}_{cc}(er(M)) \quad \text{(by ind. hyp.)} \\
&\equiv \mathcal{C}_{cc}(er(\mathsf{let}\ x = B\ \mathsf{in}\ M)) \ .
\end{aligned}$$

$\mathsf{let}\ x = \lambda x^+.N\ \mathsf{in}\ M, \mathsf{fv}(\lambda x^+.N) = \{z_1, \ldots, z_k\}$ We have:

$$\begin{aligned}
&er(\mathcal{C}_{cc}(\mathsf{let}\ x = \lambda x^+.N\ \mathsf{in}\ M)) \\
&\equiv er(\ \mathsf{let}\ y = \lambda z, x^+.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\
&\qquad \mathsf{let}\ x = (y, z_1, \ldots, z_k)\ \mathsf{in}\ \mathcal{C}_{cc}(M)\ ) \\
&\equiv \mathsf{let}\ y = \lambda z, x^+.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ er(\mathcal{C}_{cc}(N))\ \mathsf{in} \\
&\qquad \mathsf{let}\ x = (y, z_1, \ldots, z_k)\ \mathsf{in}\ er(\mathcal{C}_{cc}(M)) \\
&\equiv \mathsf{let}\ y = \lambda z, x^+.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ \mathcal{C}_{cc}(er(N))\ \mathsf{in} \\
&\qquad \mathsf{let}\ x = (y, z_1, \ldots, z_k)\ \mathsf{in}\ \mathcal{C}_{cc}(er(M)) \qquad \text{(by ind. hyp.)} \\
&\equiv \mathcal{C}_{cc}(er(\mathsf{let}\ x = \lambda x^+.N\ \mathsf{in}\ M)) \ .
\end{aligned}$$

$\ell > M$ We have:

$$\begin{array}{l} er(\mathcal{C}_{cc}(\ell > M)) \\ \equiv er(\ell > \mathcal{C}_{cc}(M)) \\ \equiv er(\mathcal{C}_{cc}(M)) \\ \equiv \mathcal{C}_{cc}(er(M)) \qquad \text{(by ind. hyp.)} \\ \equiv \mathcal{C}_{cc}(er(\ell > M)) \ . \end{array}$$

□

B.6 Proof of proposition 6 [CC simulation]

As a first step we check that the closure conversion function commutes with name substitution:

$$\mathcal{C}_{cc}([x/y]M) \equiv [x/y]\mathcal{C}_{cc}(M) \ .$$

This is a direct induction on the structure of the term M. Then we extend the closure conversion function to contexts as follows:

$$\begin{array}{ll} \mathcal{C}_{cc}([\,]) & = [\,] \\ \mathcal{C}_{cc}(\mathsf{let}\ x = (y^+)\ \mathsf{in}\ E) & = \mathsf{let}\ x = (y^+)\ \mathsf{in}\ \mathcal{C}_{cc}(E) \\ \mathcal{C}_{cc}(\mathsf{let}\ x = \lambda x^+.M\ \mathsf{in}\ E) & = \mathsf{let}\ y = \lambda z, x^+.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ \mathcal{C}_{cc}(M)\ \mathsf{in} \\ & \quad \mathsf{let}\ x = (y, z_1, \ldots, z_k)\ \mathsf{in}\ \mathcal{C}_{cc}(E) \\ & \text{where: } \mathsf{fv}(\lambda x^+.M) = \{z_1, \ldots, z_k\} \ . \end{array}$$

We note that for any evaluation context E, $\mathcal{C}_{cc}(E)$ is again an evaluation context, and moreover for any term M we have:

$$\mathcal{C}_{cc}(E[M]) \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}(M)] \ .$$

Finally we prove the simulation property by case analysis of the reduction rule being applied.

- Suppose $M \equiv E[@(x, y^+)] \to E[[y^+/x^+]M]$ where $E(x) = \lambda x^+.M$. Then:

  $$\mathcal{C}_{cc}(E[@(x, y)]) \equiv \mathcal{C}_{cc}(E)[\mathsf{let}\ z = \pi_1(z)\ \mathsf{in}\ @(z, x, y^+)]$$

  with $\mathcal{C}_{cc}(E)(x) = (y, z_1, \ldots, z_k)$ and
  $\mathcal{C}_{cc}(E)(y) = \lambda z, x^+.\mathsf{let}\ z_1 = \pi_2(z), \ldots, z_k = \pi_{k+1}(z)\ \mathsf{in}\ \mathcal{C}_{cc}(M)$. Therefore:

  $$\begin{array}{l} \mathcal{C}_{cc}(E)[\mathsf{let}\ z = \pi_1(z)\ \mathsf{in}\ @(z, x, y^+)] \\ \to \mathcal{C}_{cc}(E)[@(y, x, y^+)] \\ \to \mathcal{C}_{cc}(E)[\mathsf{let}\ z_1 = \pi_2(x), \ldots, z_k = \pi_{k+1}(x)\ \mathsf{in}\ [y^+/x^+]\mathcal{C}_{cc}(M)] \\ \xrightarrow{*} \mathcal{C}_{cc}(E)[[y^+/x^+]\mathcal{C}_{cc}(M)] \\ \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}([y^+/x^+]M)] \qquad \text{(by substitution commutation)} \\ \equiv \mathcal{C}_{cc}(E[[y^+/x^+]M]) \ . \end{array}$$

- Suppose $M \equiv E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M] \to E[[z_i/x]M]$ where $E(y) = (z_1, \ldots, z_k)$, $1 \leq i \leq k$. Then:

  $$\mathcal{C}_{cc}(E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M]) \equiv \mathcal{C}_{cc}(E)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{cc}(M)]$$

with $\mathcal{C}_{cc}(E)(y) = (z_1, \ldots, z_k)$. Therefore:

$$\begin{array}{l}\mathcal{C}_{cc}(E)[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ \mathcal{C}_{cc}(M)] \\ \to \mathcal{C}_{cc}(E)[[z_i/x]\mathcal{C}_{cc}(M)] \\ \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}([z_i/x]M)] \qquad \text{(by substitution commutation)} \\ \equiv \mathcal{C}_{cc}(E[[z_i/x]M])\ .\end{array}$$

– Suppose $M \equiv E[\ell > M] \xrightarrow{\ell} E[M]$. Then:

$$\begin{array}{l}\mathcal{C}_{cc}(E[\ell > M]) \\ \equiv \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}(\ell > M)] \\ \equiv \mathcal{C}_{cc}(E)[\ell > \mathcal{C}_{cc}(M)] \\ \xrightarrow{\ell} \mathcal{C}_{cc}(E)[\mathcal{C}_{cc}(M)] \\ \equiv \mathcal{C}_{cc}(E[M])\ .\end{array}$$

□

B.7 Proof of proposition 7 [on hoisting transformations]

As a preliminary remark, note that the hoisting contexts D can be defined in an equivalent way as follows:

$$D ::= [\ ] \mid D[\mathsf{let}\ x = B\ \mathsf{in}\ [\ ]] \mid D[\mathsf{let}\ x = \lambda y^+.[\ ]\ \mathsf{in}\ M] \mid D[\ell > [\ ]]$$

If D is a hoisting context and x is a variable we define $D(x)$ as follows:

$$D(x) = \begin{cases} \lambda z^+.T & \text{if } D = D'[\mathsf{let}\ x = \lambda z^+.T\ \mathsf{in}\ [\ ]] \\ D'(x) & \text{o.w. if } D = D'[\mathsf{let}\ y = B\ \mathsf{in}\ [\ ]], x \neq y \\ D'(x) & \text{o.w. if } D = D'[\mathsf{let}\ y = \lambda y^+.[\ ]\ \mathsf{in}\ M], x \notin \{y^+\} \\ \text{undefined} & \text{o.w.} \end{cases}$$

The intuition is that $D(x)$ checks whether D binds x to a simple function $\lambda z^+.T$. If this is the case it returns the simple function as a result, otherwise the result is undefined.

Let I be the set of terms of the $\lambda^{\ell}_{cps,a}$ such that if $M \equiv D[\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ N]$ and $z \in \mathsf{fv}(\lambda y^+.T)$ then $D(z) = \lambda z^+.T'$. Thus a name free in a simple function must be bound to another simple function. We prove the following properties:

1. The hoisting transformations terminate.
2. The hoisting transformations are confluent (hence the result of the hoisting transformations is unique).
3. If a term M of the $\lambda^{\ell}_{cps,a}$-calculus contains a function definition then $M \equiv D[\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ N]$ for some D, T, N.
4. All terms in $\lambda^{\ell}_{cc,a}$ belong to the set I (trivially).
5. The set I is an invariant of the hoisting transformations, *i.e.*, if $M \in I$ and $M \rightsquigarrow N$ then $N \in I$.
6. If a term satisfying the invariant above is not a program then a hoisting transformation applies.

(1) To prove the termination of the hoisting transformations we introduce a size function from terms to positive natural numbers as follows:

$$\begin{array}{ll} |@(x, x^+)| & = 1 \\ |\mathsf{let}\ x = \lambda y^+.M\ \mathsf{in}\ N| & = 2 \cdot |M| + |N| \\ |\mathsf{let}\ x = C\ \mathsf{in}\ N| & = 2 \cdot |N| \\ |\ell > N| & = 2 \cdot |N| \ . \end{array}$$

Then we check that if $M \rightsquigarrow N$ then $|M| > |N|$. Note that the hoisting context D induces a function which is strictly monotone on natural numbers. Thus it is enough to check that the size of the redex term is larger than the size of the reduced term.

(h_1)

$$\begin{array}{l} |\mathsf{let}\ x = C\ \mathsf{in}\ \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M| \\ = 2 \cdot (2 \cdot |T| + |M|) \\ > 2 \cdot |T| + 2 \cdot |M| \\ = |\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = C\ \mathsf{in}\ M| \ . \end{array}$$

(h_2)

$$\begin{array}{l} |\mathsf{let}\ x = \lambda w^+.\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M\ \mathsf{in}\ N| \\ = 2 \cdot (2 \cdot |T| + |M|) + |N| \\ > 2 \cdot |T| + 2 \cdot |M| + |N| \\ = |\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = \lambda w^+.M\ \mathsf{in}\ N| \ . \end{array}$$

(h_3)

$$\begin{array}{l} |\ell > \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M| \\ = 2 \cdot (2 \cdot |T| + |M|) \\ > 2 \cdot |T| + 2 \cdot |M| \\ = |\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \ell > M| \ . \end{array}$$

(2) Since the hoisting transformation is terminating, by Newman's lemma it is enough to prove local confluence. There are $9 = 3 \cdot 3$ cases to consider. In each case one checks that the two redexes cannot superpose. Moreover, since the hoisting transformations neither duplicate nor erase terms, one can close the diagrams in one step.

For instance, suppose the term $D[\mathsf{let}\ x = \lambda w^+.\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M\ \mathsf{in}\ N]$ contains a distinct redex Δ of the same type (a function definition containing a *simple* function definition). Then the root of this redex can be in the subterms M or N or in the context D. Moreover if it is in D, then either it is disjoint from the first redex or it contains it strictly. Indeed, the second let of the second redex cannot be the first let of the first redex since the latter is not defining a simple function.

(3) By induction on M. Let F be an abbreviation for $\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ N$

$@(x, x^+)$ The property holds trivially.

$\mathsf{let}\ y = C\ \mathsf{in}\ M$ Then M must contain a function definition. Then by inductive hypothesis, $M \equiv D'[F]$. We conclude by taking $D \equiv \mathsf{let}\ y = C\ \mathsf{in}\ D$.

$\mathsf{let}\ y = \lambda x^+.M'\ \mathsf{in}\ M$ If M is a restricted term then we take $D \equiv [\,]$. Otherwise, M' must contain a function definition and by inductive hypothesis, $M' \equiv D'[F]$. Then we take $D \equiv \mathsf{let}\ y = \lambda x^+.D'\ \mathsf{in}\ M$.

$\ell > M$ Then M contains a function definition and by inductive hypothesis $M \equiv D'[F]$. We conclude by taking $D \equiv \ell > D'$.

(4) In the terms of the $\lambda^{\ell}_{cc,a}$ calculus all functions are closed and therefore the condition is vacuously satisfied.

(5) We proceed by case analysis on the hoisting transformations.

(6) We proceed by induction on the structure of the term M.

$@(x, y^+)$ This is a program.

$\mathsf{let}\ x = B\ \mathsf{in}\ M'$ There are two cases:

- If M' is not a program then by inductive hypothesis a hoisting transformation applies and the same transformation can be applied to M.
- If M' is a program then it has a function definition on top (otherwise M is a program). Because M belongs to I the side condition of (h_1) is satisfied.

$\mathsf{let}\ x = \lambda y^+.M'\ \mathsf{in}\ M''$ Again there are two cases:

- If M' or M'' are not programs then by inductive hypothesis a hoisting transformation applies and the same transformation can be applied to M.
- Otherwise, M' is a program with a function definition on top (otherwise M is a program). Because M belongs to I the side condition of (h_2) is satisfied.

$\ell > M'$ Again there are two cases:

- If M' is not a program then by inductive hypothesis a hoisting transformation applies and the same transformation can be applied to M.
- If M' is a program then it has a function definition on top (otherwise M is a program) and (h_3) applies to M. □

B.8 Proof of proposition 8 [hoisting commutation]

As a preliminary step, extend the erasure function to the hoisting contexts in the obvious way and notice that (i) if D is a hoisting context then $er(D)$ is a hoisting context too, and (ii) $er(D[M]) \equiv er(D)[er(M)]$.

(1) We proceed by case analysis on the hoisting transformation applied to M. The case where $er(M) \equiv er(N)$ arises in (h_3):

$$\begin{array}{rcl} D[\ell > \mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ M] & \leadsto & D[\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ \ell > M] \\ er(D[\ell > \mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ M]) & \equiv & er(D[\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ \ell > M]) \end{array}$$

(2) We show that $er(M) \rightsquigarrow$ entails that $M \rightsquigarrow$. Since $er(M)$ has no labels, either (h_1) or (h_2) apply. Then M is a term that is derived from $er(M)$ by inserting (possibly empty) sequences of pre-labelling before each subterm. We check that either the hoisting transformation applied to $er(M)$ can be applied to M too or (h_3) applies.

(3) If $\mathcal{C}_h(M) \equiv N$ then by definition we have $M \rightsquigarrow^* N \not\rightsquigarrow$. By (1) $er(M) \rightsquigarrow^* er(N)$, and by (2) $er(N) \not\rightsquigarrow$. Hence $\mathcal{C}_h(er(M)) \equiv er(N) \equiv er(\mathcal{C}_h(M))$. □

B.9 Proof of proposition 9 [hoisting simulation]

Definition 2. *A (strong) simulation on the terms of the $\lambda^{\ell}_{cps,a}$-calculus is a binary relation R such that if $M\ R\ N$ and $M \xrightarrow{\alpha} M'$ then there is N' such that $N \xrightarrow{\alpha} N'$ and $M'\ R\ N'$.*

Definition 3. *A (pre-)congruence on the terms of the $\lambda^{\ell}_{cps,a}$-calculus is an equivalence relation (a pre-order) which is preserved by the operators of the calculus.*

Definition 4. *Let $\simeq$ be the smallest congruence on terms of the $\lambda^{\ell}_{cps,a}$-calculus which is induced by structural equivalence and the following commutation of let-definitions:*

$$\mathsf{let}\ x_1 = V_1\ \mathsf{in}\ \mathsf{let}\ x_2 = V_2\ \mathsf{in}\ M \simeq \mathsf{let}\ x_2 = V_2\ \mathsf{in}\ \mathsf{let}\ x_1 = V_1\ \mathsf{in}\ M$$

where: $x_1 \neq x_2, x_1 \notin \mathsf{fv}(V_2), x_2 \notin \mathsf{fv}(V_1)$.

The relation $\simeq$ is preserved by name substitution and it is a simulation.

Definition 5. *Let $\succeq$ the smallest pre-congruence on terms of the $\lambda^{\ell}_{cps,a}$-calculus which is induced by structural equivalence and the following collapse of let-definitions:*

$$\mathsf{let}\ x = V\ \mathsf{in}\ \mathsf{let}\ x = V\ \mathsf{in}\ M \simeq \mathsf{let}\ x = V\ \mathsf{in}\ M$$

where: $x \notin \mathsf{fv}(V)$.

The relation $\succeq$ is preserved by name substitution and it is a simulation.

Definition 6. *Let S_h be the relation $\simeq \circ \succeq$.*

Note that S_h is a simulation too. Then we can state the main lemma.

Lemma 1. *Let M be a term of the $\lambda^{\ell}_{cps,a}$-calculus. If $M \xrightarrow{\alpha} M'$ and $M \rightsquigarrow N$ then there is N' such that $N \xrightarrow{\alpha} N'$ and $M'\ (\rightsquigarrow^*) \circ S_h\ N'$.*

Proof. As a preliminary remark we notice that the hoisting transformations are preserved by name substitution. Namely if $M \rightsquigarrow N$ then $[y^+/x^+]M \rightsquigarrow [y^+/x^+]N$.

There are three reduction rules and three hoisting transformations hence there are 9 cases to consider and for each case we have to analyse how the two redexes can superpose.

As usual a term can be regarded as a tree and an occurrence in the tree is identified by a path π which is a sequence of natural numbers.

– The reduction rule is

$$E[@(x, y^+)] \to E[[y^+/z^+]M]$$

where $E(x) = \lambda z^+.M$. We suppose that π is the path which corresponds to the let-definition of the variable x and π' is that path that determines the redex of the hoisting transformation.

(h_1) There are two critical cases.

1. The let-definition that defines a function of the hoisting transformation coincides with the let-definition of x. In this case M is actually a restricted term T. The diagram is closed in one step.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

(h_2) Again there are two critical situations.

