INFORMATION AND COMMUNICATION TECHNOLOGIES (ICT) PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report D5.3 Case study

Version 1.0

Main Authors:
Roberto M. Amadio, Nicolas Ayache, François Bobot,
Antoine Madet, Yann Régis-Gianas

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

Outline The deliverable D5.3 is composed of the following parts:

1. A summary.
2. The papers [1] and [3] and the related software Cost[1].
3. The paper [2] and the related software LamCost[2].
4. The papers [5, 4].

References

[1] N. Ayache, R.M. Amadio, Y. Régis-Gianas. Certifying and Reasoning on Cost Annotations in C Programs. Proc. FMICS, Springer LNCS 7437: 32-46, 2012.

[2] R.M. Amadio, Y. Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In Journal *Higher-order and Symbolic Computation*, 61 pages, to appear.

[3] F. Bobot, J.-C.. Filliatre. Separation Predicates: A Taste of Separation Logic in First-Order Logic. Proc. ICFEM, Springer LNCS 7635:167-181, 2012.

[4] A. Brunel, A. Madet. Indexed Realizability for Bounded-Time Programming with References and Type Fixpoints. Proc. APLAS, Springer LNCS 7705:264-279 , 2012.

[5] A. Madet. A polynomial time λ-calculus with multithreading and side effects. In Proc. ACM-PPDP, pages 55-66, 2012.

[1] `http://www.pps.univ-paris-diderot.fr/~yrg/cerco/`

[2] `http://www.pps.univ-paris-diderot.fr/~yrg/fun-cca/index.php`

Summary

The main aim of WP5 is to develop proof of concept prototypes where the (untrusted) compiler implemented in WP2 is interfaced with existing tools and languages in order to synthesize complexity assertions on the execution time of programs.

In particular, Deliverable 5.3 should contain a case study (under the form of a software prototype) which is described as follows in the contract.

> Case study: analysis of synchronous code. Automatic generation of invariants for the C code generated by a synchronous language compiler. Application to the computation of a certified reaction time bound for synchronous programs and testing on significant examples.

The synchronous language we chose to carry on this case study is Lustre. 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 completely automatic application of the Cost plug-in (cf. Deliverable D5.1).

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 1. It simply executes a command line, reads the results, and sends them to the next command.

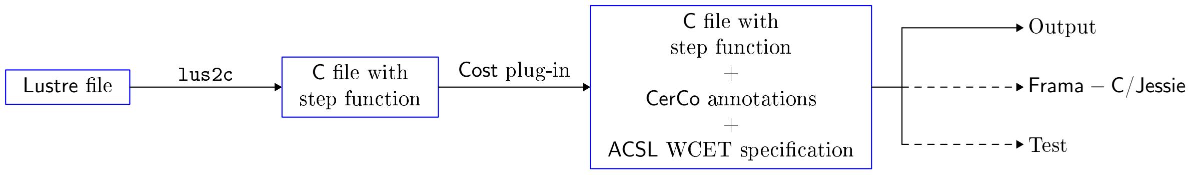

Lustre file
lus2c
C file with step function
Cost plug-in
C file with step function + CerCo annotations + ACSL WCET specification
Output
Frama − C/Jessie
Test

Figure 1: Flow of the Lustre wrapper

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_O_parity` or `parity_O_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 syntactically 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.

The prototype described above was already completed and presented at the second review along with an unplanned case study on applying the labelling method to a functional language. In particular, the work on the Lustre case study has been published in [1] along with results that relate to Deliverables 2.2 (untrusted compiler implementing the labelling method) and 5.1 (Cost plug-in). Therefore the human power left during the third period was dedicated to develop case studies which were unplanned in the contract and which are described below.

Enlarge the scope of the Cost synthesis tool.

At mentioned above the structure of the C programs produced by a Lustre compiler is particularly simple. During the third period of the project, we worked to extend the class of programs that can be handled in an automatic way. The two main contributions are as follows:

1. We showed [1] that the Cost tool can handle automatically programs with simple loops such as stream ciphers and sorting (the quoted paper got the best paper award at the conference).

2. In order to handle simple programs with pointers (such as in-place list reversal), we have developed a proof methodology that adapts some ideas of *separation logic* to the Frama − C tool [3].

Along the way, the internals of the Cost plug-in have also been revisited. In particular, the abstract interpretation technique described in [1] has been streamlined and a program instrumentation to measure stack bounds has been added. This work is not described in the quoted papers [1, 3] but it is part of the prototype software deliverable and was demonstrated at the third and final review. We stress that all this work is based on the *untrusted* CerCo compiler developed in WP2 as the *partially trusted* CerCo compiler was delivered when the man power devoted to this task was exhausted.

The labelling approach for a higher-order functional language.

At the second review meeting, we had presented an adaptation of the so called *labelling method* to a standard compiler for a higher-order functional language. The target code produced by this compiler corresponds to the source code of the back-end of the CerCo C compiler. During the third period, we have shown that the method can be enhanced to account for the cost of *safe memory management*. Specifically, we have relied on a *region based management system* and this in turn has required an analysis of the way the compilation chain preserves typing. The whole approach is described in the included paper [2].

Feasible bounds

We have worked on a type system for a multi-threaded functional language that guarantees termination in *polynomial time* [5, 4]. A long term goal of this work is to establish a connection between the research on *implicit computational complexity* (ICC) and *worst case execution time* (WCET). Researchers in ICC design type/logical systems that guarantee asymptotic bounds for the source language. What needs to be done is to develop methods to turn these asymptotic bounds for the source language into *certified* and *concrete* bounds for the compiled code. We also believe that in a practical approach one should be able to mix 'well-typed' programs whose resource bounds are guaranteed with 'untyped' ones whose resource bounds must be explicitly proved in a general purpose logic. The *realizability framework* developed in [4] appears as a promising approach to this task.

Certifying and Reasoning on Cost Annotations in C Programs

Nicolas Ayache[1,2], 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 enrich a compiler in order to turn it into a "cost annotating compiler", that is, a compiler which can *lift* pieces of information on the execution cost of the object code as cost annotations on the source code. These cost annotations characterize the execution costs of code fragments of constant complexity. The first contribution of this paper is a proof methodology that extends standard simulation proofs of compiler correctness to ensure that the cost annotations on the source code are sound and precise with respect to an execution cost model of the object code.

As a second contribution, we demonstrate that our label-based instrumentation is scalable because it consists in a modular extension of the compilation chain. To that end, we report our successful experience in implementing and testing the labelling approach on top of a prototype compiler written in ocaml for (a large fragment of) the C language.

As a third and last contribution, we provide evidence for the usability of the generated cost annotations as a mean to reason on the concrete complexity of programs written in C. For this purpose, we present a Frama-C plugin that uses our cost annotating compiler to automatically infer trustworthy logic assertions about the concrete worst case execution cost of programs written in a fragment of the C language. These logic assertions are synthetic in the sense that they characterize the cost of executing the entire program, not only constant-time fragments. (These bounds may depend on the size of the input data.) We report our experimentations on some C programs, especially programs generated by a compiler for the synchronous programming language Lustre used in critical embedded software.

1 Introduction

The formal description and certification of software components is reaching a certain level of maturity with impressing case studies ranging from compilers to kernels of operating systems. A well-documented example is the proof of functional correctness of a moderately optimizing compiler from a large subset of the C language to a typical assembly language of the kind used in embedded systems [11].

In the framework of the *Certified Complexity* (CerCo) project[1] [4], we aim to refine this line of work by focusing on the issue of the *execution cost* of

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

the compiled code. Specifically, we aim to build a formally verified C compiler that given a source program produces automatically a functionally equivalent object code plus an annotation of the source code which is a sound and precise description of the execution cost of the object code.

We target in particular the kind of C programs produced for embedded applications; these programs are eventually compiled to binaries executable on specific processors. The current state of the art in commercial products such as Scade[2] [8] is that the *reaction time* of the program is estimated by means of abstract interpretation methods (such as those developed by AbsInt[3] [7]) that operate on the binaries. These methods rely on a specific knowledge of the architecture of the processor and may require explicit (and uncertified) annotations of the binaries to determine the number of times a loop is iterated (see, *e.g.*, [14] for a survey of the state of the art).

In this context, our aim is to produce a mechanically verified compiler which can *lift* in a provably correct way the pieces of information on the execution cost of the binary code to cost annotations on the source C code. Then the produced cost annotations are manipulated with the Frama − C[4] [5] automatic tool to infer synthetic cost annotations. We stress that the practical relevance of the proposed approach depends on the possibility of obtaining accurate information on the execution cost of relatively short sequences of binary instructions. This seems beyond the scope of current Worst-Case Execution Time (WCET) tools such as AbsInt or Chronos[5] which do not support a *compositional* analysis of WCET. For this reason, we focus on processors with a simple architecture for which manufacturers can provide accurate information on the execution cost of the binary instructions. In particular, our experiments are based on the 8051 [10][6]. This is a widely popular 8-bits processor developed by Intel for use in embedded systems with no cache and no pipeline. An important characteristic of the processor is that its cost model is 'additive': the cost of a sequence of instructions is exactly the sum of the costs of each instruction.

The rest of the paper is organized as follows. Section 2 describes the labelling approach and its formal application to a toy compiler. The report [2] gives standard definitions for the toy compiler and sketches the proofs. A formal and browsable Coq development composed of 1 *Kloc* of specifications and 3.5 *Kloc* of proofs is available at `http://www.pps.univ-paris-diderot.fr/cerco`. Section 3 reports our experience in implementing and testing the labelling approach for a compiler from C to 8051 binaries. The CerCo compiler is composed of 30 *Kloc* of ocaml code; it can be both downloaded and tested as a web application at the URL above. More details are available in report [2] Section 4 introduces the automatic Cost tool that starting from the cost annotations produces certified synthetic cost bounds. This is a Frama − C plug-in composed of 5 *Kloc* of ocaml code also available at the URL above.

[2] Esterel Technologies. `http://www.esterel-technologies.com`
[3] AbsInt Angewandte Informatik. `http://www.absint.com/`
[4] Frama − C software analyzers. `http://frama-c.com/`
[5] Chronos tool. `www.comp.nus.edu.sg/~rpembed/chronos`
[6] The recently proposed ARM Cortex M series would be another obvious candidate.

2 A "Labelling" Method for Cost Annotating Compilation

In this section, we explain in general terms the so-called "labelling" method to turn a compiler into a cost annotating compiler while minimizing the impact of this extension on the proof of the semantic preservation. Then to make our purpose technically precise, we apply the method to a toy compiler.

2.1 Overview

As a first step, we need a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to provide them being sound and precise, and (iii) the way such proofs can be composed. The execution cost of the source programs we are interested in depends on their control structure. Typically, the source programs are composed of mutually recursive procedures and loops and their execution cost depends, up to some multiplicative constant, on the number of times procedure calls and loop iterations are performed. Producing a *cost annotation* of a source program amounts to:

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

Thus, producing a cost-annotation of a source program P amounts to build an *annotated program* $An(P)$ which behaves as P while self-monitoring its execution cost. In particular, if we do *not* observe the cost variables then we expect the annotated program $An(P)$ to be functionally equivalent to P. Notice that in the proposed approach an annotated program is a program in the source language. Therefore, the meaning of the cost annotations is automatically defined by the semantics of the source language and tools developed to reason on the source programs can be directly applied to the annotated programs too. Finally, notice that the annotated program $An(P)$ is *only* meant to *reason* on the execution cost of the unannotated program P and it will never be compiled or executed.

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

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

2.2 The Labelling Approach, Formally

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

$$\begin{tikzcd}
L_1 & L_{1,\ell} \arrow[l, "\mathcal{I}"] \arrow[r, "\mathcal{C}_1"] \arrow[d, bend left, "er_1"] & L_{2,\ell} \arrow[d, "er_2"] & \cdots \arrow[r, "\mathcal{C}_k"] & L_{k+1,\ell} \arrow[d, "er_{k+1}"] \\
 & L_1 \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}_1"] & L_2 & \cdots \arrow[r, "\mathcal{C}_k"] & L_{k+1}
\end{tikzcd}
\qquad \Bigg\| \qquad
\begin{array}{rcl}
er_{i+1} \circ \mathcal{C}_i & = & \mathcal{C}_i \circ er_i \\
er_1 \circ \mathcal{L} & = & id_{L_1} \\
An & = & \mathcal{I} \circ \mathcal{L}
\end{array}$$

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

For each labelled language there is an obvious function er_i erasing all labels and producing a program in the corresponding unlabelled language. The compilation functions $\mathcal{C}_i$ are extended from the unlabelled to the labelled language so that they enjoy commutation with the erasure functions. Moreover, we lift the soundness properties of the compilation functions from the unlabelled to the labelled languages and transition systems.

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

Suppose s is some adequate representation of the state of a program. Let P be a source program. The judgement $(P, s) \Downarrow s'$ is the big-step evaluation of P transforming state s into a state s'. Let us write $s[v/x]$ to denote a state s in which the variable x is assigned a value v. Suppose now that its annotation satisfies the following property:

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

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

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

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

To answer this question, we observe that the object code we are interested in is some kind of assembly code and its control flow can be easily represented as a control flow graph. The idea is then to perform two simple checks on the control flow graph. The first check is to verify that all loops go through a labelled node. If this is the case then we can associate a finite cost with every label and prove that the cost annotations are sound. The second check amounts to verify that all paths starting from a label have the same cost. If this check is successful then we can conclude that the cost annotations are precise.

2.3 A Toy Compiler

As a first case study, we apply the labelling approach to a *toy compiler*.

The syntax of the source, intermediate and target languages is given in Figure 1. The three languages considered can be shortly described as follows: Imp is a very simple imperative language with pure expressions, branching and looping commands, Vm is an assembly-like language enriched with a stack, and Mips is a Mips-like assembly language [9] with registers and main memory.

The semantics of Imp is defined over configurations (S, K, s) where S is a statement, K is a continuation and s is a state. A *continuation* K is a list of

Syntax for Imp

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

Syntax for Vm

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

Syntax for Mips

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

Fig. 1. Syntax definitions

commands which terminates with a special symbol halt. The semantics of Vm is defined over stack-based machine configurations $C \vdash (i, \sigma, s)$ where C is a program, i is a program counter, σ is a stack and s is a state. The semantics of Mips is defined over register-based machine configurations $C \vdash (i, m)$ where C is a program, i is a program counter and m is a machine memory (with registers and main memory).

The first compilation function $\mathcal{C}$ relies on the stack of the Vm language to implement expression evaluation while the second compilation function $\mathcal{C}'$ allocates (statically) the base of the stack in the registers and the rest in main memory. This is of course a naive strategy but it suffices to expose some of the problems that arise in defining a compositional approach. The formal definitions of these compilation functions $\mathcal{C}$ from Imp to Vm and $\mathcal{C}'$ from Vm to Mips are standard and thus eluded. (See report [2] for formal details about semantics and the compilation chain.)

Applying the labelling approach to this toy compiler results in the following diagram. The next sections aim at describing this diagram in details.

$$\begin{tikzcd}
\mathsf{Imp} & \mathsf{Imp}_\ell \arrow[l, "\mathcal{I}"'] \arrow[r, "\mathcal{C}"] \arrow[d, bend left, "er_{\mathsf{Imp}}"] & \mathsf{Vm}_\ell \arrow[r, "\mathcal{C}'"] \arrow[d, "er_{\mathsf{Vm}}"] & \mathsf{Mips}_\ell \arrow[d, "er_{\mathsf{Mips}}"] \\
 & \mathsf{Imp} \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}"] & \mathsf{Vm} \arrow[r, "\mathcal{C}'"] & \mathsf{Mips}
\end{tikzcd}$$

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

2.4 Labelled languages: Syntax and Semantics

Syntax The syntax of Imp is extended so that statements can be labelled: $S ::= \ldots \parallel \ell : S$. A new instruction $\mathsf{emit}(\ell)$ (resp. $(\mathsf{emit}\ \ell)$) is introduced in the syntax of Vm (resp. Mips).

Semantics. The small step semantics of Imp statements is extended as described by the following rule.

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

We denote with $\lambda, \lambda', \ldots$ finite sequences of labels. In particular, the empty sequence is written ϵ. We also identify an unlabelled transition with a transition labelled with ϵ. Then, the small step reduction relation we have defined on statements becomes a *labelled transition system*. We derive a *labelled* big-step semantics as follows: $(S, s) \Downarrow (s', \lambda)$ if $(S, \mathsf{halt}, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (\mathsf{skip}, \mathsf{halt}, s')$ and $\lambda = \lambda_1 \cdots \lambda_n$.

Following the same pattern, the small step semantics of Vm and Mips are turned into a labelled transition system as follows:

$$\begin{array}{ll} C \vdash (i, \sigma, s) \xrightarrow{\ell} (i+1, \sigma, s) & \text{if } C[i] = \mathsf{emit}(\ell) \ . \\ M \vdash (i, m) \xrightarrow{\ell} (i+1, m) & \text{if } M[i] = (\mathsf{emit}\ \ell) \ . \end{array}$$

The evaluation predicate for labelled Vm is defined as $(C, s) \Downarrow (s', \lambda)$ if $C \vdash (0, \epsilon, s) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (i, \epsilon, s')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $C[i] = \mathsf{halt}$. The evaluation predicate for labelled Mips is defined as $(M, m) \Downarrow (m', \lambda)$ if $M \vdash (0, m) \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} (j, m')$, $\lambda = \lambda_1 \cdots \lambda_n$ and $M[j] = \mathsf{halt}$.

2.5 Erasure Functions

There is an obvious *erasure* function er_{Imp} from the labelled language to the unlabelled one which is the identity on expressions and boolean conditions, and traverses commands removing all labels.

The erasure function er_{Vm} amounts to remove from a Vm code C all the $\mathsf{emit}(\ell)$ instructions and recompute jumps accordingly. Specifically, let $n(C, i, j)$ be the number of emit instructions in the interval $[i, j]$. Then, assuming $C[i] = \mathsf{branch}(k)$ we replace the offset k with an offset k' determined as follows:

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

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

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

2.6 Compilation of Labelled Languages

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

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

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

The following proposition relates Vm_ℓ code and its compilation and it is similar to proposition 1. Here $m \Vdash \sigma, s$ means "the low-level Mips memory m realizes the Vm stack σ and state s".

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

2.7 Labellings and Instrumentations

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

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

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

Proposition 3. *Let S be an Imp_ℓ command. If* $(\mathcal{I}(S), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *then* $\exists \lambda \ \ \kappa(\lambda) = \delta$ *and* $(S, s[c/cost]) \Downarrow (s'[c/cost], \lambda)$.

Definition 1. *A* labelling *is a function $\mathcal{L}$ from an unlabelled language to the corresponding labelled one such that $er_{\mathsf{Imp}} \circ \mathcal{L}$ is the identity function on the* Imp *language.*

Proposition 4. *For any labelling function $\mathcal{L}$, and* Imp *program P, the following holds:*

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

Proposition 5. *Given a function κ for the labels and a labelling function $\mathcal{L}$, for all programs P of the source language if* $(\mathcal{I}(\mathcal{L}(P)), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *and* $m \Vdash \epsilon, s[c/cost]$ *then* $(\mathcal{C}'(\mathcal{C}(\mathcal{L}(P))), m) \Downarrow (m', \lambda)$, $m' \Vdash \epsilon, s'[c/cost]$ *and* $\kappa(\lambda) = \delta$.

2.8 Sound and Precise Labellings

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

Definition 2. *A* simple path *in a* Mips_ℓ *code* M *is a directed finite path in the graph associated with* M *where the first node is labelled, the last node is the predecessor of either a labelled node or a leaf, and all the other nodes are unlabelled.*

Definition 3. *A* Mips_ℓ *code* M *is* soundly labelled *if in the associated graph the root node* 0 *is labelled and there are no loops that do not go through a labelled node. Besides, we say that a soundly labelled code is* precise *if for every label* ℓ *in the code, the simple paths starting from a node labelled with* ℓ *have the same cost.*

In a soundly labelled graph there are finitely many simple paths. Thus, given a soundly labelled Mips code M, we can associate with every label ℓ a number $\kappa(\ell)$ which is the maximum (estimated) cost of executing a simple path whose first node is labelled with ℓ. Thus for a soundly labelled Mips code the sequence of labels associated with a computation is a significant information on the execution cost.

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

Proposition 6. *If* M *is soundly (resp. precisely) labelled and* $(M, m) \Downarrow (m', \lambda)$ *then the cost of the computation is bounded by* $\kappa(\lambda)$ *(resp. is exactly* $\kappa(\lambda)$*).*

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

Table 1. A labelling for the Imp language

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

Proposition 7. *For all* Imp *programs* P*,* $\mathcal{C}'(\mathcal{C}(\mathcal{L}_p(P))$ *is a soundly and precisely labelled* Mips *code.*

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

Definition 4. *Given a labelling function* $\mathcal{L}$ *for the source language* Imp *and a program* P *in the* Imp *language, we define an annotation for the source program as follows:*

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

Proposition 8. *If* P *is a program and* $\mathcal{C}'(\mathcal{C}(\mathcal{L}(P)))$ *is a sound (sound and precise) labelling then* $(An_{\mathsf{Imp}}(P), s[c/cost]) \Downarrow s'[c+\delta/cost]$ *and* $m \Vdash \epsilon, s[c/cost]$ *entails that* $(\mathcal{C}'(\mathcal{C}(P)), m) \Downarrow m'$*,* $m' \Vdash \epsilon, s'[c/cost]$ *and the cost of the execution is bounded by (is exactly)* δ*.*

3 A C Compiler Producing Cost Annotations

We now consider an untrusted C compiler prototype in ocaml in order to experiment with the scalability of our approach. Its architecture is described below:

$$\begin{tikzcd}
 & \mathsf{C} \arrow[r] & \mathsf{Clight} \arrow[r] & \mathsf{Cminor} \arrow[r] & \mathsf{RTLAbs} \arrow[d] & \text{(front end)} \\
\text{Mips or 8051} & \mathsf{LIN} \arrow[l] & \mathsf{LTL} \arrow[l] & \mathsf{ERTL} \arrow[l] & \mathsf{RTL} \arrow[l] & \text{(back-end)}
\end{tikzcd}$$

The most notable difference with CompCert [11] is that we target the Intel 8051 [10] and Mips assembly languages (rather than PowerPc). The compilation from C to Clight relies on the CIL front-end [13]. The one from Clight to RTL has been programmed from scratch and it is partly based on the Coq definitions available in the CompCert compiler. Finally, the back-end from RTL to Mips is based on a compiler developed in ocaml for pedagogical purposes[7]; we extended this back-end to target the Intel 8051. The main optimizations the back-end performs are liveness analysis and register allocation, and graph compression. We ran some benchmarks to ensure that our prototype implementation is realistic. The results are given in report [2].

This section informally describes the labelled extensions of the languages in the compilation chain (see report [2] for details), the way the labels are propagated by the compilation functions, and the (sound and precise) labelling of the source code. A related experiment concerning a higher-order functional language of the ML family is described in [3].

3.1 Labelled Languages

Both the Clight and Cminor languages are extended in the same way by labelling both statements and expressions (by comparison, in the toy language Imp we

[7] `http://www.enseignement.polytechnique.fr/informatique/INF564/`

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

The labelled versions of RTLAbs and the languages in the back-end simply consist in adding a new instruction whose semantics is to emit a label without modifying the state. For the CFG based languages (RTLAbs to LTL), this new instruction is emit *label* $\to$ *node*. For LIN, Mips and 8051, it is emit *label*. The translation of these label instructions is immediate.

3.2 Labelling of the Source Language

As for the toy compiler, the goals of a labelling are soundness, precision, and possibly economy. Our labelling for Clight resembles that of Imp for their common instructions (e.g. loops). We only consider the instructions of Clight that are not present in Imp[8].

Ternary expressions. They may introduce a branching in the control flow. We achieve precision by associating a label with each branch.

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

Function calls. In the general case, the address of the callee cannot be inferred statically. But in the compiled assembly code, we know for a fact that the callee ends with a return statement that transfers the control back to the instruction following the function call in the caller. As a result, we treat function calls according to the following invariants: (1) the instructions of a function are covered by the labels inside this function, (2) we assume a function call always returns and runs the instruction following the call. Invariant (1) entails in particular that each function must contain at least one label. Invariant (2) is of course an over-approximation of the program behavior as a function might fail to return because of an error or an infinite loop. In this case, the proposed labelling remains correct: it just assumes that the instructions following the function call will be executed, and takes their cost into consideration. The final computed cost is still an over-approximation of the actual cost.

4 A Tool for Reasoning on Cost Annotations

Frama − C is a set of analysers for C programs with a specification language called ACSL. New analyses can be dynamically added through a plug-in system.

[8] We do not consider expressions with side-effects because they are eliminated by CIL.

For instance, the Jessie plug-in allows deductive verification of C programs with respect to their specification in ACSL, with various provers as back-end tools.

We developed the Cost plug-in for the Frama − C platform as a proof of concept of an automatic environment exploiting the cost annotations produced by the CerCo compiler. It consists of an ocaml program of 5 *Kloc* 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 the C functions of the program as a function of the value of their parameters, (4) the user can then call the Jessie tool to discharge the related proof obligations. In the following we elaborate on the soundness of the framework, the algorithms underlying the plug-in, and the experiments we performed with the Cost tool.

4.1 Soundness

The soundness of the whole framework depends on the cost annotations added by the CerCo compiler, the synthetic costs produced by the Cost plug-in, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. The synthetic costs being in ACSL format, Jessie can be used to verify them. Thus, even if the added synthetic costs are incorrect (relatively to the cost annotations), the process in its globality is still correct: indeed, Jessie will not validate incorrect costs and no conclusion can be made about the WCET of the program in this case. In other terms, the soundness does not really depend on the action of the Cost plug-in, which can in principle produce *any* synthetic cost. However, in order to be able to actually prove a WCET of a C function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce their validity. In practice this is not straightforward even for very simple programs composed of branching and assignments (no loops and no recursion) because a fine analysis of the VCs associated with branching may lead to a complexity blow up.

4.2 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. The plug-in proceeds as follows.

- Each function is independently processed and is associated a WCET that may depend on the cost of the other functions. This is done with a mix between abstract interpretation [6] and syntactic recognition of specific loops.

- As result of the previous step, a system of inequations is built and its solution is attempted by an iterative process. At each iteration, one 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. This step is repeated till a fixpoint is reached.
- ACSL annotations are added to the program according to the result of the above fixpoint. The two previous steps may fail in finding a concrete WCET for some functions, because of imprecision inherent in abstract interpretation, and because of recursive definitions 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.
- The most difficult instructions to handle are loops. We consider loops for which we can syntactically find a counter (its initial, increment and last values are domain dependent). Other loops are associated an undefined cost ($\top$). When encountering a loop, the analysis first sets the cost of its entry point to 0. The cost inside the loop is thus relative to the loop. Then, for each exit point, we fetch the value of the cost at that point and multiply it by an upper bound of the number of iterations (obtained through arithmetic over the initial, increment and last values of the counter); this results in an upper bound of the cost of the whole loop, which is sent to the successors of the considered exit point.

Figure 2 shows the action of the Cost plug-in on a C program. The most notable differences are the added so-called *cost variable*, some associated update (increment) instructions inside the code, and an `ensures` clause that specifies the WCET of the `is_sorted` function with respect to the cost variable. One can notice that this WCET depends on the inputs of the function. Running Jessie on the annotated and specified program generates VCs that are all proved by the automatic prover AltErgo[9].

4.3 Experiments

The Cost plug-in has been developed in order to validate CerCo's framework for modular WCET analysis. The plug-in allows (semi-)automatic generation and certification of WCET for C programs. Also, we designed a wrapper for supporting Lustre files. Indeed, Lustre is a data-flow language to program synchronous systems and the language comes with a compiler to C. The C function produced by the compiler implements the *step function* of the synchronous system and computing the WCET of the function amounts to obtain a bound on the reaction time of the system.

We tested the Cost plug-in and the Lustre wrapper on the C programs generated by the Lustre compiler. We also tested it on some basic algorithms and cryptographic functions; these examples, unlike those generated by the Lustre

[9] AltErgo prover. `http://ergo.lri.fr/`

```
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) The initial C source code.

```
int _cost = 0;

/*@ 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 += 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 += 91;
    if (tab[i] > tab[i+1]) { _cost += 104; res = 0; }
    else _cost += 84; }
  _cost += 4; return res; }
```

(b) The annotated source code generated by Cost.

Fig. 2. An example of the Cost plug-in action

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 function L: C generated from a Lustre file				

Fig. 3. Experiments on CerCo and the Cost plug-in

compiler include arrays and for-loops. Table 3 provides a list of concrete programs and describes their type, functionality, the number of lines of the source code, and the number of VCs generated. In each case, the Cost plug-in computes a WCET and AltErgo is able to discharge all VCs. Obviously the generation of synthetic costs is an undecidable and open-ended problem. Our experience just shows that there are classes of C programs which are relevant for embedded applications and for which the synthesis and verification tasks can be completely automatized.

Acknowledgement. The master students Kayvan Memarian and Ronan Saillard contributed both to the Coq proofs and the CerCo compiler in the early stages of their development.

References

1. Amadio, R.M., Ayache, N., Memarian, K., Saillard, R., Régis-Gianas, Y.: Compiler Design and Intermediate Languages. Deliverable 2.1 of [4]
2. Ayache, N., Amadio, R.M., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of C programs. Research Report 00702665 (June 2012)
3. Amadio, R.M., Régis-Gianas, Y.: Certifying and Reasoning on Cost Annotations of Functional Programs. In: Peña, R., van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2011. LNCS, vol. 7177, pp. 72–89. Springer, Heidelberg (2012)
4. Certified complexity (Project description). ICT-2007.8.0 FET Open, Grant 243881
5. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual. CEA-LIST, Software Safety Laboratory, Saclay, F-91191, `http://frama-c.com/`
6. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. Jou. of Logic and Computation 2(4), 511–547 (1992)
7. Ferdinand, C., Heckmann, R., Le Sergent, T., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In: Embedded Real Time Software (2008)
8. Fornari, X.: Understanding how SCADE suite KCG generates safe C code. White paper, Esterel Technologies (2010)
9. Larus, J.: Assemblers, linkers, and the SPIM simulator. Appendix of Computer Organization and Design: the hw/sw interface. Hennessy and Patterson (2005)
10. MCS 51 Microcontroller Family User's Manual. Publication number 121517. Intel Corporation (1994)
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
12. Leroy, X.: Mechanized semantics, with applications to program proof and compiler verification. In: Marktoberdorf Summer School (2009)
13. Necula, G., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
14. Wilhelm, R., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)

Separation Predicates: A Taste of Separation Logic in First-Order Logic[⋆]

François Bobot and Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay-Île-de-France, ProVal, Orsay F-91893

Abstract. This paper introduces *separation predicates*, a technique to reuse some ideas from separation logic in the framework of program verification using a traditional first-order logic. The purpose is to benefit from existing specification languages, verification condition generators, and automated theorem provers. Separation predicates are automatically derived from user-defined inductive predicates. We illustrate this idea on a non-trivial case study, namely the composite pattern, which is specified in C/ACSL and verified in a fully automatic way using SMT solvers Alt-Ergo, CVC3, and Z3.

1 Introduction

Program verification has recently entered a new era. It is now possible to prove rather complex programs in a reasonable amount of time, as demonstrated in recent program verification competitions [17,12,10]. One of the reasons for this is tremendous progress in automated theorem provers. SMT solvers, in particular, are tools of choice to discharge verification conditions, for they combine full first-order logic with equality, arithmetic, and a handful of other theories relevant to program verification, such as arrays, bit vectors, or tuples. Notable examples of SMT solvers include Alt-Ergo [4], CVC3 [1], Yices [9], and Z3 [8].

Yet, when it comes to verifying programs involving pointer-based data structures, such as linked lists, trees, or graphs, the use of traditional first-order logic to specify, and of SMT solvers to verify, shows some limitations. Separation logic [22] is then an elegant alternative. Designed at the turn of the century, it is a program logic with a new notion of conjunction to express spatial separation. Separation logic requires dedicated theorem provers, implemented in tools such as Smallfoot [2] or VeriFast [13,15]. One drawback of such provers, however, is to either limit the expressiveness of formulas (*e.g.* to the so-called symbolic heaps), or to require some user-guidance (*e.g.* open/close commands in Verifast).

In an attempt to conciliate both approaches, we introduce the notion of *separation predicates*. The idea is to introduce some ideas from separation logic into a traditional verification framework where the specification language, the

[⋆] This work was (partially) supported by the Information and Communication Technologies (ICT) Programme as Project FP7-ICT-2009-C-243881 CerCo and by the U3CAT project (ANR-08-SEGI-021) of the French national research organization.

verification condition generator, and the theorem provers were not designed with separation logic in mind. Separation predicates are automatically derived from user-defined inductive predicates, on demand. Then they can be used in program annotations, exactly as other predicates, *i.e.*, without any constraint. Simply speaking, where one would write $P \star Q$ in separation logic, one will here ask for the generation of a separation predicate *sep* and then use it as $P \wedge Q \wedge sep(P, Q)$.

We have implemented separation predicates within Frama-C's plug-in Jessie for deductive verification [21]. This paper demonstrates the usefulness of separation predicates on a realistic, non-trivial case study, namely the composite pattern from the VACID-0 benchmark [20]. We achieve a fully automatic proof using three existing SMT solvers.

This paper is organized as follows. Section 2 gives a quick overview of what separation predicates are, using the classic example of list reversal. Section 3 formalizes the notion of separation predicates and briefly describes our implementation. Then, Section 4 goes through the composite pattern case study. Section 5 presents how this framework can be extended to express the set of pointers modified by a function. We conclude with related work in Section 6.

2 Motivating Example

As an example, let us consider the classic in-place list reversal algorithm:

```
$rev(p) \equiv$
  $q :=$ NULL
  while $p \neq$ NULL do $t := p{\to}$next; $p{\to}$next $:= q$; $q := p$; $p := t$ done
  return $q$
```

We may want to verify that, whenever p points to a finite singly-linked list, then $rev(p)$ returns a finite list. (Proving that lists are indeed reversed requires more space than available here.) To do so, we first define the notion of finite singly-linked lists, for instance using the following inductive predicate *islist*:

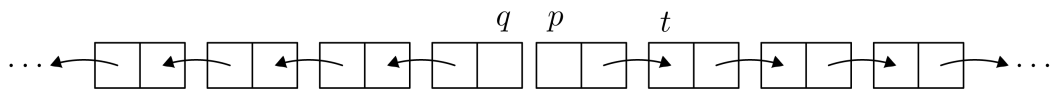
```
inductive $islist(p) \equiv$
  | $C_0 : islist(\texttt{NULL})$
  | $C_1 : \forall p.\, p \neq \texttt{NULL} \Rightarrow islist(p{\to}\texttt{next}) \Rightarrow islist(p)$
```

Then we specify function *rev* using the following Hoare triple:

$$\{islist(p)\}\ q := rev(p)\ \{islist(q)\}$$

To perform the proof, we need a loop invariant. A natural invariant expresses that both p and q are finite lists, that is $islist(p) \wedge islist(q)$.

Unfortunately, this is not enough for the proof to be carried out. Indeed, we lack the crucial information that assigning $p{\to}$`next` will not modify lists q and t. Therefore, we cannot prove that the invariant above is preserved.



Separation logic proposes an elegant solution to this problem. It introduces a new logical connective $P \star Q$ that acts as the conjunction $P \wedge Q$ and expresses spatial separation of P and Q at the same time. In the list reversal example, it is used at two places. First, it is used in the definition of *islist* to express that the first node of a list is disjoint from the remaining nodes:

$$\mathit{islist}(p) \ \equiv \ \mathsf{if}\ p = \mathtt{NULL}\ \mathsf{then\ emp\ else}\ \exists q.\, p{\rightarrow}\mathtt{next} \mapsto q \star \mathit{islist}(q)$$

This way, we can now prove that list t is preserved when $p{\rightarrow}\mathtt{next}$ is assigned. Second, the connective $\star$ is also used in the loop invariant to express that lists p and q do not share any pointer:

$$\mathit{islist}(p) \star \mathit{islist}(q).$$

This way, we can now prove that list q is preserved when $p{\rightarrow}\mathtt{next}$ is assigned. Using a dedicated prover for separation logic, list reversal can be proved correct using this loop invariant.

In our attempt to use traditional SMT solvers instead, we introduce the notion of *separation predicates*: the $\star$ connective of separation logic is replaced by new predicate symbols, which are generated on a user-demand basis. Our annotated C code for list reversal using separation predicates is given in Fig. 1.

We define predicate `islist` inductively (lines 4–8), as we did earlier in this section. In this definition `\valid(p)` express that `p` is a pointer that can be safely dereferenced (allocated and not freed). It captures finite lists only and, consequently, the first node of a list is disjoint from the remaining nodes. However, such a proof requires induction and thus is out of reach of SMT solvers. We add this property as a lemma (lines 11–12), using a separation predicate `sep_node_islist` (introduced at line 10). This lemma is analogous to the $\star$ used in the definition of `islist` in separation logic. To account for the $\star$ in the loop invariant, we first introduce a new separation predicate `sep_islist_islist` (line 14) and then we use it in the loop invariant (line 21).

With these annotations, the axiomatizations and the definitions automatically generated for `sep_node_islist` and `sep_islist_islist` allow a general-purpose SMT solver such as Alt-Ergo or CVC3 to discharge all verification conditions obtained by weakest precondition for the code in Fig. 1, in no time.

3 Separation Predicates

3.1 Inductive Definitions

A separation predicate is generated from user-defined inductive predicates. The generation is sound only if the definitions of the inductive predicates obey several constraints, the main one being that two distinct cases should not overlap. Fortunately, this is the case for most common inductive predicates. For instance, predicate `islist` from Fig. 1 (lines 4–8) trivially satisfies the non-overlapping constraint, since `p` cannot be both null and non-null.

Generally speaking, we consider inductive definitions following the syntax given in Fig. 2. The constraints are then the following:

```
struct node { int hd; struct node *next; };

/*@
inductive islist(struct node *p) {
 case nil: islist(\null);
 case cons: \forall struct node *p; p != \null ==> \valid(p) ==>
 islist(p->next) ==> islist(p);
}

#Gen_Separation sep_node_islist(struct node*, islist)
lemma list_sep:
  \forall struct node *p; p!=null ==>
    islist(p) ==> sep_node_islist(p, p->next);

#Gen_Separation sep_islist_islist(islist, islist)
@*/

/*@ requires islist(p); ensures islist(\result); @*/
struct node * rev(struct node *p) {
  struct node *q = NULL;
  /*@ loop invariant
        islist(p) && islist(q) && sep_islist_islist(p,q); @*/
  while(p != NULL) {
    struct node *tmp = p->next;
    p->next = q;
    q = p;
    p = tmp;
  }
  return q;
}
```

Fig. 1. List Reversal

$$
\begin{array}{rrcl}
\text{(terms)} & t & ::= & x \mid t{\rightarrow}\texttt{field} \mid \phi(\boldsymbol{t}) \\
\text{(formulas)} & f & ::= & t = t \mid \neg(t = t) \mid p(\boldsymbol{x}) \\
\text{(inductive case)} & c & ::= & \texttt{C} : \forall \boldsymbol{x}.f \Rightarrow \ldots \Rightarrow f \Rightarrow p(\boldsymbol{x}) \\
\text{(inductive definition)} & d & ::= & \texttt{inductive}\, p(\boldsymbol{x}) = c \texttt{|} \ldots \texttt{|} c
\end{array}
$$

Fig. 2. Inductive Definitions

- in a term t, a function symbol ϕ cannot refer to the memory state;
- in a formula f, a predicate symbol p can refer to the memory state only if it is an inductively defined predicate following the constraints (which includes the predicate being defined);
- if $\texttt{C}_\texttt{i} : \forall \boldsymbol{x}.f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ and $\texttt{C}_\texttt{j} : \forall \boldsymbol{x}.f_{j,1} \Rightarrow \ldots \Rightarrow f_{j,n_j} \Rightarrow p(\boldsymbol{x})$ are two distinct cases of $\texttt{inductive}\, p(\boldsymbol{x}) = c_1 \texttt{|} \ldots \texttt{|} c_n$, then we should have

$$\forall \boldsymbol{x}. \neg(f_{i,1} \wedge \cdots \wedge f_{i,n_i} \wedge f_{j,1} \wedge \cdots \wedge f_{j,n_j}).$$

It is worth pointing out that an inductive predicate which is never used to define a separation predicate does not have to follow these restrictions.

3.2 An Axiomatization of Footprints

The footprint of an inductive predicate p is the set of pointers which it depends on. More precisely, in a memory state m where $p(\boldsymbol{x})$ is true, the pointer q is in the footprint of $p(\boldsymbol{x})$ if we can modify the value q points at such that $p(\boldsymbol{x})$ does not hold anymore. Such a definition is too precise to be used in practice. We use instead a coarser notion of footprint, which is derived from the definition of p and over-approximates the precise footprint.

Let us consider the definition of `islist`. First, we introduce a new type `ft` for footprints. Then we declare a function symbol $\texttt{ft}_{\texttt{islist}}$ and a predicate symbol $\in$. The intended semantics is the following: $\texttt{ft}_{\texttt{islist}}(m,p)$ is the footprint of $\texttt{islist}(p)$ in memory state m and $q \in \texttt{ft}_{\texttt{islist}}(m,p)$ means that q belongs to the footprint $\texttt{ft}_{\texttt{islist}}(m,p)$. Both symbols are axiomatized simultaneously as follows:

$$\forall q.\forall m.\forall p.\, q \in \texttt{ft}_{\texttt{islist}}(m,p) \Leftrightarrow \left(\begin{array}{c} p \neq \texttt{NULL} \wedge \texttt{islist}(m, \{p\!\to\!\texttt{next}\}_m) \\ \wedge(q = p \vee q \in \texttt{ft}_{\texttt{islist}}(m, \{p\!\to\!\texttt{next}\}_m)) \end{array}\right)$$

where $\{p\!\to\!\texttt{next}\}_m$ stands for expression $p\!\to\!\texttt{next}$ in memory state m.

Then separation predicates are easily defined from footprints. The pragma from line 10 in Fig. 1 generates the definition

$$\texttt{sep\_node\_islist}(m,q,p) \triangleq q \notin \texttt{ft}_{\texttt{islist}}(m,p)$$

and pragma from line 14 generates the definition

$$\begin{array}{l}\texttt{sep\_islist\_islist}(m,p_1,p_2) \triangleq \\ \quad \forall q.\, q \notin \texttt{ft}_{\texttt{islist}}(m,p_1) \vee q \notin \texttt{ft}_{\texttt{islist}}(m,p_2)\end{array}$$

(where $q \notin s$ stands for $\neg(q \in s)$). The predicate symbols and the types that appears in the pragma specify the signature of the separation predicate and which inductive predicate must be used to defined the separation predicate. A type is viewed as the predicate symbol of an unary predicate of this type whose footprint is reduced to its argument. The signature of the defined separation predicate is the concatenation of the signature of the predicate symbols.

Generally speaking, in order to axiomatize the footprint of an inductive predicate, we first introduce a meta-operation $\texttt{FT}_{m,q}(e)$ that builds a formula expressing that q is in the footprint of a given expression e in memory state m:

$$\begin{aligned}
\texttt{FT}_{m,q}(x) &= \bot \\
\texttt{FT}_{m,q}(t\!\to\!\texttt{j}) &= \texttt{FT}_{m,q}(t) \vee q = t \\
\texttt{FT}_{m,q}(\phi(\boldsymbol{t})) &= \bigvee_j \texttt{FT}_{m,q}(t_j) \\
\texttt{FT}_{m,q}(t_1 = t_2) &= \texttt{FT}_{m,q}(\neg(t_1 = t_2)) = \texttt{FT}_{m,q}(t_1) \vee \texttt{FT}_{m,q}(t_2) \\
\texttt{FT}_{m,q}(p(\boldsymbol{t})) &= \bigvee_j \texttt{FT}_{m,q}(t_j) \vee q \in \texttt{ft}_p(m,\boldsymbol{t})
\end{aligned}$$

We pose $q \in \mathtt{ft}_p(m, \boldsymbol{t}) \triangleq \bot$ whenever predicate p does not depend on the memory state. Then the footprint of an inductive predicate p defined by $\mathtt{inductive}\, p(\boldsymbol{x}) = c_1 | \ldots | c_n$ with c_i being $\mathtt{C_i} : \forall \boldsymbol{x}. f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ is axiomatized as follows:

$$\forall q. \forall m. \forall \mathbf{x}.\, q \in \mathtt{ft}_p(m, \mathbf{x}) \Leftrightarrow \bigvee_i \left(\bigwedge_j \overline{f_{i,j}} \wedge \bigvee_j \mathtt{FT}_{m,q}(f_{i,j}) \right)$$

where $\overline{f_{i,j}}$ is the version of $f_{i,j}$ with the memory explicited (eg. $\overline{t \to \mathtt{j}} = \{t \to \mathtt{j}\}_m$). In the axiom above for the footprint of `islist`, we simplified the `NULL` case since it is equivalent to $\bot$.

With the footprints of the inductive predicates you can now define the separation predicate. A separation predicate that define the separation of n inductive predicates is defined as the conjunction of all the disjunction $q \in \mathtt{ft}_{p_i}(m, \boldsymbol{x_i}) \vee q \in \mathtt{ft}_{p_j}(m, \boldsymbol{x_j})$ between the footprint of the inductive predicate. The soundness of this construction have been proved in [3].

The separation predicates allow you to translate a large set of separation logic formulas, namely first-order separation logic formula without magic wand and with separation conjunction used only on inductive predicates which definitions satisfy our constraints.

3.3 Mutation Axioms

The last ingredient we generate is a mutation axiom. It states the main property of the footprint, namely that an assignment outside the footprint does not invalidate the corresponding predicate. In the case of `islist`, the mutation axiom is

$$\forall m, p, q, v.\, q \notin \mathtt{ft_{islist}}(m, p) \Rightarrow \mathtt{islist}(m, p) \Rightarrow \mathtt{islist}(m[q \to \mathtt{next} := v], p)$$

where $m[q \to \mathtt{next} := v]$ stands for a new memory state obtained from m by assigning value v to memory location $q \to \mathtt{next}$. Actually, this property could be proved from the definition of $\mathtt{ft_{islist}}$, but this would require induction. Since this is out of reach of SMT solvers, we state it as an axiom. We do not require the user to discharge it as a lemma, since it is proved sound in the meta-theory [3]. This is somehow analogous to the mutation rule of separation logic, which is proved sound in the meta-theory. The mutation rule of separation logic also allows proving that two formulas stay separated if you modify something separated from both of them. We can prove the same by adding an autoframe axiom, which is reminiscent of the autoframe concept in dynamic frames [16]:

$$\begin{aligned} \forall m, p, q, v.\; & q \notin \mathtt{ft_{islist}}(m, p) \Rightarrow \mathtt{islist}(m, p) \Rightarrow \\ & \mathtt{ft_{islist}}(m, p) = \mathtt{ft_{islist}}(m[q \to \mathtt{next} := v], p) \end{aligned}$$

Generally speaking, for each inductive predicate p and for each field `field` we add the following axioms :

$$\forall q. \forall v. \forall m. \forall \boldsymbol{x}. \neg q \in \mathtt{ft}_p(m, \boldsymbol{x}) \Rightarrow p(m, \boldsymbol{x}) \Rightarrow p(m[q \to \mathtt{field} := v], \boldsymbol{x})$$

and

$$\forall q.\forall v.\forall m.\forall \boldsymbol{x}.\ \neg q \in \mathtt{ft}^c_p(m, \boldsymbol{x}) \Rightarrow p(m, \boldsymbol{x}) \Rightarrow \\ \mathtt{ft}_p(m, \mathtt{x}) = \mathtt{ft}_p(m[q \rightarrow \mathtt{field} := v], \boldsymbol{x}).$$

The distinctness of the cases of the inductive predicate p appears in the proof of the autoframe property.

3.4 Implementation

Our generation of separation predicates is implemented in the Frama-C/Jessie tool chain for the verification of C programs [11,21,5]. This tool chain can be depicted as follows:

$$\begin{tikzcd} \mathtt{file.c} \arrow[r] & \boxed{\text{Frama-C}} \arrow[r] & \boxed{\text{Jessie}} \arrow[r] & \boxed{\text{Why3}} \arrow[r] & \text{theorem provers} \end{tikzcd}$$

From a technical point of view, our implementation is located in the Jessie tool, since this is the first place where the memory model is made explicit[1]. Jessie uses the component-as-array model also known as the Burstall-Bornat memory model [7,6]. Each structure field is modeled using a distinct applicative array. Consequently, function and predicate symbols such as $\mathtt{ft}_{\mathtt{islist}}$ or `islist` do not take a single argument m to denote memory state, but one or several applicative arrays instead, one for each field mentioned in the inductive definition. Similarly, a quantification $\forall m$ in our meta-theory (Sec. 3.2 and 3.3 above) is materialized in the implementation by one or several quantifications over applicative arrays, one for each field appearing in the formula. In the case of `islist`, for instance, quantification $\forall m$ becomes $\forall \mathtt{next}$, expression $\{p \rightarrow \mathtt{next}\}_m$ becomes $\mathtt{get}(\mathtt{next}, p)$, and expression $m[q \rightarrow \mathtt{next} := v]$ becomes $\mathtt{set}(\mathtt{next}, p, v)$, where `get` and `set` are access and update operations over applicative arrays. Additionally, we have to define one footprint symbol for each field.

It is worth pointing out that we made no modification at all in Why3 to support our separation predicates. Only Jessie has been modified.

4 A Case Study: Composite Pattern

To show the usefulness of separation predicates, we consider the problem of verifying an instance of the *Composite Pattern*, as proposed in the VACID-0 benchmark [20].

4.1 The Problem

We consider a forest, whose nodes are linked upward through parent links. Each node carries an integer value, as well as the sum of the values for all nodes in its subtree (including itself). The corresponding C structure is thus defined as follows:

[1] Since we could not extend the ACSL language with the new pragmas for separation, we have to modify the Jessie input file manually at each run. Furthermore we use in the assigns clauses the keyword `\all` that does not exist yet in ACSL.

```
struct node {
  int val, sum;
  struct node *parent;
};
typedef struct node *NODE;
```

The operations considered here are the following: `NODE create(int v);`, creates a new node; `void update(NODE p, int v);`, assigns value `v` to node `p`; `void addChild(NODE p, NODE q);`, set node `p` as `q`'s parent, assuming node `q` has no parent; `void dislodge(NODE p);`, disconnects `p` from its parent, if any.

One challenge with such a data structure is that operations `update`, `addChild`, and `dislodge` have non-local consequences, as the `sum` field must be updated for all ancestors. Another challenge is to prevent `addChild` from creating a cycle, *i.e.*, to express that node `q` is not already an ancestor of node `p`. Thus we prove the memory safety and the correct behavior of these operations.

4.2 Code and Specification

Our annotated C code for this instance of the composite pattern is given in the appendix. In this section, we comment on the key aspects of our solution. The annotations are written in the ACSL specification language. The behavior of the functions are defined by contract: the keyword `requires` introduces the precondition expressed by a first-order formula, the keyword `ensures` introduces the post-conditions, and the keyword `assigns` introduces the set of memory location that can be modified by a call to the function. The precondition and this set are interpreted before the execution of the function, the post-conditions is interpreted after. One can refer in the post-condition to the state before the execution of the function using the keyword `\old`. It must be remarked that if a field of a type is never modified in the body of a function you don't need to mention it in the assigns clauses. Moreover the component-as-array memory model ensures without reasoning that any formulas that depend only of such fields remain true after a call to the function.

Separation Predicate. For the purpose of `addChild`'s specification, we use a separation predicate. It states that a given node is disjoint from the list of ancestors of another node. Such a list is defined using predicate `parents` (lines 7–12), which is similar to predicate `islist` in the previous section. The separation predicate, `sep_node_parents`, is then introduced on line 14 and used in the precondition of `addChild` on line 84.

This is a crucial step, since otherwise assignment `q->parent = p` on line 95 could break property `parents(p)`. Such a property is indeed required by `upd_inv` to ensure its termination.

Restoring the Invariant. As suggested in VACID-0 [20], we introduce a function to restore the invariant (function `upd_inv` on lines 68–77). Given a node `p` and an offset `delta`, it adds `delta` to the `sum` field of `p` and of all its ancestors.

This way, we reuse this function in `addChild` (with the new child's sum), in `update` (with the difference), and in `dislodge` (with the opposite of the child's sum).

Local and Global Invariant. Another key ingredient of the proof is to ensure the invariant property that, for each node, the `sum` field contains the sum of values in all nodes beneath, including itself. To state such a property, we need to access children nodes. Since the data structure does not provide any way to do that (we only have parent links), we augment the data structure with ghost children links. To make it simple, we assume that each node has at most two children, stored in ghost fields `left` and `right` (line 4). Structural invariants relating fields `parent`, `left`, and `right` are gathered in predicate `wf` (lines 28–37).

To state the invariant for `sum` fields, we first introduce a predicate `good` (lines 20–23). It states that the `sum` field of a given node `p` has a correct value when `delta` is added to it. It is important to notice that predicate `good` is a *local* invariant, which assumes that the left and right children of `p` have correct sums. Then we introduce a predicate `inv` (lines 25–26) to state that any node `p` verifies `good(p, 0)`, with the possible exception of node `except`. Using an exception is convenient to state that the invariant is locally violated during `upd_inv`. To state that the invariant holds for all nodes, we simply use `inv(NULL)`.

Our local invariant is convenient, as it does not require any induction. However, to convince the reader that we indeed proved the expected property, we also show that this local invariant implies a global, inductively-defined invariant. Lines 130–137 introduce the sum of all values in a tree, as an inductive predicate `treesum`, and a lemma to state that local invariant `inv(NULL)` implies `treesum`$(p, p \to$ `sum`$)$ for any node p.

4.3 Proof

The proof was performed using Frama-C Carbon[2] and its Jessie plug-in [21], using SMT solvers Alt-Ergo 0.92.3, CVC3 2.2, and Z3 2.19, on an Intel Core Duo 2.4 GHz. As explained in Sec. 3.4, we first run Frama-C on the annotated C code and then we insert the separation pragmas in the generated Jessie code (this is a benign modification). All verification conditions are discharged automatically within a total time of 30 seconds.

The two lemmas `parents_sep` and `global_invariant` were proved interactively using the Coq proof assistant version 8.3pl3 [26]. A total of 100 lines of tactics is needed. It doesn't take more than three days for one of the author to find the good specifications and make the proofs.

5 Function Footprints

In the case of the composite pattern, it is easy to specify the footprints of the C functions. Indeed, we can simply say that any `sum` field may be modified

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

(using `\all->sum` in `assigns` clauses), since the invariant provides all necessary information regarding the contents of `sum` fields. For a function such as list reversal, however, we need to be more precise. We want to know that any list separated from the one being reversed is left unmodified. For instance, we would like to be able to prove the following piece of code:

```
/*@
requires islist(p) && islist(q) && sep_list_list(p,q);
ensures  islist(p) && islist(q) && sep_list_list(p,q);
@*/
void bar(struct node * p, struct node * q) {
  p = rev(p);
}
```

For that purpose we must strengthen the specification and loop invariant of function `rev` with a suitable frame property. One possibility is to proceed as follows:

```
/*@
#Gen_Frame: list_frame list
#Gen_Sub: list_sub list list

requires list(p);
ensures  list(\result) && list_frame{Old,Here}(p,result);
@*/
struct node * rev(struct node * p);
  ...
  /*@ loop invariant
        list(p) && list(q) && sep_list_list(p,q)
        && list_frame{Init,Here}(\at(p,Init),q)
        && list_sub{Init,Here}(\at(p,Init),p); @*/
...
```

Two pragmas introduce new predicates `list_frame` and `list_sub`. Both depend on two memory states. The formula `list_frame{Old,Here}(p,result)` expresses in the post-condition that, between pre-state `Old` and post-state `Here`, all modified pointers belong to list `p`. It also specifies that the footprint of list `result` is included in the (old) footprint of list `p`. On the example of function `bar`, we now know that only pointers from `p` have been modified, so we can conclude that `islist(q)` is preserved. Additionally, we know that the footprint of `islist(p)` has not grown so we can conclude that it is still separated from `islist(q)`. The formula `list_sub{Init,Here}(\at(p,Init),p)` specifies only the inclusion of the footprint of the lists.

These two predicates could be axiomatized using membership only. For instance, $\texttt{list\_sub}(p,q)$ could be simply axiomatized as $\forall x, x \in \texttt{ft}_{\texttt{islist}}(p) \Rightarrow x \in \texttt{ft}_{\texttt{islist}}(q)$. But doing so has a rather negative impact on SMT solvers, as they have to first instantiate this axiom and then to resort to other axioms related to membership. Moreover this axiom is very generic and can be applied when not needed. For that reason we provide, in addition to axioms related to membership, axioms for footprint inclusion, to prove either $s \subset \texttt{ft}_p(p)$ or

$\mathtt{ft}_p(p) \subset s$ directly. With such axioms, functions `rev` and `bar` are proved correct automatically.

6 Related and Future Work

VeriFast [13,15] allows user-defined predicates but requires user annotations to fold or unfold these predicates. In our work, we rely instead on the capability of first-order provers to fold and unfold definitions. VeriFast uses the SMT solver Z3, but only as a constraint solver on ground terms.

The technique of *implicit dynamic frames* [24] is closer to our work, except that formulas are restricted. Additionally, implicit dynamic frames make use of a set theory, whereas we do not require any, as we directly encode the relevant parts of set theory inside our footprint definition axioms.

Both these works do not allow a function to access (and thus modify) a pointer that is not in the footprint of the function's precondition — except if it is allocated inside the function. In our work, we do not have such a restriction. When necessary, we may define the footprint of a function using separation predicates, as explained in the first author's thesis [3].

There exist already several proofs of the composite pattern. One is performed using VeriFast [14]. It requires many lemmas and many `open`/`close` statements, whereas our proof does not contain much proof-related annotations.

The use of a local invariant in our proof is not new. It was first described in [19]. The proof by Rosenberg, Banerjee, and Naumann [23] also makes use of it. In order to prove that `addChild` is not creating cycles, the latter proof introduces two ghost fields, one for the set of descendants and one for the root node of the tree. Updating these ghost fields must be done at several places. In our case, we could manage to perform the case only with the generated predicate `sep_node_parents` without need of extra ghost fields which leads to a simpler proof.

The composite pattern has also been proved using *considerate reasoning* [25], a technique that advocates for local invariant like the one we used. Our predicate `inv` is similar to their `broken` declaration. As far as we understand, this proof is not mechanized, though.

Our future work includes generalizing the frame pragma used to describe the footprint of a function. One solution is to compute the footprint directly from ACSL's `assigns` clause, if any. Another is to describe the footprint using the linear maps framework [18]. One valuable future work would be to formally prove the consistency of our axioms, either using a meta-theoretical formalization, or, in a more tractable way, by producing proofs for each generated axiom.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)

2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
3. Bobot, F.: Logique de séparation et vérification déductive. Thèse de doctorat, Université Paris-Sud (December 2011)
4. Bobot, F., Conchon, S., Contejean, É., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), `http://alt-ergo.lri.fr/`
5. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
6. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
7. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
8. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. de Moura, L., Dutertre, B.: Yices: An SMT Solver, `http://yices.csl.sri.com/`
10. Filliâtre, J.-C., Paskevich, A., Stump, A.: The 2nd Verified Software Competition (November 2011), `https://sites.google.com/site/vstte2012/compet`
11. The Frama-C platform for static analysis of C programs (2008), `http://www.frama-c.cea.fr/`
12. Huisman, M., Klebanov, V., Monahan, R.: (October 2011), `http://foveoos2011.cost-ic0701.org/verification-competition`
13. Jacobs, B., Piessens, F.: The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (August 2008)
14. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. In: Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track (November 2008)
15. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
16. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
17. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience Report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011), Materials available at `www.vscomp.org`
18. Lahiri, S.K., Qadeer, S., Walker, D.: Linear maps. In: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification, PLPV 2011, pp. 3–14. ACM, New York (2011)
19. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing (2007)
20. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Proceedings of Tools and Experiments Workshop at VSTTE (2010)

21. Moy, Y., Marché, C.: The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual. INRIA & LRI (2011), http://krakatoa.lri.fr/
22. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science. IEEE Comp. Soc. Press (2002)
23. Rosenberg, S., Banerjee, A., Naumann, D.A.: Local Reasoning and Dynamic Framing for the Composite Pattern and Its Clients. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 183–198. Springer, Heidelberg (2010)
24. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
25. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Design Pattern. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
26. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), http://coq.inria.fr

A Annotated Source Code

```
typedef struct node {
  int val, sum;
  struct node *parent;
  //@ ghost struct node *left, *right;
} *NODE;

/*@ inductive parents(NODE p) {
  case nil: \forall NODE p; p==NULL ==> parents(p);
  case cons: \forall NODE p;
    p != NULL ==> \valid(p) ==>
    parents(p->parent) ==> parents(p);
}

#Gen_Separation sep_node_parents(NODE, parents)

lemma parents_sep:
  \forall NODE p; p!=NULL ==>
    parents(p) ==> sep_node_parents(p, p->parent);

predicate good(NODE p, int delta) =
  p->sum + delta == p->val +
   (p->left == NULL? 0 : p->left->sum) +
   (p->right == NULL? 0 : p->right->sum);

predicate inv(NODE except) =
  \forall NODE p; \valid(p) ==> p != except ==> good(p, 0);

predicate wf(NODE except) =
  \forall NODE p; \valid(p) ==> p != except ==>
    (p->right != NULL ==>
       p->right->parent == p && \valid(p->right)) &&
    (p->left != NULL ==>
       p->left->parent == p && \valid(p->left)) &&
    (p->right == p->left ==> p->right == NULL) &&
    (p->parent != NULL ==> \valid(p->parent)) &&
    (p->parent != NULL ==>
       p->parent->left == p || p->parent->right == p);

predicate newnode(NODE p, integer v) =
       parents(p) && p->right == NULL && p->left == NULL &&
       p->parent == NULL && p->val == v && \valid(p);
@*/

/*@ requires
      inv(NULL) && wf(NULL);
    ensures
      inv(NULL) && wf(NULL) && newnode(\result, v) &&
      \forall NODE n; \old(\valid(n)) ==>
         \result != n && \valid(n) &&
         \old(n->val) == n->val && \old(n->parent) == n->parent &&
         \old(n->left) == n->left && \old(n->right) == n->right;
@*/
NODE create(int v) {
 Before:
    {
  NODE p = (NODE)malloc(sizeof(struct node));
  /*@ assert \forall NODE n; n != p ==>
             \valid(n) ==> \at(\valid(n),Before); @*/
  p->val = p->sum = v;
  p->parent = p->left = p->right = NULL;
  return p;
}}

/*@ requires inv(p) && parents(p) && wf(NULL) && good(p,delta);
    ensures inv(NULL);
    assigns \all->sum;
 @*/
void upd_inv(NODE p, int delta) {
   NODE n = p;
   /*@ loop invariant
         inv(n) && parents(n) && (n != NULL ==> good(n,delta));
    @*/
    while (n != NULL) {
      n->sum = n->sum + delta;
      n = n->parent;
    }
};

/*@
  requires
    inv(NULL) && wf(NULL) &&
    \valid(q) && q->parent == NULL &&
    parents(p) && p != NULL && sep_node_parents(p, p->parent) &&
    (p->left == NULL || p->right == NULL) && sep_node_parents(q,p);
  ensures
    parents(q) && parents(p) && inv(NULL) && wf(NULL) &&
    (\old(p->left) == NULL ==>
       p->left == q && \old(p->right) == p->right) &&
    (\old(p->left) != NULL ==>
       p->right == q && \old(p->left) == p->left);
  assigns p->left, p->right, q->parent, \all->sum;
@*/
void addChild(NODE p, NODE q) {
  if (p->left == NULL) p->left = q; else p->right = q;
  q->parent = p;
  upd_inv(p, q->sum);
}

/*@ requires parents(p) && p != NULL && inv(NULL) && wf(NULL);
    ensures p->val == v && parents(p) && inv(NULL) && wf(NULL);
    assigns p->val, \all->sum;
@*/
void update(NODE p, int v) {
   int delta = v - p->val;
   p->val = v;
   upd_inv(p, delta);
}

/*@
  requires
    parents(p) && p != NULL && p->parent != NULL &&
    inv(NULL) && wf(NULL);
  ensures
    parents(p) && p->parent == NULL && inv(NULL) && wf(NULL) &&
    (\old(p->parent->left) == p ==>
       \old(p->parent)->left == NULL) &&
    (\old(p->parent->right) == p ==>
       \old(p->parent)->right == NULL);
  assigns p->parent->left, p->parent->right, p->parent, \all->sum;
@*/
void dislodge(NODE p) {
  NODE n = p->parent;
  if(p->parent->left == p) p->parent->left = NULL;
  if(p->parent->right == p) p->parent->right = NULL;
  p->parent = NULL;
  upd_inv(n, -p->sum);
}

/*@
inductive treesum{L}(NODE p, integer v) {
  case treesum_null{L}:
    treesum(NULL, 0);
  case treesum_node{L}:
    \forall NODE p; p != NULL ==> \forall integer sl, sr;
    treesum(p->left, sl) ==> treesum(p->right, sr) ==>
    treesum(p, p->val + sl + sr);
}

lemma global_invariant{L}:
  inv(NULL) ==> wf(NULL) ==>
  \forall NODE p; \valid(p) ==> treesum(p, p->sum);
@*/
```

Certifying and reasoning about cost annotations of functional programs *

Roberto M. Amadio(1) Yann Régis-Gianas(2)
(1) Université Paris Diderot (UMR-CNRS 7126)
(2) Université Paris Diderot (UMR-CNRS 7126) and INRIA (Team πr^2)

January 11, 2013

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, typable compilation chain to assembly code including safe memory management, and to reason about them in a higher-order Hoare logic.

1 Introduction

In previous work [2, 3], 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 by A. Perlis [22]: *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.

1.1 A standard compilation chain

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 by Morrisett *et al.* [21] (with an emphasis on typing but with no simulation proofs) and by Chlipala [9] (for type-free languages but with machine certified simulation proofs).

*An extended abstract with the same title without proofs and *not* accounting for the typing of the compilation chain and the memory management of the compiled code has appeared in [4]. Also the present version introduces a prototype implementation available in [24]. The authors were supported by the *Information and Communication Technologies (ICT) Programme* as Project FP7-ICT-2009-C-243881 CerCo.

$$\begin{tikzcd}
\lambda^{\mathcal{M}} & \lambda^{\ell} \arrow[l, "\mathcal{I}"'] \arrow[r, "\mathcal{C}_{cps}"] \arrow[d, bend left, "er"] & \lambda^{\ell}_{cps} \arrow[r, shift left, "\mathcal{C}_{vn}"] \arrow[d, "er"] & \lambda^{\ell}_{cps,vn} \arrow[l, shift left, "\mathcal{R}"] \arrow[r, "\mathcal{C}_{cc}"] \arrow[d, "er"] & \lambda^{\ell}_{cc,vn} \arrow[r, "\mathcal{C}_{h}"] \arrow[d, "er"] & \lambda^{\ell}_{h,vn} \arrow[d, "er"] \\
& \lambda \arrow[u, bend left, "\mathcal{L}"] \arrow[r, "\mathcal{C}_{cps}"] & \lambda_{cps} \arrow[r, shift left, "\mathcal{C}_{vn}"] & \lambda_{cps,vn} \arrow[l, shift left, "\mathcal{R}"] \arrow[r, "\mathcal{C}_{cc}"] & \lambda_{cc,vn} \arrow[r, "\mathcal{C}_{h}"] & \lambda_{h,vn}
\end{tikzcd}$$

Table 1: The compilation chain with its labelling and instrumentation.

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 that gives names to values, 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. One advantage of this approach, as already noted for instance by Fradet and Le Métayer [14], is to have a homogeneous notation that makes correctness proofs simpler.

Notable differences with respect to Chlipala's compilation chain [9] is a different choice of the intermediate languages and the fact that we rely on a small-step operational semantics. We also diverge from Chlipala [9] 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.

The final language of this compilation chain can be directly mapped to a 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.

1.2 The labelling approach to cost certification

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, written 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 over-approximates 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).

In our previous work [2, 3], 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 labelling of 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 [2, 3] 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 Chlipala's Coq development [9] including fixpoints, sums, exceptions, and side effects. Let us also mention that our approach has 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 which, roughly speaking, is a type-free, stack aware fragment of C defined in the COMPCERT project [17].

1.3 Reasoning about the certified cost annotations

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 Gurr's PhD thesis [15] 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 the higher-order Hoare logic co-developed by one of the authors [25] can be employed to reason about the concrete complexity of source programs by proving properties on their instrumented versions (see Table 11 for an example of a source program with complexity assertions).

1.4 Accounting for the cost of memory management

In a realistic implementation of a functional programming language, the runtime environment usually includes a garbage collector. In spite of considerable progress in *real-time garbage collectors* (see, *e.g.*, the work of Bacon *et al.* [6]), it seems to us that this approach does not offer yet a viable path to a certified and usable WCET prediction of the running time of functional programs. Instead, the approach we shall adopt, following the seminal work of Tofte *et al.* [27], is to enrich the last calculus of the compilation chain described in Table 1, (1) with a notion of *memory region*, (2) with operations to allocate and dispose memory regions, and (3) with a type and effect system that guarantees the safety of the dispose operation. This allows to further extend to the right with one more commuting square the compilation chain mentioned above and then to include the cost of safe memory management in our analysis. Actually, because effects are intertwined with types, what we shall actually do, following the work of Morrisett *et al.* [21], is to extend a *typed* version of the compilation chain.

1.5 Related work

There is a long tradition starting from the work of Wegbreit [30] which reduces the complexity analysis of *first-order* functional programs to the solution of finite difference equations. Much less is known about higher-order functional programs. Most 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 [26] 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.

A more recent work by Bonenfant *et al.* [7] addresses the problem of determining the worst case execution time of a specialised functional language called Hume. The compilation chain considered consists in first compiling 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 [5] 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. First, the Hume approach follows a tradition of compiling functional programs to the instructions of an abstract machine which is then implemented in a C like language. In contrast, we have considered a compilation chain that brings a functional program directly to RTL form. Then the back-end of a C like compiler is used to generate binary instructions. Second, the cited work [7] does not address at all the proof of correctness of the cost annotations; this is left for future work. Third, the problem of producing synthetic cost statements starting from the cost estimations of the abstract instructions of the Hume machine is not considered. Fourth, the cost of dynamic memory management, which is crucial in higher-order functional programs, is not addressed at all. Fifth, the granularity of the cost annotations is fixed in Hume [7] (the instructions of the Hume abstract machine) while it can vary in our approach.

We also share with the Hume approach one limitation. The precision of our analyses depends on the possibility of having accurate predictions of the execution time of relatively short sequences of binary code on a given processor. Unfortunately, as of today, user interfaces for WCET systems such as the aiT tool mentioned above or Chronos [19] do not support modular reasoning on execution times and therefore experimental work focuses on processors with simple and predictable architectures. In a related direction, another potential loss of precision comes from the introduction of *aggressive* optimisations in the back-end of the compiler such as loop transformations. An ongoing work by Tranquilli [28] addresses this issue by introducing a refinement of the labelling approach.

1.6 Paper organisation

In the following, section 2 describes the certification of the cost-annotations, section 3 a method to reason about the cost annotations, section 4 the typing of the compilation chain, and section 5 an extension of the compilation chain to account for safe memory deallocation. Proofs are available in the appendix A.

2 The compilation chain: commutation and simulation

We describe the intermediate languages and the compilation functions from an ordinary λ-calculus to a hoisted, value named 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 assumed to be acquainted with the type-free and typed λ-calculus, its evaluation strategies, and its continuation passing style translations [29]. 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. With this assumption, we can omit obvious side conditions on binders and free variables. Similar conventions are applied when reasoning about 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 $\ell, \ell_1, \ldots$ For each calculus, we specify the syntactic categories and the reduction rules. For the sake of clarity, the meta-variables of these syntactic categories are sometimes shared between several calculus: the context is always sufficiently precise to determine to which syntax definitions we refer. We let α range over labels and the empty word ϵ. 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 write $\xrightarrow{*}$ for the reflexive and transitive closure of $\to$. We also define $M \overset{\alpha}{\Rightarrow} N$ as $M \xrightarrow{*} N$ if $\alpha = \epsilon$ and as $M \xrightarrow{*}\xrightarrow{\alpha}\xrightarrow{*} N$ otherwise.

Given a term M in one of the labelled languages we write $M \Downarrow_\Lambda N$ if $M \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} N$, $\Lambda = \alpha_1 \cdots \alpha_n$, and N cannot reduce (in general this does not imply that N is a value). We write $M \Downarrow_\Lambda$ for $\exists N \ M \Downarrow_\Lambda N$. Also, if the term M is unlabelled, Λ is always the empty word and we abbreviate $M \Downarrow_\epsilon N$ with $M \Downarrow N$.

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 $\lambda x_1 \ldots x_n.M$, $[V^+/x^+]M$ stands for $[V_1/x_1, \ldots, V_n/x_n]M$, 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, left-to-right call-by-value λ-calculus. The calculus includes *let-definitions* and *polyadic abstraction* and *tupling* 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. Roughly speaking, every λ-abstraction is a potential starting point for a loop in the control-flow graph. Thus, we will need the body of every λ-abstraction to be pre-labelled so as to maintain the invariant that all loops go through a labelled node in the control-flow graph. As the CPS translation introduces new λ-abstractions that are not present in the source code but correspond to the image of some applications, we will also need to post-label these particular applications so that the freshly introduced λ-abstraction can be assigned a label.

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. To avoid all ambiguity, let us assume that $(V_1, \ldots, V_n) \mid K$ is translated according to the case for values, but note that if we follow the general case for tuples we obtain the same result. We recall that in a CPS translation each function takes its evaluation context as a fresh additional parameter (see, *e.g.*, the work of Wand [29], for an elaboration of this idea). The results of the evaluation of subterms (of tuples and of applications) are also named using fresh parameters $x_0, \ldots, x_n$. The initial evaluation context is defined relatively to a fresh variable named '*halt*'. 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

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

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

Label erasure (selected equations)

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

Table 2: An ordinary call-by-value λ-calculus: λ^{ℓ}

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. To that end, we make sure that post-labelling is only applied to terms $M \in W_0$, that is, terms that are not the immediate body of a λ-abstraction (which are in W_1).

Example 2 (labelling and commutation) *Let $M \equiv \lambda x.(@(x, x) > \ell)$. Then $M \notin W_0$ because the rule for abstraction requires $@(x, x) > \ell \in W_1$ while we can only show $@(x, x) > \ell \in W_0$. Notice that we have:*

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

So, for M, the commutation of the CPS translation and the erasure function only holds up to η.

Proposition 3 (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 Plotkin's seminal work [23]. 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 4 (CPS simulation) *Let M be a term of the λ^ℓ-calculus. If $M \xrightarrow{\alpha} N$ then $\mathcal{C}_{cps}(M) \overset{\alpha}{\Rightarrow} \mathcal{C}_{cps}(N)$.*

We illustrate this result on the following example.

Example 5 (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:

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

2.4 Transformation in value named CPS form

Table 4 introduces a *value named* λ-calculus in CPS form: $\lambda^\ell_{cps,vn}$. In the ordinary λ-calculus, the application of a λ-abstraction to an argument (which is a value) may duplicate the argument as in: $@(\lambda x.M, V) \to [V/x]M$. In the value named 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:

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

We also remark that in the value named 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 (see, *e.g.*, [13]).

Table 5 defines the compilation into value named form along with a readback translation. (Only the case for the local binding of values is interesting.) The latter is useful to state the simulation property. Indeed, it is not true that if $M \to M'$ in λ^ℓ_{cps} then $\mathcal{C}_{vn}(M) \xrightarrow{*} \mathcal{C}_{vn}(M')$ in $\lambda^\ell_{cps,vn}$. For instance, consider $M \equiv (\lambda x.xx)I$ where $I \equiv (\lambda y.y)$. Then $M \to II$ but $\mathcal{C}_{vn}(M)$ does not reduce to $\mathcal{C}_{vn}(II)$ but rather to a term where the 'sharing' of the duplicated value I is explicitly represented.

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

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

CPS translation

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

Table 3: CPS λ-calculus (λ^{ℓ}_{cps}) and CPS translation

Example 6 (value named 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 5.) The corresponding term in value named form is:*

$$\begin{array}{l}
\mathsf{let}\ z_1 = \lambda x, k.(\mathsf{let}\ z_{11} = \lambda y.@(x, y, k)\ \mathsf{in}\ @(x, x, z_{11}))\ \mathsf{in} \\
\mathsf{let}\ z_2 = I'\ \mathsf{in} \\
\mathsf{let}\ z_3 = H\ \mathsf{in} \\
@(z_1, z_2, z_3)\ .
\end{array}$$

Proposition 7 (VN commutation) *Let M be a λ-term in CPS form. Then:*

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

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

Proposition 8 (VN simulation) *Let N be a λ-term in CPS value named form. If $\mathcal{R}(N) \equiv M$ and $M \xrightarrow{\alpha} M'$ then there exists N' such that $N \xrightarrow{\alpha} N'$ and $\mathcal{R}(N') \equiv M'$.*

Syntax

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

Reduction Rules

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

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

Table 4: A value named CPS λ-calculus: $\lambda^{\ell}_{cps,vn}$

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 and in representing functions using closures. Our closure conversion implements a closure using a pair whose first component is the code of the translated function and whose second component is a tuple of the values of the free variables.

It will be convenient to write "$\mathsf{let}\ (y_1, \ldots, y_n) = x\ \mathsf{in}\ M$" for "$\mathsf{let}\ y_1 = \pi_1(x)\ \mathsf{in}\ \cdots \mathsf{let}\ y_n = \pi_n(x)\ \mathsf{in}\ M$" and "$\mathsf{let}\ x_1 = C_1 \ldots x_n = C_n\ \mathsf{in}\ M$" for "$\mathsf{let}\ x_1 = C_1\ \mathsf{in} \ldots \mathsf{let}\ x_n = C_n\ \mathsf{in}\ M$". The transformation is described in Table 6. The output of the transformation is such that all functional values are closed. In our opinion, this is the only compilation step where the proofs are rather straightforward.

Example 9 (closure conversion) *Let $M \equiv \mathcal{C}_{vn}(\mathcal{C}_{cps}(\lambda x.y))$, 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:

$$
\begin{array}{l}
\mathsf{let}\ c = \lambda e, x, k.(\mathsf{let}\ (y) = e, (c, e) = k\ \mathsf{in}\ @(c, e, y))\ \mathsf{in} \\
\mathsf{let}\ e = (y), z_1 = (c, e), (c, e) = halt\ \mathsf{in} \\
@(c, e, z_1)\ .
\end{array}
$$

Proposition 10 (CC commutation) *Let M be a CPS term in value named form. Then $er(\mathcal{C}_{cc}(M)) \equiv \mathcal{C}_{cc}(er(M))$.*

Proposition 11 (CC simulation) *Let M be a CPS term in value named form. If $M \xrightarrow{\alpha} M'$ then $\mathcal{C}_{cc}(M) \overset{\alpha}{\Rightarrow} \mathcal{C}_{cc}(M')$.*

Transformation in value named form (from λ^{ℓ}_{cps} to $\lambda^{\ell}_{cps,vn}$)

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

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

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

Table 5: Transformations in value named 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 $\lambda^{\ell}_{cps,vn}$-calculus. As a first step, we analyse the hoisting transformations.

Proposition 12 (on hoisting transformations) *The iteration of the hoisting transformation on a term in $\lambda^{\ell}_{cc,vn}$ (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 13 (hoisting commutation) *Let M be a term of the $\lambda^{\ell}_{cps,vn}$-calculus.*

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

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

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

The proof of the simulation property requires some work because to close the diagram we need to collapse repeated definitions, which may occur, as illustrated in the example below.

Example 14 (hoisting transformations and transitions) *Let*

$$M \equiv \mathsf{let}\ x_1 = \lambda y_1.N\ \mathsf{in}\ @(x_1,z)$$

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

Closure Conversion

$$
\begin{array}{lll}
\mathcal{C}_{cc}(@(x,y^+)) & = \mathsf{let}\ (c,e) = x\ \mathsf{in}\ @(c,e,y^+) & \\[1em]
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \begin{array}{l}\mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\dots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\ \mathsf{let}\ e = (z_1,\dots,z_k)\ \mathsf{in} \\ \mathsf{let}\ x = (c,e)\ \mathsf{in} \\ \mathcal{C}_{cc}(M) \qquad\qquad (\text{if } C = \lambda x^+.N, \mathsf{fv}(C) = \{z_1,\dots,z_k\})\end{array} \\[1em]
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = \mathsf{let}\ x = C\ \mathsf{in}\ \mathcal{C}_{cc}(M) & (\text{if } C \text{ not a function}) \\[1em]
\mathcal{C}_{cc}(\ell > M) & = \ell > \mathcal{C}_{cc}(M) &
\end{array}
$$

Table 6: Closure conversion on value named CPS terms

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:*

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

or hoist and then reduce:

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

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

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 15 (hoisting simulation) *There is a simulation relation* $\mathcal{T}_h$ *on the terms of the* $\lambda^{\ell}_{cps,vn}$*-calculus such that for all terms* M *of the* $\lambda^{\ell}_{cc,vn}$*-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:

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

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

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

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

Theorem 16 (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* $N \overset{\alpha}{\Rightarrow} N'$ *and* $M'\ \mathcal{R}_C\ N'$*.*

Syntax for λ^{ℓ}_{h}
Syntactic restrictions on $\lambda^{\ell}_{cps,vn}$ 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

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

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

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

Table 7: Hoisting transformation

3 Reasoning about 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. When the index i in $\mathcal{L}_i$ is equal to 1, it attests that M is an immediate body of a λ-abstraction. In that case, even if M is an application, it is not post-labelled. Otherwise, when i is equal to 0, the term M is not an immediate body of a λ-abstraction, and, thus is post-labelled if it is an application.

Example 17 (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.*

Proposition 18 (labelling properties) *Let M be a term of the λ-calculus.*

(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: $\mathcal{C}(M) \equiv er(\mathcal{C}(\mathcal{L}(M)))$.*

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

Table 8: 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,* $\mathcal{C}(\mathcal{L}(M))$ *is a term* P *specified by the following grammar:*

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

Point (2) of the proposition above depends on the commutation property of the compilation function (theorem 16(1)). The point (3) entails that a RTL program generated by the compilation function is composed of a set of routines and that each routine is composed of a sequence of assignments on pseudo-registers and a terminal call to another routine. Points (2) and (3) entail that the only difference between the compiled code and the compiled *labelled* code is that in the latter, upon entering a routine, a label uniquely associated with the routine is emitted.

Now suppose we can compute the cost of running once each routine, where the cost is an element of a suitable commutative monoid $\mathcal{M}$ with binary operation $\oplus$ and identity $\mathbf{0}$ (the reader may just think of the natural numbers). Then we can define a function costof which associates with every label the cost of running once the related routine; the function costof is extended to words of labels in the standard way. A run of a terminating program M corresponds to a finite sequence of routine calls which in turn correspond to the finite sequence of labels that we can observe when running the labelled program. We summarise this argument in the following proviso (a modelling hypothesis rather than a mathematical proposition).

Proviso 19 *For any term* M *of the source language, if* $\mathcal{C}(\mathcal{L}(M)) \Downarrow_\Lambda$ *then* $\mathsf{costof}(\Lambda)$ *is the cost of running* M*.*

We stress that the model at the level of the RTL programs is not precise enough to obtain useful predictions on the execution cost in terms, say, of CPU cycles. However, the compilation chain of this paper can be composed with the back-end of a moderately optimising C compiler described in our previous work [2, 3]. For RTL programs such as those characterized by the grammar above, the back end produces binary code which satisfies the checks for soundness and precision that we outlined in the introduction. This remains true even if the source language is enriched with other constructions such as branching and loops as long as the labelling function is extended to handle these cases.

$$
\begin{array}{lcl}
\psi(x) & = & x \\
\psi(\lambda x^+.M) & = & \lambda x^+.\mathcal{I}(M) \\
\psi(V_1,\dots,V_n) & = & (\psi(V_1),\dots,\psi(V_n)) \\
\\
\mathcal{I}(V) & = & (\mathbf{0},\psi(V)) \\
\mathcal{I}(@(M_0,\dots,M_n)) & = & \mathsf{let}\ (m_0,x_0)=\mathcal{I}(M_0)\cdots(m_n,x_n)=\mathcal{I}(M_n), \\
 & & (m_{n+1},x_{n+1}) = @(x_0,\dots,x_n)\ \mathsf{in} \\
 & & (m_0\oplus m_1\oplus\cdots\oplus m_{n+1},x_{n+1}) \\
\mathcal{I}((M_1,\dots,M_n)) & = & \mathsf{let}\ (m_1,x_1)=\mathcal{I}(M_1)\cdots(m_n,x_n)=\mathcal{I}(M_n)\ \mathsf{in} \\
 & & (m_1\oplus\cdots\oplus m_n,(x_1,\dots,x_n)) \quad ((M_1,\dots,M_n)\text{ not a value}) \\
\mathcal{I}(\pi_i(M)) & = & \mathsf{let}\ (m,x)=\mathcal{I}(M)\ \mathsf{in}\ (m,\pi_i(x)) \\
\mathcal{I}(\mathsf{let}\ x=M_1\ \mathsf{in}\ M_2) & = & \mathsf{let}\ (m_1,x)=\mathcal{I}(M_1)\ \mathsf{in}\ (m_2,x_2)=\mathcal{I}(M_2)\ \mathsf{in} \\
 & & (m_1\oplus m_2,x_2) \\
\mathcal{I}(\ell > M) & = & \mathsf{let}\ (m,x)=\mathcal{I}(M)\ \mathsf{in}\ (m_\ell\oplus m,x) \\
\mathcal{I}(M > \ell) & = & \mathsf{let}\ (m,x)=\mathcal{I}(M)\ \mathsf{in}\ (m\oplus m_\ell,x)
\end{array}
$$

Table 9: Instrumentation of labelled λ-calculus.

3.2 Instrumentation

As already mentioned, given a cost monoid $\mathcal{M}$, we assume the analysis of the RTL code associates with each label ℓ in the term an element $m_\ell = \mathsf{costof}(\ell)$ of the cost monoid. Table 9 describes a monadic transformation, extensively analysed in Gurr's PhD thesis [15], 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.

We assume that the reduction rules of the source language (λ) are extended to account for a call-by-value evaluation of the monoidal expressions, where each element of the monoid is regarded as a value. Then instrumentation and labelling are connected as follows.

Proposition 20 (instrumentation vs. labelling) *Let M be a term of the source labelled language λ^ℓ. If $\mathcal{I}(M) \Downarrow (m,V)$ where V is a value then $M \Downarrow_\Lambda U$, $\mathsf{costof}(\Lambda) = m$, and $\mathcal{I}(U) = (\mathbf{0},V)$.*

The following result summarizes the labelling approach to certified cost annotations.

Theorem 21 (certified cost) *Let M be a term of the source language λ. If $\pi_1(\mathcal{I}(\mathcal{L}(M))) \Downarrow m$ then the cost of running $\mathcal{C}(M)$ is m.*

Proof. We take the following steps:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\pi_1(\mathcal{I}(\mathcal{L}(M))) \Downarrow m} \\
\text{implies} & \mathcal{L}(M) \Downarrow_\Lambda \text{ and } \mathsf{costof}(\Lambda) = m & \text{(by proposition 20 above)} \\
\text{implies} & \mathcal{C}(\mathcal{L}(M)) \Downarrow_\Lambda \text{ and } \mathsf{costof}(\Lambda) = m & \text{(by the simulation theorem 16(2)).}
\end{array}
$$

By proposition 18 and the following proviso 19, we conclude that m is the cost of running the compiled code $\mathcal{C}(M)$. □

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 about its instrumentation.

Syntax

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

Logical reflection of types

$$\begin{array}{rcl}
\lceil \iota \rceil & = & \iota \\
\lceil A_1 \times \ldots \times A_n \rceil & = & \lceil A_1 \rceil \times \ldots \lceil A_n \rceil \\
\lceil A_1 \to A_2 \rceil & = & (\lceil A_1 \rceil \to \mathsf{prop}) \times (\lceil A_1 \rceil \times \lceil A_2 \rceil \to \mathsf{prop})
\end{array}$$

Logical reflection of values

$$\begin{array}{rcl}
\lceil id \rceil & = & id \\
\lceil (V_1, \ldots, V_n) \rceil & = & (\lceil V_1 \rceil, \ldots, \lceil V_n \rceil) \\
\lceil \lambda(x_1 : A_1)/F_1 : (x_2 : A_2)/F_2.\ M \rceil & = & (F_1, F_2)
\end{array}$$

Table 10: The surface language.

We illustrate this point using a Hoare logic dedicated to call-by-value purely functional programs [25]. Given a well-typed program annotated by logic assertions, this system computes a set of proof obligations, whose validity ensures the 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 A ranges over simple types, whose syntax is $A ::= \iota \mid A \times A \mid A \to A$ where ι are the basic types including a data type cm for the values of the cost monoid. The metavariable θ ranges over logical types. prop is the type of propositions. Notice that the inhabitants of arrow types on the logical side are purely logical (and terminating) functions, while on the programming language's side they are computational (and potentially non-terminating) functions. Types are lifted to the logical level through a logical reflection $\lceil \bullet \rceil$ defined in Table 10.

We write "$\mathsf{let}\ x : A/F = M\ \mathsf{in}\ M$" to annotate a let definition by a postcondition F of type $\lceil A \rceil \to \mathsf{prop}$. We write "$\lambda(x_1 : A_1)/F_1 : (x_2 : A_2)/F_2.\ M$" to ascribe to a λ-abstraction a precondition F_1 of type $\lceil A_1 \rceil \to \mathsf{prop}$ and a postcondition F_2 of type $\lceil A_1 \rceil \times \lceil A_2 \rceil \to \mathsf{prop}$. 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 of 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 typing system of λ^{ℓ} is quite standard and will be presented formally in the following section 4). The instrumentation $\mathcal{I}$ defined in Table 9 is extended to types by replacing each type annotation A by its monadic interpretation $\mathcal{I}(A)$ defined by $\mathcal{I}(A) = \mathsf{cm} \times \overline{A}, \overline{\iota} = \iota, \overline{A_1 \times A_2} = (\overline{A_1} \times \overline{A_2})$ and $\overline{A_1 \to A_2} = \overline{A_1} \to \mathcal{I}(A_2)$.

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 "$\mathsf{let}\ x : A/F = M\ \mathsf{in}\ M$", F has type $\lceil \mathcal{I}(A) \rceil \to \mathsf{prop}$, that is to say a predicate over pairs of which the first component is the execution cost.

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 these 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 "$x : A/F$" for "$x : A/\lambda(\mathrm{cost} : \mathsf{cm}, x : \lceil \mathcal{I}(A) \rceil).F$". 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 [24] in OCaml ($\sim$ 3.5Kloc). In addition to the distributed source code, a web application enables basic experiments without any installation process.

This compiler accepts a program P written in the surface language extended with fixpoints and algebraic datatypes. We found no technical difficulty in handling these extensions and this is the reason why they are excluded from the core language in the presented formal development. Specifications are written in the Coq proof assistant [11]. 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 [2, 3], 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 $\lambda_{h,vn}$ (Table 7) . The annotating compiler uses the cost model that counts 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 implying the validity of the user-written specifications. These proof obligations are either proved automatically using first-order theorem provers or manually in Coq.

3.5 Examples

In this section, we present two examples that are idiomatic of functional programming: an inductive function and a higher-order function. These examples were checked using our prototype implementation. More involved examples are distributed with the software. These examples include several standard functions on lists (fold, map, . . .), combinators written in continuation-passing style, and functions over binary search trees.

An inductive function Table 11 contains an example of a simple inductive function: the standard concatenation of two lists. In the code, one can distinguish three kinds of toplevel definitions: the type definitions prefixed by the **type** keyword, the definitions at the logical level surrounded by **logic** $\{\ \dots\}$, and the program definitions introduced by the **let** keyword.

On lines 1 and 2, the type definitions introduce a type list for lists of natural numbers as well as a type bool for booleans. Between lines 3 and 9, at the logical level, a Coq inductive function defines the length of lists so that we can use this measure in the cost annotation of concat. Notice that the type definitions are automatically lifted at the Coq level, provided that they respect the strict positivity criterion imposed by Coq to ensure well-foundedness of inductive definitions.

The concatenation function takes two lists $\mathsf{l1}$ and $\mathsf{l2}$ as input, and it is defined, as usual, by induction on $\mathsf{l1}$. In order to write a precise cost annotation, each part of the function body is labelled so that every piece of code is dominated by a label: ℓ_{match} dominates "**match** $\mathsf{l1}$ **with** $\mathsf{Nil} \Rightarrow \bullet \mid \mathsf{Cons}(\mathsf{x},\mathsf{xs}) \Rightarrow \bullet$", ℓ_{nil} dominates "Nil", ℓ_{cons} dominates "$\mathsf{Cons}(\mathsf{x},\bullet)$", and ℓ_{rec} dominates "$\mathsf{concat}(\mathsf{xs},\mathsf{l2})$". Looking at the compiled code in Table 12, it is easy to check that the covering of the code by the labels is preserved through the compilation process. One can also check that the computed costs are *correct* with respect to a cost model that simply counts the number of instructions, *i.e.*, $\mathrm{costof}(\ell_{\mathrm{nil}}) = 2$, $\mathrm{costof}(\ell_{\mathrm{rec}}) = 6$, $\mathrm{costof}(\ell_{\mathrm{cons}}) = 5$ and $\mathrm{costof}(\ell_{\mathrm{match}}) = 1$. Here we are simply assuming one unit of time per low-level instruction, but a more refined analysis is possible by propagating the binary instructions till the binary code (cf. [2, 3]).

Finally, the specification says that the cost of executing concat (l1, l2) is proportional to the size of l1. Recall that the 'cost' and 'result' variables are implicitly bound in the post-condition. Notice that the specification is very specific on the concrete time constants that are involved in that linear function. Following the proof system of the higher-order Hoare logic [25], the verification condition generator produces 37 proof obligations out of this annotated code. All of them are automatically discharged by Coq (using, in particular, the linear arithmetic decision procedure omega).

A higher-order function Let us consider a 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 13.

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 $k \cdot n$. Notice that there is no need to add a label in front of $BTrue$ in the first branch of the inner pattern-matching since the specification only characterizes the cost of an unsuccessful search.

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 man-

```
type list = Nil | Cons (nat, list)
type bool = BTrue | BFalse
logic {
 Fixpoint length (l : list) : nat :=
    match l with
    | Nil ⇒ 0
    | Cons (x, xs) ⇒ 1 + length (xs)
    end.
}
let rec concat (l1: list, l2: list) : list {
  cost = costof($\ell_{\rm match}$) + costof($\ell_{\rm nil}$)
           + (costof($\ell_{\rm rec}$) + costof($\ell_{\rm match}$) + costof($\ell_{\rm cons}$)) × length (l1)
}
=
  $\ell_{\rm match}$ >
  match l1 with
  | Nil → $\ell_{\rm nil}$ > l2
  | Cons (x, xs) → $\ell_{\rm cons}$ > Cons (x, concat (xs, l2) > $\ell_{\rm rec}$)
  end
```

$$\text{with} \begin{cases} \mathsf{costof}(\ell_{\rm nil}) & = & 2 \\ \mathsf{costof}(\ell_{\rm rec}) & = & 6 \\ \mathsf{costof}(\ell_{\rm cons}) & = & 5 \\ \mathsf{costof}(\ell_{\rm match}) & = & 1 \end{cases}$$

Table 11: A function that concatenates two lists, and its cost annotation.

```
routine x19 (c20, x7)
ℓrec:
 k2 ← proj 1 c20 ;
 x ← proj 2 c20 ;
 x14 ← make_int 1 ;
 x15 ← make_tuple (x14, x, x7) ;
 x22 ← proj 0 k2 ;
 call x22 (k2, x15)

routine x16 (c17, l1, l2, k2)
ℓmatch:
 switch l1
 0 : ℓnil :
    x18 ← proj 0 k2 ;
    call x18 (k2, l2)
 1 : ℓcons :
    xs ← proj 2 l1 ;
    x ← proj 1 l1 ;
    x13 ← make_tuple (x19, k2, x) ;
    x21 ← proj 0 c17 ;
    call x21 (c17, xs, l2, x13)
```

Table 12: The compiled code of concat.