1. The top level let-definition of the hoisting transformation coincides with the let-definition of the variable x in the reduction. This is the case illustrated by the example 5. If we reduce first then we have to apply the hoisting transformation twice (again using preservation under name substitution). After this we have to commute the let-definitions and finally collapse two identical ones.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

(h_3) There are two critical cases.

1. The function let-definition in the hoisting transformation coincides with the let-definition of the variable x in the reduction. We close the diagram in one step.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

– The reduction rule is

$$E[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M] \to E[[z_i/x]M]$$

where $E(y) = (z_1, \ldots z_n)$ and $1 \leq i \leq n$.

(h_1) There are two critical cases.

1. The first let-definition in the hoisting transformation coincides with the let-definition of the tuple in the reduction. We close the diagram in one step
2. The first let-definition in the hoisting transformation coincides with the projection in the reduction. If we reduce first then there is no need to apply a hoisting transformation to close the diagram because the projection disappears.

(h_2) The only critical case arises when the redex for the hoisting transformation is contained in M. We close the diagram in one step using the fact that the transformations are preserved by name substitution.

(h_3) Same argument as in the previous case.

– The reduction rule is

$$E[\ell > M] \xrightarrow{\ell} E[M]$$

The hoisting transformations can be either in E or in M. In both cases we close the diagram in one step. □

We conclude by proving by diagram chasing the following proposition. We rely on the previous lemma and the fact that S_h is a simulation.

Proposition 11. *The relation $T_h = ((\rightsquigarrow^*) \circ S_h)^*$ is a simulation and for all terms of the $\lambda^{\ell}_{cc,a}$-calculus, $M\ T_h\ \mathcal{C}_h(M)$.*

B.10 Proof of theorem 1 [commutation and simulation]

By composition of the commutation and simulation properties of the four compilation steps.

B.11 Proof of proposition 10 [labelling properties]

(1) Both properties are proven by induction on M. The first is immediate. We spell out the second.

x Then $\mathcal{L}_i(x) = x \in W_1 \subseteq W_0$.

$\lambda x^+.M$ Then $\mathcal{L}_i(\lambda x^+.M) = \lambda x^+.\ell > \mathcal{L}_1(M)$ and by inductive hypothesis $\mathcal{L}_1(M) \in W_1$.
Hence, $\ell > \mathcal{L}_1(M) \in W_1$ and $\lambda x^+.\ell > \mathcal{L}_1(M) \in W_1$.

$(M_1, \ldots, M_n)$ Then $\mathcal{L}_i((M_1, \ldots, M_n)) = (\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n))$ and by inductive hypothesis $\mathcal{L}_0(M_j) \in W_0$ for $j = 1, \ldots, n$.
Hence, $(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) \in W_1 \subseteq W_0$.

$\pi_j(M)$ Same argument as for the pairing.

$\mathsf{let}\ x = M\ \mathsf{in}\ N$ Then $\mathcal{L}_i(\mathsf{let}\ x = M\ \mathsf{in}\ N) = \mathsf{let}\ x = \mathcal{L}_0(M)\ \mathsf{in}\ \mathcal{L}_i(N)$ and by inductive hypothesis $\mathcal{L}_0(M) \in W_0$ and $\mathcal{L}_i(N) \in W_1$. Hence $\mathsf{let}\ x = \mathcal{L}_0(M)\ \mathsf{in}\ \mathcal{L}_i(N) \in W_i$.

$@(M_1, \ldots, M_n)$ **and** $i = 0$ Then $\mathcal{L}_0(@(M_1, \ldots, M_n)) = @(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) > \ell$ and by inductive hypothesis $\mathcal{L}_0(M_j) \in W_0$ for $j = 1, \ldots, n$. Hence $@(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) > \ell \in W_0$.

$@(M_1, \ldots, M_n)$ **and** $i = 1$ Same argument as in the previous case to conclude that
$@(\mathcal{L}_1(M_1), \ldots, \mathcal{L}_1(M_n)) \in W_1$.

(2) By (1) we know that $er(\mathcal{L}(M)) \equiv M$ and $\mathcal{L}(M) \in W_0$. Then:

$$\begin{aligned} P &\equiv \mathcal{C}(M) \\ &\equiv \mathcal{C}(er(\mathcal{L}(M))) \\ &\equiv er(\mathcal{C}(\mathcal{L}(M))) \text{ (by 1(1))} . \end{aligned}$$

(3) The main point is to show that the CPS compilation of a labelled term is a term where a pre-labelling appears exactly after each λ-abstraction. The following compilation steps (administrative, closure conversion, hoisting) neither destroy nor introduce new λ-abstractions while maintaining the invariant that the body of each function definition contains exactly one pre-labelling.

As a preliminary step, we define a restricted syntax for the λ^{ℓ}_{cps}-calculus where labels occur exactly after each λ-abstraction.

$$\begin{array}{lll} V & ::= id \mid \lambda id^{+}.\ell > M \mid (V^{+}) & \text{(restricted values)} \\ M & ::= @(V, V^{+}) \mid \mathsf{let}\ id = \pi_i(V)\ \mathsf{in}\ M & \text{(restricted CPS terms)} \\ K & ::= id \mid \lambda id.M & \text{(restricted continuations)} \end{array}$$

Let us call this language $\lambda^{\ell}_{cps,r}$ (r for restricted). First we remark that if V is a restricted value and M is a restricted CPS term then $[V/x]M$ is again a restricted CPS term. Then we show the following property.

> For all terms M of the λ-calculus and all continuations K of the $\lambda^{\ell}_{cps,r}$-calculus the term $\mathcal{L}_i(M) : K$ is again a term of the $\lambda^{\ell}_{cps,r}$-calculus provided that $i = 0$ if K is a function and $i = 1$ if K is a variable.

Notice that the initial continuation $K_0 = \lambda x.@(halt, x)$ is a functional continuation in the restricted calculus and recall that by definition $\mathcal{C}_{cps}(\mathcal{L}(M)) = \mathcal{L}_0(M) : K_0$. We proceed by induction on M and case analysis assuming that if $i = 0$ then $K = \lambda y.N$.

x, $i = 0$ We have: $\mathcal{L}_0(x) : K = x : K = [x/y]N$.
x, $i = 1$ We have: $\mathcal{L}_0(x) : k = x : k = @(k, x)$.
$\lambda x^{+}.M$, $i = 0$ We have:

$$\mathcal{L}_0(\lambda x^{+}.M) : K = \lambda x^{+}.\ell > \mathcal{L}_1(M) : K = [\lambda x^{+}, k.\ell > \mathcal{L}_1(M) : k/y]N$$

and we apply the inductive hypothesis on $\mathcal{L}_1(M) : k$ and closure under value substitution.
$\lambda x^{+}.M$, $i = 1$ We have:

$$\mathcal{L}_1(\lambda x^{+}.M) : k = \lambda x^{+}.\ell > \mathcal{L}_1(M) : k = @(k, \lambda x^{+}, k.\ell > \mathcal{L}_1(M) : k)$$

and we apply the inductive hypothesis on $\mathcal{L}_1(M) : k$.
$@(M_1, \ldots, M_n)$, $i = 0$ We have:

$$\begin{aligned} &\mathcal{L}_i(@(M_1, \ldots, M_n)) : K \\ &\equiv @(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) > \ell : K \\ &\equiv @(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) : K' \\ &\equiv \mathcal{L}_0(M_1) : \lambda x_1 \ldots \mathcal{L}_0(M_n) : \lambda x_n.@(x_1, \ldots, x_n, K') \end{aligned}$$

where $K' = \lambda y.\ell > N$. Then we apply the inductive hypothesis on $M_n, \ldots, M_1$ with the suitable functional continuations.

$@(M_1, \ldots, M_n)$, $i = 1$ We have:

$$\begin{array}{l}\mathcal{L}_i(@(M_1, \ldots, M_n)) : K \\ \equiv @(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) : K \\ \equiv \mathcal{L}_0(M_1) : \lambda x_1 \ldots \mathcal{L}_0(M_n) : \lambda x_n.@(x_1, \ldots, x_n, K)\ .\end{array}$$

Again we apply the inductive hypothesis on $M_n, \ldots, M_1$ with the suitable functional continuations.

$(M_1, \ldots, M_n)$ We have:

$$\begin{array}{l}\mathcal{L}_i((M_1, \ldots, M_n)) : K \\ \equiv (\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) : K \\ \equiv \mathcal{L}_0(M_1) : \lambda x_1 \ldots \mathcal{L}_0(M_n) : \lambda x_n.@(x_1, \ldots, x_n, K)\ .\end{array}$$

We apply the inductive hypothesis on $M_n, \ldots, M_1$ with the suitable functional continuations.

$\pi_j(M)$ We have:

$$\begin{array}{l}\mathcal{L}_i(\pi_j(M)) : K \\ \equiv \pi_j(\mathcal{L}_0(M)) : K \\ \equiv \mathcal{L}_0(M) : \lambda x.\mathsf{let}\ y = \pi_j(x)\ \mathsf{in}\ y : K\ .\end{array}$$

We apply the inductive hypothesis on M with a functional continuation.

$\mathsf{let}\ x = N\ \mathsf{in}\ M$ We have:

$$\begin{array}{l}\mathcal{L}_i(\mathsf{let}\ x = N\ \mathsf{in}\ M) : K \\ \equiv \mathsf{let}\ x = \mathcal{L}_0(N)\ \mathsf{in}\ \mathcal{L}_i(M) : K \\ \equiv \mathcal{L}_0(N) : \lambda x.\mathcal{L}_i(M) : K\ .\end{array}$$

We apply the inductive hypothesis on M and then on N with a functional continuation. □

An Elementary Affine λ-calculus with Multithreading and Side Effects*

Antoine Madet Roberto M. Amadio

Laboratoire PPS, Université Paris Diderot

{madet,amadio}@pps.jussieu.fr

Abstract

Linear logic provides a framework to control the complexity of higher-order functional programs. We present an extension of this framework to programs with multithreading and side effects focusing on the case of elementary time. Our main contributions are as follows. First, we provide a new combinatorial proof of termination in elementary time for the functional case. Second, we develop an extension of the approach to a call-by-value λ-calculus with multithreading and side effects. Third, we introduce an elementary affine type system that guarantees the standard subject reduction and progress properties. Finally, we illustrate the programming of iterative functions with side effects in the presented formalism.

*Work partially supported by project ANR-08-BLANC-0211-01 "COMPLICE" and the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881 (project CerCo).

Contents

1 Introduction

There is a well explored framework based on Linear Logic to control the complexity of higher-order functional programs. In particular, *light logics* [11, 10, 3] have led to a polynomial light affine λ-calculus [13] and to various type systems for the standard λ-calculus guaranteeing that a well-typed term has a bounded complexity [9, 8, 5]. Recently, this framework has been extended to a higher-order process calculus [12] and a functional language with recursive definitions [4]. In another direction, the notion of *stratified region* [7, 1] has been used to prove the termination of higher-order multithreaded programs with side effects.

Our general goal is to extend the framework of light logics to a higher-order functional language with multithreading and side effects by focusing on the case of elementary time [10]. The key point is that termination does not rely anymore on stratification but on the notion of depth which is standard in light logics. Indeed, light logics suggest that complexity can be tamed through a fine analysis of the way the depth of the occurrences of a λ-term can vary during reduction.

Our core functional calculus is a λ-calculus extended with a constructor '!' (the modal operator of linear logic) marking duplicable terms and a related $\mathsf{let}\,!$ destructor. The depth of an occurrence in a λ-term is the number of $!'s$ that must be crossed to reach the occurrence. Our contribution can be described as follows.

1. In Section 2 we propose a formal system called *depth system* that controls the depth of the occurrences and which is a variant of a system proposed in [13]. We show that terms well-formed in the depth system are guaranteed to terminate in elementary time under an arbitrary reduction strategy. The proof is based on an original combinatorial analysis of the depth system ([10] assumes a specific reduction strategy while [13] relies on a standardization theorem).

2. In Section 3, following previous work on an affine-intuitionistic system [2], we extend the functional core with parallel composition and operations producing side effects on an 'abstract' notion of state. We analyse the impact of side-effects operations on the depth of the occurrences and deduce an extended depth system. We show that it still guarantees termination of programs in elementary time under a natural call-by-value evaluation strategy.

3. In Section 4, we refine the depth system with a second order (polymorphic) elementary affine type system and show that the resulting system enjoys subject reduction and progress (besides termination in elementary time).

4. Finally, in Section 5, we discuss the expressivity of the resulting type system. On the one hand we check that the usual encoding of elementary functions goes through. On the other hand, and more interestingly, we provide examples of iterative (multithreaded) programs with side effects.

The λ-calculi introduced are summarized in Table 1.1. For each concurrent language there is a corresponding functional fragment and each language (functional or concurrent) refines the one on its left hand side. The elementary complexity bounds are obtained for the $\lambda^!_\delta$ and $\lambda^{!\mathsf{R}}_\delta$ calculi while the progress property and the expressivity results refer to their typed refinements $\lambda^!_{EA}$ and $\lambda^{!\mathsf{R}}_{EA}$, respectively. Proofs are available in Appendix A.

Functional	$\lambda^!$	$\supset$	$\lambda^!_\delta$	$\supset$	$\lambda^!_{EA}$
$\cap$					
Concurrent	$\lambda^{!\mathsf{R}}$	$\supset$	$\lambda^{!\mathsf{R}}_\delta$	$\supset$	$\lambda^{!\mathsf{R}}_{EA}$

Table 1.1: Overview of the λ-calculi considered

2 Elementary Time in a Modal λ-calculus

In this section, we present our core functional calculus, a related depth system, and show that every term which is well-formed in the depth system terminates in elementary time under an arbitrary reduction strategy.

2.1 A Modal λ-calculus

We introduce a modal λ-calculus called $\lambda^!$. It is very close to the *light affine λ-calculus* of Terui [13] where the paragraph modality '§' used for polynomial time is dropped and where the '!' modality is relaxed as in elementary linear logic [10].

2.1.1 Syntax

Terms are described by the grammar in Table 2.1: We find the usual set of

$$M, N ::= x, y, z \ldots \mid \lambda x.M \mid MN \mid !M \mid \mathsf{let}\ !x = N\ \mathsf{in}\ M$$

Table 2.1: Syntax of $\lambda^!$

variables, λ-abstraction and application, plus a modal operator '!' (read *bang*) and a $\mathsf{let}\,!$ operator. We define $!^0 M = M$ and $!^{n+1} M = !(!^n M)$. In the terms $\lambda x.M$ and $\mathsf{let}\ !x = N\ \mathsf{in}\ M$ the occurrences of x in M are bound. The set of free variables of M is denoted by $\mathsf{FV}(M)$. The number of free occurrences of x in M is denoted by $\mathsf{FO}(x, M)$. $M[N/x]$ denotes the term M in which each free occurrence of x has been substituted by the term N.

Each term has an *abstract syntax tree* as exemplified in Figure 2.1(a). A path starting from the root to a node of the tree denotes an *occurrence* of the program that is denoted by a word $w \in \{0, 1\}^*$ (see Figure 2.1(b)).

We define the notion of *depth*:

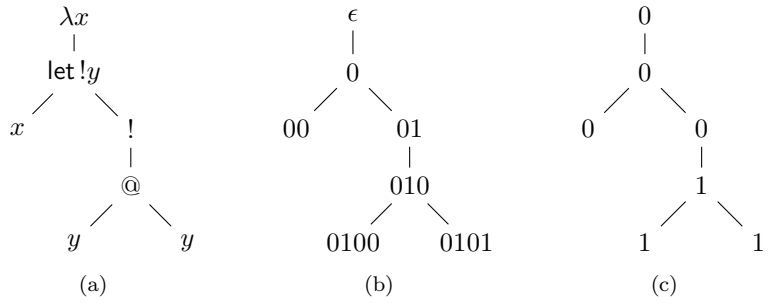

Figure 2.1: Syntax tree of the term $\lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)$, addresses and depths

Definition 2.1 (depth)**.** The *depth* $d(w)$ of an occurrence w is the number of !'s that the path leading to w crosses. The depth $d(M)$ of a term M is the maximum depth of its occurrences.

In Figure 2.1(c), each occurrence is labelled with its depth. Thus $d(\lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)) = 1$. In particular, the occurrence 01 is at depth 0; what matters in computing the depth of an occurrence is the number of ! that precedes strictly the occurrence.

2.1.2 Operational Semantics

We consider an arbitrary reduction strategy. Hence, an evaluation context E can be any term with exactly one occurrence of a special variable $[\,]$, the 'hole'. $E[M]$ denotes E where the hole has been substituted by M. The reduction rules are given in Table 2.2. The $\mathsf{let}!$ is 'filtering' modal terms and 'destructs' the

$$\begin{array}{rcl} E[(\lambda x.M)N] & \to & E[M[N/x]] \\ E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] & \to & E[M[N/x]] \end{array}$$

Table 2.2: Operational semantics of $\lambda^!$

bang of the term $!N$ after substitution. In the sequel, $\stackrel{*}{\to}$ denotes the reflexive and transitive closure of $\to$.

2.2 Depth System

By considering that deeper occurrences have less weight than shallow ones, the proof of termination in elementary time [10] relies on the observation that when reducing a redex at depth i the following holds:

(1) the depth of the term does not increase,

(2) the number of occurrences at depth $j < i$ does not increase,

(3) the number of occurrences at depth i strictly decreases,

(4) the number of occurrences at depth $j > i$ may be increased by a multiplicative factor k bounded by the number of occurrences at depth $i + 1$.

Theses properties can be guaranteed by the following requirements:

(i) in $\lambda x.M$, x may occur at most once in M and at depth 0,

(ii) in $\mathsf{let}\ !x = M\ \mathsf{in}\ N$, x may occur arbitrarily many times in N and at depth 1.