```
type list = Nil | Cons (nat, list)
type bool = BTrue | BFalse
logic {
 Definition bound (p : nat ⟶ (nat ∗ bool)) (k : nat) : Prop : =
     ∀ x m: nat, ∀ r: bool, post p x (m, r) ⇒ m ≤ k.
 Definition k0 : = costof(ℓ_m) + costof(ℓ_nil).
 Definition k1 : = costof(ℓ_m) + costof(ℓ_p) + costof(ℓ_c) + costof(ℓ_f) + costof(ℓ_r).
}
let rec pexists (p : nat → bool, l: list) { ∀ x, pre p x } : bool {
 ((result = BTrue) ⇔ (∃ x c: nat, mem x l ∧ post p x (c, BTrue))) ∧
 (∀ k: nat, bound p k ∧ (result = BFalse) ⇒ cost ≤ k0 + (k + k1) × length (l))
} = ℓ_m> match l with
   | Nil → ℓ_nil> BFalse
   | Cons (x, xs) → ℓ_c> match p (x) > ℓ_p with
                              | BTrue → BTrue
                              | BFalse → ℓ_f> (pexists (p, xs) > ℓ_r)
```

Table 13: A higher-order function and its specification.

ually proved in Coq.

4 Typing the compilation chain

We describe a (simple) typing of the compilation chain. Specifically, each λ-calculus of the compilation chain is equipped with a type system which enjoys *subject reduction*: if a term has a type then all terms to which it reduces have the same type. Then the compilation functions are extended to types and are shown to be *type preserving*: if a term has a type then its compilation has the corresponding compiled type.

Besides providing insight into the compilation chain, typing is used in two ways. First, the tool for reasoning about cost annotations presented in section 3 takes as input a typed λ-term and second, and more importantly, in section 5, we rely on an enrichment of a type system which is expressive enough to type a compiled code with explicit memory deallocations.

The two main steps in typing the compilation chain are well studied, see, *e.g.*, the work of Morrisett *et al.* [21], and concern the CPS and the closure conversion steps. In the former, the basic idea is to type the continuation/the evaluation context of a term of type A with its negated type $(A \to R)$, where R is traditionally taken as the type of 'results'. In the latter, one relies on existential types to hide the details of the representation of the 'environment' of a function, *i.e.* the tuple of variables occurring free in its body.

4.1 Typing conventions

We shall denote with *tid* the syntactic category of *type variables* with generic elements $t, s, \ldots$ and with A the syntactic category of *types* with generic elements $A, B, \ldots$ A *type context* is denoted with $\Gamma, \Gamma', \ldots$, and it stands for a finite domain partial function from variables to types. To explicit a type context, we write $x_1 : A_1, \ldots, x_n : A_n$ where the variables $x_1, \ldots, x_n$ must be all distinct and the order is irrelevant. Also we write $x^* : A^*$ for a possibly empty sequence $x_1 : A_1, \ldots, x_n : A_n$, and $\Gamma, x^* : A^*$ for the context resulting from Γ by adding

Syntax types

$$A \quad ::= tid \mid A^+ \to A \mid \times(A^*) \quad \text{(types)}$$

Typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\begin{array}{c}\Gamma, x : A \vdash N : B\\ \Gamma \vdash M : A\end{array}}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}$$

$$\frac{\Gamma, x^+ : A^+ \vdash M : B}{\Gamma \vdash \lambda x^+ . M : A^+ \to B} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A^+ \to B\\ \Gamma \vdash N^+ : A^+\end{array}}{\Gamma \vdash @(M, N^+) : B}$$

$$\frac{\Gamma \vdash M^* : A^*}{\Gamma \vdash (M^*) : \times(A^*)} \qquad \frac{\Gamma \vdash M : \times(A_1, \ldots, A_n) \quad 1 \leq i \leq n}{\Gamma \vdash \pi_i(M) : A_i}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \ell > M : A} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M > \ell : A}$$

Restricted syntax CPS types, R type of results

$$A \quad ::= tid \mid A^+ \to R \mid \times(A^*) \quad \text{(CPS types)}$$

CPS type compilation

$$\begin{array}{ll}
\mathcal{C}_{cps}(t) & = t\\
\mathcal{C}_{cps}(\times(A^*)) & = \times(\mathcal{C}_{cps}(A)^*)\\
\mathcal{C}_{cps}(A^+ \to B) & = (\mathcal{C}_{cps}(A))^+, \neg\mathcal{C}_{cps}(B) \to R\\
 & \text{where: } \neg A \equiv (A \to R)
\end{array}$$

Table 14: Type system for λ^ℓ and λ^ℓ_{cps}

the sequence $x^* : A^*$. Hence the variables in x^* must not be in the domain of Γ. If A is a type, we write $\mathsf{ftv}(A)$ for the set of *type variables occurring free* in it and, by extension, if Γ is a type context $\mathsf{ftv}(\Gamma)$ is the union of the sets $\mathsf{ftv}(A)$ where A is a type in the codomain of Γ. A *typing judgement* is typically written as $\Gamma \vdash M : A$ where M is some term. We shall write $\Gamma \vdash M^* : A^*$ for $\Gamma \vdash M_1 : A_1, \ldots, \Gamma \vdash M_n : A_n$. Similar conventions apply if we replace the symbol '$*$' with the symbol '$+$' except that in this case the sequence is assumed not-empty. A type transformation, say $\mathcal{T}$, is *lifted to type contexts* by defining $\mathcal{T}(x_1 : A_1, \ldots x_n : A_n) = x_1 : \mathcal{T}(A_1), \ldots, x_n : \mathcal{T}(A_n)$. Whenever we write:

if $\Gamma \vdash^{S_1} M : A$ then $\mathcal{T}(\Gamma) \vdash^{S_2} \mathcal{T}(M) : \mathcal{T}(A)$

what we actually mean is that if the judgement in the hypothesis is *derivable* in a certain 'type system S_1' then the transformed judgement in derivable in the 'type system S_2'.

4.2 The type system of the source language

Table 14 describes the typing rules for the source language defined in Table 2. These rules are standard except those for the labellings, and, as announced, they are preserved by reduction.

Proposition 22 (subject reduction) *If M is a term of the λ^ℓ calculus, $\Gamma \vdash M : A$ and $M \to N$ then $\Gamma \vdash N : A$.*

The typing rules described in Table 14 apply to the CPS λ-calculus too. Table 14 describes the restricted syntax of the CPS types and the CPS type translation. Then the CPS term translation defined in Table 3 preserves typing in the following sense.

Proposition 23 (type CPS) *If* $\Gamma \vdash M : A$ *then* $\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A) \vdash \mathcal{C}_{cps}(M) : R$.

4.3 Type system for the value named calculi

Table 15 describes the typing rules for the value named calculi. For the sake of brevity, we shall omit the type of a term since this type is always the type of results R and write $\Gamma \vdash^{vn} M$ rather than $\Gamma \vdash^{vn} M : R$. The first 6 typing rules are just a specialization of the corresponding rules in Table 14. The last two rules allow for the introduction and elimination of *existential types*; we shall see shortly how they are utilised in typing closure conversion.

In the proposed formalisation, we rely on the tuple constructor to introduce an existential type and the first projection to eliminate it. This has the advantage of leaving unchanged the syntax and the operational semantics of the value named λ-calculus. An alternative presentation consists in introducing specific operators to introduce and eliminate existential types which are often denoted with pack and unpack, respectively. The reader who is familiar with this notation may just read (x) as $\mathsf{pack}(x)$ and $\pi_1(x)$ as $\mathsf{unpack}(x)$ when x has an existential type. With this convention, the rewriting rule which allows to *unpack* a *packed* value is just a special case of the rule for projection.

As in the previous system, typing is preserved by reduction.

Proposition 24 (subject reduction, value named) *If* M *is a term of the* $\lambda^{\ell}_{cps,vn}$*-calculus,* $\Gamma \vdash^{vn} M$ *and* $M \to N$ *then* $\Gamma \vdash^{vn} N$.

The transformation into value named CPS form specified in Table 5 affects the terms but not the types.

Proposition 25 (type value named) *If* M *is a term of the* λ^{ℓ}_{cps}*-calculus and* $\Gamma \vdash M : R$ *then* $\Gamma \vdash^{vn} \mathcal{C}_{vn}(M)$.

On the other hand, to type the closure conversion we rely on existential types to abstract/hide the type of the environment as specified in Table 15. Then the term translation of the function definition and application given in Table 6 has to be slightly modified to account for the introduction and elimination of existential types. The revised definition is as follows:

$$\begin{array}{lcl}
\mathcal{C}_{cc}(@(x, y^+)) & = & \begin{array}{l}\mathsf{let}\ x = \pi_1(x)\ \mathsf{in} \quad (\leftarrow \text{EXISTENTIAL ELIMINATION})\\ \mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+)\end{array} \\
\\
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \begin{array}{l}\mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1, \ldots, z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in}\\ \mathsf{let}\ e = (z_1, \ldots, z_k)\ \mathsf{in}\\ \mathsf{let}\ x = (c, e)\ \mathsf{in}\\ \mathsf{let}\ x = (x)\ \mathsf{in} \quad (\leftarrow \text{EXISTENTIAL INTRODUCTION})\\ \mathcal{C}_{cc}(M) \qquad (\text{if } C = \lambda x^+.N, \mathsf{fv}(C) = \{z_1, \ldots, z_k\})\end{array}
\end{array}$$

This modified closure conversion does not affect the commutation and simulation properties stated in propositions 10 and 11 and moreover it preserves typing as follows.

Syntax types

$$A \quad ::= tid \mid (A^+ \to R) \mid \times(A^*) \mid \exists tid.A$$

Typing rules

$$\frac{\Gamma, x^+ : A^+ \vdash^{vn} M}{\Gamma \vdash^{vn} \lambda x^+.M : A^+ \to R} \qquad \frac{x : A^+ \to R, y^+ : A^+ \in \Gamma}{\Gamma \vdash^{vn} @(x, y^+)}$$

$$\frac{x^* : A^* \in \Gamma}{\Gamma \vdash^{vn} (x^*) : \times(A^*)} \qquad \frac{y : \times(A_1, \ldots, A_n) \in \Gamma \quad 1 \le i \le n \quad \Gamma, x : A_i \vdash^{vn} M}{\Gamma \vdash^{vn} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M}$$

$$\frac{\Gamma \vdash^{vn} V : A \quad \Gamma, x : A \vdash^{vn} M}{\Gamma \vdash^{vn} \mathsf{let}\ x = V\ \mathsf{in}\ M} \qquad \frac{\Gamma \vdash^{vn} M}{\Gamma \vdash^{vn} \ell > M}$$

$$\frac{x : [B/t]A \in \Gamma}{\Gamma \vdash^{vn} (x) : \exists t.A} \qquad \frac{y : \exists t.A \in \Gamma \quad \Gamma, x : A \vdash^{vn} M \quad t \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash^{vn} \mathsf{let}\ x = \pi_1(y)\ \mathsf{in}\ M}$$

Closure conversion type compilation

$$\begin{array}{ll}
\mathcal{C}_{cc}(t) & = t \\
\mathcal{C}_{cc}(\times(A^*)) & = \times(\mathcal{C}_{cc}(A)^*) \\
\mathcal{C}_{cc}(A^+ \to R) & = \exists t. \times ((t, \mathcal{C}_{cc}(A)^+ \to R),\ t) \\
\mathcal{C}_{cc}(\exists t.A) & = \exists t.\mathcal{C}_{cc}(A)
\end{array}$$

Table 15: Type system for the value named calculi and closure conversion

Proposition 26 (type closure conversion) *If M is a term in $\lambda^{\ell}_{cps,vn}$ and $\Gamma \vdash^{vn} M$ then $\mathcal{C}_{cc}(\Gamma) \vdash^{vn} \mathcal{C}_{cc}(M)$.*

Similarly to the transformation in value named form, the hoisting transformations affect the terms but not the types.

Proposition 27 (type hoisting) *If M is a term in $\lambda^{\ell}_{cps,vn}$, $\Gamma \vdash^{vn} M$, and $M \leadsto N$ then $\Gamma \vdash^{vn} N$.*

4.4 Typing the compiled code

We can now extend the compilation function to types by defining:

$$\mathcal{C}(A) = \mathcal{C}_{cc}(\mathcal{C}_{cps}(A))$$

and by composing the previous results we derive the following type preservation property of the compilation function.

Theorem 28 (type preserving compilation) *If M is a term of the λ^{ℓ}-calculus and $\Gamma \vdash M : A$ then*

$$\mathcal{C}(\Gamma), halt : \exists t. \times (t, \mathcal{C}(A) \to R, t) \vdash^{vn} \mathcal{C}(M)\ .$$

Remark 29 *The 'halt' variable introduced by the CPS translation can occur only in a subterm of the shape $@(halt, x)$ in the intermediate code prior to closure conversion. Then in the closure conversion translation, it suffices that $\mathcal{C}_{cc}(@(halt, x)) = @(halt, x)$ and give to '$halt'$ a functional rather than an existential type. With this proviso, theorem 28 above can be restated as follows:*

If M is a term of the λ^{ℓ}-calculus and $\Gamma \vdash M : A$ then $\mathcal{C}(\Gamma), halt : \neg\mathcal{C}(A) \vdash^{vn} \mathcal{C}(M)$.

Example 30 (typing the compiled code) *We consider again the compilation of the term $\lambda x.y$ (cf. example 9) which can be typed,* e.g., *as follows:*

$$y : t_1 \vdash \lambda x.y : (t_2 \to t_1) \ .$$

Its CPS translation is then typed as:

$$y : t_1, halt : \neg\mathcal{C}_{cps}(t_2 \to t_1) \vdash @(halt, \lambda x, k.@(k, y)) : R \ .$$

The value named translation does not affect the types:

$$y : t_1, halt : \neg\mathcal{C}_{cps}(t_2 \to t_1) \vdash^{vn} \mathsf{let}\ z_1 = \lambda x, k.@(k, y)\ \mathsf{in}\ @(halt, z_1) \ .$$

After closure conversion we obtain the following term M:

$\mathsf{let}\ c = \lambda e, x, k.\mathsf{let}\ y = \pi_1(e), k = \pi_1(k), c = \pi_1(k), e = \pi_2(k)\ \mathsf{in}\ @(c, e, y)\ \mathsf{in}$
$\mathsf{let}\ e = (y), z_1 = (c, e), z_1 = (z_1), halt = \pi_1(halt), c = \pi_1(halt), e = \pi_2(halt)\ \mathsf{in}$
$@(c, e, z_1)$

which is typed as follows:

$$y : t_1, halt : \exists t. \times (t, \mathcal{C}(t_2 \to t_1) \to R, t) \vdash^{vn} M \ .$$

In this case no further hoisting transformation applies. If we adopt the optimised compilation strategy sketched in remark 29 then after closure conversion we obtain the following term M':

$\mathsf{let}\ c = \lambda e, x, k.\mathsf{let}\ y = \pi_1(e), k = \pi_1(k), c = \pi_1(k), e = \pi_2(k)\ \mathsf{in}\ @(c, e, y)\ \mathsf{in}$
$\mathsf{let}\ e = (y), z_1 = (c, e), z_1 = (z_1),\ \mathsf{in}$
$@(halt, z_1)$

which is typed as follows:

$$y : t_1, halt : \mathcal{C}(t_2 \to t_1) \to R \vdash^{vn} M' \ .$$

5 Memory management

We describe an enrichment of the $\lambda^{\ell}_{h,vn}$-calculus called $\lambda^{\ell,r}_{h,vn}$-calculus which explicitly handles allocation and deallocation of *memory regions*. At our level of abstraction, the memory regions are just names $r, r', \ldots$ of a countable set. Intuitively, the 'live' locations of a memory are partitioned into regions. The three new operations the enriched calculus may perform are: (1) allocate a new region, (2) allocate a value (in our case a non-empty tuple) in a region, and (3) dispose a region. The additional operation of reading a value from a region is implicit in the projection operation which is already available in the non-enriched calculus $\lambda^{\ell}_{h,vn}$. In order to gain some expressivity we shall also allow a function to be parametric in a collection of region names which are provided as arguments at the moment of the function call.

From our point of view, the important property of this approach to memory management is both its cost predictability and the possibility of formalising and certifying it using techniques similar to those presented in section 2. Indeed the operations (1-3) inject short sequences of instructions in the compiled code that can be executed in constant time as stressed by Tofte and Talpin [27] (more on this at the end of section 5.2).

Because of the operation (3) described above (dispose), the following memory errors may arise at run-time: (i) write a value in a disposed region, (ii) access (project) a value in a disposed region, and (iii) dispose an already disposed region. To avoid these errors, we formulate a *type and effect* system in the sense of Lucassen and Gifford [18] that over-approximates the visible set of regions and guarantees safe memory disposal, following Tofte and Talpin [27]. This allows to further extend to the right with one more commuting square a typed version of the compilation chain described in Table 1 and then to include the cost of safe memory management in our analysis.

5.1 Region conventions

We introduce a syntactic category of *regions rid* with generic elements $r, r', \dots$ and a syntactic category of *effects* e with generic elements $e, e', \dots$ An effect is a finite set of region variables. We keep denoting *types* with $A, B, \dots$ However types may depend on both regions and effects. Regions can be *bound* when occurring either in types or in terms. In the first case, the binder is a universal quantifier $\forall r.A$, while in the second it is either a new region allocation operator $\mathsf{let\ all}(r)\ \mathsf{in}\ T$ or a region λ-abstraction. On the other hand, we stress that the disposal operator $\mathsf{dis}(r)\ \mathsf{in}\ T$ is *not* a binder. Because of the universal quantification and the λ-abstraction both the untyped and the typed region enriched calculi include a notion of *region substitution*. Note that such a substitution operates on the effects contained in the types too and that, as a result, it may reduce the cardinality of the set of regions which composes an effect. The change of cardinality however, can only arise in the untyped calculus. In the typed calculus, all the region substitutions are guaranteed to be injective. We denote with $\mathsf{frv}(A)$ the set of regions occurring free in the type A, and $\mathsf{frv}(\Gamma)$ denotes the obvious extension to type contexts.

5.2 A region enriched calculus

A formalisation of the operations and the related memory errors is given through the region enriched calculus presented in Tables 16 and 17. Notice that an empty tuple is stored in a local variable rather than in a region and that a similar stategy would be adopted for basic data values such as booleans or integers. The usual formalisation of the operational semantics relies on a rather concrete representation of a heap as a (finite domain) function mapping regions to stores which are (finite domain) functions from locations to values satisfying some coherence conditions (see, *e.g.* [27, 1, 8]). In the following, we will take a slightly more abstract approach by representing the heap implicitly as a *heap context* H. The latter is simply a list of region allocations, value allocations at a region, and region disposals.

It turns out that it is possible to formulate the coherence conditions on the memory directly on this list so that we do not have to commit to a concrete representation of the heap as the one sketched above. A first consequence of this design choice, is that we can dispense with the introduction of additional syntactic entities like that of 'location' or 'address' and consequently avoid the non-deterministic rules that choose fresh regions or locations (as in, say, the π-calculus, α-renaming will take care of that). A second consequence is that the proof of the simulation property of the standard calculus by the region-enriched calculus is rather direct.

Coherent heap context relative to live regions

$$\frac{}{\mathit{Coh}([\,],L)} \qquad \frac{\mathit{Coh}(H,L)}{\mathit{Coh}(\mathsf{let}\ x=()\ \mathsf{in}\ H,L)} \qquad \frac{\mathit{Coh}(H,L) \quad r\in L}{\mathit{Coh}(\mathsf{let}\ x=(y^+)\mathsf{at}(r)\ \mathsf{in}\ H,L)}$$

$$\frac{\mathit{Coh}(H,L\cup\{r\})}{\mathit{Coh}(\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ H,L)} \qquad \frac{\mathit{Coh}(H,L\backslash\{r\}) \quad r\in L}{\mathit{Coh}(\mathsf{dis}(r)\ \mathsf{in}\ H,L)}$$

Region not-disposed in a heap context

$$\frac{}{\mathit{NDis}(r,[\,])} \qquad \frac{\mathit{NDis}(r,H)}{\mathit{NDis}(r,\mathsf{let}\ x=()\ \mathsf{in}\ H)} \qquad \frac{\mathit{NDis}(r,H)}{\mathit{NDis}(r,\mathsf{let}\ x=(y^+)\mathsf{at}(r')\ \mathsf{in}\ H)}$$

$$\frac{(r=r')\ \text{or}\ (r\neq r'\ \text{and}\ \mathit{NDis}(r,H))}{\mathit{NDis}(r,\mathsf{let}\ \mathsf{all}(r')\ \mathsf{in}\ H)} \qquad \frac{r\neq r' \quad \mathit{NDis}(r,H)}{\mathit{NDis}(r,\mathsf{dis}(r')\ \mathsf{in}\ H)}$$

Table 16: Coherence predicate on heap contexts

Continuing the comparison with formalisations found in the literature, we notice that the fact that region disposal is decoupled from allocation avoids the introduction of a special 'disposed' or 'free' region which is sometimes used in the operational semantics to represent the situation where a region becomes inaccessible (see, *e.g.*, [27, 1]). What we do instead is to keep track of the disposal operation in the *heap context*.

Finally, let us notice that we certainly take advantage of the fact that our formalisation of region management targets an intermediate RTL language where the execution order and the operations of writing and reading a value from memory are completely explicit. The formalisation of region management at the level of the source language, *e.g.*, the λ-calculus, *appears* a bit more involved because one has to enrich the language with operations that really refer to the way the language is compiled. For instance, one has to distinguish between the act of storing a value in memory and the act of referring to it without exploring its internal structure.

Table 16 specifies the coherence predicate $\mathit{Coh}(H,L)$ of the heap context H relatively to a set of 'live' regions L. Briefly, a heap context is *coherent* if whenever the context contains an allocation for a tuple in a region, or a disposal of a region, the region in question is alive. This is defined by induction on the structure of the heap context H.

The reduction rules in Table 17 are a refinement of those of the value named λ-calculus described in Table 4. The main novelties are that a transition can be fired only if the heap context is coherent relatively to an empty set of regions in the sense described above and moreover that a tuple can be projected only if it is allocated in a region which has not been disposed. To formalise this last property we have refined the definition of the function $E(x)$ which looks for the value bound to a variable in an evaluation context. The refined function, upon success, returns both the value and the part of the heap context, say H, which has been explored. Then the predicate $\mathit{NDis}(r,H)$ defined in Table 16 checks that the region r where the tuple is allocated is not disposed by H. Again, this predicate is defined by induction on the structure of the heap context H.

We remark the following decomposition property of region-enriched programs.

Proposition 31 (decomposition) *A program P in the region enriched λ-calculus can be uniquely decomposed as $F[H[\Delta]]$ where F is a function context, H a heap context, and Δ is either an application of the shape $@(x,r^*,y^+)$ or a projection of the shape* $\mathsf{let}\ x=\pi_i(y)\ \mathsf{in}\ T$,

Syntax

rid	$::= r \mid r' \mid \cdots$	(region identifiers)
C	$::= () \mid (id^+)\mathsf{at}(rid) \mid \pi_i(id)$	(restricted let-bindable terms)
T	$::= @(id, rid^*, id^+) \mid \mathsf{let}\ id = C\ \mathsf{in}\ T \mid \ell > T \mid$ $\mathsf{let\ all}(rid)\ \mathsf{in}\ T \mid \mathsf{dis}(rid)\ \mathsf{in}\ T$	(restricted terms)
P	$::= T \mid \mathsf{let}\ id = \lambda rid^*, id^+.T\ \mathsf{in}\ P$	(programs)
F	$::= [\ ] \mid \mathsf{let}\ id = \lambda rid^*, id^+.T\ \mathsf{in}\ F$	(function contexts)
H	$::= [\ ] \mid \mathsf{let}\ id = ()\ \mathsf{in}\ H \mid \mathsf{let}\ id = (id^+)\mathsf{at}(rid)\ \mathsf{in}\ H \mid$ $\mathsf{let\ all}(rid)\ \mathsf{in}\ H \mid \mathsf{dis}(rid)\ \mathsf{in}\ H$	(heap contexts)
E	$::= F[H]$	(evaluation contexts)

Reduction rules

$$E[@(x, r'_1, \ldots, r'_m, z_1, \ldots, z_n)] \to E[[r'_1/r_1, \ldots, r'_m/r_m, z_1/y_1, \ldots, z_n/y_n]T]$$
$$\text{if } \pi_1(E(x)) \equiv \lambda r_1, \ldots, r_m, y_1, \ldots, y_n.T, E \equiv F[H],\ \mathit{Coh}(H, \emptyset)$$

$$E[\mathsf{let}\ z = \pi_i(x)\ \mathsf{in}\ T] \to E[[y_i/z]T]]$$
$$\text{if } E(x) = ((y_1, \ldots, y_n)\mathsf{at}(r), H'),\ 1 \le i \le n,\ E \equiv F[H],\ \mathit{Coh}(H, \emptyset),\ \mathit{NDis}(r, H')$$

$$E[\ell > T] \xrightarrow{\ell} E[T] \qquad \text{if } E \equiv F[H], \mathit{Coh}(H, \emptyset)$$

$$\text{where: } \left[\begin{array}{l} E(x) = \begin{cases} (V, [\ ]) & \text{if } E = E'[\mathsf{let}\ x = V\ \mathsf{in}\ [\ ]] \\ (V, E''[El]) & \text{otherwise if } E = E'[El], E'(x) = (V, E'') \\ \text{undefined} & \text{otherwise} \end{cases} \\ \\ V ::= () \mid (id^*)\mathsf{at}(rid) \mid \lambda rid^*, id^+.T \\ El ::= \mathsf{let}\ id = V\ \mathsf{in}\ [\ ] \mid \mathsf{let\ all}(rid)\ \mathsf{in}\ [\ ] \mid \mathsf{dis}(rid)\ \mathsf{in}\ [\ ] \end{array} \right]$$

Table 17: The region-enriched calculus: $\lambda^{\ell,r}_{h,vn}$

or a labelling of the shape $\ell > T$.

We define an obvious *erasure function* on the region-enriched types, values, and terms that just erases all the region related pieces of information (please refer to the formal definition in Table 19 of the appendix for details).

Because of the possible memory errors described above, a region enriched program does not necessarily simulate its region erasure.

Example 32 (memory errors) *Consider the following program P in $\lambda^{\ell}_{h,vn}$ (not necessarily the result of a compilation):*

$$\begin{array}{rl} P \equiv & F[@(\mathit{pair}, v_1, v_2)] \\ F \equiv & \mathsf{let}\ \mathit{prj1} = \lambda x.\mathsf{let}\ y = \pi_1(x)\ \mathsf{in}\ @(\mathit{halt}, y)\ \mathsf{in} \\ & \mathsf{let}\ \mathit{pair} = \lambda x_1, x_2.\mathsf{let}\ y = (x_1, x_2)\ \mathsf{in}\ @(\mathit{prj1}, y)\ \mathsf{in}\ [\ ]\ . \end{array}$$

One strategy to manage memory regions in P is to allocate a region upon entering the pair function and to dispose it just before calling the prj1 function as in the following program P_1 in $\lambda^{\ell,r}_{h,vn}$.

$$\begin{array}{rcl} P_1 & \equiv & F_1[@(\mathit{pair}, v_1, v_2)] \\ F_1 & \equiv & \mathsf{let}\ \mathit{prj1} = \lambda x.\mathsf{let}\ z = \pi_1(x)\ \mathsf{in}\ @(\mathit{halt}, z)\ \mathsf{in} \\ & & \mathsf{let}\ \mathit{pair} = \lambda x_1, x_2.\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ \mathsf{let}\ y = (x_1, x_2)\mathsf{at}(r)\ \mathsf{in}\ \mathsf{dis}(r)\ \mathsf{in}\ @(\mathit{prj1}, y)\ \mathsf{in}\ [\ ]\ . \end{array}$$

Unfortunately this strategy leads to a memory error as:

$$\begin{array}{rcl} P_1 & \xrightarrow{*} & F_1[H_1[\mathsf{let}\ z = \pi_1(y)\ \mathsf{in}\ @(\mathit{halt}, z)]] \\ H_1 & \equiv & \mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ \mathsf{let}\ y = (v_1, v_2)\mathsf{at}(r)\ \mathsf{in}\ \mathsf{dis}(r)\ \mathsf{in}\ [\ ] \end{array}$$

Formally, $F_1[H_1](y) = ((v_1, v_2)\mathsf{at}(r), H_2)$, $H_2 = \mathsf{dis}(r)\ \mathsf{in}\ [\ ]$, and the predicate $NDis(r, H_2)$ does not *hold. In plain words, the problem with this strategy is that it disposes the region r before the value (v_1, v_2) allocated into it is projected. A better strategy is to pass the region created in the function pair to the function prj1 and let this function dispose the region once the value (v_1, v_2) has been projected. This strategy is described by the following program P_2 in $\lambda^{\ell,r}_{h,vn}$.*

$$\begin{array}{rcl} P_2 & \equiv & F_2[@(\mathit{pair}, v_1, v_2)] \\ F_2 & \equiv & \mathsf{let}\ \mathit{prj1} = \lambda r, x.\mathsf{let}\ z = \pi_1(x)\ \mathsf{in}\ \mathsf{dis}(r)\ \mathsf{in}\ @(\mathit{halt}, z)\ \mathsf{in} \\ & & \mathsf{let}\ \mathit{pair} = \lambda x_1, x_2.\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ \mathsf{let}\ y = (x_1, x_2)\mathsf{at}(r)\ \mathsf{in}\ @(\mathit{prj1}, r, y)\ \mathsf{in}\ [\ ]\ . \end{array}$$

This time the reduction leads to a normal termination:

$$\begin{array}{rcl} P_2 & \xrightarrow{*} & F_2[H_2[@(\mathit{halt}, v_1)]] \\ H_2 & \equiv & \mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ \mathsf{let}\ y = (v_1, v_2)\mathsf{at}(r)\ \mathsf{in}\ \mathsf{dis}(r)\ \mathsf{in}\ [\ ]\ . \end{array}$$

We conclude this section with an overview of a rather standard *implementation scheme* of region based memory management (see, *e.g.*, [20] for more details). Initially, the available memory is partitioned in pages which constitute a free list. A region is a pointer to a 'region descriptor' that contains a pointer to the beginning and the end of a list of pages and a counter which gives the amount of memory available in the last page of the list. A value (a non-empty tuple in our case) is just a pointer to a memory address and an access to a value is direct. Storing a value in a region means storing the value in the last page of the list related to the region and updating the region descriptor. If the space available is not sufficient, then one or more pages are taken from the free list and appended to the end of the region list and again the region descriptor is updated. This operation can be executed in constant time as long as the size of the values to be allocated can be determined at compile time (which is obviously true in our case). Deallocating a region means concatenating the list related to the region to the free list. We refrain from going into the details of the implementation scheme mentioned above which really belong to the backend of the compiler. Indeed, the scheme is rather independent from the source language (for instance, Christiansen *et al.* [10] rely on it to implement an object-oriented language) while depending for its efficiency on the memory organisation of the processor and possibly the operating system.

5.3 A type and effect system

In order to have the simulation property, we require that the region-enriched program is typable with respect to an enhanced *type and effect* system described in Table 18 whose purpose is precisely to avoid memory errors at run time. The system defines judgment (i) $\Gamma \vdash^{rg} T : e$ read "restricted term T has effect e under Γ" ; (ii) $\Gamma \vdash^{rg} C : A$ read "restricted let-bindable term C has type A under Γ" and (iii) $\Gamma \vdash^{rg} P : e$ read "program P has effect e under Γ". The formalisation follows the work of Aiken *et al.* [1] in that allocation and disposal of a region are decoupled (see also Henglein *et al.* [16] for a survey and Boudol [8] for a discussion). Then a region can be disposed only if the following computation neither accesses nor disposes it. Note that in typing values we omit the effect (which is always empty) and in typing terms we omit the type (which is always the type of results R). The typing rules are designed to maintain several invariants. First, if a program P has effect e then the set of regions e over-approximates the *visible* regions that the program P may dispose or access for allocating or reading a value. Second, all the region names have been allocated (and possibly disposed afterwards). Third, distinct region names in the program correspond at run time to different regions, *i.e.*, all region substitutions are *injective.* With respect to the system described in Table 15, we notice that we distinguish the rules for typing an empty and a non-empty tuple as the former has no effect on the heap. For similar reasons, we split the rule for typing a value definition in three depending on whether the value is an empty tuple, a function, or a non-empty tuple (possibly of existential type). Only in the last case an effect on the heap is recorded. As already mentioned, empty tuples do not affect the heap and function definitions eventually become sequences of assembly language instructions which are stored in a statically allocated and read-only zone of memory separated by the data memory.

Example 33 (types and effects) *Going back to example 32, let us assume the types $t_i : v_i$, $i = 1, 2$ and halt* $: t_1 \xrightarrow{\emptyset} R$*. Then the reader may check that the program P_2 is typable (has an effect) assuming the following types for the functions:*

$$\mathit{pair} \ : \ t_1, t_2 \xrightarrow{\emptyset} R \qquad\qquad \mathit{prj1} \ : \ \forall r. \times (t_1, t_2)\mathsf{at}(r) \xrightarrow{\{r\}} R$$

On the other hand, any attempt at typing P_1 fails trivially because the type of the function prj1 must be of the shape $\times(t_1, \ldots)\mathsf{at}(r) \xrightarrow{\{r\}} R$ and it cannot match the type of the pair allocated by the function pair in a new *region. If we fix this problem by abstracting the function prj1 w.r.t. a region so that it has the type $\forall r. \times (t_1, \ldots)\mathsf{at}(r) \xrightarrow{\{r\}} R$ we stumble on the main*

Types and effects syntax

$$
\begin{array}{lll}
e & ::= \{rid, \ldots, rid\} & \text{(effects)} \\
A & ::= tid \mid \forall rid^*.A^+ \xrightarrow{e} R \mid \times() \mid \times(A^+)\mathsf{at}(rid) \mid (\exists tid.A)\mathsf{at}(rid) & \text{(types)}
\end{array}
$$

Typing rules

$$\frac{\Gamma, y^+ : A^+ \vdash^{rg} T : e \quad \{r^*\} \cap \mathsf{frv}(\Gamma) = \emptyset \quad \mathsf{frv}(\lambda r^*, y^+.T) = \emptyset}{\Gamma \vdash^{rg} \lambda r^*, y^+.T : \forall r^*.A^+ \xrightarrow{e} R}$$

$$\frac{B \equiv \forall r_1^*.A^+ \xrightarrow{e} R \quad x : B \in \Gamma \quad y^+ : [r^*/r_1^*]A^+ \in \Gamma \quad r^* \text{ distinct} \quad \mathsf{frv}(B) = \emptyset}{\Gamma \vdash^{rg} @(x, r^*, y^+) : [r^*/r_1^*]e}$$

$$\frac{x^+ : A^+ \in \Gamma}{\Gamma \vdash^{rg} (x^+)\mathsf{at}(r) : \times(A^+)\mathsf{at}(r)}$$

$$\frac{y : \times(A_1, \ldots, A_n)\mathsf{at}(r) \in \Gamma \quad 1 \le i \le n \quad \Gamma, x : A_i \vdash^{rg} T : e}{\Gamma \vdash^{rg} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T : e \cup \{r\}}$$

$$\frac{x : [B/t]A \in \Gamma}{\Gamma \vdash^{rg} (x)\mathsf{at}(r) : (\exists t.A)\mathsf{at}(r)}$$

$$\frac{y : (\exists t.A)\mathsf{at}(r) \in \Gamma \quad \Gamma, x : A \vdash^{rg} T : e \quad t \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash^{rg} \mathsf{let}\ x = \pi_1(y)\ \mathsf{in}\ T : e \cup \{r\}}$$

$$\frac{}{\Gamma \vdash^{rg} () : \times()}$$

$$\frac{\Gamma \vdash^{rg} () : A \quad \Gamma, x : A \vdash^{rg} T : e}{\Gamma \vdash^{rg} \mathsf{let}\ x = ()\ \mathsf{in}\ T : e}$$

$$\frac{\Gamma \vdash^{rg} \lambda r^*, y^+.T : A \quad \Gamma, x : A \vdash^{rg} P : e}{\Gamma \vdash^{rg} \mathsf{let}\ x = \lambda r^*, y^+.T\ \mathsf{in}\ P : e}$$

$$\frac{\Gamma \vdash^{rg} (y^+)\mathsf{at}(r) : A \quad \Gamma, x : A \vdash^{rg} T : e}{\Gamma \vdash^{rg} \mathsf{let}\ x = (y^+)\mathsf{at}(r)\ \mathsf{in}\ T : e \cup \{r\}}$$

$$\frac{\Gamma \vdash^{rg} T : e \cup \{r\} \quad r \notin \mathsf{frv}(\Gamma), e}{\Gamma \vdash^{rg} \mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ T : e}$$

$$\frac{\Gamma \vdash^{rg} T : e \quad r \notin e}{\Gamma \vdash^{rg} \mathsf{dis}(r)\ \mathsf{in}\ T : e \cup \{r\}}$$

$$\frac{\Gamma \vdash^{rg} T : e}{\Gamma \vdash^{rg} \ell > T : e}$$

$$\frac{\Gamma \vdash^{rg} P : e \quad e \subseteq e'}{\Gamma \vdash^{rg} P : e'}$$

Table 18: Type and effect system for the region-enriched calculus