Hence, the rest of this section is devoted to the introduction of a set of inferences rules called depth system. Every term which is valid in the depth system will terminate in elementary time. First, we introduce the judgement:

$$\Gamma \vdash^{\delta} M$$

where δ is a natural number and the context Γ is of the form $x_1 : \delta_1, \ldots, x_n : \delta_n$. We write $dom(\Gamma)$ for the set $\{x_1, \ldots, x_n\}$. It should be interpreted as follows:

> The free variables of $!^{\delta}M$ may only occur at the depth specified by the context Γ.

The inference rules of the depth system are presented in Table 2.3.

$$\frac{}{\Gamma, x : \delta \vdash^{\delta} x}$$

$$\frac{\Gamma, x : \delta \vdash^{\delta} M \qquad \mathsf{FO}(x, M) \leq 1}{\Gamma \vdash^{\delta} \lambda x.M} \qquad \frac{\Gamma \vdash^{\delta} M \qquad \Gamma \vdash^{\delta} N}{\Gamma \vdash^{\delta} MN}$$

$$\frac{\Gamma \vdash^{\delta} N \qquad \Gamma, x : (\delta + 1) \vdash^{\delta} M}{\Gamma \vdash^{\delta} \mathsf{let}\ !x = N\ \mathsf{in}\ M} \qquad \frac{\Gamma \vdash^{\delta+1} M}{\Gamma \vdash^{\delta} !M}$$

Table 2.3: Depth system: $\lambda^{!}_{\delta}$

We comment on the rules. The variable rule says that the current depth of a free variable is specified by the context. The λ-abstraction rule requires that the occurrence of x in M is at the same depth as the formal parameter; moreover it occurs at most once so that no duplication is possible at the current depth (Property (3)). The application rule says that we may only apply two terms if they are at the same depth. The $\mathsf{let}\,!$ rule requires that the bound occurrences of x are one level deeper than the current depth; note that there is no restriction on the number of occurrences of x since duplication would happen one level deeper than the current depth. Finally, the bang rule is better explained in a bottom-up way: crossing a modal occurrence increases the current depth by one.

Definition 2.2 (well-formedness)**.** A term M is *well-formed* if for some Γ and δ a judgement $\Gamma \vdash^{\delta} M$ can be derived.

Example 2.3. The term of Figure 1(a) is well-formed according to our depth system:

$$\dfrac{\dfrac{x : \delta \vdash^{\delta} x \qquad \dfrac{\dfrac{x : \delta, y : \delta + 1 \vdash^{\delta+1} y \qquad x : \delta, y : \delta + 1 \vdash^{\delta+1} y}{x : \delta, y : \delta + 1 \vdash^{\delta+1} yy}}{x : \delta, y : \delta + 1 \vdash^{\delta} !(yy)}}{x : \delta \vdash^{\delta} \mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)}}{\vdash^{\delta} \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(yy)}$$

On the other hand, the followings term is not valid:

$$P = \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(y!(yz))$$

Indeed, the second occurrence of y in $!(y!(yz))$ is too deep of one level, hence reduction may increase the depth by one. For example, $P!!N$ of depth 2 reduces to $!(!N!(!N)z)$ of depth 3.

Proposition 2.4 (properties on the depth system)**.** *The depth system satisfies the following properties:*

1. *If $\Gamma \vdash^{\delta} M$ and x occurs free in M then $x : \delta'$ belongs to Γ and all occurrences of x in $!^{\delta}M$ are at depth δ'.*
2. *If $\Gamma \vdash^{\delta} M$ then $\Gamma, \Gamma' \vdash^{\delta} M$.*
3. *If $\Gamma, x : \delta' \vdash^{\delta} M$ and $\Gamma \vdash^{\delta'} N$ then $d(!^{\delta}M[N/x]) \leq max(d(!^{\delta}M), d(!^{\delta'}N))$ and $\Gamma \vdash^{\delta} M[N/x]$.*
4. *If $\Gamma \vdash^{0} M$ and $M \to N$ then $\Gamma \vdash^{0} N$ and $d(M) \geq d(N)$.*

2.3 Elementary Bound

In this section, we prove that well-formed terms terminate in elementary time under an arbitrary reduction strategy. To this end, we define a measure on terms based on the number of occurrences at each depth.

Definition 2.5 (measure)**.** Given a term M and $0 \leq i \leq d(M)$, let $\omega_i(M)$ be the number of occurrences in M of depth i increased by 2 (so $\omega_i(M) \geq 2$). We define $\mu_n^i(M)$ for $n \geq i \geq 0$ as follows:

$$\mu_n^i(M) = (\omega_n(M), \ldots, \omega_{i+1}(M), \omega_i(M))$$

We write $\mu_n(M)$ for $\mu_n^0(M)$. We order the vectors of $n+1$ natural number with the (well-founded) lexicographic order $>$ from right to left.

We derive a termination property by observing that the measure strictly decreases during reduction.

Proposition 2.6 (termination)**.** *If M is well-formed, $M \to M'$ and $n \geq d(M)$ then $\mu_n(M) > \mu_n(M')$.*

Proof. We do this by case analysis on the reduction rules:

- $M = E[(\lambda x.M_1)M_2] \to M' = E[M_1[M_2/x]]$
  Let the occurrence of the redex $(\lambda x.M_1)M_2$ be at depth i. The restrictions on the formation of terms require that x occurs at most once in M_1 at depth 0. Then $\omega_i(M) - 3 \geq \omega_i(M')$ because we remove the nodes for application and λ-abstraction and either M_2 disappears or the occurrence of the variable x in M_1 disappears (both being at the same depth as the redex). Clearly $\omega_j(M) = \omega_j(M')$ if $j \neq i$, hence

  $$\mu_n(M') \leq (\omega_n(M), \ldots, \omega_{i+1}(M), \omega_i(M) - 3, \mu_{i-1}(M)) \tag{2.1}$$

  and $\mu_n(M) > \mu_n(M')$.

- $M = E[\mathsf{let}\ !x =!M_2\ \mathsf{in}\ M_1] \to M' = E[M_1[M_2/x]]$
  Let the occurrence of the redex $\mathsf{let}\ !x =!M_2\ \mathsf{in}\ M_1$ be at depth i. The restrictions on the formation of terms require that x may only occur in M_1 at depth 1 and hence in M at depth $i+1$. We have that $\omega_i(M) = \omega_i(P)-2$ because the $\mathsf{let}!$ node disappear. Clearly, $\omega_j(M) = \omega_j(M')$ if $j < i$. The number of occurrences of x in M_1 is bounded by $k = \omega_{i+1}(M) \geq 2$. Thus if $j > i$ then $\omega_j(M') \leq k \cdot \omega_j(M)$. Let's write, for $0 \leq i \leq n$:

  $$\mu_n^i(M) \cdot k = (\omega_n(M) \cdot k, \omega_{n-1}(M) \cdot k, \ldots, \omega_i(M) \cdot k)$$

  Then we have

  $$\mu_n(M') \leq (\mu_n^{i+1}(M) \cdot k, \omega_i(M) - 2, \mu_{i-1}(M)) \tag{2.2}$$

  and finally $\mu_n(M) > \mu_n(M')$.

□

We now want to show that termination is actually in elementary time. We recall that a function f on integers is elementary if there exists a k such that for any n, $f(n)$ can be computed in time $\mathcal{O}(t(n,k))$ where:

$$t(n,0) = 2^n, \qquad t(n,k+1) = 2^{t(n,k)}\ .$$

Definition 2.7 (tower functions)**.** We define a family of tower functions $t_\alpha(x_1, \ldots, x_n)$ by induction on n where we assume $\alpha \geq 1$ and $x_i \geq 2$:

$$\begin{array}{rcll} t_\alpha() & = & 0 & \\ t_\alpha(x_1, x_2, \ldots, x_n) & = & (\alpha \cdot x_1)^{2^{t_\alpha(x_2,\ldots,x_n)}} & n \geq 1 \end{array}$$

Then we need to prove the following crucial lemma.

Lemma 2.8 (shift)**.** *Assuming $\alpha \geq 1$ and $\beta \geq 2$, the following property holds for the tower functions with $x, \mathbf{x}$ ranging over numbers greater or equal to 2:*

$$t_\alpha(\beta \cdot x, x', \mathbf{x}) \leq t_\alpha(x, \beta \cdot x', \mathbf{x})$$

Now, by a closer look at the shape of the lexicographic ordering during reduction, we are able to compose the decreasing measure with a tower function.

Theorem 2.9 (elementary bound)**.** *Let M be a well-formed term with $\alpha = d(M)$ and let t_α denote the tower function with $\alpha + 1$ arguments. If $M \to M'$ then $t_\alpha(\mu_\alpha(M)) > t_\alpha(\mu_\alpha(M'))$.*

Proof. We illustrate the proof for $\alpha = 2$ and the crucial case where

$$M = \mathsf{let}\ !x = !M_1\ \mathsf{in}\ M_2 \to M' = M_1[M_2/x]$$

Let $\mu_2(M) = (x, y, z)$ such that $x = \omega_2(M)$, $y = \omega_1(M)$ and $z = \omega_0(M)$. We want to show that:

$$t_2(\mu_2(M')) < t_2(\mu_2(M))$$

We have:

$$\begin{array}{rcll} t_2(\mu_2(M')) & \leq & t_2(x \cdot y, y \cdot y, z - 2) & \text{by inequality (2.2)} \\ & \leq & t_2(x, y^3, z - 2) & \text{by Lemma 2.8} \end{array}$$

Hence we are left to show that:

$$t_2(y^3, z - 2) < t_2(y, z) \quad \textit{i.e.} \quad (2y^3)^{2^{2(z-2)}} < (2y)^{2^{2z}}$$

We have:

$$(2y^3)^{2^{2(z-2)}} \leq (2y)^{3 \cdot 2^{2(z-2)}}$$

Thus we need to show:

$$3 \cdot 2^{2(z-2)} < 2^{2z}$$

Dividing by 2^{2z} we get:

$$3 \cdot 2^{-4} < 1$$

which is obviously true. Hence $t_2(\mu_2(M')) < t_2(\mu_2(M))$. □

This shows that the number of reduction steps of a term M is bound by an elementary function where the height of the tower depends on $d(M)$. We also note that if $M \xrightarrow{*} M'$ then $t_\alpha(\mu_\alpha(M))$ bounds the size of M'. Thus we can conclude with the following corollary.

Corollary 2.10 (elementary time normalisation)**.** *The normalisation of terms of bounded depth can be performed in time elementary in the size of the terms.*

3 Elementary Time in a Modal λ-calculus with Side Effects

In this section, we extend our functional language with side effects operations. By analysing the way side effects act on the depth of occurrences, we extend our depth system to the obtained language. We can then lift the proof of termination in elementary time to programs with side effects that run with a call-by-value reduction strategy.

3.1 A Modal λ-calculus with Multithreading and Regions

We introduce a call-by-value modal λ-calculus endowed with parallel composition and operations to read and write *regions*. We call it $\lambda^{!\mathsf{R}}$. A region is an *abstraction* of a set of dynamically generated values such as imperative references or communication channels. We regard $\lambda^{!\mathsf{R}}$ as an abstract, highly non-deterministic language which entails complexity bounds for more concrete languages featuring references or channels (we will give an example of such a language in Section 5). To this end, it is enough to map the dynamically generated values to their respective regions and observe that the reductions in the concrete languages are simulated in $\lambda^{!\mathsf{R}}$ (see, *e.g.*, [2]).

3.1.1 Syntax

The syntax of the language is described in Table 3.1. We describe the new

$x, y, \ldots$			(Variables)
$r, r', \ldots$			(Regions)
V	::=	$* \mid r \mid x \mid \lambda x.M \mid !V$	(Values)
M	::=	$V \mid MM \mid !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ M$	
		$\mathsf{set}(r, V) \mid \mathsf{get}(r) \mid (M \mid M)$	(Terms)
S	::=	$(r \leftarrow V) \mid (S \mid S)$	(Stores)
P	::=	$M \mid S \mid (P \mid P)$	(Programs)
E	::=	$[\,] \mid EM \mid VE \mid !E \mid \mathsf{let}\ !x = E\ \mathsf{in}\ M$	(Evaluation Contexts)
C	::=	$[\,] \mid (C \mid P) \mid (P \mid C)$	(Static Contexts)

Table 3.1: Syntax of programs: $\lambda^{!\mathsf{R}}$

operators. We have the usual set of variable $x, y, \ldots$ and a set of regions $r, r', \ldots$. The set of values V contains the unit constant $*$, variables, regions, λ-abstraction and modal values $!V$ which are marked with the *bang* operator '!'. The set of terms M contains values, application, modal terms $!M$, a $\mathsf{let}\,!$ operator, $\mathsf{set}(r, V)$ to write the value V at region r, $\mathsf{get}(r)$ to fetch a value from region r and $(M \mid N)$ to evaluate M and N in parallel. A store S is the composition of several stores $(r \leftarrow V)$ in parallel. A program P is a combination of terms and

stores. Evaluation contexts follow a call-by-value discipline. Static contexts C are composed of parallel compositions. Note that stores can only appear in a static context, thus $M(M' \mid (r \leftarrow V))$ is not a legal term. We define $!^n(P \mid P) = (!^n P \mid !^n P)$, and $!^n(r \leftarrow V) = (r \leftarrow V)$. As usual, we abbreviate $(\lambda z.N)M$ with $M; N$, where z is not free in N.

Each program has an *abstract syntax tree* as exemplified in Figure 3.1(a).

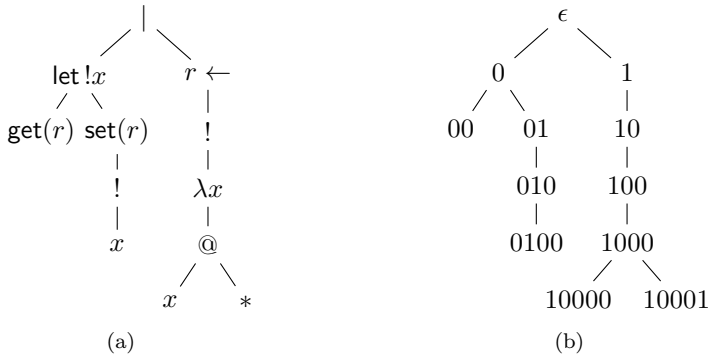

Figure 3.1: Syntax tree and addresses of $P = \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x) \mid (r \leftarrow !(\lambda x.x*))$

3.1.2 Operational Semantics

The operational semantics of the language is described in Table 3.2. Programs

$$\begin{array}{rcll} P \mid P' & \equiv & P' \mid P & \text{(Commutativity)} \\ (P \mid P') \mid P'' & \equiv & P \mid (P' \mid P'') & \text{(Associativity)} \end{array}$$

$$\begin{array}{lllll} E[(\lambda x.M)V] & & \to & E[M[V/x]] & \\ E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] & & \to & E[M[V/x]] & \\ E[\mathsf{set}(r, V)] & & \to & E[*] & \mid\ (r \leftarrow V) \\ E[\mathsf{get}(r)] \mid (r \leftarrow V) & & \to & E[V] & \\ E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V) & & \to & E[M[V/x]] & \mid\ (r \leftarrow !V) \end{array}$$

Table 3.2: Semantics of $\lambda^{!\mathsf{R}}$ programs

are considered up to a structural equivalence $\equiv$ which is the least equivalence relation preserved by static contexts, and which contains the equations for α-renaming and for the commutativity and associativity of parallel composition. The reduction rules apply modulo structural equivalence and in a static context C.

When writing to a region, values are accumulated rather than overwritten (remember that $\lambda^{!\mathsf{R}}$ is an abstract language that can simulate more concrete ones where values relating to the same region are associated with distinct addresses).

On the other hand, reading a region amounts to select non-deterministically one of the values associated with the region. We distinguish two rules to read a region. The first *consumes* the value from the store, like when reading a communication channel. The second *copies* the value from the store, like when reading a reference. Note that in this case the value read must be duplicable (of the shape $!V$).

Example 3.1. Program P of Figure 3.1 reduces as follows:

$$\begin{array}{ll} & \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x) \mid (r \leftarrow !(\lambda x.x*)) \\ \rightarrow & \mathsf{set}(r, !(\lambda x.x*)) \mid (r \leftarrow !(\lambda x.x*)) \\ \rightarrow & * \mid (r \leftarrow !(\lambda x.x*)) \mid (r \leftarrow !(\lambda x.x*)) \end{array}$$

3.2 Extended Depth System

We start by analysing the interaction between the depth of the occurrences and side effects. We observe that side effects may increase the depth or generate occurrences at lower depth than the current redex, which violates Property (1) and (2) (see Section 2.2) respectively. Then to find a suitable notion of depth, it is instructive to consider the following program examples where $M_r = \mathsf{let}\ !z = \mathsf{get}(r)\ \mathsf{in}\ !(z*)$.

$$\begin{array}{ll} (A) & E[\mathsf{set}(r, !V)] \\ (B) & \lambda x.\mathsf{set}(r, x); !\mathsf{get}(r) \\ (C) & !(M_r) \mid (r \leftarrow !(\lambda y.M_{r'})) \mid (r' \leftarrow !(\lambda y.*)) \\ (D) & !(M_r) \mid (r \leftarrow !(\lambda y.M_r)) \end{array}$$

(A) Suppose the occurrence $\mathsf{set}(r, !V)$ is at depth $\delta > 0$ in E. Then when evaluating such a term we always end up in a program of the shape $E[*] \mid (r \leftarrow !V)$ where the occurrence $!V$, previously at depth δ, now appears at depth 0. This contradicts Property (2).

(B) If we apply this program to $!V$ we obtain $!!V$, hence Property (1) is violated because from a program of depth 1, we reduce to a program of depth 2. We remark that this is because the read and write operations do not execute at the same depth.

(C) According to our definition, this program has depth 2, however when we reduce it we obtain a term $!^3*$ which has depth 3, hence Property (1) is violated. This is because the occurrence $\lambda y.M_{r'}$ originally at depth 1 in the store, ends up at depth 2 in the place of z applied to $*$.

(D) If we accept circular stores, we can even write diverging programs whose depth is increased by 1 every two reduction steps.

Given these remarks, the rest of this section is devoted to a *revised notion of depth* and to depth system extended with side effects. First, we introduce the following contexts:

$$\Gamma = x_1 : \delta_1, \ldots, x_n : \delta_n \qquad\qquad R = r_1 : \delta_1, \ldots, r_n : \delta_n$$

where δ_i is a natural number. We write $dom(R)$ for the set $\{r_1, \ldots, r_n\}$. We write $R(r_i)$ for the depth δ_i associated with r_i in the context R.

In the sequel, we shall call the notion of depth introduced in Definition 2.1 *naive depth*. We revisit the notion of naive depth as follows.

Definition 3.2 (revised depth)**.** Let P be a program, R a region context where $dom(R)$ contains all the regions of P and $d_n(w)$ the naive depth of an occurrence w of P. If w does not appear under an occurrence $r \leftarrow$ (a store), then the revised depth $d_r(w)$ of w is $d_n(w)$. Otherwise, $d_r(w)$ is $R(r)+d_n(w)$. The revised depth $d_r(P)$ of the program is the maximum revised depth of its occurrences.

Note that the revised depth is relative to a fixed region context. In the sequel we write $d(\_)$ for $d_r(\_)$. On functional terms, this notion of depth is equivalent to the one given in Definition 2.1. However, if we consider the program of Figure 3.1, we now have $d(10) = R(r)$ and $d(100) = d(1000) = d(10000) = d(10001) = R(r) + 1$.

A judgement in the depth system has the shape

$$R; \Gamma \vdash^{\delta} P$$

and it should be interpreted as follows:

> The free variables of $!^{\delta}P$ may only occur at the depth specified by the context Γ, where depths are computed according to R.

The inference rules of the extended depth system are presented in Table 3.3. We comment on the new rules. A region and the constant $*$ may appear at

$$\frac{}{R; \Gamma, x : \delta \vdash^{\delta} x} \qquad \frac{}{R; \Gamma \vdash^{\delta} r} \qquad \frac{}{R; \Gamma \vdash^{\delta} *}$$

$$\frac{\mathsf{FO}(x, M) \leq 1 \qquad R; \Gamma, x : \delta \vdash^{\delta} M}{R; \Gamma \vdash^{\delta} \lambda x.M} \qquad \frac{R; \Gamma \vdash^{\delta} M_i \qquad i = 1, 2}{R; \Gamma \vdash^{\delta} M_1 M_2}$$

$$\frac{R; \Gamma \vdash^{\delta+1} M}{R; \Gamma \vdash^{\delta} !M} \qquad \frac{R; \Gamma \vdash^{\delta} M_1 \qquad R; \Gamma, x : (\delta + 1) \vdash^{\delta} M_2}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2}$$

$$\frac{}{R, r : \delta; \Gamma \vdash^{\delta} \mathsf{get}(r)} \qquad \frac{R, r : \delta; \Gamma \vdash^{\delta} V}{R, r : \delta; \Gamma \vdash^{\delta} \mathsf{set}(r, V)}$$

$$\frac{R, r : \delta; \Gamma \vdash^{\delta} V}{R, r : \delta; \Gamma \vdash^{0} (r \leftarrow V)} \qquad \frac{R; \Gamma \vdash^{\delta} P_i \qquad i = 1, 2}{R; \Gamma \vdash^{\delta} (P_1 \mid P_2)}$$

Table 3.3: Depth system for programs: $\lambda^{!\mathsf{R}}_{\delta}$

any depth. The key cases are those of read and write: the depth of these two operations is specified by the region context. The current depth of a store is

always 0, however, the depth of the value in the store is specified by R (note that it corresponds to the revised definition of depth). We remark that R is constant in a judgement derivation.

Definition 3.3 (well-formedness)**.** A program P is *well-formed* if for some R, Γ, δ a judgement $R;\Gamma \vdash^{\delta} P$ can be derived.

Example 3.4. The program of Figure 3.1 is well-formed with the following derivation where $R(r) = 0$:

$$\dfrac{\dfrac{R;\Gamma \vdash^{0} \mathsf{get}(r) \qquad \dfrac{\dfrac{R;\Gamma, x:1 \vdash^{1} x}{R;\Gamma, x:1 \vdash^{0} !x}}{R;\Gamma, x:1 \vdash^{0} \mathsf{set}(r,!x)}}{R;\Gamma \vdash^{0} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x)} \qquad \dfrac{\vdots}{R;\Gamma \vdash^{0} (r \leftarrow !(\lambda x.x*))}}{R;\Gamma \vdash^{0} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r,!x) \mid (r \leftarrow !(\lambda x.x*))}$$

We reconsider the troublesome programs with side effects. Program (A) is well-formed with judgement (i):

$$R;\Gamma \vdash^{0} E[\mathsf{set}(r,!V)] \quad \text{with } R = r:\delta \quad (i)$$
$$R;\Gamma \vdash^{0} !M_r \mid (r \leftarrow !(\lambda y.M_{r'})) \mid (r' \leftarrow !(\lambda y.*)) \quad \text{with } R = r:1, r':2 \quad (ii)$$

Indeed, the occurrence $!V$ is now preserved at depth δ in the store. Program (B) is not well-formed since the read operation requires $R(r) = 1$ and the write operations require $R(r) = 0$. Program (C) is well-formed with judgement (ii); indeed its depth does not increase anymore because $!M_r$ has depth 2 but since $R(r) = 1$ and $R(r') = 2$, $(r \leftarrow !(\lambda y.M_{r'}))$ has depth 3 and $(r' \leftarrow !(\lambda y.*))$ has depth 2. Hence program (C) has already depth 3. Finally, it is worth noticing that the diverging program (D) is not well-formed since $\mathsf{get}(r)$ appears at depth 1 in $!M_r$ and at depth 2 in the store.

Theorem 3.5 (properties on the extended depth system)**.** *The following properties hold:*

1. *If $R;\Gamma \vdash^{\delta} M$ and x occurs free in M then $x : \delta'$ belongs to Γ and all occurrences of x in $!^{\delta}M$ are at depth δ'.*
2. *If $R;\Gamma \vdash^{\delta} P$ then $R;\Gamma,\Gamma' \vdash^{\delta} P$.*
3. *If $R;\Gamma, x:\delta' \vdash^{\delta} M$ and $R;\Gamma \vdash^{\delta'} V$ then $R;\Gamma \vdash^{\delta} M[V/x]$ and $d(!^{\delta}M[V/x]) \leq max(d(!^{\delta}M), d(!^{\delta'}V))$.*
4. *If $R;\Gamma \vdash^{0} P$ and $P \to P'$ then $R;\Gamma \vdash^{0} P'$ and $d(P) \geq d(P')$.*

3.3 Elementary Bound

In this section, we prove that well-formed programs terminate in elementary time. The measure of Definition 2.5 extends trivially to programs except that

to simplify the proofs of the following properties, we assume the occurrences labelled with $|$ and $r \leftarrow$ do not count in the measure and that $\mathsf{set}(r)$ counts for two occurrences such that the measure strictly decreases on the rule $E[\mathsf{set}(r, V)] \to E[*] \mid (r \leftarrow V)$.

We derive a similar termination property:

Proposition 3.6 (termination)**.** *If P is well-formed, $P \to P'$ and $n \geq d(P)$ then $\mu_n(P) > \mu_n(P')$.*

Proof. By a case analysis on the new reduction rules.

- $P \equiv E[\mathsf{set}(r, V)] \to P' \equiv E[*] \mid (r \leftarrow V)$
  If $R; \Gamma \vdash^{\delta} \mathsf{set}(r, V)$ then by 3.5(4) we have $R; \Gamma \vdash^{0} (r \leftarrow V)$ with $R(r) = \delta$. Hence, by definition of the depth, the occurrences in V stay at depth δ in $(r \leftarrow V)$. However, the node $\mathsf{set}(r, V)$ disappears, and both $*$ and $(r \leftarrow V)$ are null occurrences, thus $\omega_\delta(P') = \omega_\delta(P) - 1$. The number of occurrences at other depths stay unchanged, hence $\mu_n(P) > \mu_n(P')$.

- $P \equiv E[\mathsf{get}(r)] \mid (r \leftarrow V) \to P' \equiv E[V]$
  If $R; \Gamma \vdash^{0} (r \leftarrow V)$ with $R(r) = \delta$, then $\mathsf{get}(r)$ must be at depth δ in $E[\ ]$. Hence, by definition of the depth, the occurrences in V stay at depth δ, while the node $\mathsf{get}(r)$ and $|$ disappear. Thus $\omega_\delta(P') = \omega_\delta(P) - 1$ and the number of occurrences at other depths stay unchanged, hence $\mu_n(P) > \mu_n(P')$.

- $P \equiv E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow\ !V) \to P' \equiv E[M[V/x]] \mid (r \leftarrow\ !V)$
  This case is the only source of duplication with the reduction rule on $\mathsf{let}\,!$. Suppose $R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M$. Then we must have $R; \Gamma \vdash^{\delta+1} V$. The restrictions on the formation of terms require that x may only occur in M at depth 1 and hence in P at depth $\delta+1$. Hence the occurrences in V stay at the same depth in $M[V/x]$, while the let, $\mathsf{get}(r)$ and some x nodes disappear, hence $\omega_\delta(P) \leq \omega_\delta(P') - 2$. The number of occurrences of x in M is bound by $k = \omega_{\delta+1}(P) \geq 2$. Thus if $j > \delta$ then $\omega_j(P') \leq k \cdot \omega_j(P)$. Clearly, $\omega_j(M) = \omega_j(M')$ if $j < i$. Hence, we have

  $$\mu_n(P') \leq (\mu_n^{i+1}(P) \cdot k, \omega_i(P) - 2, \mu_{i-1}(P)) \tag{3.1}$$

  and $\mu_n(P) > \mu_n(P')$.

□

Then we have the following theorem.

Theorem 3.7 (elementary bound)**.** *Let P be a well-formed program with $\alpha = d(P)$ and let t_α denote the tower function with $\alpha+1$ arguments. Then if $P \to P'$ then $t_\alpha(\mu_\alpha(P)) > t_\alpha(\mu_\alpha(P'))$.*

Proof. From the proof of termination, we remark that the only new rule that duplicates occurrences is the one that copies from the store. Moreover, the derived inequality (3.1) is exactly the same as the inequality (2.2). Hence the arithmetic of the proof is exactly the same as in the proof of elementary bound for the functional case. □

Corollary 3.8. *The normalisation of programs of bounded depth can be performed in time elementary in the size of the terms.*

4 An Elementary Affine Type System

The depth system entails termination in elementary time but does *not* guarantee that programs 'do not go wrong'. In particular, the introduction and elimination of bangs during evaluation may generate programs that deadlock, *e.g.*,

$$\mathsf{let}\ !y = (\lambda x.x)\ \mathsf{in}\ !(yy) \tag{4.1}$$

is well-formed but the evaluation is stuck. In this section we introduce an elementary affine type system ($\lambda^{!\mathsf{R}}_{EA}$) that guarantees that programs cannot deadlock (except when trying to read an empty store).

The upper part of Table 4.1 introduces the syntax of types and contexts. Types are denoted with $\alpha, \alpha', \ldots$. Note that we distinguish a special behaviour

$t, t', \ldots$			(Type variables)
α	$::=$	$\mathbf{B} \mid A$	(Types)
A	$::=$	$t \mid \mathbf{1} \mid A \multimap \alpha \mid\ !A \mid \forall t.A \mid \mathsf{Reg}_r A$	(Value-types)
Γ	$::=$	$x_1 : (\delta_1, A_1), \ldots, x_n : (\delta_n, A_n)$	(Variable contexts)
R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$	(Region contexts)

$$\frac{}{R \downarrow t} \qquad \frac{}{R \downarrow \mathbf{1}} \qquad \frac{}{R \downarrow \mathbf{B}} \qquad \frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)}$$

$$\frac{R \downarrow A}{R \downarrow\ !A} \qquad \frac{r : (\delta, A) \in R}{R \downarrow \mathsf{Reg}_r A} \qquad \frac{R \downarrow A \quad t \notin R}{R \downarrow \forall t.A}$$

$$\frac{\forall r : (\delta, A) \in R \quad R \downarrow A}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha}$$

$$\frac{\forall x : (\delta, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Table 4.1: Types and contexts

type $\mathbf{B}$ which is given to the entities of the language which are not supposed

to return a value (such as a store or several terms in parallel) while types of entities that may return a value are denoted with A. Among the types A, we distinguish type variables $t, t', \ldots$, a terminal type $\mathbf{1}$, an affine functional type $A \multimap \alpha$, the type $!A$ of terms of type A that can be duplicated, the type $\forall t.A$ of polymorphic terms and the type $\mathsf{Reg}_r A$ of the region r containing values of type A. Hereby types may depend on regions.

In contexts, natural numbers δ_i play the same role as in the depth system. Writing $x : (\delta, A)$ means that the variable x ranges on values of type A and may occur at depth δ. Writing $r : (\delta, A)$ means that addresses related to region r contain values of type A and that read and writes on r may only happen at depth δ. The typing system will additionally guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains an hypothesis $r : (\delta, A)$.

Because types depend on regions, we have to be careful in stating in Table 4.1 when a region-context and a type are compatible ($R \downarrow \alpha$), when a region context is well-formed ($R \vdash$), when a type is well-formed in a region context ($R \vdash \alpha$) and when a context is well-formed in a region context ($R \vdash \Gamma$). A more informal way to express the condition is to say that a judgement $r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n) \vdash \alpha$ is well formed provided that: (1) all the region names occurring in the types $A_1, \ldots, A_n, \alpha$ belong to the set $\{r_1, \ldots, r_n\}$, (2) all types of the shape $\mathsf{Reg}_{r_i} B$ with $i \in \{1, \ldots, n\}$ and occurring in the types $A_1, \ldots, A_n, \alpha$ are such that $B = A_i$. We notice the following substitution property on types.

Proposition 4.1. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

Example 4.2. One may verify that $r : (\delta, \mathbf{1} \multimap \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ can be derived while the following judgements cannot: $r : (\delta, \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$, $r : (\delta, \mathsf{Reg}_r \mathbf{1}) \vdash \mathbf{1}$.

A typing judgement takes the form:

$$R; \Gamma \vdash^{\delta} P : \alpha$$

It attributes a type α to the program P at depth δ, in the region context R and the context Γ. Table 4.2 introduces an elementary affine type system *with regions*. One can see that the δ's are treated as in the depth system. Note that a region r may occur at any depth. In the $\mathsf{let}!$ rule, M should be of type $!A$ since x of type A appears one level deeper. A program in parallel with a store should have the type of the program since we might be interested in the value the program reduces to; however, two programs in parallel cannot reduce to a single value, hence we give them a behaviour type. The polymorphic rules are straightforward where $t \notin (R; \Gamma)$ means t does not occur free in a type of R or Γ.

Example 4.3. The well-formed program (C) can be given the following typing judgement: $R; \_ \vdash^0 !(M_r) \mid (r \leftarrow !(\lambda y.M_{r'})) \mid (r' \leftarrow !(\lambda y.*)) : !!\mathbf{1}$ where: $R = r : (1, !(\mathbf{1} \multimap \mathbf{1})), r' : (2, !(\mathbf{1} \multimap \mathbf{1}))$. Also, we remark that the deadlocking program (4.1) admits no typing derivation.

$$\frac{R \vdash \Gamma \quad x : (\delta, A) \in \Gamma}{R; \Gamma \vdash^{\delta} x : A} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} * : \mathbf{1}} \qquad \frac{R \vdash \Gamma \quad r : (\delta', A) \in R}{R; \Gamma \vdash^{\delta} r : \mathsf{Reg}_r A}$$

$$\frac{\mathsf{FO}(x, M) \leq 1 \quad R; \Gamma, x : (\delta, A) \vdash^{\delta} M : \alpha}{R; \Gamma \vdash^{\delta} \lambda x.M : A \multimap \alpha} \qquad \frac{R; \Gamma \vdash^{\delta} M : A \multimap \alpha \quad R; \Gamma \vdash^{\delta} N : A}{R; \Gamma \vdash^{\delta} MN : \alpha}$$

$$\frac{R; \Gamma \vdash^{\delta+1} M : A}{R; \Gamma \vdash^{\delta} !M : !A} \qquad \frac{R; \Gamma \vdash^{\delta} M : !A \quad R; \Gamma, x : (\delta + 1, A) \vdash^{\delta} N : B}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = M\ \mathsf{in}\ N : B}$$

$$\frac{R; \Gamma \vdash^{\delta} M : A \quad t \notin (R; \Gamma)}{R; \Gamma \vdash^{\delta} M : \forall t.A} \qquad \frac{R; \Gamma \vdash^{\delta} M : \forall t.A \quad R \vdash B}{R; \Gamma \vdash^{\delta} M : A[B/t]}$$

$$\frac{r : (\delta, A) \in R \quad R \vdash \Gamma}{R; \Gamma \vdash^{\delta} \mathsf{get}(r) : A} \qquad \frac{r : (\delta, A) \in R \quad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, V) : \mathbf{1}} \qquad \frac{r : (\delta, A) \in R \quad R; \Gamma \vdash^{\delta} V : A}{R; \Gamma \vdash^{0} (r \leftarrow V) : \mathbf{B}}$$

$$\frac{R; \Gamma \vdash^{\delta} P : \alpha \quad R; \Gamma \vdash^{\delta} S : \mathbf{B}}{R; \Gamma \vdash^{\delta} (P \mid S) : \alpha} \qquad \frac{P_i \text{ not a store } i = 1, 2 \quad R; \Gamma \vdash^{\delta} P_i : \alpha_i}{R; \Gamma \vdash^{\delta} (P_1 \mid P_2) : \mathbf{B}}$$

Table 4.2: An elementary affine type system: $\lambda^{!\mathsf{R}}_{EA}$

Theorem 4.4 (subject reduction and progress)**.** *The following properties hold.*

1. (Well-formedness) *Well-typed programs are well-formed.*
2. (Weakening) *If* $R; \Gamma \vdash^{\delta} P : \alpha$ *and* $R \vdash \Gamma, \Gamma'$ *then* $R; \Gamma, \Gamma' \vdash^{\delta} P : \alpha$.
3. (Substitution) *If* $R; \Gamma, x : (\delta', A) \vdash^{\delta} M : \alpha$ *and* $R; \Gamma' \vdash^{\delta'} V : A$ *and* $R \vdash \Gamma, \Gamma'$ *then* $R; \Gamma, \Gamma' \vdash^{\delta} M[V/x] : \alpha$.
4. (Subject Reduction) *If* $R; \Gamma \vdash^{\delta} P : \alpha$ *and* $P \to P'$ *then* $R; \Gamma \vdash^{\delta} P' : \alpha$.
5. (Progress) *Suppose* P *is a closed typable program which cannot reduce. Then* P *is structurally equivalent to a program*

$$M_1 \mid \cdots \mid M_m \mid S_1 \mid \cdots \mid S_n \quad m, n \geq 0$$

where M_i *is either a value or can be decomposed as a term* $E[\mathsf{get}(r)]$ *such that no value is associated with the region* r *in the stores* $S_1, \ldots, S_n$.