problem, namely the function pair disposes a region which is used in the continuation; this is forbidden by the typing rule for region disposal.

Example 34 (injective region substitutions) *The soundness and relative simplicity of the type and effect system bear on the fact that region substitutions are injective. Technically this property is enforced by the rules for typing an application and an abstraction. In an application, say $@(x, r^*, y^+)$, the region variables r^* are* distinct *and the type of x is region closed. In an abstraction, say $\lambda r^*, x^+.T$, the function is region closed. It is instructive to see what can go wrong if we drop these conditions. Consider a function x of the following shape with its possible (region closed) type:*

$$x = \lambda r_1, r_2, y.\mathsf{dis}(r_1) \text{ in let } z = (y)\mathsf{at}(r_2) \text{ in } T : \forall r_1, r_2. \times () \xrightarrow{\{r_1,r_2\}} R \ .$$

An application $@(x, r, r, y)$ where we pass twice the same region name r will produce a memory error since we dispose r before writing into it. A similar phenomenon arises with a function of the following shape and related (region open!) type:

$$x = \lambda r_1, y.\mathsf{dis}(r_1) \text{ in let } z = (y)\mathsf{at}(r_2) \text{ in } T : \forall r_1. \times () \xrightarrow{\{r_1,r_2\}} R \ .$$

Then an application $@(x, r_2, y)$ where we pass a region name which is free in the type of the function will also produce a memory error. As a final example, consider a function

$$x = \lambda r_1, y.\text{let } z = () \text{ in } @(y, r_1, r_2, z)$$

with a free region variable r_2. Note that r_1, r_2 may not appear in the type of the function x because, e.g., *y makes no use of them. Then if we apply x as in $@(x, r_2, y)$ we end up with an application $@(y, r_2, r_2, z)$ which is not typable because it violates the condition that all the region variables passed as arguments are distinct.*

5.4 Properties of the type and effect system

We notice that the region erasing function preserves typing.

Proposition 35 (region erasure) *If $\Gamma \vdash^{rg} P : e$ then $rer(\Gamma) \vdash^{vn} rer(P)$.*

In the other direction, it is always possible to insert region annotations in a typable program of $\lambda^{\ell}_{h,vn}$ so as to produce a typable region-enriched program. A simple but *not* very interesting way to do this is to allocate one region at the very beginning of the computation which is never disposed and which is shared by all functions.

Proposition 36 (region enrichment) *Let Γ_0 be a type context such that if $x : A \in \Gamma_0$ then A is not a type of the shape $\times(B^+)$ or $\exists t.B$. If $\Gamma_0 \vdash^{vn} P$ then it is always possible to find a region enriched typable program P' such $rer(P') \equiv P$.*

Fortunately, more interesting strategies are available; we refer to Aiken *et al.* [1] for their description and for an encouraging experimental evaluation and to Henglein *et al.* [16] for a survey of region inference techniques. For our purposes, it is enough to know that it is always possible to define a compilation function $\mathcal{C}_{rg}$ from the typed $\lambda^{\ell}_{h,vn}$-calculus to the typed $\lambda^{\ell,r}_{h,vn}$-calculus which is a right inverse of the region erasing function, *i.e.*, $rer(\mathcal{C}_{rg}(P)) \equiv P$, and which commutes with the label erasure functions, *i.e.*, $er(\mathcal{C}_{rg}(P)) \equiv \mathcal{C}_{rg}(er(P))$. Also, following remark 29, we notice that the typing context of the compiled code satisfies the conditions in proposition 36 provided that if Γ is the typing context of the source code and $x : A \in \Gamma$ then the type A is not of the shape $\times(B^+)$.

Next we remark that the type system entails the coherence of the heap context of a program; this leads to the following *progress* property.

Proposition 37 (progress) *Let P be a typable program in the region enriched calculus such that $\mathsf{frv}(P) = \emptyset$. Then P decomposes as $F[H[\Delta]]$ (proposition 31) and either (i) P reduces or (ii) Δ has the shape $@(x, r^*, y^+)$ or $\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ T$, where $x \in \mathsf{fv}(P)$.*

Of course, we must also prove that the region enriched types are preserved by reduction.

Proposition 38 (subject reduction, types and effects) *If $\Gamma \vdash^{rg} P : e$ and $P \rightarrow P'$ then $\Gamma \vdash^{rg} P' : e$.*

Finally, we can show that a well-typed region enriched program does indeed simulate its region erasure.

Theorem 39 (region simulation) *If $\Gamma \vdash^{rg} P : e$, $\mathsf{frv}(P) = \emptyset$, and $rer(P) \xrightarrow{\alpha} Q$ then $P \xrightarrow{\alpha} P'$ and $rer(P') \equiv Q$.*

6 Conclusion

We have shown that our approach, that we call the 'labelling' approach, can be used to obtain certified execution costs on functional programs following a standard compilation chain which composes well with the back-end of a moderately optimising C compiler. The technique allows to compute the cost of the compiled code while reasoning abstractly at the level of the source language and it accounts precisely for the cost of memory management for a particular memory management strategy that uses regions. To provide technical evidence for this claim has required to have an in-depth and sometimes novel look at the formal properties of the compilation chain; notable examples are the commutation property of the CPS transformation and the simulation property for the hoisting and the region aware transformations.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that significantly helped to improve this paper, as well as Anindya Banerjee and Olivier Danvy for their efficient work in the edition process.

References

[1] A. Aiken, M. Fähndrich, R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In Proc. ACM-PLDI, pp 174-185, 1995.

[2] 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.

[3] N. Ayache, R.M. Amadio, Y. Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In Proc. Formal Methods for Industrial Critical Systems (FMICS), Springer-Verlag 7437:32–46, 2012.

[4] R.M. Amadio, Y. Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In Proc. FOPARA, Springer LNCS 7177:72–88, 2012.

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

[6] D. Bacon, P. Cheng, V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In Proc. ACM-POPL, pp 285-298, 2003.

[7] 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.

[8] G. Boudol. Typing safe deallocation. In Proc. ESOP, Springer LNCS 4960:116-130, 2008.

[9] A. Chlipala. A verified compiler for an impure functional language. In Proc. ACM-POPL:93-106, 2010.

[10] M. Christiansen, F. Henglein, H. Niss, P. Velshow. Safe region-based memory management for objects. TOPPS Report D-397 Department of Computer Science, University of Copenhagen (DIKU), October 1998.

[11] The Coq Development Team. The Coq proof assistant. INRIA-Rocquencourt, December 2001. `http://coq.inria.fr`.

[12] K. Crary, D. Walker, G. Morrisett. Typed memory management in a calculus of capabilities. In Proc. ACM-POPL, pp 262-275, 1999.

[13] P.-L. Curien. An abstract framework for environment machines. Theoret. Comput. Sci., 82(2):389-402, 1991.

[14] P. Fradet, D. Le Métayer. Compilation of functional languages by program transformation. ACM Transactions on Programming Languages and Systems, 13(1):21–51, 1991.

[15] D. Gurr. Semantic frameworks for complexity. PhD thesis, University of Edinburgh, 1991.

[16] F. Henglein, H. Makholm, H. Niss. Effect types and region-based memory management. In *Advanced topics in types and programming languages*, B. Pierce (ed.), MIT Press, 2005.

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

[18] J. Lucassen, D. Gifford. Polymorphic effect systems. In Proc. ACM-POPL, pp 47-57, 1988.

[19] X. Li, L. Yun, T. Mitra, A. Roychoudhury. Chronos: A timing analyzer for embedded software. Sci. Comput. Program. 69(1-3): 56–67, 2007.

[20] H. Makholm. A language-independent framework for region inference. PhD thesis, University of Copenhagen, 2003.

[21] 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.

[22] A. Perlis. Epigrams on programming. SIGPLAN Notices Vol. 17(9):7-13, 1982.

[23] G. Plotkin. Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2):125-159, 1975.

[24] Y. Régis-Gianas. An annotating compiler for MiniML. `http://www.pps.univ-paris-diderot.fr/~yrg/fun-cca`.

[25] Y. Régis-Gianas, F. Pottier. A Hoare logic for call-by-value functional programs. In Proc. Mathematics of Program Construction, pp 305-335, 2008.

[26] D. Sands. Complexity analysis for a lazy higher-order language. In Proc. ESOP, Springer LNCS 432:361-376, 1990.

[27] M. Tofte, J.-P. Talpin. Region-based memory management. Information and Computation. 132(2):109-176, 1997.

[28] P. Tranquilli. Indexed labelling for loop iteration dependent costs. Deliverable 5.1, Project CerCo, FP7-ICT-2009-C-243881, 2012.

[29] M. Wand. Continuation-based program transformation strategies. Journal of ACM, 27(1):164–180, 1980.

[30] B. Wegbreit. Mechanical Program Analysis. Commun. ACM, 18(9):528–539, 1975.

A Proofs

We outline the proofs of the results we have stated.

Proof of proposition 3 [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 $\lambda x.(x \mid K) \equiv K$ holds, for all continuations K such that K is an abstraction.

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:

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

   We proceed by induction on M.

   x We expand the definition of $x \mid K$ 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 $\lambda x^{+}.M \mid K$ 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:

$$\begin{array}{l} er(@(M_0, \ldots, M_n)) \mid er(K) \\ \equiv er(M_0) \mid \lambda x_0. \ldots er(M_n) \mid \lambda x_n.@(x_0, \ldots, x_n, er(K)) \\ \equiv er(M_0) \mid \lambda x_0. \ldots er(M_n \mid \lambda x_n.@(x_0, \ldots, x_n, K)) \\ \equiv \cdots \\ \equiv er(M_0 \mid \lambda x_0. \ldots M_n \mid \lambda x_n.@(x_0, \ldots, x_n, K)) \\ \equiv er(@(M_0, \ldots, M_N) \mid K) \ . \end{array}$$

   $\ell > M$ 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:

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

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

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

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

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

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

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

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

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

□

Proof of proposition 4 [CPS simulation]

The proof takes the following steps.

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

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

We proceed by induction on M.

M **is a 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 $K \equiv \lambda y.N$. We observe:

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

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

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

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:

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

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

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

where $M^* \mid \lambda x^*.N$ stands for $M_0 \mid \lambda x_0 \ldots M_n \mid \lambda x_n.N$ 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:

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

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

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

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:

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

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:

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

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

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

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

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

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

$$\begin{array}{l} E[\ell > M] \mid K_{[\ ]} \\ \equiv \ell > M \mid K_E \\ \equiv \ell > (M \mid K_E) \\ \xrightarrow{\ell} (M \mid K_E) \\ \equiv E[M] \mid K_{[\ ]}\ . \end{array}$$

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

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

□

Proof of proposition 7 [VN 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 $\mathcal{R}(\mathcal{C}_{vn}(P)) \equiv P$.

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{l} \mathcal{R}(\mathcal{E}_{vn}((y^*), x)[N]) \\ \equiv \mathcal{R}(\mathsf{let}\ x = (y^*)\ \mathsf{in}\ N) \\ \equiv [(y^*)/x]\mathcal{R}(N) . \end{array}$$

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

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

(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 $er(\mathcal{C}_{vn}(P)) \equiv \mathcal{C}_{vn}(er(P))$.

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

Proof of proposition 8 [VN simulation]

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

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

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 $N \xrightarrow{\ell} E[N'']$. ☐

Proof of proposition 10 [CC commutation]

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

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

$$\begin{array}{l} er(\mathcal{C}_{cc}(@(x, y^+))) \\ \equiv er(\mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+)) \\ \equiv \mathsf{let}\ (c, e) = x\ \mathsf{in}\ @(c, e, y^+) \\ \equiv \mathcal{C}_{cc}(@(x, y^+)) \\ \equiv er(\mathcal{C}_{cc}(@(x, y^+)))\ . \end{array}$$

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

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

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

$$\begin{array}{l}
er(\mathcal{C}_{cc}(\mathsf{let}\ x = \lambda x^+.N\ \mathsf{in}\ M)) \\
\equiv er(\ \mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\ldots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\
\quad\ \mathsf{let}\ e = (z_1,\ldots,z_k), x = (c,e)\ \mathsf{in}\ \mathcal{C}_{cc}(M)\ ) \\
\equiv \mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\ldots,z_k) = e\ \mathsf{in}\ er(\mathcal{C}_{cc}(N))\ \mathsf{in} \\
\quad\ \mathsf{let}\ e = (z_1,\ldots,z_k), x = (c,e)\ \mathsf{in}\ er(\mathcal{C}_{cc}(M)) \\
\equiv \mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\ldots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(er(N))\ \mathsf{in} \\
\quad\ \mathsf{let}\ e = (z_1,\ldots,z_k), x = (c,e)\ \mathsf{in}\ \mathcal{C}_{cc}(er(M)) \qquad \text{(by ind. hyp.)} \\
\equiv \mathcal{C}_{cc}(er(\mathsf{let}\ x = \lambda x^+.N\ \mathsf{in}\ M))\ .
\end{array}$$

$\ell > M$ We have:

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

□

Proof of proposition 11 [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:

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

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 $E(x) = \lambda x^+.M$ and $\mathsf{fv}(\lambda x^+.M) = \{z_1,\ldots,z_k\}$. Then:

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

  with $\mathcal{C}_{cc}(E)(x) = (c,e)$, $\mathcal{C}_{cc}(E)(c) = \lambda e, x^+.\mathsf{let}\ (z_1,\ldots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(M)$ and $\mathcal{C}_{cc}(E)(e) = (z_1,\ldots,z_k)$. Therefore:

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

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

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

□

Proof of proposition 12 [on hoisting transformations]

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

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

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

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

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 $\lambda^{\ell}_{cps,vn}$ 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 $\lambda^{\ell}_{cps,vn}$-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 $\lambda^{\ell}_{cc,vn}$ belong to the set I (trivially).

5. The set I is an invariant of the hoisting transformations, *i.e.*, if $M \in I$ and $M \leadsto N$ 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 $M \leadsto N$ 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 $\lambda^{\ell}_{cc,vn}$ 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.

$\mathsf{let}\ x = C\ \mathsf{in}\ M'$ 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. □

Proof of proposition 13 [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 $er(M) \leadsto$ entails that $M \leadsto$. 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 $M \leadsto^* N \not\leadsto$. By (1) $er(M) \leadsto^* er(N)$, and by (2) $er(N) \not\leadsto$. Hence $\mathcal{C}_h(er(M)) \equiv er(N) \equiv er(\mathcal{C}_h(M))$. $\Box$

Proof of proposition 15 [hoisting simulation]

Definition 40 *A (strong) simulation on the terms of the $\lambda^{\ell}_{cps,vn}$-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 41 *A (pre-)congruence on the terms of the $\lambda^{\ell}_{cps,vn}$-calculus is an equivalence relation (a pre-order) which is preserved by the operators of the calculus.*

Definition 42 *Let $\simeq$ be the smallest congruence on terms of the $\lambda^{\ell}_{cps,vn}$-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 43 *Let $\succeq$ the smallest pre-congruence on terms of the $\lambda^{\ell}_{cps,vn}$-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 44 *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 45 *Let M be a term of the $\lambda^{\ell}_{cps,vn}$-calculus. If $M \xrightarrow{\alpha} M'$ and $M \leadsto N$ then there is N' such that $N \xrightarrow{\alpha} N'$ and $M'\ (\leadsto^*) \circ S_h\ N'$.*

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

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 14. 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 $E(y) = (z_1, \dots z_n)$ and $1 \le i \le n$.

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

Proof of theorem 16 [commutation and simulation]

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

Proof of proposition 18 [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 $\mathcal{L}_i(N) \in W_i$. 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 $@(\mathcal{L}_0(M_1), \ldots, \mathcal{L}_0(M_n)) \in W_1$.

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

$$\begin{aligned} P &\equiv \mathcal{C}(M) \\ &\equiv \mathcal{C}(er(\mathcal{L}(M))) \\ &\equiv er(\mathcal{C}(\mathcal{L}(M))) \quad \text{(by theorem 16(1))} \; . \end{aligned}$$

(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 (value named, 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 $\mathcal{L}_i(M) \mid K$ 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 $\mathcal{C}_{cps}(\mathcal{L}(M)) = \mathcal{L}_0(M) \mid K_0$. We proceed by induction on M and case analysis assuming that if $i = 0$ then $K = \lambda y.N$.

x, $i = 0$ We have: $\mathcal{L}_0(x) \mid K = x \mid K = [x/y]N$.

x, $i = 1$ We have: $\mathcal{L}_i(x) \mid k = x \mid k = @(k, x)$.

$\lambda x^{+}.M$, $i = 0$ We have:

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

and we apply the inductive hypothesis on $\mathcal{L}_1(M) \mid k$ and closure under value substitution.

$\lambda x^{+}.M$, $i = 1$ We have:

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

and we apply the inductive hypothesis on $\mathcal{L}_1(M) \mid k$.

$@(M_1, \ldots, M_n)$, $i = 0$ We have:

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

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

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

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

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

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

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

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

$\pi_j(M)$ We have:

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

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

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

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

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

Proof of proposition 20 [instrumentation vs. labelling]

As a preliminary step, we show that for all terms M and values V, V' of the λ^{ℓ}-calculus the following (mutually dependent) properties hold.

1. $[\psi(V)/x]\psi(V') \equiv \psi([V/x]V')$.
2. $[\psi(V)/x]\mathcal{I}(M) \equiv \mathcal{I}([V/x]M)$.

Let $S=[V_1/x_1,\dots,V_n/x_n]$ denote a substitution in the λ^{ℓ}-calculus. Then let $\psi(S)$ be the substitution $[\psi(V_1)/x_1,\dots,\psi(V_n)/x_n]$. We prove the following generalisation of the proposition.

For all terms M and substitutions S, if $\psi(S)\mathcal{I}(M) \Downarrow (m,V)$ then $\mathcal{I}(SM) \Downarrow_{\Lambda} V'$, $\mathsf{costof}(\Lambda)=m$ and $\psi(V')\equiv V$.

We proceed by induction on the length of the derivation of the judgement $\psi(S)\mathcal{I}(M) \Downarrow (m,V)$ and case analysis on M.

We consider the case for application which explains the need for the generalisation. Suppose $\psi(S)\mathcal{I}(@(M_0, M_1, \ldots, M_n)) \Downarrow (m, V)$. By the shape of $\mathcal{I}(@(M_0, M_1, \ldots, M_n))$ this entails that $\psi(S)\mathcal{I}(M_i) \Downarrow (m_i, V_i)$ for $i = 0, \ldots, n$. By induction hypothesis, $\mathcal{I}(SM_i) \Downarrow_{\Lambda_i} V_i'$, $\mathsf{costof}(\Lambda_i) = m_i$, and $\psi(V_i') = V_i$, for $i = 0, \ldots, n$. We also have $@(V_0, V_1, \ldots, V_n) \Downarrow (m_{n+1}, V)$ and $m = m_0 \oplus \cdots \oplus m_{n+1}$. Since $\psi(V_0') = V_0$, this requires $V_0' = \lambda x_1 \cdots x_n.M'$ and $V_0 = \lambda x_1 \cdots x_n.\mathcal{I}(M')$. So we must have $[V_1/x_1, \ldots, V_n/x_n]\mathcal{I}(M') \Downarrow (m_{n+1}, V)$. Again by inductive hypothesis, this entails that $\mathcal{I}([V_1'/x_1, \ldots, V_n'/x_n]M') \Downarrow_{\Lambda_{n+1}} V'$, $\mathsf{costof}(\Lambda_{n+1}) = m_{n+1}$, and $\psi(V') = V$. We conclude that $\mathcal{I}(S(@(M_0, M_1, \ldots, M_n))) \Downarrow_\Lambda V'$ with $\Lambda = \Lambda_0 \cdots \Lambda_{n+1}$. $\square$

Proof of proposition 22 [subject reduction]

The proof of this result is standard, so we just recall the main steps.

1. Prove a weakening lemma: $\Gamma \vdash M : A$ implies $\Gamma, x : B \vdash M : A$ for x fresh.

2. Prove a substitution lemma: $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : A$ implies $\Gamma \vdash [N/x]M : B$; by induction on the proof of M.

3. Derive by iteration the following substitution lemma: $\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ and $\Gamma \vdash N_i : A_i$ for $i = 1, \ldots, n$ implies $\Gamma \vdash [N_1/x_1, \ldots, N_n/x_n]M : B$.

4. With reference to Table 2, notice that in an evaluation context one does not cross any binder. Then we have that $E[M] \equiv [M/x]E[x]$ for x fresh variable. Moreover, if $\Gamma \vdash E[M] : A$ then for some B, $\Gamma \vdash M : B$.

5. Now examine the 5 possibilities for reduction specified in Table 2. They all have the shape $E[\Delta] \to E[\Delta']$. By the previous remark, it suffices to show that if $\Gamma \vdash \Delta : B$ then $\Gamma \vdash \Delta' : B$. Note that the typing rules in Table 14 are driven by the syntax of the term. Then the property is checked by case analysis while appealing, for the first two rewriting rules, to the substitution properties mentioned above.

Proof of proposition 23 [type CPS]

First, we prove the following properties at once by induction on the structure of the term (possibly a value).

1. If $\Gamma \vdash V : A$ then $\mathcal{C}_{cps}(\Gamma) \vdash \psi(V) : \mathcal{C}_{cps}(A)$.

2. If $\Gamma \vdash M : A$ then $\mathcal{C}_{cps}(\Gamma), k : \neg\mathcal{C}_{cps}(A) \vdash (M \mid k) : R$.

3. If $\Gamma \vdash M : A$ and $\mathcal{C}_{cps}(\Gamma), \Gamma', x : \mathcal{C}_{cps}(A) \vdash N : R$ then $\mathcal{C}_{cps}(\Gamma), \Gamma' \vdash (M \mid (\lambda x.N)) : R$.

We illustrate the analysis for the cases of abstraction and application.

Abstraction Suppose $\Gamma \vdash \lambda x^+.M : A^+ \to B$ is derived from $\Gamma, x^+ : A^+ \vdash M : B$. We prove the 3 properties above.

(1) By induction hypothesis (property 2), we know:

$$\mathcal{C}_{cps}(\Gamma), x^+ : \mathcal{C}_{cps}(A)^+, k : \neg\mathcal{C}_{cps}(B) \vdash (M \mid k) : R \; .$$

Then, recalling that:

$$\psi(\lambda x^+.M) \equiv \lambda x^+, k.(M \mid k) \text{ and } \mathcal{C}_{cps}(A^+ \to B) = \mathcal{C}_{cps}(A)^+, \neg\mathcal{C}_{cps}(B) \to R \ ,$$

we derive:

$$\mathcal{C}_{cps}(\Gamma) \vdash \psi(\lambda x^+.M) : \mathcal{C}_{cps}(A^+ \to B) \ .$$

(2) Recall that $(\lambda x^+.M) \mid k \equiv @(k, \psi(\lambda x^+.M))$. By property 1, we derive:

$$\mathcal{C}_{cps}(\Gamma) \vdash \psi(\lambda x^+.M) : \mathcal{C}_{cps}(A^+ \to B)$$

Then by weakening and substitution we derive

$$\mathcal{C}_{cps}(\Gamma), k : \neg\mathcal{C}_{cps}(A^+ \to B) \vdash \psi(\lambda x^+.M) : \mathcal{C}_{cps}(A^+ \to B)$$

Finally the application rule gives

$$\mathcal{C}_{cps}(\Gamma), k : \neg\mathcal{C}_{cps}(A^+ \to B) \vdash ((\lambda x^+.M) : k) : R \ .$$

(3) Suppose additionally that $\mathcal{C}_{cps}(\Gamma), \Gamma', y : \mathcal{C}_{cps}(A^+ \to B) \vdash N : R$. Recall that $(\lambda x^+.M \mid \lambda y.N) \equiv [\psi(\lambda x^+.M)/y]N$. By property 1, we know that:

$$\mathcal{C}_{cps}(\Gamma) \vdash \psi(\lambda x^+.M) : \mathcal{C}_{cps}(A^+ \to B) \ .$$

Then by weakening and substitution we derive that:

$$\mathcal{C}_{cps}(\Gamma), \Gamma' \vdash [\psi(\lambda x^+.M)/y]N : R \ .$$

Application Suppose $\Gamma \vdash @(M_0, \ldots, M_n) : B$ is derived from $\Gamma \vdash M_0 : A$, $A \equiv A_1, \ldots, A_n \to B$, and $\Gamma \vdash M_i : A_i$ for $i = 1, \ldots, n$. In this case, we just look at the last two properties since an application cannot be a value.

(2) Clearly:

$$\mathcal{C}_{cps}(\Gamma), \Gamma', x_n : \mathcal{C}_{cps}(A_n) \vdash @(x_0, x_1, \ldots, x_n, k) : R \ ,$$

where $\Gamma' \equiv x_0 : \mathcal{C}_{cps}(A), \ldots, x_{n-1} : \mathcal{C}_{cps}(A_{n-1}), k : \neg\mathcal{C}_{cps}(B)$. By induction hypothesis (property 3) on M_n, we derive:

$$\mathcal{C}_{cps}(\Gamma), \Gamma' \vdash (M_n : \lambda x_n.@(x_0, x_1, \ldots, x_n, k)) : R \ .$$

Then by applying the inductive hypothesis (property 3) on $M_{n-1}, \ldots, M_0$ we obtain:

$$\mathcal{C}_{cps}(\Gamma), k : \neg\mathcal{C}_{cps}(B) \vdash (M_0 \mid \lambda x_0. \cdots M_n \mid \lambda x_n.@(x_0, x_1, \ldots, x_n, k)) : R \ .$$

(3) Suppose additionally that $\mathcal{C}_{cps}(\Gamma), \Gamma'', y : \mathcal{C}_{cps}(B) \vdash N : R$. Then we have:

$$\mathcal{C}_{cps}(\Gamma), \Gamma', \Gamma'', x_n : \mathcal{C}_{cps}(A_n) \vdash @(x_0, x_1, \ldots, x_n, \lambda y.N) : R \ ,$$

where $\Gamma' \equiv x_0 : \mathcal{C}_{cps}(A), \ldots, x_{n-1} : \mathcal{C}_{cps}(A_{n-1})$. Then proceed as in the previous case by applying the inductive hypothesis (property 3) on $M_n, \ldots, M_0$.

The proof of the omitted cases follows a similar pattern. Now to derive proposition 23, recall that $\mathcal{C}_{cps}(M) \equiv M \mid \lambda x.@(\mathit{halt}, x)$. Then we obtain the desired statement from the property 3 above observing that if $\Gamma \vdash M : A$ and $\mathcal{C}_{cps}(\Gamma), \mathit{halt} : \neg\mathcal{C}_{cps}(A), x : \mathcal{C}_{cps}(A) \vdash @(\mathit{halt}, x) : R$ then $\mathcal{C}_{cps}(\Gamma), \mathit{halt} : \neg\mathcal{C}_{cps}(A) \vdash \mathcal{C}_{cps}(M) : R$.

Proof of proposition 24 [subject reduction, value named]

First, we prove some standard properties for the type system described in Table 15.

Weakening If $\Gamma \vdash^{vn} M$ then $\Gamma, x : A \vdash^{vn} M$ with x fresh.

Variable substitution If $\Gamma, x : A \vdash^{vn} M$ and $y : A \in \Gamma$ then $\Gamma \vdash^{vn} [y/x]M$. This property generalizes to $\Gamma, x^+ : A^+ \vdash^{vn} M$ and $y^+ : A^+ \in \Gamma$ implies $\Gamma \vdash^{vn} [y^+/x^+]M$.

Type substitution If $\Gamma \vdash^{vn} M$ then $[B/t]\Gamma \vdash^{vn} M$.

Next, suppose $\Gamma \vdash^{vn} M$ and $M \to N$ according to the rules specified in Table 4. This means $M \equiv E[\Delta]$ where for some Γ', we have $\Gamma, \Gamma' \vdash^{vn} \Delta$ and Δ is either an application, or a projection or a labelling. We consider each case in turn.

$\Delta \equiv @(x, y^+)$. Then $y^+ : A^+ \in \Gamma, \Gamma'$, $\Gamma' \equiv \Gamma_1, x : A^+ \to R, \Gamma_2$, x is bound to some function $\lambda z^+.M'$, and $\Gamma, \Gamma_1, z^+ : A^+ \vdash^{vn} M'$. By weakening, we have $\Gamma, \Gamma', z^+ : A^+ \vdash^{vn} M'$ and by substitution $\Gamma, \Gamma' \vdash^{vn} [y^+/z^+]M'$. Then we derive $\Gamma \vdash^{vn} E[[y^+/z^+]M']$ as required.

$\Delta \equiv \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ M'$. This case splits in two sub-cases: the first for product types and the second for existential types.

Product $\Gamma' \equiv \Gamma_1, y : \times(A_1, \ldots, A_n), \Gamma_2$, $1 \le i \le n$, $\Gamma, \Gamma', x : A_i \vdash^{vn} M'$, and for some $z_1, \ldots, z_n$, $z_1 : A_1, \ldots, z_n : A_n \in \Gamma, \Gamma_1$. By substitution, $\Gamma, \Gamma' \vdash^{vn} [z_i/x]M'$. Then we derive $\Gamma \vdash^{vn} E[[z_i/x]M']$ as required.

Existential $i = 1$, $\Gamma' \equiv \Gamma_1, y : \exists t.A, \Gamma_2$, $\Gamma, \Gamma', x : A \vdash^{vn} M'$ with $t \notin \mathsf{ftv}(\Gamma, \Gamma')$, and for some z, B, we have $z : [B/t]A \in \Gamma, \Gamma_1$. By type substitution, $\Gamma, \Gamma', x : [B/t]A \vdash^{vn} M'$ and by substitution, $\Gamma, \Gamma' \vdash^{vn} [z/x]M'$. Then we derive $\Gamma \vdash^{vn} E[[z/x]M']$ as required.

$\Delta \equiv \ell > M'$. Then $\Gamma, \Gamma' \vdash^{vn} M'$ and we derive $\Gamma \vdash^{vn} E[M']$ as required.

Proof of proposition 25 [type value named]

We prove at once the following two properties:

1. If $\Gamma \vdash M : R$ then $\Gamma \vdash^{vn} \mathcal{C}_{vn}(M)$.
2. If $\Gamma \vdash V : A$, $V \neq id$ and $\Gamma, y : A \vdash^{vn} N : R$ then $\Gamma \vdash^{vn} \mathcal{E}_{vn}(V, y)[N]$.

We proceed by induction on the structure of M and V along the pattern of the definition of the value named translation in Table 5. We spell out two typical cases.

$M \equiv @(x^*, V, V^*)$, $V \neq id$. Suppose $\Gamma \vdash @(x^*, V, V^*) : R$. This entails $\Gamma \vdash V : A$ for some type A. We also have $\Gamma, y : A \vdash @(x^*, y, V^*) : R$ and by inductive hypothesis (property 1) $\Gamma, y : A \vdash^{vn} \mathcal{C}_{vn}(@(x^*, y, V^*))$. Then, we apply the inductive hypothesis on V (property 2) to derive that: $\Gamma \vdash^{vn} \mathcal{E}_{vn}(V, y)[\mathcal{C}_{vn}(@(x^*, y, V^*))]$, and this last term equals $\mathcal{C}_{vn}(@(x^*, V, V^*))$.

$V \equiv (x^*, V', V^*)$, $V' \neq id$. Suppose $\Gamma, y : A \vdash^{vn} N$ and $\Gamma \vdash (x^*, V', V^*) : A$. This entails $\Gamma \vdash V' : B$ for some type B and $\Gamma, z : B \vdash (x^*, z, V^*) : A$. By weakening and inductive hypothesis (property 2) on (x^*, z, V^*) we derive $\Gamma, z : B \vdash^{vn} \mathcal{E}_{vn}((x^*, z, V^*), y)[N]$. Then by inductive hypothesis on V' (again by property 2) we derive:

$$\Gamma \vdash^{vn} \mathcal{E}_{vn}(V', z)[\mathcal{E}_{vn}((x^*, z, V^*), y)[N]] \ ,$$

and this last term equals $\mathcal{E}_{vn}((x^*, V', V^*), y)[N]$.

Proof of proposition 26 [type closure conversion]

By induction on the typing of $\Gamma \vdash^{vn} M$ according to the rules specified in Table 15. We detail the cases of abstraction and application.

Abstraction Suppose $\Gamma \vdash^{vn} \mathsf{let}\ x = \lambda y^+.M\ \mathsf{in}\ N$ is derived from $\Gamma, y^+ : A^+ \vdash^{vn} M$ and $\Gamma, x : A^+ \to R \vdash^{vn} N$. Let us pose $\{z^*\} = \mathsf{fv}(\lambda y^+.M)$. Then for some C^* we have $z^* : C^* \in \Gamma$. We have to show that:

$$\begin{array}{ll}\mathcal{C}_{cc}(\Gamma) \vdash^{vn} & \mathsf{let}\ c = \lambda e, y^+.\mathsf{let}\ (z^*) = e\ \mathsf{in}\ \mathcal{C}_{cc}(M)\ \mathsf{in} \\ & \mathsf{let}\ e = (z^*)\ \mathsf{in} \\ & \mathsf{let}\ x' = (c, e)\ \mathsf{in} \\ & \mathsf{let}\ x = (x')\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ .\end{array}$$

By inductive hypothesis on M, variable substitution, and weakening we derive: $\mathcal{C}_{cc}(\Gamma), \Gamma' \vdash^{vn} \mathcal{C}_{cc}(M)$, with $\Gamma' \equiv c : \times(C^*), \mathcal{C}_{cc}(A)^+ \to R, e : \times(C^*)$.

Also, by inductive hypothesis on N and weakening we derive:

$$\mathcal{C}_{cc}(\Gamma), \Gamma', \Gamma'' \vdash^{vn} \mathcal{C}_{cc}(N) \ ,$$

with $\Gamma'' \equiv x' : [\times(C^*)/t]B, x : \exists t.B$ and $B \equiv \times((t, \mathcal{C}_{cc}(A)^+ \to R), t)$.

Application Suppose $\Gamma \vdash^{vn} @(x, y^+)$ is derived from $x : A^+ \to R, y^+ : A^+ \in \Gamma$. We have to show:

$$\begin{array}{ll}\mathcal{C}_{cc}(\Gamma) \vdash^{vn} & \mathsf{let}\ x' = \pi_1(x)\ \mathsf{in} \\ & \mathsf{let}\ (c, e) = x'\ \mathsf{in}\ @(c, e, y^+)\ .\end{array}$$

Since $\mathcal{C}_{cc}(A^+ \to R) \equiv \exists t. \times((t, \mathcal{C}_{cc}(A)^+ \to R), t)$, the judgement above is derived from:

$$\mathcal{C}_{cc}(\Gamma), x' : \times((t, \mathcal{C}_{cc}(A)^+ \to R), t), c : (t, \mathcal{C}_{cc}(A)^+ \to R), e : t \vdash^{vn} @(c, e, y^+) \ .$$

Proof of proposition 27 [type hoisting]

First, we show the following property.

Strengthening If $\Gamma, x : A \vdash^{vn} P$ and $x \notin \mathsf{fv}(P)$ then $\Gamma \vdash^{vn} P$.

Then we proceed by case analysis (3 cases) on the hoisting transformations specified in Table 7. They all have the shape $D[\Delta] \leadsto D[\Delta']$, so it suffices to show that if $\Gamma \vdash^{vn} \Delta$ then $\Gamma \vdash^{vn} \Delta'$. We detail the analysis for the transformation (h_2). Suppose:

$$\Gamma \vdash^{vn} \mathsf{let}\ x = \lambda w^+.\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M\ \mathsf{in}\ N,$$

$$
\begin{aligned}
&rer(t) && = t\\
&rer(A^+ \xrightarrow{e} R) && = rer(A)^+ \to R\\
&rer(\times()) && = \times()\\