5 Expressivity

In this section, we consider two results that illustrate the expressivity of the elementary affine type system. First we show that all elementary functions can be represented and second we develop an example of iterative program with side effects.

5.1 Completeness

The representation result just relies on the functional core of the language $\lambda^!_{EA}$. Building on the standard concept of Church numeral, Table 5.1 provides a representation for natural numbers and the multiplication function. We denote with

N	$=$	$\forall t.!(t \multimap t) \multimap !(t \multimap t)$	(type of numerals)
$\overline{n}$	$:$	N	(numerals)
$\overline{n}$	$=$	$\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f(\cdots(fx)\cdots))$	
mult	$:$	$\mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{N})$	(multiplication)
mult	$=$	$\lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ n(m!f)$	

Table 5.1: Representation of natural numbers and the multiplication function

$\mathbb{N}$ the set of natural numbers. The precise notion of representation is spelled out in the following definitions where by strong β-reduction we mean that reduction under λ's is allowed.

Definition 5.1 (number representation)**.** Let $\emptyset \vdash^\delta M : \mathsf{N}$. We say M *represents* $n \in \mathbb{N}$, written $M \Vdash n$, if, by using a strong β-reduction relation, $M \xrightarrow{*} \overline{n}$.

Definition 5.2 (function representation)**.** Let $\emptyset \vdash^\delta F : (\mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_k) \multimap !^p\mathsf{N}$ where $p \geq 0$ and $f : \mathbb{N}^k \to \mathbb{N}$. We say F *represents* f, written $F \Vdash f$, if for all M_i and $n_i \in \mathbb{N}$ where $1 \leq i \leq k$ such that $\emptyset \vdash^\delta M_i : N$ and $M_i \Vdash n_i$, $FM_1 \ldots M_k \Vdash f(n_1, \ldots, n_k)$.

Elementary functions are also characterized as the smallest class of functions containing zero, successor, projection, subtraction and which is closed by composition and bounded summation/product. These functions can be represented in the sense of Definition 5.2 by adapting the proofs from Danos and Joinet [10].

Theorem 5.3 (completeness)**.** *Every elementary function is representable in* $\lambda^!_{EA}$.

5.2 Iteration with Side Effects

We rely on a slightly modified language where reads, writes and stores relate to concrete addresses rather than to abstract regions. In particular, we introduce

terms of the form $\nu x\ M$ to generate a fresh address name x whose scope is M. One can then write the following program:

$$\nu x\ ((\lambda y.\mathsf{set}(y, V))x) \xrightarrow{*} \nu x\ * \mid (x \leftarrow V)$$

where x and y relate to a region r, *i.e.* they are of type $\mathsf{Reg}_r A$. Our type system can be easily adapted by associating region types with the address names. Next we show that it is possible to program the iteration of operations producing a side effect on an inductive data structure. Specifically, in the following we show how to iterate, possibly in parallel, an update operation on a list of addresses of the store. The examples have been tested on a running implementation of the language.

Following Church encodings, we define the representation of lists and the associated iterator in Table 5.2. Here is the function multiplying the numeral

$$\begin{array}{rcll}
\mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap !(t \multimap t) & \text{(type of lists)} \\
\\
[u_1, \ldots, u_n] & : & \mathsf{List}\ A & \text{(list represent.)} \\
[u_1, \ldots, u_n] & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f u_1 (f u_2 \ldots (f u_n x)) & \\
\\
\mathsf{list_it} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap !t \multimap !t & \text{(iterator)} \\
\mathsf{list_it} & = & \lambda f.\lambda l.\lambda z.\mathsf{let}\ !z = z\ \mathsf{in}\ \mathsf{let}\ !y = l f\ \mathsf{in}\ !(yz) &
\end{array}$$

Table 5.2: Representation of lists

pointed by an address at region r:

$$\begin{array}{rcl}
\mathsf{update} & : & !\mathsf{Reg}_r\mathsf{N} \multimap !\mathbf{1} \multimap !\mathbf{1} \\
\mathsf{update} & = & \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ \lambda z.!((\lambda y.\mathsf{set}(x, y))(\mathsf{mult}\ \overline{2}\ \mathsf{get}(x))
\end{array}$$

Consider the following list of addresses and stores:

$$[!x, !y, !z] \mid (x \leftarrow \overline{m}) \mid (y \leftarrow \overline{n}) \mid (z \leftarrow \overline{p})$$

Note that the bang constructors are needed to match the type $!\mathsf{Reg}_r\mathsf{N}$ of the argument of update. Then we define the iteration as:

$$\mathsf{run} : !!\mathbf{1} \qquad \mathsf{run} = \mathsf{list\_it}\ !\mathsf{update}\ [!x, !y, !z]\ !!*$$

Notice that it is well-typed with $R = r : (2, \mathsf{N})$ since both the read and the write appear at depth 2. Finally, the program reduces by updating the store as expected:

$$\begin{array}{rl}
 & \mathsf{run} \mid (x \leftarrow \overline{m}) \mid (y \leftarrow \overline{n}) \mid (z \leftarrow \overline{p}) \\
\xrightarrow{*} & !!\mathbf{1} \mid (x \leftarrow \overline{2m}) \mid (y \leftarrow \overline{2n}) \mid (z \leftarrow \overline{2p})
\end{array}$$

Building on this example, suppose we want to write a program with three concurrent threads where each thread multiplies by 2 the memory cells pointed by

a list. Here is a function waiting to apply a functional f to a value x in three concurrent threads:

$$\begin{array}{lcl} \mathsf{gen\_threads} & : & \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B} \\ \mathsf{gen\_threads} & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ !(fx) \mid !(fx) \mid !(fx) \end{array}$$

We define the functional F as run but parametric in the list:

$$F : \mathsf{List}\ !\mathsf{Reg}_r\mathsf{N} \multimap !!\mathbf{1} \qquad F = \lambda l.\mathsf{list\_it}\ !\mathsf{update}\ l\ !!*$$

And the final term is simply:

$$\mathsf{run\_threads} : \mathbf{B} \qquad \mathsf{run\_threads} = \mathsf{gen\_threads}\ !F\ ![!x, !y, !z]$$

where $R = r : (3, !\mathsf{N})$. Our program then reduces as follows:

$$\begin{array}{rlllll} & \mathsf{run\_threads} & \mid & (x \leftarrow \overline{m}) & \mid & (y \leftarrow \overline{n}) \quad \mid \quad (z \leftarrow \overline{p}) \\ \xrightarrow{*} & !!!\mathbf{1} \mid !!!\mathbf{1} \mid !!!\mathbf{1} & \mid & (x \leftarrow \overline{8m}) & \mid & (y \leftarrow \overline{8n}) \quad \mid \quad (z \leftarrow \overline{8p}) \end{array}$$

Note that different thread interleavings are possible but in this particular case the reduction is confluent.

6 Conclusion

We have introduced a type system for a higher-order functional language with multithreading and side effects that guarantees termination in elementary time thus providing a significant extension of previous work that had focused on purely functional programs. In the proposed approach, the depth system plays a key role and allows for a relatively simple presentation. In particular we notice that we can dispense both with the notion of *stratified region* that arises in recent work on the termination of higher-order programs with side effects [1, 7] and with the distinction between affine and intuitionistic hypotheses [6, 2].

As a future work, we would like to adapt our approach to polynomial time. In another direction, one could ask if it is possible to program in a simplified language without bangs and then try to infer types or depths.

Acknowledgements We would like to thank Patrick Baillot for numerous helpful discussions and a careful reading on a draft version of this report.

References

[1] R. M. Amadio. On stratified regions. In *APLAS'09*, volume 5904 of *LNCS*, pages 210–225. Springer, 2009.

[2] R. M. Amadio, P. Baillot, and A. Madet. An affine-intuitionistic system of types and effects: confluence and termination. Technical report, Laboratoire PPS, 2009. `http://hal.archives-ouvertes.fr/hal-00438101/`.

[3] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Trans. Comput. Log.*, 3(1):137–175, 2002.

[4] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP'10*, volume 6012 of *LNCS*, pages 104–124. Springer, 2010.

[5] P. Baillot and K. Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA'05*, volume 3461 of *LNCS*, pages 55–70. Springer, 2005.

[6] A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, The Laboratory for Foundations of Computer Science, University of Edinburgh, 1996.

[7] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208(6):716–736, 2010.

[8] P. Coppola, U. Dal Lago, and S. Ronchi Della Rocca. Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science*, 4(4), 2008.

[9] P. Coppola and S. Martini. Optimizing optimal reduction: A type inference algorithm for elementary affine logic. *ACM Trans. Comput. Log.*, 7:219–260, 2006.

[10] V. Danos and J.-B. Joinet. Linear logic and elementary time. *Inf. Comput.*, 183(1):123 – 137, 2003.

[11] J.-Y. Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.

[12] U. D. Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 46–60, 2010.

[13] K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007.

A Proofs

A.1 Proof of theorem 3.5

1. We consider the last rule applied in the typing of M.

   - $\Gamma, x : \delta \vdash^\delta x$. The only free variable is x and indeed it is at depth δ in $!^\delta x$.
   - $\Gamma \vdash^\delta \lambda y.M$ is derived from $\Gamma, y : \delta \vdash^\delta M$. If x is free in $\lambda y.M$ then $x \neq y$ and x is free in M. By inductive hypothesis, $x : \delta' \in \Gamma, y : \delta$ and all occurrences of x in $!^\delta M$ are at depth δ'. By definition of depth, the same is true for $!^\delta(\lambda y.M)$.
   - $\Gamma \vdash^\delta (M_1 M_2)$ is derived from $\Gamma \vdash^\delta M_i$ for $i = 1, 2$. By inductive hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^\delta M_i$, $i = 1, 2$ are at depth δ'. By definition of depth, the same is true for $!^\delta(M_1 M_2)$.
   - $\Gamma \vdash^\delta !M$ is derived from $\Gamma \vdash^{\delta+1} M$. By inductive hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^{\delta+1} M$ are at depth δ' and notice that $!^{\delta+1} M = !^\delta(!M)$.
   - $\Gamma \vdash^\delta \mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2$ is derived from $\Gamma \vdash^\delta M_1$ and $\Gamma, y : (\delta + 1) \vdash^\delta M_2$. Without loss of generality, assume $x \neq y$. By inductive hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^\delta M_i$, $i = 1, 2$ are at depth δ'. By definition of depth, the same is true for $!^\delta(\mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2)$.
   - $M \equiv *$ or $M \equiv r$ or $M \equiv \mathsf{get}(r)$. There is no free variable in these terms.
   - $M \equiv \mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ N$. We have

     $$\frac{R, r : \delta; \Gamma, y : (\delta + 1) \vdash^\delta N}{R, r : \delta; \Gamma \vdash^\delta \mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ N}$$

     If x occurs free in M then x occurs free in N. By induction hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^\delta N$ are at depth δ'. By definition of the depth, this is also true for $!^\delta(\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ N)$.
   - $M \equiv \mathsf{set}(r, V)$. We have

     $$\frac{R, r : \delta; \Gamma \vdash^\delta V}{R, r : \delta; \Gamma \vdash^\delta \mathsf{set}(r, V)}$$

     If x occurs free in $\mathsf{set}(r, V)$ then x occurs free in V. By induction hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^\delta V$ are at depth δ'. By definition of the depth, this is also true for $!^\delta(\mathsf{set}(r, V))$.
   - $M \equiv (M_1 \mid M_2)$. We have

     $$\frac{R; \Gamma \vdash^\delta M_i \qquad i = 1, 2}{R; \Gamma \vdash^\delta (M_1 \mid M_2)}$$

If x occurs free in M then x occurs free in M_i, $i = 1, 2$. By induction hypothesis, $x : \delta' \in \Gamma$ and all occurrences of x in $!^{\delta} M_i$, $i = 1, 2$, are at depth δ'. By definition of depth, the same is true of $!^{\delta}(M_1 \mid M_2)$.

2. All the rules can be weakened by adding a context Γ'.

3. If x is not free in M, we just have to check that any proof of $\Gamma, x : \delta' \vdash^{\delta} M$ can be transformed into a proof of $\Gamma \vdash^{\delta} M$.

   So let us assume x is free in M.

   We consider first the bound on the depth. By (1), we know that all occurrences of x in $!^{\delta} M$ are at depth δ'. By definition of depth, it follows that $\delta' \geq \delta$ and the occurrences of x in M are at depth $(\delta' - \delta)$. An occurrence in $!^{\delta'} V$ at depth $\delta' + \delta''$ will generate an occurrence in $!^{\delta} M[V/x]$ at the same depth $\delta + (\delta' - \delta) + \delta''$.

   Next, we proceed by induction on the derivation of $\Gamma, x : \delta' \vdash^{\delta} M$.

   - $\Gamma, x : \delta \vdash^{\delta} x$. Then $\delta = \delta'$, $x[V/x] = V$, and by hypothesis $\Gamma \vdash^{\delta'} V$.
   - $\Gamma, x : \delta' \vdash^{\delta} \lambda y.M$ is derived from $\Gamma, x : \delta', y : \delta \vdash^{\delta} M$, with $x \neq y$ and y not occurring in N. By (2), $\Gamma, y : \delta \vdash^{\delta'} V$. By inductive hypothesis, $\Gamma, y : \delta \vdash^{\delta} M[V/x]$, and then we conclude $\Gamma \vdash^{\delta} (\lambda y.M)[V/x]$.
   - $\Gamma, x : \delta' \vdash^{\delta} (M_1 M_2)$ is derived from $\Gamma, x : \delta' \vdash^{\delta} M_i$, for $i = 1, 2$. By inductive hypothesis, $\Gamma \vdash^{\delta} M_i[V/x]$, for $i = 1, 2$ and then we conclude $\Gamma \vdash^{\delta} (M_1 M_2)[V/x]$.
   - $\Gamma, x : \delta' \vdash^{\delta} !M$ is derived from $\Gamma, x : \delta' \vdash^{\delta+1} M$. By inductive hypothesis, $\Gamma \vdash^{\delta+1} M[V/x]$, and then we conclude $\Gamma \vdash^{\delta} !M[V/x]$.
   - $\Gamma, x : \delta' \vdash^{\delta} \mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2$, with $x \neq y$ and y not free in V is derived from $\Gamma, x : \delta' \vdash^{\delta} M_1$ and $\Gamma, x : \delta', y : (\delta + 1) \vdash^{\delta} M_2$. By inductive hypothesis, $\Gamma \vdash^{\delta} M_1[V/x]$ $\Gamma, y : (\delta + 1) \vdash^{\delta} M_2[V/x]$, and then we conclude $\Gamma \vdash^{\delta} (\mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2)[V/x]$.
   - $M \equiv \mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ M_1$. We have

     $$\frac{R, r : \delta; \Gamma, x : \delta', y : (\delta + 1) \vdash^{\delta} M_1}{R, r : \delta; \Gamma, x : \delta' \vdash^{\delta} \mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ M_1}$$

     By induction hypothesis we get

     $$R, r : \delta; \Gamma, y : (\delta + 1) \vdash^{\delta} M_1[V/x]$$

     and hence we derive

     $$R, r : \delta; \Gamma \vdash^{\delta} (\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ M_1)[V/x]$$

   - $M \equiv \mathsf{set}(r, V')$. We have

     $$\frac{R, r : \delta; \Gamma, x : \delta' \vdash^{\delta} V'}{R, r : \delta; \Gamma, x : \delta' \vdash^{\delta} \mathsf{set}(r, V')}$$

By induction hypothesis we get

$$R, r : \delta; \Gamma \vdash^{\delta} V'[V/x]$$

and hence we derive

$$R, r : \delta; \Gamma \vdash^{\delta} (\mathsf{set}(r, V'))[V/x]$$

- $M \equiv (M_1 \mid M_2)$. We have

$$\frac{R; \Gamma, x : \delta' \vdash^{\delta} M_i \quad i = 1, 2}{R; \Gamma, x : \delta' \vdash^{\delta} (M_1 \mid M_2)}$$

By induction hypothesis we derive

$$R; \Gamma \vdash^{\delta} M_i[V/x]$$

and hence we derive

$$R; \Gamma \vdash^{\delta} (M_1 \mid M_2)[V/x]$$

4. We proceed by case analysis on the reduction rules.

- Suppose $\Gamma \vdash^0 E[(\lambda x.M)V]$. Then for some Γ' extending Γ and $\delta \geq 0$ we must have $\Gamma' \vdash^{\delta} (\lambda x.M)V$. This must be derived from $\Gamma', x : \delta \vdash^{\delta} M$ and $\Gamma' \vdash^{\delta} V$. By (3), with $\delta = \delta'$, it follows that $\Gamma' \vdash^{\delta} M[V/x]$ and that the depth of an occurrence in $E[M[V/x]]$ is bounded by the depth of an occurrence which is already in $E[(\lambda x.M)V]$. Moreover, we can derive $\Gamma \vdash^0 E[M[V/x]]$.
- Suppose $\Gamma \vdash^0 E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M]$. Then for some Γ' extending Γ and $\delta \geq 0$ we must have $\Gamma' \vdash^{\delta} \mathsf{let}\ !x = !V\ \mathsf{in}\ M$. This must be derived from $\Gamma', x : (\delta + 1) \vdash^{\delta} M$ and $\Gamma' \vdash^{(\delta+1)} V$. By (3), with $(\delta + 1) = \delta'$, it follows that $\Gamma' \vdash^{\delta} M[V/x]$ and that the depth of an occurrence in $E[M[V/x]]$ is bounded by the depth of an occurrence which is already in $E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M]$. Moreover, we can derive $\Gamma \vdash^0 E[M[V/x]]$.
- $E[\mathsf{set}(r, V)] \to E[*] \mid (r \leftarrow V)$
  We have $R; \Gamma \vdash^0 E[\mathsf{set}(r, V)]$ from which we derive

$$\frac{R; \Gamma \vdash^{\delta} V}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, V)}$$

for some $\delta \geq 0$, with $r : \delta \in R$. Hence we can derive

$$\frac{R; \Gamma \vdash^{\delta} V}{R; \Gamma \vdash^{0} (r \leftarrow V)}$$

Moreover, we have as an axiom $R; \Gamma \vdash^{\delta} *$ thus we can derive $R; \Gamma \vdash^0 E[*]$. Applying the parallel rule we finally get

$$R; \Gamma \vdash^0 E[*] \mid (r \leftarrow V)$$

Concerning the depth bound, clearly we have $d(E[*] \mid (r \leftarrow V)) = d(E[\mathsf{set}(r, V)])$.

- $E[\mathsf{get}(r)] \mid (r \leftarrow V) \to E[M[V/x]]$
  We have $R; \Gamma \vdash^0 E[\mathsf{get}(r)] \mid (r \leftarrow V)$ from which we derive

$$\frac{}{R; \Gamma \vdash^\delta \mathsf{get}(r)}$$

  and

$$\frac{}{R; \Gamma, x : \delta \vdash^\delta M}$$

  for some $\delta \geq 0$, with $r : \delta \in R$, and

$$\frac{R; \Gamma \vdash^\delta V}{R; \Gamma \vdash^0 (r \leftarrow V)}$$

  Hence we can derive

$$R; \Gamma \vdash^0 E[V]$$

  Concerning the depth bound, clearly we have $d(E[V]) = d(E[\mathsf{get}(r)] \mid (r \leftarrow V))$.

- $E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V) \to E[M[V/x]] \mid (r \leftarrow !V)$
  We have $R; \Gamma \vdash^0 E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid \mathsf{r}!V$ from which we derive

$$\frac{R; \Gamma', x : (\delta + 1) \vdash^\delta M}{R; \Gamma' \vdash^\delta \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M}$$

  for some $\delta \geq 0$ with $r : \delta \in R$, and some Γ' extending Γ. We also derive

$$\frac{\dfrac{R; \Gamma \vdash^{\delta+1} V}{R; \Gamma \vdash^\delta !V}}{R; \Gamma \vdash^0 (r \leftarrow !V)}$$

  By (2) we get $R; \Gamma' \vdash^{\delta+1} V$. By (3) we derive

$$R; \Gamma' \vdash^\delta M[V/x]$$

  hence

$$R; \Gamma \vdash^0 E[M[V/x]]$$

  and finally

$$R; \Gamma \vdash^0 E[M[V/x]] \mid (r \leftarrow !V)$$

  Concerning the depth bound, by (3), the depth of an occurrence in $E[M[V/x]] \mid (r \leftarrow !V)$ is bounded by the depth of an occurrence which is already in $E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V)$, hence $d(E[M[V/x]] \mid (r \leftarrow !V)) \leq d(E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V))$.

A.2 Proof of proposition 3.6

We do this by case analysis on the reduction rules.

- $P = E[(\lambda x.M)V] \to P' = E[M[V/x]]$
  Let the occurrence of the redex $(\lambda x.M)V$ be at depth i. The restrictions on the formation of terms require that x occurs at most once in M at depth 0. Then $\omega_i(P) - 3 \geq \omega_i(P')$ because we remove the nodes for application and λ-abstraction and either V disappears or the occurrence of the variable x in M disappears (both being at the same depth as the redex). Clearly $\omega_j(P) = \omega_j(P')$ if $j \neq i$, hence

  $$\mu_n(P') \leq (\omega_n(P), \ldots, \omega_{i+1}(P), \omega_i(P) - 3, \mu_{i-1}(P)) \tag{A.1}$$

  and $\mu_n(P) > \mu_n(P')$.

- $P = E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] \to P' = E[M[V/x]]$
  Let the occurrence of the redex $\mathsf{let}\ !x = !V\ \mathsf{in}\ M$ be at depth i. The restrictions on the formation of terms require that x may only occur in M at depth 1 and hence in P at depth $i+1$. We have that $\omega_i(P') = \omega_i(P) - 2$ because the $\mathsf{let}\,!$ node disappear. Clearly, $\omega_j(P) = \omega_j(P')$ if $j < i$. The number of occurrences of x in M is bounded by $k = \omega_{i+1}(P) \geq 2$. Thus if $j > i$ then $\omega_j(P') \leq k \cdot \omega_j(P)$. Let's write, for $0 \leq i \leq n$:

  $$\mu_n^i(P) \cdot k = (\omega_n(P) \cdot k, \omega_{n-1}(P) \cdot k, \ldots, \omega_i(P) \cdot k)$$

  Then we have

  $$\mu_n(P') \leq (\mu_n^{i+1}(P) \cdot k, \omega_i(P) - 2, \mu_{i-1}(P)) \tag{A.2}$$

  and finally $\mu_n(P) > \mu_n(P')$.

- $P \equiv E[\mathsf{set}(r, V)] \to P' \equiv E[*] \mid (r \leftarrow V)$
  If $R; \Gamma \vdash^{\delta} \mathsf{set}(r, V)$ then by 3.5(4) we have $R; \Gamma \vdash^{0} (r \leftarrow V)$ with $R(r) = \delta$. Hence, by definition of the depth, the occurrences in V stay at depth δ in $(r \leftarrow V)$. Moreover, the node $\mathsf{set}(r, V)$ disappears and the nodes $*$, $\mid$, and $r \leftarrow$ appear. Recall that we assume the occurrences $\mid$ and $r \leftarrow$ do not count in the measure and that $\mathsf{set}(r)$ counts for two occurrences. Thus $\omega_\delta(P') = \omega_\delta(P) - 2 + 1 + 0 + 0$. The number of occurrences at other depths stay unchanged, hence $\mu_n(P) > \mu_n(P')$.

- $P \equiv E[\mathsf{get}(r)] \mid (r \leftarrow V) \to P' \equiv E[V]$
  If $R; \Gamma \vdash^{0} (r \leftarrow V)$ with $R(r) = \delta$, then $\mathsf{get}(r)$ must be at depth δ in $E[\ ]$. Hence, by definition of the depth, the occurrences in V stay at depth δ, while the node $\mathsf{get}(r)$ and $\mid$ disappear. Thus $\omega_\delta(P') = \omega_\delta(P) - 1$ and the number of occurrences at other depths stay unchanged, hence $\mu_n(P) > \mu_n(P')$.

- $P \equiv E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V) \to P' \equiv E[M[V/x]] \mid (r \leftarrow !V)$
  This case is the only source of duplication with the reduction rule on $\mathsf{let}\,!$. Suppose $R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M$. Then we must have $R; \Gamma \vdash^{\delta+1} V$. The restrictions on the formation of terms require that x may only occur in M at depth 1 and hence in P at depth $\delta+1$. Hence the occurrences in V stay at the same depth in $M[V/x]$, while the let, $\mathsf{get}(r)$ and some x nodes disappear, hence $\omega_\delta(P) \leq \omega_\delta(P') - 2$. The number of occurrences of x in M is bounded by $k = \omega_{\delta+1}(P) \geq 2$. Thus if $j > \delta$ then $\omega_j(P') \leq k \cdot \omega_j(P)$. Clearly, $\omega_j(M) = \omega_j(M')$ if $j < i$. Hence, we have

$$\mu_n(P') \leq (\mu_n^{i+1}(P) \cdot k, \omega_i(P) - 2, \mu_{i-1}(P)) \tag{A.3}$$

  and $\mu_n(P) > \mu_n(P')$.

A.3 Proof of lemma 2.8

We start by remarking some basic inequalities.

Lemma A.1 (some inequalities)**.** *The following properties hold on natural numbers.*

1. $\forall\, x \geq 2, y \geq 0\ \ (y+1) \leq x^y$
2. $\forall\, x \geq 2, y \geq 0\ \ (x \cdot y) \leq x^y$
3. $\forall\, x \geq 2, y, z \geq 0\ \ (x \cdot y)^z \leq x^{(y \cdot z)}$
4. $\forall\, x \geq 2, y \geq 0, z \geq 1\ \ x^z \cdot y \leq x^{(y \cdot z)}$
5. *If* $x \geq y \geq 0$ *then* $(x-y)^k \leq (x^k - y^k)$

Proof. 1. By induction on y. The case for $y = 0$ is clear. For the inductive case, we notice:

$$(y+1) + 1 \leq 2^y + 2^y = 2^{y+1} \leq x^{y+1}\ .$$

2. By induction on y. The case $y = 0$ is clear. For the inductive case, we notice:

$$\begin{array}{rcll} x \cdot (y+1) & \leq & x \cdot (x^y) & \text{(by (1))} \\ & = & x^{(y+1)} & \end{array}$$

3. By induction on z. The case $z = 0$ is clear. For the inductive case, we notice:

$$\begin{array}{rcll} (x \cdot y)^{z+1} & = & (x \cdot y)^z (x \cdot y) & \\ & \leq & x^{y \cdot z}(x \cdot y) & \text{(by inductive hypothesis)} \\ & \leq & x^{y \cdot z}(x^y) & \text{(by (2))} \\ & = & x^{y \cdot (z+1)} & \end{array}$$

4. From $z \geq 1$ we derive $y \leq y^z$. Then:

$$\begin{array}{rcll} x^z \cdot y & \leq & x^z \cdot y^z & \\ & = & (x \cdot y)^z & \\ & \leq & x^{y \cdot z} & \text{(by (3))} \end{array}$$

5. By the binomial law, we have $x^k = ((x-y)+y)^k = (x-y)^k + y^k + p$ with $p \geq 0$. Thus $(x-y)^k = x^k - y^k - p$ which implies $(x-y)^k \leq x^k - y^k$.

□

We also need the following property.

Lemma A.2 (pre-shift)**.** *Assuming $\alpha \geq 1$ and $\beta \geq 2$, the following property holds for the tower functions with $x, \mathbf{x}$ ranging over numbers greater or equal than 2:*

$$\beta \cdot t_\alpha(x, \mathbf{x}) \leq t_\alpha(\beta \cdot x, \mathbf{x})$$

Proof. This follows from:

$$\beta \leq \beta^{2^{t_\alpha(\mathbf{x})}}$$

□

Then we can derive the proof of the shift lemma as follows.
Let $k = t_\alpha(x', \mathbf{x}) \geq 2$. Then

$$\begin{array}{rll} t_\alpha(\beta \cdot x, x', \mathbf{x}) = \beta \cdot (\alpha \cdot x)^{2^k} & \leq (\alpha \cdot x)^{\beta \cdot 2^k} & \text{(by lemma A.1(3))} \\ & \leq (\alpha \cdot x)^{(\beta \cdot 2)^k} & \\ & \leq (\alpha \cdot x)^{2^{(\beta \cdot k)}} & \text{(by lemma A.1(3))} \end{array}$$

and by lemma A.2 $\beta \cdot t_\alpha(x', \mathbf{x}) \leq t_\alpha(\beta \cdot x', \mathbf{x})$.
Hence

$$(\alpha \cdot x)^{2^{(\beta \cdot k)}} \leq (\alpha \cdot x)^{2^{t_\alpha(\beta \cdot x', \mathbf{x})}} = t_\alpha(x, \beta \cdot x', \mathbf{x})$$

A.4 Proof of theorem 3.7

Suppose $\mu_\alpha(P) = (x_0, \ldots, x_\alpha)$ so that x_i corresponds to the occurrences at depth $(\alpha - i)$ for $0 \leq i \leq \alpha$. Also assume the reduction is at depth $(\alpha - i)$. By looking at equations (A.1) and (A.2) in the proof of termination (Proposition 3.6), we see that the components $i+1, \ldots, \alpha$ of $\mu_\alpha(P)$ and $\mu_\alpha(P')$ coincide. Hence, let $k = 2^{t_\alpha(x_{i+1}, \ldots, x_\alpha)}$. By definition of the tower function, $k \geq 1$.

We proceed by case analysis on the reduction rules.

- $P \equiv \mathsf{let}\ !x = !V\ \mathsf{in}\ M \to P' \equiv M[V/x]$
  By inequality (A.2) we know that:

$$\begin{array}{rcl} t_\alpha(\mu_\alpha(P')) & \leq & t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2, x_{i+1}, \ldots, x_\alpha) \\ & = & t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2)^k \end{array}$$

By iterating lemma 2.8, we derive:

$$\begin{array}{ll} & t_\alpha(x_0 \cdot x_{i-1}, x_1 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2) \\ \leq & t_\alpha(x_0, x_1 \cdot x_{i-1}^2, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2) \\ \leq & \ldots \\ \leq & t_\alpha(x_0, x_1, \ldots, x_{i-1}^i, x_i - 2) \end{array}$$

Renaming x_{i-1} with x and x_i with y, we are left to show that:

$$(\alpha x^i)^{2^{(\alpha \cdot (y-2))^k}} < (\alpha x)^{2^{(\alpha \cdot y)^k}}$$

Since $i \leq \alpha$ the first quantity is bounded by:

$$(\alpha x)^{\alpha \cdot 2^{(\alpha \cdot (y-2))^k}}$$

We notice:

$$\begin{array}{lll} & \alpha \cdot 2^{(\alpha \cdot (y-2))^k} & \\ = & \alpha \cdot 2^{(\alpha \cdot y - \alpha \cdot 2)^k} & \\ \leq & \alpha \cdot 2^{(\alpha \cdot y)^k - (\alpha \cdot 2)^k} & \text{(by lemma A.1(5))} \end{array}$$

So we are left to show that:

$$\alpha 2^{(\alpha \cdot y)^k - (\alpha \cdot 2)^k)} \leq 2^{(\alpha \cdot y)^k}$$

Dividing by $2^{(\alpha \cdot y)^k}$ and recalling that $k \geq 1$, it remains to check:

$$\alpha \cdot 2^{-(\alpha \cdot 2)^k} \leq \alpha \cdot 2^{-(\alpha \cdot 2)} < 1$$

which is obviously true for $\alpha \geq 1$.

- $P \equiv (\lambda x.M)V \to P' \equiv M[V/x]$
  By equation (A.1), we have that:

$$t_\alpha(\mu_\alpha(P')) \leq t_\alpha(x_0, \ldots, x_{i-1}, x_i - 2, x_{i+1}, \ldots, x_\alpha)$$

and one can check that this quantity is strictly less than:

$$t_\alpha(\mu_\alpha(P)) = t_\alpha(x_0, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_\alpha)$$

- $P \equiv \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M \mid (r \leftarrow !V) \to P' \equiv M[V/x] \mid (r \leftarrow !V)$
  Let $k = 2^{t_\alpha(x_{i+1}, \ldots, x_\alpha)}$. By definition of the tower function, $k \geq 1$. By equation (A.3) we have

$$\begin{array}{ll} t_\alpha(\mu_\alpha(P')) & \leq t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2, x_{i+1}, \ldots, x_\alpha) \\ & = t_\alpha(x_0 \cdot x_{i-1}, \ldots, x_{i-1} \cdot x_{i-1}, x_i - 2)^k \end{array}$$

And we end up in the case of the rule for $\mathsf{let}\,!$.

- For the read that consume a value from the store, by looking at the proof of termination, we see that exactly one element of the vector $\mu_\alpha(P)$ is strictly decreasing during the reduction, hence one can check that $t_\alpha(\mu_\alpha(P)) > t_\alpha(\mu_\alpha(P'))$.
- The case for the write is similar to the read.

We conclude with the following remark that shows that the size of a program is proportional to its number of occurrences.

Remark A.3. The size of a program $|P|$ of depth d is at most twice the sum of its occurrences: $|P| \leq 2 \cdot \sum_{0 \leq i \leq d} \omega_i(P)$.

Hence the size of a program P is bounded by $t_d(\mu_d(P))$.

A.5 Proof of proposition 4.1

By induction on A.