&rer(\times(A^+)\mathsf{at}(r)) && = \times(rer(A)^+)\\
&rer((\exists t.A)\mathsf{at}(r)) && = \exists t.rer(A)\\
\\
&rer(\lambda r^*, x^+.T) && = \lambda x^+.rer(T)\\
&rer(()) && = ()\\
&rer((x^+)\mathsf{at}(r)) && = (x^+)\\
\\
&rer(\mathsf{let}\ x = V\ \mathsf{in}\ P) && = \mathsf{let}\ x = rer(V)\ \mathsf{in}\ rer(P)\\
&rer(@(x, r^*, y^+)) && = @(x, y^+)\\
&rer(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T) && = \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ rer(T)
\end{aligned}
$$

Table 19: Region erasure for types, values and terms.

with $\{w^+\} \cap \mathsf{fv}(\lambda z^+.T) = \emptyset$, is derived from:

$$
\begin{array}{ll}
(1) & \Gamma, x : A^+ \to R \vdash^{vn} N\ ,\\
(2) & \Gamma, w^+ : A^+, z^+ : B^+ \vdash^{vn} T\ ,\\
(3) & \Gamma, w^+ : A^+, y : B^+ \to R \vdash^{vn} M\ .
\end{array}
$$

Then we derive:

$$\Gamma \vdash^{vn} \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = \lambda w^+.M\ \mathsf{in}\ N\ .$$

as follows:

$$
\begin{array}{lll}
(1') & \Gamma, x : A^+ \to R, y : B^+ \to R \vdash^{vn} N & \text{(by (1) and weakening)}\\
(2') & \Gamma, z^+ : B^+ \vdash^{vn} T & \text{(by (2) and strengthening)}\\
(3') & \Gamma, w^+ : A^+, y : B^+ \to R \vdash^{vn} M & \text{(by (3))}\ .
\end{array}
$$

Proof theorem 28 [type preserving compilation]

Suppose M term of the λ^ℓ-calculus and $\Gamma \vdash M : A$. Then:

$$
\begin{array}{ll}
\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A) \vdash \mathcal{C}_{cps}(M) : R & \text{(by proposition 23)}\\
\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A) \vdash^{vn} \mathcal{C}_{vn}(\mathcal{C}_{cps}(M)) & \text{(by proposition 25)}\\
\mathcal{C}(\Gamma), halt : \exists t. \times((t, \mathcal{C}(A) \to R), t) \vdash^{vn} \mathcal{C}_{cc}(\mathcal{C}_{vn}(\mathcal{C}_{cps}(M))) & \text{(by proposition 26)}
\end{array}
$$

Next recall that the compiled term $\mathcal{C}(M)$ is the result of iterating the hoisting transformations on the term $\mathcal{C}_{cc}(\mathcal{C}_{vn}(\mathcal{C}_{cps}(M)))$ a finite number of times. Hence, by proposition 27 we conclude:

$$\mathcal{C}(\Gamma), halt : \exists t. \times((t, \mathcal{C}(A) \to R), t) \vdash^{vn} \mathcal{C}(M)\ .$$

Proof of proposition 31 [decomposition]

With reference to Table 17, we know that a program P is a list of function definitions, determining the function context F, followed by a term T. The latter is a list of value definitions and region allocations and disposals, determining the heap context H, and ending either in an application or a projection or a labelling. This last part of the program corresponds to the redex Δ.

Proof of proposition 35 [region erasure]

First, we notice that the region erasure function is invariant under region substitutions: $rer([r'/r]A) = rer(A)$. Then we prove at once the following two properties where it is intended that the judgements on the left are derivable in the type and effect system described in Table 18 and the ones on the right in the type system described in Table 15.

1. If $\Gamma \vdash^{rg} P : e$ then $rer(\Gamma) \vdash^{vn} rer(P)$.
2. If $\Gamma \vdash^{rg} V : A$ then $rer(\Gamma) \vdash^{vn} rer(V) : rer(A)$.

We detail the cases of abstraction and application.

Abstraction Suppose $\Gamma \vdash^{rg} \lambda r^*, y^+.T : \forall r^*.A^+ \xrightarrow{e} R$ is derived from $\Gamma, y^+ : A^+ \vdash^{rg} T : e$. Then by inductive hypothesis, $rer(\Gamma), y^+ : rer(A)^+ \vdash^{vn} rer(T)$. And we conclude: $rer(\Gamma) \vdash^{vn} \lambda y^+.T : rer(A)^+ \to R$ as required.

Application Suppose $\Gamma \vdash^{rg} @(x, r^+, y^+)$ is derived from $x : B \in \Gamma$, $B \equiv \forall r_1^*.A^+ \xrightarrow{e} R$, $y^+ : [r^*/r_1^*]A^+ \in \Gamma$. Then, by the invariance property of the region erasure function noticed above, $x : rer(A)^+ \to R, y^+ : rer(A)^+ \in rer(\Gamma)$. So we conclude: $rer(\Gamma) \vdash^{vn} @(x, y^+)$ as required.

Proof of proposition 36 [region enrichment]

We define a region enrichment function ren from the programs of the $\lambda^{\ell}_{h,vn}$-calculus to those of the $\lambda^{\ell,r}_{h,vn}$-calculus. With reference to Table 7, we recall that a program P of the $\lambda^{\ell}_{h,vn}$ is composed of a list of function definitions and a term. Thus P is decomposed uniquely as $F[T]$ where F is a functional context defined as follows

$$F ::= [\,] \parallel \mathsf{let}\ id = \lambda id^+.T\ \mathsf{in}\ F\ .$$

We fix one region variable r and define the region enrichment function relatively to it as follows:

$ren(F[T])$	$= ren(F)[\mathsf{let\ all}(r)\ \mathsf{in}\ ren(T)]$	(Programs)
$ren([\,])$	$= [\,]$	(Function contexts)
$ren(\mathsf{let}\ x = \lambda y^+.T\ \mathsf{in}\ F)$	$= \mathsf{let}\ x = \lambda r, y^+.ren(T)\ \mathsf{in}\ ren(F)$	
$ren(@(x, y^+))$	$= @(x, r, y^+)$	(Restricted terms)
$ren(\mathsf{let}\ x = C\ \mathsf{in}\ T)$	$= \mathsf{let}\ x = ren(C)\ \mathsf{in}\ ren(T)$	
$ren(\ell > T)$	$= \ell > ren(T)$	
$ren(())$	$= ()$	(Restricted let-bindable terms)
$ren((x^+))$	$= (x^+)\mathsf{at}(r)$	
$ren(\pi_i(x))$	$= \pi_i(x)$	

The intuition is that a region r is created initially and never disposed, that all tuples are allocated in this region, and that at every function call we pass this region as a parameter. Then all functions when applied will produce (at most) an effect $\{r\}$.

Next we extend the region enrichment function to types as follows:

$$\begin{array}{ll}
\mathit{ren}(t) & = t \\
\mathit{ren}(A^+ \to R) & = \forall r.\mathit{ren}(A)^+ \xrightarrow{\{r\}} R \\
\mathit{ren}(\times()) & = \times() \\
\mathit{ren}(\times(A^+)) & = \times(\mathit{ren}(A)^+)\mathsf{at}(r) \\
\mathit{ren}((\exists t.A)) & = (\exists t.\mathit{ren}(A))\mathsf{at}(r)
\end{array}$$

We notice that function definitions are region closed and so are the functional types in the image of the function *ren*. Let us denote with Γ_0 a type context such that if $x : A \in \Gamma_0$ then A is not a type of the shape $\times(B^+)$ or $\exists t.B$. It follows that $\mathsf{frv}(\mathit{ren}(\Gamma_0)) = \emptyset$.

We show the following *enrichment* property:

If $\Gamma_0, \Gamma \vdash^{vn} T$ then $\mathit{ren}(\Gamma_0, \Gamma) \vdash^{rg} \mathit{ren}(T) : \{r\}$.

We detail three cases.

Tuple construction Suppose $\Gamma_0, \Gamma \vdash^{vn} \mathsf{let}\ x = (y^+)\ \mathsf{in}\ T$ is derived from

$$\Gamma_0, \Gamma \vdash^{vn} (y^+) : A^+ \quad \text{and} \quad \Gamma_0, \Gamma, x : \times(A^+) \vdash^{vn} T\ .$$

Then we derive:

$$\mathit{ren}(\Gamma_0, \Gamma) \vdash^{rg} \mathsf{let}\ x = (y^+)\mathsf{at}(r)\ \mathsf{in}\ \mathit{ren}(T) : \{r\}$$

from:

$$\begin{array}{ll}
\mathit{ren}(\Gamma_0, \Gamma) \vdash^{rg} (y^+)\mathsf{at}(r) : \times(\mathit{ren}(A)^+)\mathsf{at}(r) & \text{and} \\
\mathit{ren}(\Gamma_0, \Gamma), x : \times(\mathit{ren}(A)^+)\mathsf{at}(r) \vdash^{rg} \mathit{ren}(T) : \{r\} & \text{(inductive hypothesis).}
\end{array}$$

Projection Suppose $\Gamma_0, \Gamma \vdash^{vn} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T$ is derived from $y : \times(A_1, \ldots, A_n) \in \Gamma$, $1 \leq i \leq n$, and $\Gamma_0, \Gamma, x : A_i \vdash^{vn} T$. Then:

$$y : \times(\mathit{ren}(A_1), \ldots, \mathit{ren}(A_n))\mathsf{at}(r) \in \mathit{ren}(\Gamma) \text{ and } \mathit{ren}(\Gamma_0, \Gamma, x : A_i) \vdash^{vn} \mathit{ren}(T) : \{r\}\ .$$

Hence:

$$\mathit{ren}(\Gamma_0, \Gamma) \vdash^{vn} \mathit{ren}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T) : \{r\}\ .$$

Application Suppose $\Gamma_0, \Gamma \vdash^{vn} @(x, y^+)$ is derived from $x : A^+ \to R, y^+ : A^+ \in \Gamma_0, \Gamma$. Then we derive $\mathit{ren}(\Gamma_0, \Gamma) \vdash^{rg} @(x, r, y^+) : \{r\}$ from:

$$x : \forall r.\mathit{ren}(A)^+ \xrightarrow{\{r\}} R,\ y^+ : \mathit{ren}(A)^+ \in \mathit{ren}(\Gamma_0, \Gamma)\ .$$

Finally, we derive from the enrichment property above the following two properties which suffice to derive the statement:

- If $\Gamma_0 \vdash^{vn} \lambda x^+.T : A^+ \to R$ then $\mathit{ren}(\Gamma_0) \vdash^{rg} \mathit{ren}(\lambda x^+.T) : \mathit{ren}(A^+ \to R)$.
- If $\Gamma_0 \vdash^{vn} T$ then $\mathit{ren}(\Gamma_0) \vdash^{rg} \mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ \mathit{ren}(T) : \emptyset$.

Proof of proposition 37 [progress]

First we prove by induction on the structure of a heap context the following monotonicity property of the coherence predicate:

if $Coh(H, L)$ and $L \subseteq L'$ then $Coh(H, L')$.

Let $\mathsf{frv}(H)$ denote the set of region variables free in a heap context. If the program $P \equiv F[H[\Delta]]$ is typable then a judgement of the form $\Gamma \vdash^{rg} H[\Delta] : e$ is derivable. We show by induction on the typing of such judgement the following two properties:

1. $Coh(H, \mathsf{frv}(H))$.
2. If $r \in \mathsf{frv}(H)$ then $r \in e$.

Because we assumed $\mathsf{frv}(P) = \emptyset$ we must have $\mathsf{frv}(H) = \emptyset$ and by the first property we derive $Coh(H, \emptyset)$. In other terms, in a typable program without free region variables the heap context is coherent relatively to the empty set. We look at the shape of Δ.

Labelling If Δ is a labelling then the program may reduce.

Application If Δ is an application $@(x, r^*, y^+)$ then either the variable x is not bound in the function context or it is bound to a function value. The fact that the number of parameters matches the number of arguments is forced (as usual) by typing. Then a reduction is possible.

Projection The last case is when the redex is a projection $\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T$. Similarly to the previous case, either y is not bound or it is bound to a tuple allocated at a region r. Then we must be able to type a term of the shape:

$$\Gamma, y : (A)\mathsf{at}(r) \vdash^{rg} H[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T] \tag{1}$$

The fact that the projection is in the right range is forced (as usual) by typing. To fire the transition we need to check that $NDis(r, H)$ holds. In fact let us argue that if the predicate does not hold then the judgement (1) above cannot be typed. By inspecting the definition of $NDis(r, H)$ we see that for the predicate to fail, H must have the shape $H_1[\mathsf{dis}(r)\ \mathsf{in}\ [H_2]]$ for a heap context H_1 which contains neither allocations nor disposals on the region r. But then $H_2[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T]$ must produce a visible effect on r. Indeed the typing system records an effect on r when projecting y and this effect cannot be hidden by an allocation because the region variable r is free in the context $\Gamma, y : (A)\mathsf{at}(r)$. Then the typing of $\mathsf{dis}(r)\ \mathsf{in}\ [H_2[\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T]]$ fails because the typing forbids disposing a region which is in the effect of the continuation.

Proof of proposition 38 [subject reduction, types and effects]

First we prove some standard properties (cf. proof of proposition 24) and a specific property on injective region substitutions.

Weakening If $\Gamma \vdash^{rg} P : e$ then $\Gamma, x : A \vdash^{rg} P : e$ with x fresh.

Variable substitution If $\Gamma, x : A \vdash^{rg} P : e$ and $y : A \in \Gamma$ then $\Gamma \vdash^{rg} [y/x]P : e$.

Type substitution If $\Gamma \vdash^{rg} P : e$ then $[B/t](\Gamma) \vdash^{rg} P : e$.

Injective region substitution If $\Gamma \vdash^{rg} T : e$ and σ is a (finite domain) region substitution which is injective on $\mathsf{frv}(T) \cup e$ then $\sigma\Gamma \vdash^{rg} \sigma T : \sigma e$.

We detail the proof of the last property which proceeds by induction on the typing proof of $\Gamma \vdash^{rg} T : e$.

Application Suppose $\Gamma \vdash^{rg} @(x, r^*, y^+) : [r^*/r_1^*]e$ is derived from $x : B, y^+ : [r^*/r_1^*]A^+ \in \Gamma$, $B \equiv \forall r_1.A^+ \xrightarrow{e} R$, $\mathsf{frv}(B) = \emptyset$, r^* distinct variables. Notice that $\mathsf{frv}(A^+) \cup e \subseteq \{r_1^*\}$. It follows that $\mathsf{frv}(@(x, r^*, y^+)) \cup [r^*/r_1^*]e = \{r^*\}$. So suppose σ is an injective substitution on r^* so that $r'^* = (\sigma r)^*$. We remark:

$$\begin{array}{ll}
\sigma B & \equiv B \\
\sigma[r^*/r_1^*]A^+ & \equiv [r'^*/r_1^*]A^+ \\
\sigma[r^*/r_1^*]e & \equiv [r'^*/r_1^*]e \\
\sigma @(x, r^*, y^+) & \equiv @(x, r'^*, y^+) \\
\sigma r^* & \text{distinct}
\end{array}$$

Then we can prove $\sigma\Gamma \vdash^{rg} \sigma @(x, r^*, y^+) : \sigma([r^*/r_1^*]e)$ by the typing rule for application.

Unit Suppose $\Gamma \vdash^{rg} \mathsf{let}\ x = ()\ \mathsf{in}\ T : e$ is derived from $\Gamma, x : \times() \vdash^{rg} T : e$ and σ is injective on $\mathsf{frv}(\mathsf{let}\ x = ()\ \mathsf{in}\ T) \cup e$. Then σ is injective on $\mathsf{frv}(T) \cup e$, by inductive hypothesis $\sigma\Gamma, x : \times() \vdash^{rg} \sigma T : \sigma e$, and we conclude $\sigma\Gamma \vdash^{rg} \sigma(\mathsf{let}\ x = ()\ \mathsf{in}\ T) : \sigma e$.

Product Suppose

$$\Gamma \vdash^{rg} \mathsf{let}\ x = (y^+)\mathsf{at}(r)\ \mathsf{in}\ T : e \cup \{r\}$$

is derived from

$$\Gamma, x : \times(A^+)\mathsf{at}(r) \vdash^{rg} T : e,$$

$y^+ : A^+ \in \Gamma$ and σ is injective on the set:

$$\mathsf{frv}(\mathsf{let}\ x = (y^+)\mathsf{at}(r)\ \mathsf{in}\ T) \cup e \cup \{r\} = \mathsf{frv}(T) \cup e \cup \{r\}\ .$$

Then σ is injective on $\mathsf{frv}(T) \cup e$. By inductive hypothesis:

$$\sigma\Gamma, x : (\times(\sigma A^+))\mathsf{at}(\sigma r) \vdash^{rg} \sigma T : \sigma e\ .$$

Moreover $y^+ : (\sigma A)^+ \in \sigma\Gamma$. So we conclude:

$$\sigma\Gamma \vdash^{rg} (\mathsf{let}\ x = (y^+)\mathsf{at}(\sigma r)\ \mathsf{in}\ T) : \sigma e \cup \{\sigma r\}\ .$$

Existential This case is similar to the previous one. Suppose:

$$\Gamma \vdash^{rg} \mathsf{let}\ x = (y)\mathsf{at}(r)\ \mathsf{in}\ T : e \cup \{r\}$$

is derived from:

$$\Gamma, x : (\exists t.A)\mathsf{at}(r) \vdash^{rg} T : e\ ,$$

$y : [B/t]A \in \Gamma$ and σ is injective on the set:

$$\mathsf{frv}(\mathsf{let}\ x = (y)\mathsf{at}(r)\ \mathsf{in}\ T) \cup e \cup \{r\} = \mathsf{frv}(T) \cup e \cup \{r\}\ .$$

Then σ is injective on $\mathsf{frv}(T) \cup e$. By inductive hypothesis:

$$\sigma\Gamma, x : (\exists t.\sigma A)\mathsf{at}(\sigma r) \vdash^{rg} \sigma T : \sigma e \ .$$

Moreover $y : \sigma[B/t]A \in \sigma\Gamma$. We notice $\sigma[B/t]A \equiv [\sigma B/t]\sigma A$ and $\sigma(\exists t.A) \equiv \exists t.\sigma A$. Then we conclude:

$$\sigma\Gamma \vdash^{rg} (\mathsf{let}\ x = (y)\mathsf{at}(\sigma r)\ \mathsf{in}\ T) : \sigma e \cup \{\sigma r\} \ .$$

Projection Suppose:

$$\Gamma \vdash^{rg} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T : e \cup \{r\}$$

is derived from $y : \times(A_1, \ldots, A_n)\mathsf{at}(r) \in \Gamma$, $1 \leq i \leq n$, $\Gamma, x : A_i \vdash^{rg} T : e$, and σ is injective on $\mathsf{frv}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T) \cup e \cup \{r\}$. Then σ is injective on $\mathsf{frv}(T) \cup e$ and by inductive hypothesis:

$$\sigma\Gamma, x : \sigma(\times(A_1, \ldots, A_n)\mathsf{at}(r)) \vdash^{rg} \sigma T : \sigma e \ .$$

We conclude:

$$\sigma\Gamma \vdash^{rg} \sigma(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T) : \sigma e \cup \{\sigma r\} \ .$$

The case where y has an existential type is similar.

Disposal Suppose $\Gamma \vdash^{rg} \mathsf{dis}(r)\ \mathsf{in}\ T : e \cup \{r\}$ is derived from $\Gamma \vdash^{rg} T : e$, $r \notin e$, and σ is injective on $\mathsf{frv}(\mathsf{dis}(r)\ \mathsf{in}\ T) \cup e \cup \{r\}$. Then σ is injective on $\mathsf{frv}(T) \cup e$ and by inductive hypothesis $\sigma\Gamma \vdash^{rg} \sigma T : \sigma e$. Also $\sigma r \notin \sigma e$. We conclude:

$$\sigma\Gamma \vdash^{rg} \sigma(\mathsf{dis}(r)\ \mathsf{in}\ T) : \sigma e \cup \{\sigma r\} \ .$$

Allocation Suppose $\Gamma \vdash^{rg} \mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ T : e$ is derived from $\Gamma \vdash^{rg} T : e \cup \{r\}$, $r \notin e \cup \mathsf{frv}(\Gamma)$, and σ is injective on $\mathsf{frv}(\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ T) \cup e$. Up to renaming, we can choose r so that it is not in the domain or image of σ. Then σ is injective on $\mathsf{frv}(T) \cup e \cup \{r\}$ and by inductive hypothesis $\sigma\Gamma \vdash^{rg} \sigma T : \sigma e \cup \{\sigma r\}$. Also, by the choice above, $\sigma r \notin \sigma e \cup \sigma\Gamma$. We conclude $\sigma\Gamma \vdash^{rg} \sigma(\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ T) : \sigma e$.

Labelling Suppose $\Gamma \vdash^{rg} \ell > T : e$ is derived from $\Gamma \vdash^{rg} T : e$ and σ is injective on $\mathsf{frv}(\ell > T) \cup \{e\}$. By inductive hypothesis $\sigma\Gamma \vdash^{rg} \sigma T : \sigma e$ and we conclude $\sigma\Gamma \vdash^{rg} \sigma(\ell > T) : \sigma e$.

Subeffect Suppose $\Gamma \vdash^{rg} P : e$ is derived from $\Gamma \vdash^{rg} P : e'$, $e' \subseteq e$, and σ is injective on $\mathsf{frv}(P) \cup e$. By inductive hypothesis $\sigma\Gamma \vdash^{rg} \sigma P : \sigma e'$ and we conclude $\sigma\Gamma \vdash^{rg} \sigma P : \sigma e$.

If $\Gamma_P \vdash^{rg} P : e_P$ then we know that $P \equiv F[H[\Delta]]$ and the reduced term has the shape $F[H[\Delta']]$. For some Γ we have $\Gamma \vdash^{rg} \Delta : e'$. We show that then $\Gamma \vdash^{rg} \Delta' : e'$ and $\mathsf{frv}(\Delta') \subseteq \mathsf{frv}(\Delta)$. Then we claim that the typing proof for the surrounding context $F[H]$ can be ported to the program $F[H[\Delta']]$.

We proceed by case analysis on the reduction rule applied and its typing. Notice that the typing is syntax directed except for the subeffect rule. So for instance, if $\Gamma \vdash^{rg} \Delta : e'$ and Δ is an application then for some $e'' \subseteq e'$ we can derive $\Gamma \vdash^{rg} \Delta : e''$ where the last rule being applied is the one for application. A similar argument holds for the cases where Δ is a projection or a labelling.

Application Suppose $\Gamma \vdash^{rg} @(x, r^*, y^+) : e''$ with $e'' = [r^*/r_1^*]e$ is derived from $x : B, y^+ : [r^*/r_1^*]A^+ \in \Gamma$, $B \equiv \forall r_1.A^+ \xrightarrow{e} R$, $\mathsf{frv}(B) = \emptyset$, r^* distinct variables. Since the program reduces, x must be bound to a region closed function $\lambda r_1^*, z^+.T$ in the functional context F and $\Gamma_1, z^+ : A^+ \vdash^{rg} T : e$ where Γ_1 is a prefix of Γ and $\{r_1^*\} \cap \mathsf{frv}(\Gamma_1) = \emptyset$. We notice that the substitution $\sigma = [r^*/r_1^*]$ is injective on $\mathsf{frv}(T) \cup e \subseteq \{r_1^*\}$, hence by the injective substitution property we derive:

$$\Gamma_1, z^+ : (\sigma A)^+ \vdash^{rg} \sigma T : \sigma e \ .$$

Notice that we must have $y^+ : (\sigma A)^+ \in \Gamma_1$ hence by the variable substitution property and weakening we derive:

$$\Gamma \vdash^{rg} \sigma[r^*/r_1^*, y^+/z^+]T : [r^*/r_1^*]e \ .$$

We conclude by noticing that:

$$\mathsf{frv}([r^*/r_1^*, y^+/z^+]T) \subseteq \mathsf{frv}(@(x, r^*, y^+)) = \{r^*\} \ .$$

Projection Suppose $\Gamma \vdash^{rg} \mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T : e \cup \{r\}$ is derived from $y : \times(A_1, \ldots, A_n) \in \Gamma$, $1 \leq i \leq n$ $\Gamma, x : A_i \vdash^{rg} T : e$. Since the program reduces, y must be bound to a tuple $(z_1, \ldots, z_n)\mathsf{at}(r)$ in the heap context H and $z_1 : A_1, \ldots, z_n : A_n \in \Gamma$. Then by variable substitution we derive $\Gamma \vdash^{rg} [z_i/x]T : e$. Also notice that $\mathsf{frv}([z_i/x]T) = \mathsf{frv}(\mathsf{let}\ x = \pi_i(y)\ \mathsf{in}\ T)$.

The case where y has an existential type is similar except that it relies on type substitution too (cf. proof of proposition 24).

Labelling Suppose $\Gamma \vdash^{rg} \ell > T : e$ is derived from $\Gamma \vdash^{rg} T : e$. Since $\Delta \equiv \ell > T \xrightarrow{\ell} T$, the conclusion is immediate.

Proof of theorem 39 [region simulation]

First, we observe that the region erasure function commutes with variable substitution:

$$[x/y]\mathit{rer}(T) \equiv \mathit{rer}([x/y]T) \ .$$

By proposition 31, P decomposes as $F[H[\Delta]]$ where Δ is either an application, or a projection, or a labelling. The region erasure function commutes with this decomposition too, so that we can write $\mathit{rer}(P)$ as $\mathit{rer}(F)[\mathit{rer}(H)[\mathit{rer}(\Delta)]]$, where $\mathit{rer}(F)[\mathit{rer}(H)]$ is an evaluation context. If $\mathit{rer}(P) \xrightarrow{\alpha} Q$ then we proceed by case analysis on the reduction rule being applied. We detail the case where Δ is an application $@(x, r^*, y^+)$. Then we must have $F \equiv F_1[\mathsf{let}\ x = \lambda r_1^*, z^+.T\ \mathsf{in}\ F_2]$ and

$$\begin{array}{rcl} \mathit{rer}(P) & \equiv & \mathit{rer}(F_1)[\mathsf{let}\ x = \lambda z^+.\mathit{rer}(T)\ \mathsf{in}\ \mathit{rer}(F_2)[\mathit{rer}(H)[@(x, y^+)]] \\ & \rightarrow & \mathit{rer}(F_1)[\mathsf{let}\ x = \lambda z^+.\mathit{rer}(T)\ \mathsf{in}\ \mathit{rer}(F_2)[\mathit{rer}(H)[[y^+/z^+]\mathit{rer}(T)]] \ . \end{array}$$

Since P is typable, the heap context is coherent and then P can simulate the reduction above as follows:

$$P \rightarrow F[H[[r^*/r_1^*, y^+/z^+]T]]$$

noticing that $\mathit{rer}([r^*/r_1^*, y^+/z^+]T) \equiv [y^+/z^+]\mathit{rer}(T)$ (initial remark and invariance of the region erasure function under region substitutions).

A Polynomial Time λ-calculus with Multithreading and Side Effects [*]

Antoine Madet

Univ Paris Diderot, Sorbonne Paris Cité
PPS, UMR 7126, CNRS, F-75205 Paris, France
madet@pps.univ-paris-diderot.fr

Abstract

The framework of *light logics* has been extensively studied to control the complexity of higher-order functional programs. We propose an extension of this framework to multithreaded programs with side effects, focusing on the case of polynomial time. After introducing a modal λ-calculus with parallel composition and *regions*, we prove that a realistic call-by-value evaluation strategy can be computed in polynomial time for a class of well-formed programs. The result relies on the simulation of call-by-value by a polynomial *shallow-first* strategy which preserves the evaluation order of side effects. Then, we provide a polynomial type system that guarantees that well-typed programs do not go wrong. Finally, we illustrate the expressivity of the type system by giving a programming example of concurrent iteration producing side effects over an inductive data structure.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Formal Definitions and Theory; F.2 [*Analysis of Algorithms and Problem Complexity*]: General

Keywords λ-calculus, side effect, region, thread, resource analysis.

1. Introduction

Quantitative resource analysis of programs is a challenging task in computer science. Besides being essential for the development of safety-critical systems, it provides interesting viewpoints on the structure of programs.

The framework of *light logics* (see *e.g.* **LLL** [12], **ELL** [10], **SLL** [13]) which originates from Linear Logic [11], have been deeply studied to control the complexity of higher-order functional programs. In particular, polynomial time λ-calculi [5, 18] have been proposed as well as various type systems [8, 9] guaranteeing complexity bounds of functional programs. Recently, Amadio and the author proposed an extension of the framework to a higher-order functional language with multithreading and side effects [16], focusing on the case of elementary time (**ELL**).

In this paper, we consider a more reasonable complexity class: polynomial time. The functional core of the language is the *light* λ-calculus [18] that features the modalities *bang* (written '!') and *paragraph* (written '§') of **LLL**. The notion of *depth* (the number of nested modalities) which is standard in light logics is used to control the duplication of data during the execution of programs. The language is extended with side effects by means of read and write operations on *regions* which were introduced to represent areas of the store [15]. Threads can be put in parallel and interact through a shared state.

There appears to be no direct combinatorial argument to bound a call-by-value evaluation strategy by a polynomial. However, the *shallow-first* strategy (*i.e.* redexes are eliminated in a depth-increasing order) is known to be polynomial in the functional case [4, 12]. Using this result, Terui shows [18] that a class of *well-formed* light λ-terms strongly terminates in polynomial time (*i.e.* every reduction strategy is polynomial) by proving that any reduction sequence can be simulated by a *longer* one which is shallow-first. Following this method, our contribution is to show that a class of well-formed call-by-value programs with side effects and multithreading can be simulated in polynomial time by shallow-first reductions. The bound covers any scheduling policy and takes thread generation into account.

Reordering a reduction sequence into a shallow-first one is non-trivial: the evaluation order of side effects must be kept unchanged in order to preserve the semantics of the program. An additional difficulty is that reordering produces non call-by-value sequences but fails for an arbitrary larger relation (which may even require exponential time). We identify an intermediate *outer-bang* relation $\longrightarrow_{\mathsf{ob}}$ which can be simulated by shallow-first ordering and this allows us to simulate the call-by-value relation $\longrightarrow_{\mathsf{v}}$ which is contained in the outer-bang relation. We illustrate this development in Figure 1.

The paper is organized as follows. We start by presenting the language with multithreading and regions in Section 2 and define the largest reduction relation. Then, we introduce a *polynomial depth system* in Section 3 to control the depth of program occurrences. Well-formed programs in the depth system follow Terui's discipline [18] on the functional side and the *stratification of regions* by depth level that we introduced previously [16]. We prove in Section 4 that the class of outer-bang strategies (containing call-by-value) can be simulated by shallow-first reductions of exactly the same length. We review the proof of polynomial soundness of the shallow-first strategy in Section 5. We provide a *polynomial type system* in Section 6 which results from a simple decoration of the polynomial depth system with linear types. We derive the stan-

[*] 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).

PPDP'12, September 19–21, 2012, Leuven, Belgium.
Copyright © 2012 ACM 978-1-4503-1522-7/12/09…$10.00

$$\begin{tikzcd}
 & & & & \mathbf{PTIME} \arrow[d, phantom, "\not\cap"] \\
 & & & & {} \\
\longrightarrow_{\mathsf{v}} \arrow[r, phantom, "\subsetneq"] \arrow[ddrr, Rightarrow] & {} & \longrightarrow_{\mathsf{ob}} \arrow[r, phantom, "\subsetneq"] \arrow[dd, Rightarrow] & {} & \longrightarrow \arrow[d, Rightarrow] \\
 & & & \text{implies} \ \longleftarrow & \mathit{fail} \\
\longrightarrow_{\mathsf{v}} \arrow[r, phantom, "\subsetneq"] & {} & \longrightarrow_{\mathsf{ob}} \arrow[r, phantom, "\subseteq"] & {} & \mathbf{PTIME}
\end{tikzcd}$$

$\Longrightarrow$: reordering into shallow-first preserving length and semantics of reductions

Figure 1. Simulation by shallow-first ordering

dard subject reduction proposition and progress proposition which states that well-types programs reduce to values. Finally, we illustrate the expressivity of the type system in Section 7 by showing that it is polynomially complete in the extensional sense and we give a programming example of a concurrent iteration producing side effects over an inductive data structure.

2. A modal λ-calculus with multithreading and regions

As mentioned previously, the functional core of the language is a modal λ-calculus with constructors and destructors for the modalities '!' and '§' that are used to control the duplication of data. The global store is partitioned into a finite number of regions where each region abstracts a set of memory locations. Following [1], side effects are produced by read and write operators on regions. A parallel operator allows to evaluate concurrently several terms which can communicate through regions. As we shall see in Section 7, this abstract non-deterministic language entails complexity bounds for languages with concrete memory locations representing *e.g.* references, channels or signals.

The syntax of the language is presented in Figure 2. We have

-variables	$x, y, \ldots$		
-regions	$r, r', \ldots$		
-terms	M	$::=$	$x \mid r \mid \star \mid \lambda x.M \mid MM \mid !M \mid \S M$
			$\mathsf{let}\ !x = M\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ M$
			$\mathsf{get}(r) \mid \mathsf{set}(r, M) \mid (M \parallel M)$
-stores	S	$::=$	$r \Leftarrow M \mid (S \parallel S)$
-programs	P	$::=$	$M \mid S \mid (P \parallel P)$

Figure 2. Syntax of the language

the usual set of variables $x, y, \ldots$ and a set of regions $r, r', \ldots$ The set of terms M contains variables, regions, the terminal value (unit) $\star$, λ-abstractions, applications, modal terms $!M$ and $\S M$ (resp. called !-terms and §-terms) and the associated let !-binders and let §-binders. We have an operator $\mathsf{get}(r)$ to read a region r, an operator $\mathsf{set}(r, M)$ to assign a term M to a region r and a parallel operator $(M \parallel N)$ to evaluate M and N in parallel. A store S is the composition of several assignments $r \Leftarrow M$ in parallel and a program P is the combination of several terms and stores in parallel. Note that stores are global, *i.e.* they always occur in empty contexts.

In the following we write $\dagger$ for $\dagger \in \{!, \S\}$ and we define $\dagger^0 M = M$ and $\dagger^{n+1} M = \dagger(\dagger^n M)$. Terms $\lambda x.M$ and $\mathsf{let}\ \dagger x = N\ \mathsf{in}\ M$ bind occurrences of x in M. 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 $\mathsf{FO}(x, M)$. The number of free occurrences in M is denoted by $\mathsf{FO}(M)$. $M[N/x]$ denotes the term M in which each free occurrence of x has been substituted by N.

Each program has an *abstract syntax tree* where variables, regions and unit constants are leaves, λ-abstractions and $\dagger$-terms have one child, and applications and let $\dagger$-binders have two children. An example is given in Figure 3. A path starting from the root to a

$$P = \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, (!x)(\S x)) \parallel r \Leftarrow !(\lambda x.x\star)$$

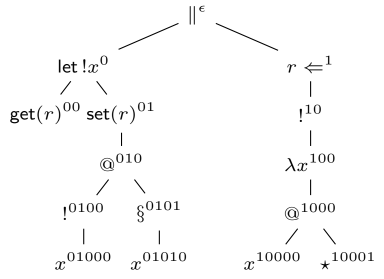


Figure 3. Syntax tree and addresses of P

node of the tree denotes an *occurrence* of the program whose address is a word $w \in \{0, 1\}^*$ hereby denoted in exponent form. We write $w \sqsubseteq w'$ when w is a prefix of w'. We denote the number of occurrences in P by $|P|$.

The operational semantics of the language is given in Figure 4. In order to prove the later simulation result, the largest reduction relation $\longrightarrow$ (which shall contain call-by-value) is presented.

-structural rules-

$$\begin{array}{rcl} P \parallel P' & \equiv & P' \parallel P \\ (P \parallel P') \parallel P'' & \equiv & P \parallel (P' \parallel P'') \end{array}$$

-evaluation contexts-

$$\begin{array}{rcl} E & ::= & [\cdot] \mid \lambda x.E \mid EM \mid ME \mid !E \mid \S E \\ & & \mathsf{let}\ !x = E\ \mathsf{in}\ M \mid \mathsf{let}\ \S x = E\ \mathsf{in}\ M \\ & & \mathsf{let}\ !x = M\ \mathsf{in}\ E \mid \mathsf{let}\ \S x = M\ \mathsf{in}\ E \\ & & \mathsf{set}(r, E) \mid r \Leftarrow E \mid (E \parallel P) \mid (P \parallel E) \end{array}$$

-reduction rules-

$$\begin{array}{lrcl} (\beta) & E[(\lambda x.M)N] & \longrightarrow & E[M[N/x]] \\ (!) & E[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] & \longrightarrow & E[M[N/x]] \\ (\S) & E[\mathsf{let}\ \S x = \S N\ \mathsf{in}\ M] & \longrightarrow & E[M[N/x]] \\ (\mathsf{get}) & E[\mathsf{get}(r)] \parallel r \Leftarrow M & \longrightarrow & E[M] \\ (\mathsf{set}) & E[\mathsf{set}(r, M)] & \longrightarrow & E[\star] \parallel r \Leftarrow M \text{-if } \mathsf{FV}(M) = \emptyset \\ (\mathsf{gc}) & E[\star \parallel M] & \longrightarrow & E[M] \end{array}$$

Figure 4. Operational semantics

Programs are considered up to a structural equivalence $\equiv$ which contains the equations for α-renaming, commutativity and associativity of parallel composition. Reduction rules apply modulo structural equivalence, in an evaluation context E which can be any program with exactly one occurrence of a special variable '$[\cdot]$', called the *hole*. We write $E[M]$ for $E[M/[\cdot]]$. Each rule is identified by its name. (β) is the usual β-reduction. $(\dagger)$ are rules for filtering modal terms. (get) is for *consuming* a term from a region. (set) is for *assigning* a closed term to a region. (gc) is for *erasing* a terminated thread.

First, note that the reduction rule (set) generates a *global* assignment, that is out of the evaluation context E. In turn, we require M

to be closed such that it does not contain variables bound in E. Second, several terms can be assigned to a single region. This cumulative semantics allows the simulation of several memory locations by a single region. In turn, reading a region consists in consuming *non-deterministically* one of the assigned terms.

The reduction is very 'liberal' with side effects. The contexts $(P \parallel E)$ and $(E \parallel P)$ embed any scheduling of threads. Moreover, contexts of the shape $r \Leftarrow E$ allow evaluation in the store as exemplified in the following possible reduction:

$$\begin{aligned}\mathsf{set}(r, \lambda x.\mathsf{get}(r)) \parallel r \Leftarrow M &\longrightarrow \star \parallel r \Leftarrow \lambda x.\mathsf{get}(r) \parallel r \Leftarrow M \\ &\longrightarrow \star \parallel r \Leftarrow \lambda x.M\end{aligned}$$

In the rules (β), $(\dagger)$, (gc), the *redex* denotes the term inside the context of the left hand-side and the *contractum* denotes the term inside the context of the right hand-side. In the rule (get), the redex is $\mathsf{get}(r)$ and the contractum is M. In the rule (set), the redex is $\mathsf{set}(r, M)$ and the contractum is M. Finally, $\longrightarrow^+$ denotes the transitive closure of $\longrightarrow$ and $\longrightarrow^*$ denotes the reflexive closure of $\longrightarrow^+$.