- $A \equiv t'$
  We have
  $$\frac{R \vdash t' \qquad t \notin R}{R \vdash \forall t.t'}$$
  If $t \neq t'$ we have $t'[B/t] \equiv t'$ hence $R \vdash [B/t]t'$. If $t \equiv t'$ then we have $t'[B/t] \equiv B$ hence $R \vdash t'[B/t]$.
- $A \equiv \mathbf{1}$
  We have
  $$\frac{R \vdash \mathbf{1} \qquad t \notin R}{R \vdash \forall t.\mathbf{1}}$$
  from which we deduce $R \vdash \mathbf{1}[B/t]$.
- $A \equiv (C \multimap D)$
  By induction hypothesis we have $R \vdash C[B/t]$ and $R \vdash D[B/t]$. We then derive
  $$\frac{R \vdash C[B/t] \qquad R \vdash D[B/t]}{R \vdash (C \multimap D)[B/t]}$$
- $A \equiv !C$
  By induction hypothesis we have $R \vdash C[B/t]$, from which we deduce
  $$\frac{R \vdash C[B/t]}{R \vdash !C[B/t]}$$
- $A \equiv \mathsf{Reg}_r C$
  We have
  $$\frac{\dfrac{R \vdash \qquad r : C \in R}{R \vdash \mathsf{Reg}_r C} \qquad t \notin R}{R \vdash \forall t.\mathsf{Reg}_r C}$$

As $t \notin R$ and $r : (\delta, C) \in R$, we have $r : (\delta, C[B/t]) \in R$, from which we deduce

$$\frac{R \vdash \quad r : (\delta, C[B/t] \in R}{R \vdash \mathsf{Reg}_r C[B/t]}$$

- $A \equiv \forall t'.C$
  If $t \neq t'$: From $R \vdash \forall t.(\forall t'.C)$ we have $t' \notin R$ and by induction hypothesis we have $R \vdash C[B/t]$, from which we deduce

$$\frac{R \vdash C[B/t] \quad t' \notin R}{R \vdash (\forall t'.C)[B/t]}$$

  If $t \equiv t'$ we have $(\forall t'.C)[B/t] \equiv \forall t'.C$. Since we have

$$\frac{R \vdash \forall t'.C \quad t \notin R}{R \vdash \forall t.(\forall t'.C)}$$

  we conclude $R \vdash (\forall t'.C)[B/t]$.

A.6 Proof of theorem 4.4

Properties 1 and 2 are easily checked.

A.6.1 Substitution

If x is not free in M, we just have to check that any proof of $\Gamma, x : (\delta', A) \vdash^{\delta} M$ can be transformed into a proof of $\Gamma \vdash^{\delta} M$.

So let us assume x is free in M. Next, we proceed by induction on the derivation of $\Gamma, x : \delta' \vdash^{\delta} M$.

- $\Gamma, x : (\delta, A) \vdash^{\delta} x : A$. Then $\delta = \delta'$, $x[V/x] = V$, and by hypothesis $\Gamma \vdash^{\delta} V : A$.
- $\Gamma, x : (\delta', A) \vdash^{\delta} \lambda y.M : B \multimap C$ is derived from $\Gamma, x : (\delta', A), y : (\delta, B) \vdash^{\delta} M : C$, with $x \neq y$ and y not occurring in V. By (2), $\Gamma, y : (\delta, B) \vdash^{\delta'} V : A$. By inductive hypothesis, $\Gamma, (y : \delta, B) \vdash^{\delta} M[V/x] : C$, and then we conclude $\Gamma \vdash^{\delta} (\lambda y.M)[V/x] : B \multimap C$.
- $\Gamma, x : (\delta', A) \vdash^{\delta} (M_1 M_2) : C$ is derived from $\Gamma, x : (\delta', A) \vdash^{\delta} M_1 : B \multimap C$ and $\Gamma, x : (\delta', A) \vdash^{\delta} M_1 : B \multimap C$. By inductive hypothesis, $\Gamma \vdash^{\delta} M_1[V/x] : B \multimap C$ and $\Gamma \vdash^{\delta} M_2[V/x] : C$, and then we conclude $\Gamma \vdash^{\delta} (M_1 M_2)[V/x] : C$.
- $\Gamma, x : (\delta', A) \vdash^{\delta} !M : !B$ is derived from $\Gamma, x : (\delta', A) \vdash^{\delta+1} M : B$. By inductive hypothesis, $\Gamma \vdash^{\delta+1} M[V/x] : B$, and then we conclude $\Gamma \vdash^{\delta} !M[V/x] : !B$.
- $\Gamma, x : (\delta', A) \vdash^{\delta} \mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2 : B$, with $x \neq y$ and y not free in V is derived from $\Gamma, x : (\delta', A) \vdash^{\delta} M_1 : C$ and $\Gamma, x : (\delta', A), y : (\delta + 1, C) \vdash^{\delta} M_2 : B$. By inductive hypothesis, $\Gamma \vdash^{\delta} M_1[V/x] : C$ $\Gamma, y : (\delta + 1, C) \vdash^{\delta} M_2[V/x] : B$, and then we conclude $\Gamma \vdash^{\delta} (\mathsf{let}\ !y = M_1\ \mathsf{in}\ M_2)[V/x] : B$.

- $M \equiv \mathsf{get}(r)$. We have $R, r : (\delta, B); \Gamma, x : (\delta', A) \vdash^{\delta} \mathsf{get}(r) : B$. Since $\mathsf{get}(r)[V/x] = \mathsf{get}(r)$ and $x \notin \mathsf{FV}(\mathsf{get}(r))$ then $R, r : (\delta, B); \Gamma \vdash^{\delta} \mathsf{get}(r)[V/x] : B$.

- $M \equiv \mathsf{set}(r, V')$. We have

$$\frac{R, r : (\delta, C); \Gamma, x : (\delta', A) \vdash^{\delta} V' : C}{R, r : (\delta, C); \Gamma, x : (\delta', A) \vdash^{\delta} \mathsf{set}(r, V') : \mathbf{1}}$$

  By induction hypothesis we get

$$R, r : (\delta, C); \Gamma \vdash^{\delta} V'[V/x] : C$$

  and hence we derive

$$R, r : (\delta, C); \Gamma \vdash^{\delta} (\mathsf{set}(r, V'))[V/x] : \mathbf{1}$$

- $M \equiv (M_1 \mid M_2)$. We have

$$\frac{R; \Gamma, x : (\delta', A) \vdash^{\delta} M_i : C_i \qquad i = 1, 2}{R; \Gamma, x : (\delta', A) \vdash^{\delta} (M_1 \mid M_2) : \mathbf{B}}$$

  By induction hypothesis we derive

$$R; \Gamma \vdash^{\delta} M_i[V/x] : C_i$$

  and hence we derive

$$R; \Gamma \vdash^{\delta} (M_1 \mid M_2)[V/x] : \mathbf{B}$$

A.6.2 Subject Reduction

We first state and sketch the proof of 4 lemmas.

Lemma A.4 (structural equivalence preserves typing)**.** *If $R; \Gamma \vdash^{\delta} P : \alpha$ and $P \equiv P'$ then $R; \Gamma \vdash^{\delta} P' : \alpha$.*

Proof. Recall that structural equivalence is the least equivalence relation induced by the equations stated in Table 3.2 and closed under static contexts. Then we proceed by induction on the proof of structural equivalence. This is is mainly a matter of reordering the pieces of the typing proof of P so as to obtain a typing proof of P'. □

Lemma A.5 (evaluation contexts and typing)**.** *Suppose that in the proof of $R; \Gamma \vdash^{\delta} E[M] : \alpha$ we prove $R; \Gamma' \vdash^{\delta'} M : \alpha'$. Then replacing M with a M' such that $R; \Gamma' \vdash^{\delta'} M' : \alpha'$, we can still derive $R; \Gamma \vdash^{\delta} E[M'] : \alpha$.*

Proof. By induction on the structure of E. □

Lemma A.6 (functional redexes)**.** *If* $R;\Gamma \vdash^{\delta} E[\Delta] : \alpha$ *where* Δ *has the shape* $(\lambda x.M)V$ *or* $\mathsf{let}\ !x = !V\ \mathsf{in}\ M$ *then* $R;\Gamma \vdash^{\delta} E[M[V/x]] : \alpha$.

Proof. We appeal to the substitution lemma 3. This settles the case where the evaluation context E is trivial. If it is complex then we also need lemma A.5. □

Lemma A.7 (side effects redexes)**.** *If* $R;\Gamma \vdash^{\delta} \Delta : \alpha$ *where* Δ *is one of the programs on the left-hand side then* $R;\Gamma \vdash^{\delta} \Delta' : \alpha$ *where* Δ' *is the corresponding program on the right-hand side:*

(1)	$E[\mathsf{set}(r,V)]$	$E[*] \mid (r \leftarrow V)$
(2)	$E[\mathsf{get}(r)] \mid (r \leftarrow V)$	$E[V]$
(3)	$E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] \mid (r \leftarrow !V)$	$E[M[V/x]] \mid (r \leftarrow !V)$

Proof. We proceed by case analysis.

1. Suppose we derive $R;\Gamma \vdash^{\delta} E[\mathsf{set}(r,V)] : \alpha$ from $R;\Gamma' \vdash^{\delta'} \mathsf{set}(r,V) : \mathbf{1}$. We can derive $R;\Gamma' \vdash^{\delta'} * : \mathbf{1}$ and by Lemma A.5 we derive $R;\Gamma \vdash^{\delta} E[*] : \alpha$ and finally $R;\Gamma \vdash^{\delta} E[\mathsf{set}(r,V)] \mid (r \leftarrow V) : \alpha$.

2. Suppose $R;\Gamma \vdash^{\delta} E[\mathsf{get}(r)] : \alpha$ is derived from $R;\Gamma \vdash^{\delta'} \mathsf{get}(r) : A$, where $r : (\delta', A) \in R$. Hence $R;\Gamma \vdash^{0} (r \leftarrow V) : \mathbf{B}$ is derived from $R;\Gamma \vdash^{\delta'} V : A$. Finally, by Lemma A.5 we derive $R;\Gamma \vdash^{\delta} E[V] : \alpha$.

3. Suppose $R;\Gamma \vdash^{\delta} E[\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M] : \alpha$ is derived from

$$\frac{R;\Gamma' \vdash^{\delta'} \mathsf{get}(r) : !A \qquad R;\Gamma', x : (\delta'+1, A) \vdash^{\delta'} M : \alpha'}{R;\Gamma' \vdash^{\delta'} \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M : \alpha'}$$

   where $r : (\delta', !A) \in R$. Hence $R;\Gamma \vdash^{0} (r \leftarrow !V) : \mathbf{B}$ is derived from $R;\Gamma \vdash^{\delta'+1} V : A$. By Lemma 3 we can derive $R;\Gamma' \vdash^{\delta'} M[V/x] : \alpha'$. Then by Lemma A.5 we derive $R;\Gamma \vdash^{\delta} E[M[V/x]] : \alpha$.

□

We are then ready to prove subject reduction. We recall that $P \to P'$ means that P is structurally equivalent to a program $C[\Delta]$ where C is a static context, Δ is one of the programs on the left-hand side of the rewriting rules specified in Table 3.2, Δ' is the respective program on the right-hand side, and P' is syntactically equal to $C[\Delta']$.

By lemma A.4, we know that $R;\Gamma \vdash^{\delta} C[\Delta] : \alpha$. This entails that $R';\Gamma' \vdash^{\delta'} \Delta : \alpha'$ for suitable $R', \Gamma', \alpha', \delta'$. By lemmas A.6 and A.7, we derive that $R';\Gamma' \vdash^{\delta'} \Delta' : \alpha'$. Then by induction on the structure of C we argue that $R;\Gamma \vdash^{\delta} C[\Delta'] : \alpha$.

A.6.3 Progress

To derive the progress property we first determine for each closed type A where $A = A_1 \multimap A_2$ or $A = !A_1$ the shape of a closed value of type A with the following classification lemma.

Lemma A.8 (classification)**.** *Assume* $R; - \vdash^{\delta} V : A$*. Then:*

- *if* $A = A_1 \multimap A_2$ *then* $V = \lambda x.M$*,*
- *if* $A = !A_1$ *then* $V = !V_1$

Proof. By case analysis on the typing rules.

- if $A = A_1 \multimap A_2$, the only typing rule that can be applied is

$$\frac{R; x : (\delta, A_1) \vdash^{\delta} M : A_2}{R; - \vdash^{\delta} \lambda x.M : A_1 \multimap A_2}$$

  hence $V = \lambda x.M$.

- if $A = !A_1$, the only typing rule that can be applied is

$$\frac{R; - \vdash^{\delta+1} V_1 : A_1}{R; - \vdash^{\delta} !V_1 : !A_1}$$

  hence $V = !V_1$.

□

Then we proceed by induction on the structure of the threads M_i to show that each one of them is either a value or a stuck get of the form $E[\Delta]$ where Δ can be $(\lambda x.M)\mathsf{get}(r)$ or $\mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ M$.

- $M_i = x$
  the case of variables is void since they are not closed terms.
- $M_i = *$ or $M_i = r$ or $M_i = \lambda x.M$
  these cases are trivial since $*$, r and $\lambda x.M$ are already values.
- $M_i = PQ$
  We know that PQ cannot reduce, which by looking at the evaluation contexts means that P cannot reduce. Then by induction hypothesis we have two cases: either P is a value or P is a stuck get.
  - assume P is a value. We have

$$\frac{R; - \vdash^{\delta} P : A \multimap B \qquad R; - \vdash^{\delta} Q : A}{R; - \vdash^{\delta} PQ : B}$$

    By Lemma A.8 we have $P = \lambda x.M$. Since PQ cannot reduce and $P = \lambda x.M$, by looking at the evaluation contexts we have that Q cannot reduce. Moreover Q cannot be a value, otherwise PQ is a redex. Hence by induction hypothesis Q is a stuck get of the form $E_1[\Delta]$. Hence PQ is of the form $E[\Delta]$ where $E = PE_1$.

– assume P is a stuck get of the form $E_1[\Delta]$. Then PQ is of the form $E[\Delta]$ where $E = E_1 Q$.

- $M_i = \mathsf{let}\ !x = P\ \mathsf{in}\ Q$
  We know that $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ cannot reduce, which by looking at the evaluation contexts means that P cannot reduce. Then by induction hypothesis we have two cases: either P is a value or P is a stuck get.

  – assume P is a value. We have

  $$\frac{R; - \vdash^{\delta} P : !A \qquad R; x : (\delta + 1, A) \vdash^{\delta} Q : B}{R; - \vdash^{\delta} \mathsf{let}\ !x = P\ \mathsf{in}\ Q : B}$$

  By Lemma A.8 we have $P = !V$ hence $\mathsf{let}\ !x = !V\ \mathsf{in}\ Q$ is a redex and this contradicts the hypothesis that $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ cannot reduce. Thus P cannot be a value.

  – assume P is a stuck get of the form $E_1[\Delta]$. Then $\mathsf{let}\ !x = P\ \mathsf{in}\ Q$ is of the form $E[\Delta]$ where $E = \mathsf{let}\ !x = E_1\ \mathsf{in}\ Q$.

- $M_i = !P$
  We know that $!P$ cannot reduce, which by looking at the evaluation contexts means that P cannot reduce. Then by induction hypothesis we have two cases: either P is a value or P is a stuck get.

  – assume P is a value. Then $!P$ is also a value and we are done.

  – assume P is of the form $E_1[\Delta]$. Then $!P$ is of the shape $E[\Delta]$ where $E = !E_1$.

- $M_i = \mathsf{get}(r')$
  We know that $\mathsf{get}(r')$ cannot reduce which means that M_i is of the form $E[\Delta]$ where $r' = r$ and $E = []$ and that no value is associated with r in the store.

- $M_i = \mathsf{set}(r, V)$
  This case is void since $\mathsf{set}(r, V)$ can reduce is any case.

A.7 Proof of theorem 5.3

Elementary functions are characterized as the smallest class of functions containing zero, successor, projection, subtraction and which is closed by composition and bounded summation/product. We will need the arithmetic functions defined in Table A.1. We will abbreviate $\lambda^{!}x.M$ for $\lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ M$. Moreover, in order to represent some functions, we need to manipulate pairs in the language. We define the representation of pairs in Table A.2. In the following, we show that the required functions can be represented in the sense of Definition 5.2 by adapting the proofs from Danos and Joinet [10].

$$
\begin{array}{rcll}
\mathsf{N} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) & \text{(type of numerals)} \\
\\
\mathsf{zero} & : & \mathsf{N} & \text{(zero)} \\
\mathsf{zero} & = & \lambda f.!(\lambda x.x) & \\
\\
\mathsf{succ} & : & \mathsf{N} \multimap \mathsf{N} & \text{(successor)} \\
\mathsf{succ} & = & \lambda n.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y = n!f\ \mathsf{in} !(\lambda x.f(yx)) & \\
\\
\overline{n} & : & \mathsf{N} & \text{(numerals)} \\
\overline{n} & = & \lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ !(\lambda x.f(\cdots(fx)\cdots)) & \\
\\
\mathsf{add} & : & \mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{N}) & \text{(addition)} \\
\mathsf{add} & = & \lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y = n!f\ \mathsf{in} & \\
 & & \mathsf{let}\ !y' = m!f\ \mathsf{in}\quad !(\lambda x.y(y'x)) & \\
\\
\mathsf{mult} & : & \mathsf{N} \multimap (\mathsf{N} \multimap \mathsf{N}) & \text{(multiplication)} \\
\mathsf{mult} & = & \lambda n.\lambda m.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in} & \\
 & & n(m!f) & \\
\\
\mathsf{int\_it} & : & \mathsf{N} \multimap \forall t.!(t \multimap t) \multimap !t \multimap !t & \text{(iteration)} \\
\mathsf{int\_it} & = & \lambda n.\lambda g.\lambda x.\mathsf{let}\ !y = ng\ \mathsf{in} & \\
 & & \mathsf{let}\ !y' = x\ \mathsf{in}\ !(yy') & \\
\\
\mathsf{int\_git} & : & \forall t.\forall t'.!(t \multimap t) \multimap (!(t \multimap t) \multimap t') \multimap \mathsf{N} \multimap t' & \\
\mathsf{int\_git} & = & \lambda s.\lambda e.\lambda n.e(nts) &
\end{array}
$$

Table A.1: Representation of some arithmetic functions

A.7.1 Successor, addition and multiplication

We check that succ represents the *successor* function s:

$$
\begin{array}{l}
s : \mathbb{N} \mapsto \mathbb{N} \\
s(x) = x + 1
\end{array}
$$

Proposition A.9. $\mathsf{succ} \Vdash s$.