3. A polynomial depth system

In this section, we first review the principles of well-formed light λ-terms (Subsection 3.1) and then the stratification of regions by depth level (Subsection 3.2). Eventually we combine the two as a set of inference rules that characterizes a class of *well-formed* programs (Subsection 3.3).

3.1 On light λ-terms

First, we define the notion of depth.

Definition 1. *The* depth $d(w)$ *of an occurrence* w *in a program* P *is the number of* $\dagger$ *labels that the path leading to the end node crosses. The depth* $d(P)$ *of program* P *is the maximum depth of its occurrences.*

With reference to Figure 3, $d(01000) = d(01010) = d(100) = d(1000) = d(10000) = d(10001) = 1$, whereas other occurrences have depth 0. In particular, $d(0100) = d(0101) = d(10) = 0$; what matters in computing the depth of an occurrence is the number of $\dagger$'s that precede strictly the end node. Thus $d(P) = 1$. In the sequel, we say that a program *occurs* at depth i when it corresponds to an occurrence of depth i. For example, $\mathsf{get}(r)$ occur at depth 0 in P. We write $\xrightarrow{i}$ when the redex occurs at depth i; we write $|P|_i$ for the number of occurrences at depth i of P.

Then we can define *shallow-first* reductions.

Definition 2. *A* shallow-first reduction sequence $P_1 \xrightarrow{i_1} P_2 \xrightarrow{i_2} \ldots \xrightarrow{i_n} P_n$ *is such that* $m < n$ *implies* $i_m \leq i_n$. *A* shallow-first strategy *is a strategy that produces shallow-first sequences.*

The polynomial soundness of shallow-first strategies relies on the following properties: when $P \overset{i}{\longrightarrow}{}^* P'$,

$$d(P') \leq d(P) \tag{3.1}$$

$$|P'|_j \leq |P|_j \text{ for } j < i \tag{3.2}$$

$$|P'|_i < |P|_i \tag{3.3}$$

$$|P'| \leq |P|^2 \tag{3.4}$$

To see this in a simple way, assume P is a program such that $d(P) = 2$. By properties (3.1),(3.2),(3.3) we can eliminate all the redexes of P with the shallow-first sequence $P \overset{0}{\longrightarrow}{}^* P' \overset{1}{\longrightarrow}{}^* P'' \overset{2}{\longrightarrow}{}^* P'''$. By property (3.4), $|P'''| \leq |P|^8$. By properties (3.3) the length l of the sequence is such that $l \leq |P| + |P'| + |P''| = p$. Since we can show that $p \leq |P|^8$ we conclude that the shallow-first evaluation of P can be computed in polynomial time.

The well-formedness criterions of light λ-terms are intended to ensure the above four properties. These criterions can be summarized as follows:

- λ-abstraction is affine: in $\lambda x.M$, x may occur at most once and at depth 0 in M.
- $\mathsf{let}\,!$-binders are for duplication: in $\mathsf{let}\ !x = M \ \mathsf{in}\ N$, x may occur arbitrarily many times and at depth 1 in N.
- $\mathsf{let}\,\S$-binders are affine: in $\mathsf{let}\ \S x = M \ \mathsf{in}\ N$, x may occur at most once and at depth 1 in N. The depth of x must be due to a $\S$ modality.
- a !-term may contain at most one occurrence of free variable, whereas a $\S$-term can contain many occurrences of free variables.

By the first three criterions, we observe the following. The depth of a term never increases (property (3.1)) since the reduction rules (β),$(!)$ and $(\S)$ substitute a term for a variable occurring at the same depth. Reduction rules (β) and $(\S)$ are strictly size-decreasing since the corresponding binders are affine. A reduction $(!)$ is strictly size-decreasing at the depth where the redex occurs but potentially size-increasing at deeper levels. Therefore properties (3.2) and (3.3) are also guaranteed. The fourth criterion is intended to ensure a quadratic size increase (property (3.4)). Indeed, take the term Z borrowed from [18] that respects the first three criterions but not the fourth:

$$\begin{gathered} Z = \lambda x.\mathsf{let}\ !x = x \ \mathsf{in}\ !(xx) \\ \underbrace{Z \ldots (Z(Z}_{n \text{ times}}!y)) \longrightarrow^* !(\underbrace{yy \ldots y}_{2^n \text{ times}}) \end{gathered} \tag{3.5}$$

It may trigger an exponential size explosion by repeated application of the duplicating rule $(!)$. The following term

$$\begin{aligned} &Y = \lambda x.\mathsf{let}\ !x = x \ \mathsf{in}\ \S(xx) \\ &\underbrace{Y \ldots (Y(Y}_{n \text{ times}}!y)) \\ \longrightarrow^* &\underbrace{Y \ldots (Y(Y}_{n-2 \text{ times}}(\mathsf{let}\ !x = \S(yy) \ \mathsf{in}\ \S(xx)))) \not\longrightarrow \end{aligned} \tag{3.6}$$

respects the four criterions but cannot be used to apply $(!)$ exponentially.

3.2 On the stratification of regions by depth

In our previous work on elementary time [16], we analyzed the impact of side effects on the depth of occurrences and remarked that arbitrary reads and writes could increase the depth of programs. In the reduction sequence

$$\begin{aligned} (\lambda x.\mathsf{set}(r, x) \parallel \S\mathsf{get}(r))!M &\longrightarrow^* \S\mathsf{get}(r) \parallel r \Leftarrow !M \\ &\longrightarrow \S!M \end{aligned} \tag{3.7}$$

the occurrence M moves from depth 1 to depth 2 during the last reduction step, because the read occurs at depth 0 while the write occurs at depth 1.

Following this analysis, we introduced *region contexts* in order to constrain the depth at which side effects occur. A region context

$$R = r_1 : \delta_1, \ldots, r_n : \delta_n$$

associates a natural number δ_i to each region r_i in a finite set of regions $\{r_1, \ldots, r_n\}$ that we write $dom(R)$. We write $R(r_i)$ for δ_i. Then, the rules of the elementary depth system were designed in such a way that $\mathsf{get}(r_i)$ and $\mathsf{set}(r_i, M)$ may only occur at depth δ_i, thus rejecting (3.7).

Moreover, we remarked that since stores are global, that is they always occur at depth 0, assigning a term to a region breaks stratification whenever $\delta_i > 0$. Indeed, in the reduction

$$\S\mathsf{set}(r, M) \longrightarrow \S\star \parallel r \Leftarrow M \qquad (3.8)$$

where $R(r)$ should be 1, the occurrence M moves from depth 1 to depth 0. Therefore, we revised the definition of depth as follows.

Definition 3. *Let P be a program and R a region context where $dom(R)$ contains all the regions of P. The* revised depth *$d(w)$ of an occurrence w of P is the number of $\dagger$ labels that the path leading to the end node crosses, plus $R(r)$ if the path crosses a store label $r \Leftarrow$. The revised depth $d(P)$ of a program P is the maximum revised depth of its occurrences.*

By considering this revised definition of depth, in (3.8) the occurrence M stays at depth 1. In Figure 3 we now get $d(01000) = d(01010) = 1$, $d(10) = R(r)$ and $d(100) = d(1000) = d(10000) = d(10001) = R(r) + 1$. Other occurrences have depth 0. From now on we shall say depth for the revised definition of depth.

3.3 Inference rules

Now we introduce the inference rules of the polynomial depth system. First, we define region contexts R and variable contexts Γ as follows:

$$\begin{array}{rcl} R & = & r_1 : \delta_1, \ldots, r_n : \delta_n \\ \Gamma & = & x_1 : u_1, \ldots, x_n : u_n \end{array}$$

Regions contexts are described in the previous subsection. A variable context associates each variable with a usage $u \in \{\lambda, \S, !\}$ which constrains the variable to be bound by a λ-abstraction, a let $\S$-binder or a let $!$-binder respectively. We write Γ_u if $dom(\Gamma)$ only contains variables with usage u. A depth judgement has the shape

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

where δ is a natural number. It should entail the following:

- if $x : \lambda \in \Gamma$ then x occurs at depth δ in $\dagger^\delta P$,
- if $x : \dagger \in \Gamma$ then x occurs at depth $\delta + 1$ in $\dagger^\delta P$,
- if $r : \delta' \in R$ then $\mathsf{get}(r)$/$\mathsf{set}(r)$ occur at depth δ' in $\dagger^\delta P$.

The inference rules of the depth system are presented in Figure 5. We comment on the handling of usages. Variables are introduced with usage λ. The construction of $!$-terms updates the usage of variables to $!$ if they all previously had usage λ. The construction of $\S$-terms updates the usage of variables to $\S$ for one part and $!$ for the other part if they all previously had usage λ. In both constructions, contexts with other usages can be weakened. As a result, λ-abstractions bind variables occurring at depth 0, let $!$-binders bind variables occurring at depth 1 in $!$-terms or $\S$-terms, and let $\S$-binders bind variables occurring at depth 1 in $\S$-terms.

To control the duplication of data, the rules for binders have predicates which specify how many occurrences can be bound. λ-abstractions and let $\S$-binders are linear by predicate $\mathsf{FO}(x, M) = 1$ and let $!$-binders are at least linear by predicate $\mathsf{FO}(x, M) \geq 1$.

The depth δ of the judgement is decremented when constructing $\dagger$-terms. This allows to stratify regions by depth level by requiring that $\delta = R(r)$ in the rules for $\mathsf{get}(r)$ and $\mathsf{set}(r, M)$. A store assignment $r \Leftarrow M$ is global hence its judgement has depth 0 whereas the premise has depth $R(r)$ (this reflects the revised notion of depth).

Definition 4. *(Well-formedness) A program P is* well-formed *if a judgement $R; \Gamma \vdash^\delta P$ can be derived for some R, Γ and δ.*

$$\dfrac{x : \lambda \in \Gamma}{R;\Gamma \vdash^\delta x} \qquad \dfrac{}{R;\Gamma \vdash^\delta \star} \qquad \dfrac{}{R;\Gamma \vdash^\delta r}$$

$$\dfrac{\begin{array}{c}\mathsf{FO}(x, M) = 1 \\ R;\Gamma, x : \lambda \vdash^\delta M\end{array}}{R;\Gamma \vdash^\delta \lambda x.M} \qquad \dfrac{R;\Gamma \vdash^\delta M \quad R;\Gamma \vdash^\delta N}{R;\Gamma \vdash^\delta MN}$$

$$\dfrac{\begin{array}{c}\mathsf{FO}(M) \leq 1 \\ R;\Gamma_\lambda \vdash^{\delta+1} M\end{array}}{R;\Gamma_!, \Delta_\S, \Psi_\lambda \vdash^\delta !M} \qquad \dfrac{\begin{array}{c}\mathsf{FO}(x, N) \geq 1 \quad R;\Gamma \vdash^\delta M \\ R;\Gamma, x : ! \vdash^\delta N\end{array}}{R;\Gamma \vdash^\delta \mathsf{let}\ !x = M\ \mathsf{in}\ N}$$

$$\dfrac{R;\Gamma_\lambda, \Delta_\lambda \vdash^{\delta+1} M}{R;\Gamma_!, \Delta_\S, \Psi_\lambda \vdash^\delta \S M} \qquad \dfrac{\begin{array}{c}\mathsf{FO}(x, N) = 1 \quad R;\Gamma \vdash^\delta M \\ R;\Gamma, x : \S \vdash^\delta N\end{array}}{R;\Gamma \vdash^\delta \mathsf{let}\ \S x = M\ \mathsf{in}\ N}$$

$$\dfrac{r : \delta \in R}{R;\Gamma \vdash^\delta \mathsf{get}(r)} \qquad \dfrac{r : \delta \in R \quad R;\Gamma \vdash^\delta M}{R;\Gamma \vdash^\delta \mathsf{set}(r, M)}$$

$$\dfrac{r : \delta \in R \quad R;\Gamma \vdash^\delta M}{R;\Gamma \vdash^0 r \Leftarrow M} \qquad \dfrac{i = 1, 2 \quad R;\Gamma \vdash^\delta P_i}{R;\Gamma \vdash^\delta (P_1 \parallel P_2)}$$

Figure 5. A polynomial depth system

Example 1. *The program P of Figure 3 is well-formed by composition of the two derivation trees of Figure 6. The program Z given in* (3.5) *is not well-formed.*

The depth system is strictly linear in the sense that it is not possible to bind 0 occurrences. We shall see in Section 4 that it allows for a major simplification of the proof of simulation. However, this impossibility to discard data is a notable restriction over light λ-terms. In a call-by-value setting, the sequential composition $M; N$ is usually encoded as the non well-formed term $(\lambda z.N)M$ where $z \notin \mathsf{FV}(N)$ is used to discard the terminal value of M. We show that side effects can be used to simulate the discarding of data even though the depth system is strictly linear. Assume that we dispose of a specific region gr collecting 'garbage' values at each depth level of a program. Then $M; N$ could be encoded as the well-formed program $(\lambda z.\mathsf{set}(gr, z) \parallel N)M$. Using a call-by-value semantics, we would observe the following reduction sequence

$$\begin{array}{rl} M; N \longrightarrow^* V; N \longrightarrow \mathsf{set}(gr, V) \parallel N & \longrightarrow \star \parallel N \parallel gr \Leftarrow V \\ & \longrightarrow N \parallel gr \Leftarrow V \end{array}$$

where $\star$ has been erased by (gc) and V has been garbage collected into gr.

Finally we derive the following lemmas on the depth system in order to get the subject reduction proposition.

Lemma 1 (Weakening and Substitution)**.**

1. *If $R; \Gamma \vdash^\delta P$ then $R; \Gamma, \Gamma' \vdash^\delta P$.*
2. *If $R; \Gamma, x : \lambda \vdash^\delta M$ and $R; \Gamma \vdash^\delta N$ then $R; \Gamma \vdash^\delta M[N/x]$.*
3. *If $R; \Gamma, x : \S \vdash^\delta M$ and $R; \Gamma \vdash^\delta \S N$ then $R; \Gamma \vdash^\delta M[N/x]$.*
4. *If $R; \Gamma, x : ! \vdash^\delta M$ and $R; \Gamma \vdash^\delta !N$ then $R; \Gamma \vdash^\delta M[N/x]$.*

Proposition 1 (Subject reduction)**.** *If $R; \Gamma \vdash^\delta P$ and $P \longrightarrow P'$ then $R; \Gamma \vdash^\delta P'$ and $d(P) \geq d(P')$.*

$$\dfrac{\dfrac{\dfrac{}{r : 0; - \vdash^0 r}}{r : 0; - \vdash^0 \mathsf{get}(r)} \qquad \dfrac{\dfrac{}{r : 0; x : ! \vdash^0 r} \quad \dfrac{\dfrac{\dfrac{}{r : 0; x : \lambda \vdash^1 x}}{r : 0; x : ! \vdash^0 !x} \quad \dfrac{\dfrac{}{r : 0; x : \lambda \vdash^1 x}}{r : 0; x : ! \vdash^0 \S x}}{r : 0; x : ! \vdash^0 !x\S x}}{r : 0; x : ! \vdash^0 \mathsf{set}(r, !x\S x)}}{r : 0; - \vdash^0 \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !x\S x)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r : 0; x : \lambda \vdash^1 x} \quad \dfrac{}{r : 0; x : \lambda \vdash^1 \star}}{r : 0; x : \lambda \vdash^1 x\star}}{r : 0; - \vdash^1 \lambda x.x\star}}{r : 0; - \vdash^0 !(\lambda x.x\star)}}{r : 0; - \vdash^0 r \Leftarrow !(\lambda x.x\star)}$$

Figure 6. Derivation trees

4. Simulation by shallow-first

In this section, we first explain why we need a class of *outer-bang* reduction strategies (Subsection 4.1). Then, we prove that shallow-first simulates any outer-bang strategy and that the result applies to call-by-value (Subsection 4.2).

4.1 Towards outer-bang strategies

Reordering a reduction sequence into a shallow-first one is an iterating process where each iteration consists in commuting two consecutive reduction steps which are applied in 'deep-first' order.

First, we show that this process requires a reduction which is strictly larger than an usual call-by-value relation. Informally, assume $\dagger V$ denotes a value. The following two reduction steps in call-by-value style

$$\mathsf{set}(r, \dagger M) \xrightarrow{1} \mathsf{set}(r, \dagger V) \xrightarrow{0} \star \parallel r \Leftarrow \dagger V$$

commute into the shallow-first sequence

$$\mathsf{set}(r, \dagger M) \xrightarrow{0} \star \parallel r \Leftarrow \dagger M \xrightarrow{1} \star \parallel r \Leftarrow \dagger V$$

which is obviously not call-by-value: first, we write a non-value $\dagger M$ to the store and second we reduce *in* the store! As another example, the following two reduction steps in call-by-value style

$$(\lambda x.\lambda y.xy)\dagger M \xrightarrow{1} (\lambda x.\lambda y.xy)\dagger V \xrightarrow{0} \lambda y.(\dagger V)y$$

commute into the shallow-first sequence

$$(\lambda x.\lambda y.xy)\dagger M \xrightarrow{0} \lambda y.(\dagger M)y \xrightarrow{i} \lambda y.(\dagger V)y$$

which is not call-by-value: we need to reduce inside a λ-abstraction and this is not compatible with the usual notion of value.

Second, we show that an arbitrary relation like $\longrightarrow$ is too large to be simulated by shallow-first sequences. For instance, consider the following reduction of a well-formed program:

$$\begin{aligned} &\mathsf{let}\ !x = !\mathsf{get}(r)\ \mathsf{in}\ \S(xx) \parallel r \Leftarrow M \\ \xrightarrow{1}\ &\mathsf{let}\ !x = !M\ \mathsf{in}\ \S(xx) \qquad (4.1) \\ \xrightarrow{0}\ &\S(MM) \end{aligned}$$

This sequence is deep-first; it can be reordered into a shallow-first one as follows:

$$\begin{aligned} &\mathsf{let}\ !x = !\mathsf{get}(r)\ \mathsf{in}\ \S(xx) \parallel r \Leftarrow M \\ \xrightarrow{0}\ &\S(\mathsf{get}(r)\mathsf{get}(r)) \parallel r \Leftarrow M \qquad (4.2) \\ \xrightarrow{1}\ &\S(M\mathsf{get}(r)) \not\rightarrow \end{aligned}$$

However, the sequence cannot be confluent with the previous one for we try to read the region two times by duplicating the redex $\mathsf{get}(r)$. It turns out that a non shallow-first strategy may require exponential time in the presence of side effects. Consider the well-formed λ-abstraction

$$F = \lambda x.\mathsf{let}\ \S x = x\ \mathsf{in}\ \S\mathsf{set}(r, x); !\mathsf{get}(r)$$

which transforms a $\S$-term into a !-term (think of the type $\S A \multimap !A$ that would be rejected in **LLL**). Then, building on program Z given in (3.5), take

$$Z' = \lambda x.\mathsf{let}\ !x = x\ \mathsf{in}\ F\S(xx)$$

We observe an exponential explosion of the size of the following well-formed program:

$$\begin{aligned} &\underbrace{Z'Z'\dots Z'}_{n \text{ times}}!\star \\ \longrightarrow^*\ &\underbrace{Z'Z'\dots Z'}_{n-1 \text{ times}}(F\S(\star\star)) \\ \longrightarrow^*\ &\underbrace{Z'Z'\dots Z'}_{n-1 \text{ times}}(!(\star\star)) \parallel gr \Leftarrow \S\star \\ \longrightarrow^*\ &!(\underbrace{\star\star\dots\star}_{2^n \text{ times}}) \parallel \underbrace{gr \Leftarrow \S\star \parallel \dots \parallel gr \Leftarrow \S\star}_{n \text{ times}} \end{aligned}$$

where gr is a region collecting the garbage produced by the sequential composition operator of F. This previous sequence is not shallow-first since the redexes $\mathsf{set}(r, M)$ and $\mathsf{get}(r)$ occurring at depth 1 are alternatively applied with other redexes occurring at depth 0. A shallow-first strategy would produce the reduction sequence

$$\underbrace{Z'Z'\dots Z'}_{n \text{ times}}!\star \longrightarrow^* !(\star\star\underbrace{\mathsf{get}(r)\mathsf{get}(r)\dots\mathsf{get}(r)}_{n-1 \text{ times}}) \parallel S$$

where S is the same garbage store as previously but we observe no size explosion.

Following these observations, our contribution is to identify an intermediate *outer-bang* reduction relation that can be simulated by shallow-first sequences. The keypoint is to prevent reductions inside !-terms like in sequence (4.1). For this, we define the *outer-bang* evaluation contexts F in Figure 7. They are not decomposable

$$\begin{aligned} F \quad ::= \quad & [\cdot] \mid \lambda x.F \mid FM \mid MF \mid \S F \\ & \mathsf{let}\ \dagger x = F\ \mathsf{in}\ M \mid \mathsf{let}\ \dagger x = M\ \mathsf{in}\ F \\ & \mathsf{set}(r, F) \mid (F \parallel M) \mid (M \parallel F) \mid r \Leftarrow F \end{aligned}$$

Figure 7. Outer-bang evaluation contexts

in a context of the shape $E[!E']$ and thus cannot be used to reduce in !-terms. In the sequel, $\longrightarrow_{\mathsf{ob}}$ denotes reduction modulo evaluation contexts F.

4.2 Simulation of outer-bang strategies

After identifying a proper outer-bang relation $\longrightarrow_{\mathsf{ob}}$, the main difficulty is to preserve the evaluation order of side effects by shallow-first reordering. For example, the following two reduction steps do not commute:

$$\begin{aligned} &F_1[\mathsf{set}(r, Q)] \parallel F_2[\mathsf{get}(r)] \\ \xrightarrow{i}\ &F_1[\star] \parallel F_2[\mathsf{get}(r)] \parallel r \Leftarrow Q \qquad (4.3) \\ \xrightarrow{j}\ &F_1[\star] \parallel F_2[Q] \end{aligned}$$

We claim that this is not an issue since the depth system enforces that side effects on a given region can only occur at fixed depth, hence that $i = j$. Therefore, we should never need to 'swap' a read with a write on the same region.

We can prove the following crucial lemma.

Lemma 2 (Swapping). *Let P be a well-formed program such that $P \xrightarrow{i}_{\mathsf{ob}} P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$ and $i > j$. Then, there exists P' such that $P \xrightarrow{j}_{\mathsf{ob}} P' \xrightarrow{i}_{\mathsf{ob}} P_2$.*

Proof. We write M the contractum of the reduction $P \xrightarrow{i}_{\mathsf{ob}} P_1$ and N the redex of the reduction $P_1 \xrightarrow{j}_{\mathsf{ob}} P_2$. Assume they occur at addresses w_m and w_n in P_1. We distinguish three cases: (1) M and N are separated (neither $w_m \sqsubseteq w_n$ nor $w_m \sqsupseteq w_n$); (2) M contains N ($w_m \sqsubseteq w_n$); (3) N strictly contains M ($w_m \sqsupseteq w_n$ and $w_m \neq w_n$). For each of them we discuss a crucial subcase:

1. Assume M is the contractum of a (set) rule and that N is the redex of a (get) rule related to the same region. This case has been introduced in example (4.3) where M and N are separated by a parallel node. By well-formedness of P, the redexes $\mathsf{get}(r)$ and $\mathsf{set}(r, Q)$ must occur at the same depth, that is $i = j$, and we conclude that we do not need to swap the reductions.
2. If the contractum M contains the redex N, N may not exist yet in P which makes the swapping impossible. We remark that, for any well-formed program Q such that $Q \xrightarrow{d}_{\mathsf{ob}} Q'$, both the redex and the contractum occur at depth d. In particular, this is true when a contractum occurs in the store as follows:
$$Q = F[\mathsf{set}(r, T)] \xrightarrow{d}_{\mathsf{ob}} Q' = F[\star] \parallel r \Leftarrow T$$
By well-formedness of Q, there exists a region context R such that $R(r) = d$ and the redex $\mathsf{set}(r, T)$ occurs at depth d. By the revised definition of depth, the contractum T occurs at depth d in the store. As a result of this remark, M occurs at depth i and N occurs at depth j. Since $i > j$, it is clear that the contractum M cannot contain the redex N and this case is void.
3. Let N be the redex $\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q$ and let the contractum M appears in R as in the following reduction sequence
$$\begin{aligned} P &= F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] \\ &\xrightarrow{i}_{\mathsf{ob}} P_1 = F[\mathsf{let}\ \S x = \S R\ \mathsf{in}\ Q] \\ &\xrightarrow{j}_{\mathsf{ob}} P_2 = F[Q[R/x]] \end{aligned}$$
By well-formedness, x occurs exactly once in Q. This implies that applying first $P \xrightarrow{j} P'$ cannot discard the redex in R'. Hence, we can produce the following shallow-first sequence of the same length:
$$\begin{aligned} P = F[\mathsf{let}\ \S x = \S R'\ \mathsf{in}\ Q] &\xrightarrow{j}_{\mathsf{ob}} P' = F[Q[R'/x]] \\ &\xrightarrow{i}_{\mathsf{ob}} P_2 = F[Q[R/x]] \end{aligned}$$
Moreover, the reduction $P' \xrightarrow{i}_{\mathsf{ob}} P_2$ must be outer-bang for x cannot occur in a !-term in Q. $\square$

There are two notable differences with Terui's swapping procedure. First, our procedure returns sequences of exactly the same length as the original ones while his may return longer sequences. The reason is that outer-bang contexts force redexes to be duplicated before being reduced, as in reduction (4.2), hence our swapping procedure cannot lengthen sequences more. The other difference is that his calculus is affine whereas ours is strictly linear. Therefore his procedure might shorten sequences by discarding redexes and this breaks the argument for *strong* polynomial termination. His solution is to introduce an auxiliary calculus with explicit discarding for which swapping lengthens sequences. This is at the price of introducing commutation rules which require quite a lot of extra work to obtain the simulation result. We conclude that strict linearity brings major proof simplifications while we have seen it does not cause a loss of expressivity if we use garbage collecting regions.

Using the swapping lemma, we show that any reduction sequence that uses outer-bang evaluation contexts can be simulated by a shallow-first sequence.

Proposition 2 (Simulation by shallow-first). *To any reduction sequence $P_1 \longrightarrow^*_{\mathsf{ob}} P_n$ corresponds a shallow-first reduction sequence $P_1 \longrightarrow^*_{\mathsf{ob}} P_n$ of the same length.*

Proof. By simple application of the bubble sort algorithm: traverse the original sequence from P_1 to P_n, compare the depth of each consecutive reduction steps, swap them by Lemma 2 if they are in deep-first order. Repeat the traversal until no swap is needed. Note that we never need to swap two reduction steps of the same depth, which implies that we never need to reverse the order of dependent side effects. For example, in Figure 8, the sequence $P \xrightarrow{2}_{\mathsf{ob}} P' \xrightarrow{1}_{\mathsf{ob}} P'' \xrightarrow{0}_{\mathsf{ob}} P'''$ is reordered into $P \xrightarrow{0}_{\mathsf{ob}} C \xrightarrow{1}_{\mathsf{ob}} B \xrightarrow{2}_{\mathsf{ob}} P'''$ by 3 traversals. $\square$

$$\begin{array}{ccccccc} P & \xrightarrow{2}_{\mathsf{ob}} & P' & \xrightarrow{1}_{\mathsf{ob}} & P'' & \xrightarrow{0}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{1}_{\mathsf{ob}} & A & \xrightarrow{2}_{\mathsf{ob}} & P'' & \xrightarrow{0}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{1}_{\mathsf{ob}} & A & \xrightarrow{0}_{\mathsf{ob}} & B & \xrightarrow{2}_{\mathsf{ob}} & P''' \\ P & \xrightarrow{0}_{\mathsf{ob}} & C & \xrightarrow{1}_{\mathsf{ob}} & B & \xrightarrow{2}_{\mathsf{ob}} & P''' \end{array}$$

Figure 8. Reordering of $P \longrightarrow^*_{\mathsf{ob}} P'''$ in shallow-first

As an application, we show that the simulation result applies to a call-by-value operational semantics that we define in Figure 9. We

$$\begin{array}{llll} \text{-values} & V & ::= & x \mid \star \mid r \mid \lambda x.M \mid \dagger V \\ \text{-terms} & M & ::= & V \mid MM \mid \S M \mid \mathsf{let}\ \dagger x = M\ \mathsf{in}\ M \\ & & & \mathsf{get}(r) \mid \mathsf{set}(r, M) \mid (M \parallel M) \\ \text{-stores} & S & ::= & r \Leftarrow V \mid (S \parallel S) \\ \text{-programs} & P & ::= & M \mid S \mid (P \parallel P) \\ \text{-contexts} & F_{\mathsf{v}} & ::= & [\cdot] \mid F_{\mathsf{v}} M \mid V F_{\mathsf{v}} \mid \S F_{\mathsf{v}} \\ & & & \mathsf{let}\ \dagger x = F_{\mathsf{v}}\ \mathsf{in}\ M \mid \mathsf{set}(r, F_{\mathsf{v}}) \\ & & & (F_{\mathsf{v}} \parallel P) \mid (P \parallel F_{\mathsf{v}}) \end{array}$$

-reduction rules-

$$\begin{array}{lrcl} (\beta_{\mathsf{v}}) & F_{\mathsf{v}}[(\lambda x.M)V] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (!_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (\S_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{let}\ \S x = \S V\ \mathsf{in}\ M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M[V/x]] \\ (\mathsf{get}_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{get}(r)] \parallel r \Leftarrow V & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[V] \\ (\mathsf{set}_{\mathsf{v}}) & F_{\mathsf{v}}[\mathsf{set}(r, V)] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[\star] \parallel r \Leftarrow V \\ (\mathsf{gc}_{\mathsf{v}}) & F_{\mathsf{v}}[\star \parallel M] & \longrightarrow_{\mathsf{v}} & F_{\mathsf{v}}[M] \end{array}$$

Figure 9. CBV syntax and operational semantics

revisit the syntax of programs with a notion of value V that may be a variable, unit, a region, a λ-abstraction or a $\dagger$-value. Terms and programs are defined as previously (see Figure 2) except that $!M$ cannot be constructed unless M is a value. Store assignments are restricted to values. Evaluation contexts F_{v} are left-to-right call-by-value (obviously we do not evaluate in stores). The call-by-value reduction relation is denoted by $\longrightarrow_{\mathsf{v}}$ and is defined modulo F_{v} and $\equiv$.

From a programming viewpoint, we shall only duplicate values. This explains why we do not want to construct $!M$ if M is not a value.

Call-by-value contexts F_{v} are outer-bang contexts since F_{v} cannot be decomposed as $E[!E']$. This allows the relation $\longrightarrow_{\mathsf{ob}}$ to contain the relation $\longrightarrow_{\mathsf{v}}$. As a result, we obtain the following corollary.

Corollary 1 (Simulation of CBV). *To any reduction sequence $P_1 \longrightarrow_{\mathsf{v}}^* P_n$ corresponds a shallow-first reduction sequence $P_1 \longrightarrow_{\mathsf{ob}}^* P_n$ of the same length.*

Remark that we may obtain a non call-by-value sequence but that the semantics of the program is preserved (we compute P_n).

5. Polynomial soundness of shallow-first

In this section we prove that well-formed programs admit polynomial bounds with a shallow-first strategy. We stress that this subsection is similar to Terui's [18]; the main difficulty has been to design the polynomial depth system such that we could adopt a similar proof method.

As a first step, we define an *unfolding* transformation on programs.

Definition 5. *(Unfolding) The* unfolding *at depth i of a program P, written $\sharp^i(P)$, is defined as follows:*

$$\begin{array}{rcl}
\sharp^i(x) & = & x \\
\sharp^i(r) & = & r \\
\sharp^i(\star) & = & \star \\
\sharp^i(\lambda x.M) & = & \lambda x.\sharp^i(M) \\
\sharp^i(MN) & = & \sharp^i(M)\sharp^i(N)
\end{array}$$

$$\sharp^i(\dagger M) = \begin{cases} \dagger\sharp^{i-1}(M) & \textit{if } i > 0 \\ \dagger M & \textit{if } i = 0 \end{cases}$$

$$\sharp^i(\mathsf{let}\ \dagger x = M\ \mathsf{in}\ N) = \begin{cases} \textit{if } i = 0, M = !M' \textit{ and } \dagger = ! : \\ \mathsf{let}\ !x = \underbrace{MM\ldots M}_{k\ times}\ \mathsf{in}\ \sharp^0(N) \\ \textit{where } k = \mathsf{FO}(x, \sharp^0(N)) \\ \\ \textit{otherwise:} \\ \mathsf{let}\ \dagger x = \sharp^i(M)\ \mathsf{in}\ \sharp^i(N) \end{cases}$$

$$\begin{array}{rcl}
\sharp^i(\mathsf{get}(r)) & = & \mathsf{get}(r) \\
\sharp^i(\mathsf{set}(r, M)) & = & \mathsf{set}(r, \sharp^i(M)) \\
\sharp^i(r \Leftarrow M) & = & r \Leftarrow \sharp^i(M) \\
\sharp^i(P_1 \parallel P_2) & = & \sharp^i(P_1) \parallel \sharp^i(P_2)
\end{array}$$

This unfolding procedure is intended to duplicate statically the occurrences that will be duplicated by redexes occurring at depth i. For example, in the following reductions occurring at depth 0:

$$P = \mathsf{let}\ !x = !M\ \mathsf{in}\ (\mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy) \parallel \mathsf{let}\ !y = !x\ \mathsf{in}\ \S(yy))$$
$$\overset{0}{\longrightarrow^*}\ \S(MM) \parallel \S(MM)$$

the well-formed program P duplicates the occurrence M four times. We observe that the unfolding at depth 0 of P reflects this duplication:

$$\sharp^0(P) = \mathsf{let}\ !x = !M!M!M!M\ \mathsf{in}$$
$$(\mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy) \parallel \mathsf{let}\ !y = !x!x\ \mathsf{in}\ \S(yy))$$

Unfolded programs are not intended to be reduced. However, the size of an unfolded program can be used as a non increasing measure in the following way.

Lemma 3. *Let P be a well-formed program such that $P \overset{i}{\longrightarrow} P'$. Then $|\sharp^i(P')| \leq |\sharp^i(P)|$.*

Proof. First, we assume the occurrences labelled with '$\parallel$' and '$r \Leftarrow$' do not count in the size of a program and that '$\mathsf{set}(r)$' counts for two occurrences, such that the size strictly decreases by the rule (set). Then, it is clear that $(!)$ is the only reduction rule that can make the size of a program increase, so let

$$P = F[\mathsf{let}\ !x = !N\ \mathsf{in}\ M] \overset{i}{\longrightarrow} P' = F[M[N/x]]$$

We have

$$\sharp^i(P) = F'[\mathsf{let}\ !x = \underbrace{!N!N\ldots!N}_{n\ \text{times}}\ \mathsf{in}\ \sharp^0(M)]$$
$$\sharp^i(P') = F'[\sharp^0(M[N/x])]$$

for some context F' and $n = \mathsf{FO}(x, \sharp^0(M))$. Therefore we are left to show

$$|\sharp^0(M[N/x])| \leq |\mathsf{let}\ !x = \underbrace{!N!N\ldots!N}_{n\ \text{times}}\ \mathsf{in}\ \sharp^0(M)|$$

which is clear since N must occur n times in $\sharp^0(M[N/x])$. $\square$

We observe in the following lemma that the size of an unfolded program bounds quadratically the size of the original program.

Lemma 4. *If P is well-formed, then for any depth $i \leq d(P)$:*

1. $\mathsf{FO}(\sharp^i(P)) \leq |P|$,
2. $|\sharp^i(P)| \leq |P| \cdot (|P| - 1)$,

Proof. By induction on P and i. $\square$

We can then bound the size of a program after reduction.

Lemma 5 (Squaring). *Let P be a well-formed program such that $P \overset{i}{\longrightarrow^*} P'$. Then:*

1. $|P'| \leq |P| \cdot (|P| - 1)$
2. *the length of the sequence is bounded by $|P|$*

Proof.

1. By Lemma 3 it is clear that $|\sharp^i(P')| \leq |\sharp^i(P)|$. Then by Lemma 4-2 we obtain $|\sharp^i(P')| \leq |P| \cdot (|P| - 1)$. Finally it is clear that $|P'| \leq |\sharp^i(P')|$ thus $|P'| \leq |P| \cdot (|P| - 1)$.
2. It suffices to remark $|P'|_i < |P|_i \leq |P|$. $\square$

Finally we obtain the following theorem for a shallow-first strategy using any evaluation context.

Theorem 1 (Polynomial bounds). *Let P be a well-formed program such that $d(P) = d$ and $P \longrightarrow^* P'$ is shallow-first. Then:*

1. $|P'| \leq |P|^{2^d}$
2. *the length of the reduction sequence is bounded by $|P|^{2^d}$*

Proof. The reduction $P \longrightarrow^* P'$ can be decomposed as $P = P_0 \overset{0}{\longrightarrow^*} P_1 \overset{1}{\longrightarrow^*} \ldots \overset{d-1}{\longrightarrow^*} P_d \overset{d}{\longrightarrow^*} P_{d+1} = P'$. To prove (1), we observe that by iterating Lemma 5-1 we obtain $|P_d| \leq |P_0|^{2^d}$. Moreover it is clear that $|P_{d+1}| \leq |P_d|$. Hence $|P'| \leq |P|^{2^d}$. To prove (2), we first prove by induction on d that $|P_0| + |P_1| + \ldots + |P_d| \leq |P_0|^{2^d}$. By Lemma 5-2, it is clear that the length of the reduction $P \longrightarrow^* P'$ is bounded by $|P_0| + |P_1| + \ldots + |P_d|$, which is in turn bounded by $|P_0|^{2^d}$. $\square$

It is worth noticing that the first bound takes the size of all the threads into account and that the second bound is valid for any thread interleaving.

Corollary 2 (Call-by-value is polynomial)**.** *The call-by-value evaluation of a well-formed program P of size n and depth d can be computed in time $O(n^{2^d})$.*

Proof. Let $P \longrightarrow_{\mathsf{v}}^* P'$ be the call-by-value reduction sequence of the well-formed program P. By Corollary 1 we can reorder the sequence into a shallow-first sequence $P \longrightarrow_{\mathsf{ob}}^* P'$ of the same length. By Theorem 1 we know that its length is bounded by $|P|^{2^d}$ and that $|P'| \leq |P|^{2^d}$. $\square$

6. A polynomial type system