Proof. Take $\emptyset \vdash^{\delta} M : \mathsf{N}$ and $M \Vdash n$. We have $\emptyset \vdash^{\delta} \mathsf{succ} : \mathsf{N} \multimap \mathsf{N}$. We can show that $\mathsf{succ}\, M \xrightarrow{*} \overline{s(n)}$, hence $\mathsf{succ}\, M \Vdash s(n)$. Thus $\mathsf{succ} \Vdash s$. □

We check that add represents the *addition* function a:

$$
\begin{array}{l}
a : \mathbb{N}^2 \mapsto \mathbb{N} \\
a(x, y) = x + y
\end{array}
$$

$$
\begin{array}{rcll}
A \times B & = & \forall t.(A \multimap B \multimap t) \multimap t & \text{(type of pairs)} \\
\\
\langle M, N \rangle & : & A \times B & \text{(pair representation)} \\
\langle M, N \rangle & = & \lambda x.xMN & \\
\\
\mathsf{fst} & : & \forall t, t'.t \times t' \multimap t & \text{(left destructor)} \\
\mathsf{fst} & = & \lambda p.p(\lambda x.\lambda y.x) & \\
\\
\mathsf{snd} & : & \forall t, t'.t \times t' \multimap t' & \text{(right destructor)} \\
\mathsf{snd} & = & \lambda p.p(\lambda x.\lambda y.y) &
\end{array}
$$

Table A.2: Representation of pairs

Proposition A.10. $\mathsf{add} \Vdash a$.

Proof. For $i = 1, 2$ take $\emptyset \vdash^\delta M_i : \mathsf{N}$ and $M_i \Vdash n_i$. We have $\emptyset \vdash^\delta \mathsf{add} : \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$. We can show that $\mathsf{add} M_1 M_2 \xrightarrow{*} \overline{a(n_1, n_2)}$, hence $\mathsf{add} M_1 M_2 \Vdash a(n_1, n_2)$. Thus $A \Vdash a$. □

We check that mult represents the multiplication function m:

$$
\begin{array}{l}
m : \mathbb{N}^2 \mapsto \mathbb{N} \\
m(x, y) = x * y
\end{array}
$$

Proposition A.11. $\mathsf{mult} \Vdash m$.

Proof. For $i = 1, 2$ take $\emptyset \vdash^\delta M_i : \mathsf{N}$ and $M_i \Vdash n_i$. We have $\emptyset \vdash^\delta \mathsf{mult} : \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$. We can show that $\mathsf{mult}\ M_1 M_2 \xrightarrow{*} \overline{m(n_1, n_2)}$, hence $\mathsf{mult}\ M_1 M_2 \Vdash m(n_1, n_2)$. Thus $\mathsf{mult} \Vdash m$. □

A.7.2 Iteration schemes

We check that $\mathsf{int\_it}$ represents the following iteration function it:

$$
\begin{array}{l}
it : (\mathbb{N} \mapsto \mathbb{N}) \mapsto \mathbb{N} \mapsto \mathbb{N} \mapsto \mathbb{N} \\
it(f, n, x) = f^n(x)
\end{array}
$$

Proposition A.12. $\mathsf{int\_it} \Vdash it$.

Proof. We have $\emptyset \vdash^\delta \mathsf{int\_it} : \mathsf{N} \multimap \forall t.!(t \multimap t) \multimap !t \multimap !t$. Given $\emptyset \vdash^\delta M : \mathsf{N}$ with $M \Vdash n$, $\emptyset \vdash^\delta F : \mathsf{N} \multimap \mathsf{N}$ with $F \Vdash f$ and $\emptyset \vdash^\delta X : \mathsf{N}$ with $X \Vdash x$, we observe that $\mathsf{int\_it}\ M(!F)(!X) \xrightarrow{*} F^n X$. Since $F \Vdash f$ and $X \Vdash x$, we get $F^n X \xrightarrow{*} \overline{it(f, n, x)}$. Hence $\mathsf{int\_it} \Vdash it$. □

The function it is an instance of the more general iteration scheme git:

$$
\begin{array}{l}
git : (\mathbb{N} \mapsto \mathbb{N}) \mapsto ((\mathbb{N} \mapsto \mathbb{N}) \mapsto \mathbb{N}) \mapsto \mathbb{N} \mapsto \mathbb{N} \\
git(step, exit, n) = exit(\lambda x.step^n(x))
\end{array}
$$

Indeed, we have:

$$git(f, \lambda f.fx, n) = (\lambda f.fx)(\lambda x.f^n(x)) = it(f, n, x)$$

Proposition A.13. $\mathsf{int\_git} \Vdash git$.

Proof. Take $\emptyset \vdash^\delta M : \mathsf{N}$ with $M \Vdash n$, $\emptyset \vdash^\delta E : ((\mathsf{N} \multimap \mathsf{N}) \multimap \mathsf{N}) \multimap \mathsf{N}$ with $E \Vdash exit$, $\emptyset \vdash^\delta S : \mathsf{N} \multimap \mathsf{N}$ with $S \Vdash step$. Then we have $\mathsf{int\_git}\, S\, E\, M \xrightarrow{*} E(\lambda x.S^n x)$. Since $S \Vdash step$ and $E \Vdash exit$ we have $E(\lambda x.S^n x) \xrightarrow{*} \overline{exit(\lambda x.step^n(x)}$. Hence $\mathsf{int\_git} \Vdash git$. □

A.7.3 Coercion

Let $S = \lambda n^N.S'$. For $0 \geq i$, we define S'_i inductively:

$$\begin{array}{l} S'_0 = S' \\ S'_i = \mathsf{let}\ !n = n\ \mathsf{in}\ !S'_{i-1} \end{array}$$

Let $S_i = \lambda n.S'_i$. We can derive $\emptyset \vdash^\delta S_i : !^i\mathsf{N} \multimap !^i\mathsf{N}$. For $i \geq 0$, we define C_i inductively:

$$\begin{array}{l} C_0 = \lambda x.x \\ C_{i+1} = \lambda n.\mathsf{int\_it}(!S_i)(!^{i+1}\overline{0})n \\ \emptyset \vdash^\delta C_i : \mathsf{N} \multimap !^i\mathsf{N} \end{array}$$

Lemma A.14 (integer representation is preserved by coercion)**.** *Let $\emptyset \vdash^\delta M : \mathsf{N}$ and $M \Vdash n$. We can derive $\emptyset \vdash^\delta C_i M : !^i\mathsf{N}$. Moreover $C_i M \Vdash n$.*

Proof. By induction on i. □

Lemma A.15 (function representation is preserved by coercion)**.** *Let*

$$\emptyset \vdash^\delta F : !^{i_1}\mathsf{N}_1 \multimap \ldots \multimap !^{i_k}\mathsf{N}_k \multimap !^p\mathsf{N}$$

and $\emptyset \vdash^\delta M_j : \mathsf{N}$ with $M_j \Vdash n_j$ for $1 \leq j \leq k$ such that $F(!^{i_1}M_1 \ldots (!^{i_k}M_k)) \xrightarrow{*} \overline{f(n_1, \ldots, n_k)}$. Then we can find a term $\mathcal{C}(F) = \lambda \vec{x}^{\mathsf{N}}.F((C_{i_1}x_1)\ldots(C_{i_k}x_k))$ such that

$$\emptyset \vdash^\delta \mathcal{C}(F) : \mathsf{N} \multimap \mathsf{N} \multimap \ldots \multimap \mathsf{N} \multimap !^p\mathsf{N}$$

and $\mathcal{C}(F) \Vdash f$.

A.7.4 Predecessor and subtraction

We first want to represent *predecessor*:

$$\begin{array}{l} p : \mathbb{N} \mapsto \mathbb{N} \\ p(0) = 0 \\ p(x) = x - 1 \end{array}$$

We define the following terms:

$$ST = !(\lambda z.\langle \mathsf{snd}\ z, f(\mathsf{snd}\ z)\rangle)$$
$$f : (\delta + 1, t \multimap t) \vdash^{\delta} ST : !(t \times t \multimap t \times t)$$

$$EX = \lambda g.\mathsf{let}\ !g = g\ \mathsf{in}\ !(\lambda x.\mathsf{fst}\ g\langle x, x\rangle)$$
$$\emptyset \vdash^{\delta} EX : !(t \times t \multimap t \times t) \multimap !(t \multimap t)$$

$$P = \lambda n.\lambda f.\mathsf{let}\ !f = f\ \mathsf{in}\ \mathsf{int\_git}\ ST\ EX\ n$$
$$\emptyset \vdash^{\delta} P : \mathsf{N} \multimap \mathsf{N}$$

Proposition A.16 (predecessor is representable)**.** $P \Vdash p$.

Proof. Take $\emptyset \vdash^{\delta} M : \mathsf{N}$ and $M \Vdash n$. We can show that $(PM)^{-} \xrightarrow{*} \overline{p(n)}$, hence $PM \Vdash p(n)$. Thus $P \Vdash p$. □

Now we want to represent (positive) subtraction s:

$$s : \mathbb{N}^2 \mapsto \mathbb{N}$$
$$s(x, y) = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } y \geq x \end{cases}$$

Take

$$SUB = \lambda m.\mathsf{let}\ !m = m\ \mathsf{in}\ \lambda n.\mathsf{int\_it}\ !P\ !m\ n : !\mathsf{N} \multimap \mathsf{N} \multimap !\mathsf{N}$$
$$\emptyset \vdash^{\delta} SUB : !\mathsf{N} \multimap \mathsf{N} \multimap !\mathsf{N}$$

Proposition A.17 (subtraction is representable)**.** $\mathcal{C}(SUB) \Vdash s$.

Proof. For $i = 1, 2$ take $\emptyset \vdash^{\delta} M_i : \mathsf{N}$ and $M_i \Vdash n_i$. We can show that $(SUB(!M_1)M_2)^{-} \xrightarrow{*} \overline{s(n_1, n_2)}$. Hence by Lemma A.15, $\mathcal{C}(SUB) \Vdash s$. □

A.7.5 Composition

Let g be a m-ary function and G be a term such that $\emptyset \vdash^{\delta} G : \mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_m \multimap !^{p}\mathsf{N}$ (where $p \geq 0$) and $G \Vdash g$. For $1 \leq i \leq m$, let f_i be a k-ary function and F_i a term such that $\emptyset \vdash^{\delta} F_i : \mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_k !^{q_i}\mathsf{N}$ (where $q_i \geq 0$) and $F_i \Vdash f_i$. We want to represent the composition function h such that:

$$h : \mathbb{N}^k \mapsto \mathbb{N}$$
$$h(x_1, \ldots, x_k) = g(f_1(x_1, \ldots, x_k), \ldots, f_m(x_1, \ldots, x_k))$$

For $i \geq 0$ and a term T, we define T^i inductively as:

$$T^0 = T$$
$$T^i = \lambda \vec{x}^{!^{i}\mathsf{N}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ !(T^{i-1}\vec{x})$$

Let $q = max(q_i)$. We can derive

$$\emptyset \vdash^{\delta} G^{q+1} : !^{q+1}\mathsf{N}_1 \multimap \ldots \multimap !^{q+1}\mathsf{N}_m \multimap !^{p+q+1}\mathsf{N}$$

We can also derive

$$\emptyset \vdash^{\delta} F_i^{q-q_i} : !^{q-q_i}\mathsf{N}_1 \multimap \ldots \multimap !^{q-q_i}\mathsf{N}_k \multimap !^{q}\mathsf{N}$$

Then, applying coercion we get

$$\emptyset \vdash^{\delta} \mathcal{C}(F_i^{q-q_i}) : \mathsf{N}_1 \multimap \ldots \mathsf{N}_k \multimap !^{q}\mathsf{N}$$

and we derive

$$x_1 : (\delta+1, \mathsf{N}), \ldots, x_k : (\delta+1, \mathsf{N}) \vdash^{\delta} !(\mathcal{C}(F_i^{q-q_i})x_1 \ldots x_k) : !^{q+1}\mathsf{N}$$

Let $F_i' \equiv !(\mathcal{C}(F_i^{q-q_i})x_1 \ldots x_k)$. By application we get

$$x_1 : (\delta+1, \mathsf{N}), \ldots, x_k : (\delta+1, \mathsf{N}) \vdash^{\delta} G^{q+1}F_1' \ldots F_m' : !^{p+q+1}\mathsf{N}$$

We derive

$$\emptyset \vdash^{\delta} \lambda\vec{x}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m' : !\mathsf{N}_1 \multimap \ldots \multimap !\mathsf{N}_m \multimap !^{p+q+1}\mathsf{N}$$

Applying coercion we get

$$\emptyset \vdash^{\delta} \mathcal{C}(\lambda\vec{x}^{!\mathsf{N}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m') : \mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_m \multimap !^{p+q+1}\mathsf{N}$$

Take

$$H = \mathcal{C}(\lambda\vec{x}^{!\mathsf{N}}.\mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ G^{q+1}F_1' \ldots F_m')$$

Proposition A.18 (composition is representable)**.** *$H \Vdash h$.*

Proof. We now have to show that for all M_i and n_i where $1 \leq i \leq k$ such that $M_i \Vdash n_i$ and $\emptyset \vdash^{\delta} M_i : \mathsf{N}$, we have $HM_1 \ldots M_k \Vdash h(n_1, \ldots, n_k)$. Since $F_i \Vdash f_i$, we have $F_iM_1 \ldots M_k \Vdash f_i(n_1, \ldots, n_k)$. Moreover $G \Vdash g$, hence

$$G(F_1M_1 \ldots M_k) \ldots (F_mM_1 \ldots M_k) \Vdash g(f_1(n_1, \ldots, n_k), \ldots, f_m(n_1, \ldots, n_k))$$

We can show that $HM_1 \ldots M_k \xrightarrow{*} G(F_1M_1 \ldots M_k) \ldots (F_mM_1 \ldots M_k)$, hence

$$HM_1 \ldots M_k \Vdash g(f_1(n_1, \ldots, n_k), \ldots, f_m(n_1, \ldots, n_k))$$

Thus $H \Vdash h$. □

A.7.6 Bounded sums and products

Let f be a $k+1$-ary function $f : \mathbb{N}^{k+1} \to \mathbb{N}$, where

$$\emptyset \vdash F : \mathsf{N}_i \multimap \mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_k \multimap !^{p}\mathsf{N}$$

with $p \geq 0$ and $F \Vdash f$. We want to represent

- bounded sum: $\sum_{1 \leq i \leq n} f(i, x_1, \ldots, x_k)$
- bounded product: $\prod_{1 \leq i \leq n} f(i, x_1, \ldots, x_k)$

For this we are going to represent $h : \mathbb{N}^{k+1} \to \mathbb{N}$:

$$h(0, x_1, \ldots, x_k) = f(0, x_1, \ldots, x_k)$$
$$h(n+1, x_1, \ldots, x_k) = g(f(n+1, x_1, \ldots, x_k), h(n, x_1, \ldots, x_k))$$

where g is a binary function standing for addition or multiplication, thus representable. More precisely we have $g : \mathbb{N}^2 \to \mathbb{N}$ such that $\emptyset \vdash^\delta G : \mathsf{N} \multimap \mathsf{N} \multimap \mathsf{N}$ and $G \Vdash g$.

For $i \geq 0$ and a term T we define T^i inductively:

$$T^0 = T x_1 \ldots x_k$$
$$T^i = \mathsf{let}\ !x_1 = x_1\ \mathsf{in}\ \ldots \mathsf{let}\ !x_k = x_k\ \mathsf{in}\ !T^{i-1}$$

We define the following terms:

$$ST = \lambda z.\langle S(\mathsf{fst}\ z), G^p(F x_1 \ldots x_k(S(\mathsf{fst}\ z)))(\mathsf{snd}\ z)\rangle$$
$$\emptyset; x_1 : (\delta, \mathsf{N}), \ldots, x_k : (\delta, \mathsf{N}) \vdash^\delta ST : \mathsf{N} \times !^p\mathsf{N} \multimap \mathsf{N} \times !^p\mathsf{N}$$

$$EX = \lambda h.\mathsf{let}\ !h = h\ \mathsf{in}\ !\mathsf{snd}\ h\langle\overline{0}, F x_1 \ldots x_k \overline{0}\rangle$$
$$\emptyset; x_1 : (\delta+1, \mathsf{N}), \ldots, x_k : (\delta+1, \mathsf{N}) \vdash^\delta EX : !(\mathsf{N} \times !^p\mathsf{N} \multimap \mathsf{N} \times !^p\mathsf{N}) \multimap !^{p+1}\mathsf{N}$$

We derive

$$n : (\mathsf{N}), \vec{x} : (\delta, \mathsf{N}) \vdash^\delta \mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ \mathsf{let}\ !n = n\ \mathsf{in}\ \mathsf{int\_git}\ !ST\ EX\ n : !^{p+1}\mathsf{N}$$

Let $R = \mathsf{let}\ !\vec{x} = \vec{x}\ \mathsf{in}\ \mathsf{let}\ !n = n\ \mathsf{in}\ \mathsf{int\_git}\ !ST\ EX\ n$. By coercion and abstractions we get

$$\emptyset \vdash^\delta \mathcal{C}(\lambda n.\lambda\vec{x}.R) : \mathsf{N}_i \multimap \mathsf{N}_1 \multimap \ldots \multimap \mathsf{N}_k \multimap !^{p+1}\mathsf{N}$$

Take $H = \mathcal{C}(\lambda n.\lambda\vec{x}.R)$.

Proposition A.19 (bounded sum/product is representable)**.** *$H \Vdash h$.*

Proof. Given $M_i \Vdash i$ and $M_j \Vdash n_j$ with $1 \leq j \leq k$ and taking G for addition, we remark that

$$H M_i M_1 \ldots M_k \xrightarrow{*} \overline{f(i, n_1, \ldots, n_k) + \ldots + f(1, n_1, \ldots, n_k) + f(0, n_1, \ldots, n_k)}$$

Hence $H \Vdash h$. □