The depth system entails termination in polynomial time but does *not* guarantee that programs 'do not go wrong'. In particular, the well-formed program in (3.6) get stuck on a non-value. In this section, we propose a solution to this problem by introducing a polynomial type system as a simple decoration of the polynomial depth system with linear types. Then, we derive a progress proposition which guarantees that well-typed programs cannot deadlock (except when trying to read an empty region).

We define the syntax of types and contexts in Figure 10. Types

-type variables	$t, t', \ldots$		
-types	α	$::=$	$\mathbf{B} \mid A$
-res. types	A	$::=$	$t \mid \mathbf{1} \mid A \multimap \alpha \mid \dagger A \mid \forall t.A \mid \mathsf{Reg}_r A$
-var. contexts	Γ	$::=$	$x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)$
-reg. contexts	R	$::=$	$r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)$

Figure 10. Syntax of types, effects and contexts

are denoted with $\alpha, \alpha', \ldots$. Note that we distinguish a special *behaviour* type $\mathbf{B}$ which is given to the entities of the language which are not supposed to return a result (such as a store or several terms in parallel) while types of entities that may return a result are denoted with A. Among the types A, we distinguish type variables $t, t', \ldots$, a terminal type $\mathbf{1}$, a linear functional type $A \multimap \alpha$, the type $!A$ of terms of type A that may be duplicated, the type $\S A$ of terms of type A that may have been duplicated, the type $\forall t.A$ of polymorphic terms and the type $\mathsf{Reg}_r A$ of regions r containing terms of type A. Hereby types may depend on regions.

In contexts, usages play the same role as in the depth system. Writing $x : (u, A)$ means that the variable x ranges on terms of type A and can be bound according to u. Writing $r : (\delta, A)$ means that the region r contain terms of type A and that $\mathsf{get}(r)$ and $\mathsf{set}(r, M)$ may only occur at depth δ. The typing system will additionally guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains a hypothesis $r : (\delta, A)$.

Because types depend on regions, we have to be careful in stating in Figure 11 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 constants 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 $B = A_i$.

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

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

$$\frac{\begin{array}{c}\forall r : (\delta, A) \in R \\ R \downarrow A\end{array}}{R \vdash} \qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha} \qquad \frac{\begin{array}{c}\forall x : (\delta, A) \in \Gamma \\ R \vdash A\end{array}}{R \vdash \Gamma}$$

Figure 11. Types and contexts

Example 2. *One may verify that the judgment $r : (\delta, \mathbf{1} \multimap \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ can be derived while judgements $r : (\delta, \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$ and $r : (\delta, \mathsf{Reg}_r \mathbf{1}) \vdash \mathbf{1}$ cannot.*

We notice the following substitution property on types.

Proposition 3. *If $R \vdash \forall t.A$ and $R \vdash B$ then $R \vdash A[B/t]$.*

A typing judgement takes the form: $R; \Gamma \vdash^\delta P : \alpha$. It attributes a type α to the program P occurring at depth δ, according to region context R and variable context Γ. Figure 12 introduces the polynomial type system. We comment on some of the rules. A λ-abstraction may only take a term of result-type as argument, *i.e.* two threads in parallel are not considered an argument. The typing of $\dagger$-terms is limited to result-types for we may not duplicate several threads in parallel. There exists two rules for typing parallel programs. The one on the left indicates that a program P_2 in parallel with a store or a thread producing a terminal value should have the type of P_2 since we might be interested in its result (note that we omit the symmetric rule for the program $(P_2 \parallel P_1)$). The one on the right indicates that two programs in parallel cannot reduce to a single result.

Example 3. *The program of Figure 3 is well-typed according to the following derivable judgement:*

$$R; - \vdash^\delta \mathsf{let}\ !x = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, (!x)(\S x)) \parallel r \Leftarrow !(\lambda x.x\star) : \mathbf{1}$$

where $R = r : (\delta, \forall t.!((\mathbf{1} \multimap t) \multimap t))$. Whereas the program in (3.6) *is not.*

Remark 1. *We can easily see that a well-typed program is also well-formed.*

The polynomial type system enjoys the subject reduction property for the largest relation $\longrightarrow \supseteq \longrightarrow_{\mathsf{ob}} \supseteq \longrightarrow_{\mathsf{v}}$.

Lemma 6 (Substitution)**.**

1. *If $R; \Gamma, x : (\lambda, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta N : A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*
2. *If $R; \Gamma, x : (\S, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta \S N : \S A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*
3. *If $R; \Gamma, x : (!, A) \vdash^\delta M : B$ and $R; \Gamma \vdash^\delta !N : !A$ then $R; \Gamma \vdash^\delta M[N/x] : B$.*

Proposition 4 (Subject Reduction)**.** *If $R; \Gamma \vdash^\delta P : \alpha$ and $P \longrightarrow P'$ then $R; \Gamma \vdash^\delta P' : \alpha$.*

$$\frac{R \vdash \Gamma \quad x : (\lambda, A) \in \Gamma}{R; \Gamma \vdash^{\delta} x : A} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} \star : \mathbf{1}} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash^{\delta} r : \mathsf{Reg}_r A} \qquad \frac{\begin{array}{c}\mathsf{FO}(x, M) = 1\\ R; \Gamma, x : (\lambda, A) \vdash^{\delta} M : \alpha\end{array}}{R : \Gamma \vdash^{\delta} \lambda x.M : A \multimap \alpha}$$

$$\frac{\begin{array}{c}R; \Gamma \vdash^{\delta} M : A \multimap \alpha\\ R; \Gamma \vdash^{\delta} N : A\end{array}}{R; \Gamma \vdash^{\delta} MN : \alpha} \qquad \frac{\begin{array}{c}\mathsf{FO}(M) \leq 1\\ R; \Gamma_{\lambda} \vdash^{\delta+1} M : A\end{array}}{R; \Gamma_{!}, \Delta_{\S}, \Psi_{\lambda} \vdash^{\delta} !M : !A} \qquad \frac{\begin{array}{c}R; \Gamma \vdash^{\delta} M : !A \quad \mathsf{FO}(x, N) \geq 1\\ R; \Gamma, x : (!, A) \vdash^{\delta} N : \alpha\end{array}}{R; \Gamma \vdash^{\delta} \mathsf{let}\ !x = M\ \mathsf{in}\ N : \alpha} \qquad \frac{R; \Gamma_{\lambda}, \Delta_{\lambda} \vdash^{\delta+1} M : A}{R; \Gamma_{\S}, \Delta_{!}, \Psi_{\lambda} \vdash^{\delta} \S M : \S A}$$

$$\frac{\begin{array}{c}R; \Gamma \vdash^{\delta} M : \S A \quad \mathsf{FO}(x, N) = 1\\ R; \Gamma, x : (\S, A) \vdash^{\delta} N : \alpha\end{array}}{R; \Gamma \vdash^{\delta} \mathsf{let}\ \S x = M\ \mathsf{in}\ N : \alpha} \qquad \frac{\begin{array}{c}t \notin (R; \Gamma)\\ R; \Gamma \vdash^{\delta} M : A\end{array}}{R; \Gamma \vdash^{\delta} M : \forall t.A} \qquad \frac{R; \Gamma \vdash^{\delta} M : \forall t.A \quad R \vdash B}{R; \Gamma \vdash^{\delta} M : A[B/t]} \qquad \frac{R \vdash \Gamma \quad r : (\delta, A) \in R}{R; \Gamma \vdash^{\delta} \mathsf{get}(r) : A}$$

$$\frac{\begin{array}{c}r : (\delta, A)\\ R; \Gamma \vdash^{\delta} M : A\end{array}}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, M) : \mathbf{1}} \qquad \frac{\begin{array}{c}r : (\delta, A)\\ R; \Gamma \vdash^{\delta} M : A\end{array}}{R; \Gamma \vdash^{0} r \Leftarrow M : \mathbf{B}} \qquad \frac{\begin{array}{c}R; \Gamma \vdash^{\delta} P_1 : \mathbf{1} \text{ or } P_1 = S\\ R; \Gamma \vdash^{\delta} P_2 : \alpha\end{array}}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2) : \alpha} \qquad \frac{R; \Gamma \vdash^{\delta} P_i : \alpha_i}{R; \Gamma \vdash^{\delta} (P_1 \parallel P_2) : \mathbf{B}}$$

Figure 12. A polynomial type system

Finally, we establish a progress proposition which shows that any well-typed call-by-value program (*i.e.* defined from Figure 9) reduces to several threads in parallel which are values or deadlocking reads.

Proposition 5 (Progress)**.** *Suppose P is a closed typable call-by-value program which cannot reduce. Then P is structurally equivalent to a program*

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

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

7. Expressivity

We now illustrate the expressivity of the polynomial type system. First we show that our system is complete in the extensional sense: every polynomial time function can be represented (Subsection 7.1). Then we introduce a language with memory locations representing higher-order references for which the type system can be easily adapted (Subsection 7.2). Building on this language, we give an example of polynomial programming (Subsection 7.3).

As a first step, we define some Church-like encodings in Figure 13 where we abbreviate $\lambda x.\mathsf{let}\ \ \dagger x = x\ \mathsf{in}\ M$ by $\lambda^{\dagger} x.M$. We have natural numbers of type Nat, binary natural number of type BNat and lists of type $\mathsf{List}\ A$ that contain values of type A.

7.1 Polynomial completeness

The representation of polynomial functions relies on the representation of binary words. The precise notion of representation is spelled out in the following definitions.

Definition 6. *(Binary word representation) Let $- \vdash^{\delta} M : \S^{p}\mathsf{BNat}$ for some $\delta, p \in \mathbb{N}$. We say M* represents *$w \in \{0,1\}^*$, written $M \Vdash w$, if $M \longrightarrow^* \S^{p}\overline{w}$.*

Definition 7. *(Function representation) Let $- \vdash^{\delta} F : \mathsf{BNat} \multimap \S^{d}\mathsf{BNat}$ where $\delta, d \in \mathbb{N}$ and $f : \{0,1\}^* \to \{0,1\}^*$. We say F* represents *f, written $F \Vdash f$, if for any M and $w \in \{0,1\}^*$ such that $- \vdash^{\delta} M : \mathsf{BNat}$ and $M \Vdash w$, $FM \Vdash f(w)$.*

The following theorem is a restatement of Girard [12] and Asperti [4].

$$\begin{array}{rcl}
\mathsf{Nat} & = & \forall t.!(t \multimap t) \multimap \S(t \multimap t)\\
\overline{n} & : & \mathsf{Nat}\\
\overline{n} & = & \lambda^{!} f.\S(\lambda x.\underbrace{f(\ldots(f}_{n \text{ times}} x)))\\
\\
\mathsf{add} & : & \mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}\\
\mathsf{add} & = & \lambda m.\lambda n.\lambda^{!} f.\mathsf{let}\ \S y = m!f\ \mathsf{in}\\
 & & \mathsf{let}\ \S z = n!f\ \mathsf{in}\ \S(\lambda x.y(zx))\\
\\
\mathsf{BNat} & = & \forall t.!(t \multimap t) \multimap !(t \multimap t) \multimap \S(t \multimap t)\\
\text{for } w = i_0 \ldots i_n \in \{0,1\}^* & & \\
\overline{w} & : & \mathsf{BNat}\\
\overline{w} & = & \lambda^{!} x_0.\lambda x_1^{!}.\S(\lambda z.x_{i_0}(\ldots(x_{i_n} z)))\\
\\
\mathsf{List}\ A & = & \forall t.!(A \multimap t \multimap t) \multimap \S(t \multimap t)\\
[u_1, \ldots, u_n] & : & \mathsf{List}\ A\\
[u_1, \ldots, u_n] & = & \lambda f^{!}.\S(\lambda x.fu_1(fu_2 \ldots (fu_n x)))\\
\\
\mathsf{list\_it} & : & \forall u.\forall t.!(u \multimap t \multimap t) \multimap \mathsf{List}\ u \multimap \S t \multimap \S t\\
\mathsf{list\_it} & = & \lambda f.\lambda l.\lambda^{\S} x.\mathsf{let}\ \S y = lf\ \mathsf{in}\ \S(yx)
\end{array}$$

Figure 13. Church encodings

Theorem 2 (Polynomial completeness)**.**
Every function $f : \{0,1\}^* \to \{0,1\}^*$ which can be computed by a Turing machine in time bounded by a polynomial of degree d can be represented by a term of type $\mathsf{BNat} \multimap \S^{d}\mathsf{BNat}$.

7.2 A language with higher-order references

Next, we give an application of the language with abstract regions by presenting a connection with a language with dynamic memory locations representing higher-order references.

The differences with the region-based system are presented in Figure 14. We introduce terms of the form $\nu x.M$ to generate a fresh memory location x whose scope is M. Contexts are call-by-value and allow evaluation under ν binders. The structural rule (ν) is for scope extrusion. Region constants have been removed from the syntax of terms hence reduction rules (get_{ν}) and (set_{ν}) relate to memory locations. The operational semantics of references is adopted: when assigning a value to a memory location, the previous value is *overwritten*, and when reading a memory location, the

$$\begin{array}{rcl} M & ::= & \ldots \mid \nu x.M \\ F_\nu & ::= & F_{\mathsf{v}} \mid \nu x.F_\nu \end{array}$$

$$(\nu) \qquad F_\nu[\nu x.M] \quad \equiv \quad \nu x.F_\nu[M] \quad \text{if } x \notin \mathsf{FV}(F_\nu)$$

$$\begin{array}{llcl} (\mathsf{get}_\nu) & F_\nu[\mathsf{get}(x)] \parallel x \Leftarrow V & \longrightarrow_\nu & F_\nu[V] \parallel x \Leftarrow V \\ (\mathsf{set}_\nu) & F_\nu[\mathsf{set}(x,V)] \parallel x \Leftarrow V' & \longrightarrow_\nu & F_\nu[\star] \parallel x \Leftarrow V \end{array}$$

$$\frac{R;\Gamma, x : (u, \mathsf{Reg}_r !A) \vdash^\delta M : B}{R;\Gamma \vdash^\delta \nu x.M : B} \qquad \frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r !A}{R;\Gamma \vdash^\delta \mathsf{get}(x) : !A}$$

$$\frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r !A \quad R;\Gamma \vdash^\delta M : !A}{R;\Gamma \vdash^\delta \mathsf{set}(x, M) : \mathbf{1}} \qquad \frac{R(r) = (\delta, !A) \quad R;\Gamma \vdash^\delta x : \mathsf{Reg}_r !A \quad R;\Gamma \vdash^\delta V : !A}{R;\Gamma \vdash^0 x \Leftarrow V : \mathbf{B}}$$

Figure 14. A call-by-value system with references

value is *copied* from the store. We see in the typing rules that region constants still appear in region types and that a memory location must be a free variable that relates to an abstract region r by having the type $\mathsf{Reg}_r A$.

There is a simple translation from the language with memory locations to the language with regions. It consists in replacing the (free or bound) variables with a region type of the shape $\mathsf{Reg}_r A$ by the constant r. We then observe that read access and assignments to references are mapped to several reduction steps in the system with regions. It requires the following observation: in the typing rules, memory locations only relate to regions with duplicable content of type $!A$. This allows us to simulate the *copy from memory* mechanism of references by decomposing it into a *consume* and *duplicate* mechanism in the language with regions. More precisely: an occurrence of $\mathsf{get}(x)$ where x relates to region r is translated into

$$\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !y) \parallel !y$$

such that

$$F_{\mathsf{v}}[\mathsf{let}\ !y = \mathsf{get}(r)\ \mathsf{in}\ \mathsf{set}(r, !y) \parallel !y] \parallel r \Leftarrow !V$$
$$\longrightarrow_{\mathsf{v}}^{+} F[!V] \parallel r \Leftarrow !V$$

simulates the reduction (get_ν). Also, it is easy to see that a reduction step (set_ν) can be simulated by exactly one reduction step $(\mathsf{set}_{\mathsf{v}})$. Since typing is preserved by translation, we conclude that any time complexity bound can be lifted to the language with references.

Note that this also works if we adopt the operational semantics of communication channels; in that case, memory locations can also relate to regions containing non-duplicable content since reading a channel means *consuming* the value.

7.3 Polynomial programming

Using higher-order references, we show that it is possible to program the iteration of operations producing a side effect on an inductive data structure, possibly in parallel.

Here is the function update taking as argument a memory location x related to region r and incrementing the numeral stored at that location:

$$r : (3, !\mathsf{Nat}); - \vdash^2 \mathsf{update} : !\mathsf{Reg}_r !\mathsf{Nat} \multimap \S\mathbf{1} \multimap \S\mathbf{1}$$
$$\mathsf{update} = \lambda^! x.\lambda^\S z.\S(\mathsf{set}(x, \mathsf{let}\ !y = \mathsf{get}(x)\ \mathsf{in}\ !(\mathsf{add}\ \overline{2}\ y)) \parallel z)$$

The second argument z is to be garbage collected. Then we define the program run that iterates the function update over a list $[!x, !y, !z]$ of 3 memory locations:

$$r : (3, !\mathsf{Nat}) \vdash^1 \mathsf{run} : \S\S\mathbf{1}$$
$$\mathsf{run} = \mathsf{list\_it}\ !\mathsf{update}\ [!x, !y, !z]\ \S\S\star$$

All addresses have type $!\mathsf{Reg}_r !\mathsf{Nat}$ and thus relate to the same region r. Finally, the program run in parallel with some store assignments reduces as expected:

$$\mathsf{run} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p}$$
$$\longrightarrow_\nu^* \S\S\star \parallel x \Leftarrow !\overline{2+m} \parallel y \Leftarrow !\overline{2+n} \parallel z \Leftarrow !\overline{2+p}$$

Note that due to the Church-style encoding of numbers and lists, we assume that the relation $\longrightarrow_\nu$ may reduce under binders when required.

Building on this example, suppose we want to write a program of three threads where each thread concurrently increments the numerals pointed by the memory locations of the list. Here is the function gen_threads taking a functional f and a value x as arguments and generating three threads where x is applied to f:

$$r : (3, !\mathsf{Nat}) \vdash^0 \mathsf{gen\_threads} : \forall t.\forall t'.!(t \multimap t') \multimap !t \multimap \mathbf{B}$$
$$\mathsf{gen\_threads} = \lambda^! f.\lambda^! x.\S(fx) \parallel \S(fx) \parallel \S(fx)$$

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

$$r : (3, !\mathsf{Nat}) \vdash^1 \mathsf{F} : \mathsf{List}\ !\mathsf{Reg}_r !\mathsf{Nat} \multimap \S\S\mathbf{1}$$
$$\mathsf{F} = \lambda l.\mathsf{list\_it}\ !\mathsf{update}\ l\ \S\S\star$$

Finally the concurrent iteration is defined in run_threads:

$$r : (3, !\mathsf{Nat}) \vdash^0 \mathsf{run\_threads} : \mathbf{B}$$
$$\mathsf{run\_threads} = \mathsf{gen\_threads}\ !\mathsf{F}\ ![!x, !y, !z]$$

The program is well-typed for side effects occurring at depth 3 and it reduces as follows:

$$\mathsf{run\_threads} \parallel x \Leftarrow !\overline{m} \parallel y \Leftarrow !\overline{n} \parallel z \Leftarrow !\overline{p}$$
$$\longrightarrow_\nu^* \S\S\S\star \parallel x \Leftarrow !\overline{6+m} \parallel y \Leftarrow !\overline{6+n} \parallel z \Leftarrow !\overline{6+p}$$

Note that different thread interleavings are possible but in this particular case they are confluent.

8. Conclusion and Related work

We have proposed a type system for a higher-order functional language with multithreading and side effects that guarantees termination in polynomial time, covering any scheduling of threads and taking account of thread generation. To the best of our knowledge, there appears to be no other characterization of polynomial time in such a language. The polynomial soundness of the call-by-value strategy relies on the simulation of call-by-value by a shallow-first strategy which is proved to be polynomial. The proof is a significant adaptation of Terui's methodology [18]: it is greatly simplified by a strict linearity condition and based on a clever analysis of the evaluation order of side effects which is shown to be preserved.

Related work The framework of light logics has been previously applied to a higher-order π-calculus [14] and a functional language with pattern-matching and recursive definitions [6]. The notion of *stratified region*[1] has been proposed [1, 7] to ensure the termination of a higher-order multithreaded language with side effects . In the setting of *synchronous* computing, static analyses have been developed to bound resource consumption in a synchronous π-calculus [2] and a multithreaded first-order language [3]. Recently, the framework of *complexity information flow* have been applied to characterize polynomial multithreaded imperative programs [17].

[1] Here we speak of stratification by means of a type-and-effect discipline, this is not to be confused with the notion of stratification *by depth level* that is used in the present paper.

Acknowledgments The author wishes to thank Roberto Amadio for his precious help on the elaboration of this work and Patrick Baillot for his careful reading of the paper.

References

[1] R. M. Amadio. On stratified regions. In Z. Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2009. ISBN 978-3-642-10671-2. 2, 8

[2] R. M. Amadio and F. Dabrowski. Feasible reactivity in a synchronous pi-calculus. In M. Leuschel and A. Podelski, editors, *PPDP*, pages 221–230. ACM, 2007. ISBN 978-1-59593-769-8. 8

[3] R. M. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. *Theoretical Computer Science*, 358(2-3):229–254, 2006. 8

[4] A. Asperti. Light affine logic. In *LICS*, pages 300–308. IEEE Computer Society, 1998. ISBN 0-8186-8506-9. 1, 7.1

[5] P. Baillot and V. Mogbil. Soft lambda-calculus: A language for polynomial time computation. In I. Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2004. ISBN 3-540-21298-1. 1

[6] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2010. ISBN 978-3-642-11956-9. 8

[7] G. Boudol. Typing termination in a higher-order concurrent imperative language. *Information and Computation*, 208(6): 716–736, 2010. 8

[8] P. Coppola and S. Martini. Optimizing optimal reduction: A type inference algorithm for elementary affine logic. *ACM Transaction on Computational Logic*, 7:219–260, April 2006. ISSN 1529-3785. 1

[9] 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. 1

[10] V. Danos and J.-B. Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123 – 137, 2003. ISSN 0890-5401. 1

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50: 1–102, 1987. 1

[12] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998. 1, 7.1

[13] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004. 1

[14] U. D. Lago, S. Martini, and D. Sangiorgi. Light logics and higher-order processes. In S. B. Fröschle and F. D. Valencia, editors, *EXPRESS*, volume 41 of *EPTCS*, pages 46–60, 2010. 8

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM, 1988. ISBN 0-89791-252-7. 1

[16] A. Madet and R. M. Amadio. An elementary affine λ-calculus with multithreading and side effects. In C.-H. L. Ong, editor, *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011. ISBN 978-3-642-21690-9. 1, 1, 3.2

[17] J.-Y. Marion and R. Péchoux. Complexity information flow in a multi-threaded imperative language. *CoRR*, abs/1203.6878, 2012. 8

[18] K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3-4):253–280, 2007. 1, 1, 3.1, 5, 8

Indexed Realizability for Bounded-Time Programming with References and Type Fixpoints*

Aloïs Brunel[1] and Antoine Madet[2]

[1] Laboratoire d'Informatique de Paris-Nord, Université Paris 13
[2] Univ Paris Diderot, Sorbonne Paris Cité,
PPS, UMR 7126, CNRS, F-75205 Paris, France

Abstract. The field of implicit complexity has recently produced several bounded-complexity programming languages. This kind of language allows to implement exactly the functions belonging to a certain complexity class. We present a realizability semantics for a higher-order functional language based on a fragment of linear logic called **LAL** which characterizes the complexity class **PTIME**. This language features recursive types and higher-order store. Our realizability is based on biorthogonality, indexing and is quantitative. This last feature enables us not only to derive a semantical proof of termination, but also to give bounds on the number of computational steps of typed programs.

1 Introduction

Implicit Computational Complexity — This research field aims at providing machine-independent characterizations of complexity classes (such as polynomial time or logspace functions). One approach is to use type systems based on linear logic to control the complexity of higher-order functional programs. In particular, the so-called light logics (*e.g.* **LLL** [7], **SLL** [10]) have led to various type systems for the λ-calculus guaranteeing that a well-typed term has a bounded complexity [3]. These logics introduce the modalities '!' (read *bang*) and '§' (read *paragraph*). By a fine control of the nesting of these modalities, which is called the *depth*, the duplication of data can be made explicit and the complexity of programs can be tamed. This framework has been recently extended to a higher-order process calculus [6] and a functional language with recursive definitions [19]. Also, Amadio and Madet have proposed [15] a multi-threaded λ-calculus with higher-order store that enjoys an elementary time termination.

Quantitative Realizability — Starting from Kleene, the concept of realizability has been introduced in different forms and has been shown very useful to

* Work partially supported by 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).

build models of computational systems. In a series of works [13,12], Dal Lago and Hofmann have shown how to extend Kleene realizability with quantitative informations in order to interpret subsystems of linear logic with restricted complexity. The idea behind Dal Lago and Hofmann's work is to consider bounded-time programs as realizers, where bounds are represented by elements of a *resource monoid.* In [5] the first author has shown how this quantitative extension fits well in a biorthogonality based framework (namely Krivine's classical realizability [9]) and how it relates to the notion of forcing.

Step-Indexing — In order to give a semantical account of features like recursive or reference types, one has to face troublesome circularity issues. To solve this problem, Appel and McAllester [2] have proposed step-indexed models. The idea is to define the interpretation of a type as a predicate on terms indexed by numbers. Informally, a term t belongs to the interpretation of a type τ with the index $k \in \mathbb{N}$ if when t is executed for k steps, it satisfies the predicate associated to τ. Then, it is possible to define by induction on the index k the interpretation of recursive or reference types. Step-indexing has been related to Gödel-Löb logic and the *later* operator $\triangleright$ [17].

Contributions — In this paper, we present a typed λ-calculus called $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ whose functional core is based on the light logic **LAL** [3]. We extend it with recursive types and higher-order store. Even in presence of these features, every program typable in $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ terminates in polynomial time. To prove termination in bounded-time, we propose a new quantitative realizability semantics with the following features:

- It is biorthogonality based, which permits a simple presentation and allows the possibility to interpret control operators (though it is only discussed informally in the conclusion of this paper).
- It is indexed, which permits to interpret higher-order store and recursive types. The particularity is that our model is indexed by depths (the nesting of modalities) instead of computational steps (like in step-indexing).

To our knowledge, this is the first semantics presenting at the same time quantitative, indexed and biorthogonality features.

Outline — Section 2 introduces the language $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ and its type system. In Section 3, we introduce the indexed quantitative realizability. It is then used to obtain a semantic model for $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$, which in turn implies termination in polynomial time of typed programs. Finally, we mention related works in Section 4 and in Section 5 we discuss future research directions and conclude.

2 The Language

This section presents the language $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ and its type system. Before going into details, we give some intuitions on the modalities and explain how we deal with side-effects with the notion of *region.*

On Modalities — The functional core of the language is an affine λ-calculus which means that λ-abstractions can use their argument at most once. To provide some duplication power, we introduce two modal constructors '!' and '§' that originate from **LAL**. Values of the shape $!V$ and $\S V$ can then be destructed against specific $\mathsf{let}\,!$ and $\mathsf{let}\,\S$-expressions.

On Regions — Following a standard practice in effect systems [14], the state of a program is abstracted into a finite set of regions where each region may represents several dynamically allocated values. Then, side-effects are produced by read and write operations on these regions. As noted by Amadio [1], the reduction rules of an abstract language with regions can be formalized such that they simulate the reduction rules of a concrete language with dynamic allocation. Working at the abstract level of regions allows to encompass several interaction mechanisms like references and channels (for the latter, the language should dispose of concurrency constructs). Moreover, termination in polynomial time of the language with regions entails termination in polynomial time of the language with dynamic allocation since the simulation preserves the number of reduction steps. Finally, we find it easier to give a semantic model of a type system with regions instead of dynamic locations.

How these modalities relate to polynomial time and how regions simulate dynamic locations is further explained in Section 2.2.

2.1 Syntax and Operational Semantics

The syntax of the language is the following:

$$\begin{array}{ll} \textit{Values} & V ::= x \mid \lambda x.M \mid r \mid () \mid !V \mid \S V \\ \textit{Terms} & M ::= V \mid M_1 M_2 \mid !M \mid \S M \mid \mathsf{get}(r) \mid \mathsf{set}(r, V) \\ & \qquad\quad \mathsf{let}\ !x = V \ \mathsf{in}\ M \mid \mathsf{let}\ \S x = V \ \mathsf{in}\ M \end{array}$$

We suppose having a countable set of variables denoted $x, y, \ldots$ and of regions denoted by the letters $r, r', \ldots$. The terminal value *unit* is denoted by (). Modal terms and modal values are built with the unary constructors ! and § and are destructed by the respective $\mathsf{let}\,!$ and $\mathsf{let}\,\S$-expressions. The terms $\mathsf{get}(r)$ and $\mathsf{set}(r, V)$ are respectively used to read a value from a region r and to assign a value V to a region r. We denote by $M[N/x]$ the term M in which each free occurrence of x has been substituted by N.

The operational semantics of the language is presented in the form of an abstract machine. We first define the configurations of the abstract machine:

$$\begin{array}{ll} \textit{Environments} & E ::= \diamond \mid V \cdot E \mid M \odot E \mid ! \cdot E \mid \S \cdot E \\ \textit{Stores} & S ::= r \Leftarrow V \mid S_1 \uplus S_2 \\ \textit{Configurations} & C ::= \langle M, E, S \rangle \end{array}$$

Programs are intended to be executed with a right-to-left call-by-value strategy. Hence, an environment E is either an empty frame $\diamond$, a stack of frames to evaluate on the left of a value ($V \cdot E$), on the right of a term ($M \odot E$) or in-depth of a term ($! \cdot E$ and $\S \cdot E$). Finally, a store S is a multiset of region assignments $r \Leftarrow V$. A configuration of the abstract machine is executed according to the following rules:

$$\begin{aligned}
\langle \lambda x.M, V \cdot E, S\rangle &\longrightarrow \langle M[V/x], E, S\rangle \\
\langle MN, E, S\rangle &\longrightarrow \langle N, M \odot E, S\rangle \\
\langle V, M \odot E, S\rangle &\longrightarrow \langle M, V \cdot E, S\rangle \\
\langle \dagger M, E, S\rangle &\longrightarrow \langle M, \dagger \cdot E, S\rangle \qquad\qquad \text{if } M \text{ is not a value} \\
\langle V, \dagger \cdot E, S\rangle &\longrightarrow \langle \dagger V, E, S\rangle \\
\langle \mathsf{let}\ \dagger x = \dagger V \ \mathsf{in}\ M, E, S\rangle &\longrightarrow \langle M[V/x], E, S\rangle \\
\langle \mathsf{get}(r), E, (r \Leftarrow V) \uplus S\rangle &\longrightarrow \langle V, E, S\rangle \\
\langle \mathsf{set}(r, V), E, S\rangle &\longrightarrow \langle (), E, (r \Leftarrow V) \uplus S\rangle
\end{aligned}$$

For the sake of conciseness we wrote $\dagger$ for $\dagger \in \{!, \S\}$. Observe that $\mathsf{let}\,\dagger$-expressions destruct modal values $\dagger V$ and propagate V. Reading a region amounts to *consume* the value from the store and writing to a region amounts to *add* the value to the store. We consider programs up to α-renaming and in the sequel $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Example 1. Here is a function $F = \lambda x.\mathsf{let}\ !y = x \ \mathsf{in}\ \mathsf{set}(r_1, \S y); \mathsf{set}(r_2, \S y)$ that duplicates its argument and assign it to regions r_1 and r_2. It can be used to duplicate a value from another region r_3 as follows:

$$\langle F\mathsf{get}(r_3), \diamond, r_3 \Leftarrow !V\rangle \longrightarrow^* \langle (), \diamond, (r_1 \Leftarrow \S V) \uplus (r_2 \Leftarrow \S V)\rangle$$

Remark 1. As usual, we can encode the sequential composition $M; N$ by the application $(\lambda x.N)M$ where x does not occur free in N. Thus, the reduction rule $\langle V; M, E, S\rangle \longrightarrow \langle M, E, S\rangle$ can be assumed.

Definition 1. *We define the notation $\langle M, E, S\rangle \Downarrow^n$ as the following statement:*

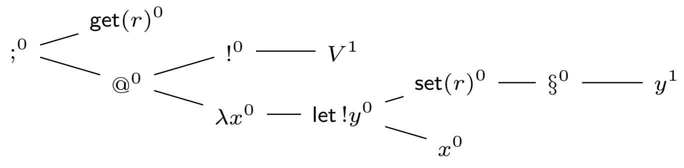
- *The evaluation of $\langle M, E, S\rangle$ in the abstract machine terminates.*
- *The number of steps needed by $\langle M, E, S\rangle$ to terminate is n.*

2.2 Type System

The light logic **LAL** relies on a *stratification* principle which is at the basis of our type system. We first give an informal explanation of this principle.

On Stratification — Each occurrence of a program can be given a depth which is the number of nested modal constructors for which the occurrence is in scope. Here is an example for the program P where each occurrence is labeled with its depth:

$$P = (\lambda x.\mathsf{let}\ !y = x \ \mathsf{in}\ \mathsf{set}(r, \S y))!V; \mathsf{get}(r)$$



The depth $d(M)$ of a term M is the maximum depth of its occurrences. The stratification principle is that the depth of every occurrence is preserved by reduction. On the functional side, it can be ensured by these two constraints: (1) if a λ-abstraction occurs at depth d, then the bound variable must occur at depth d; (2) if a $\mathsf{let}\,\dagger$-expression occurs at depth d, then the bound variable must occur at depth $d+1$. These two constraints are respected by the program P and we observe in the following reduction

$$\langle P, \diamond, \emptyset\rangle \longrightarrow^* \langle \mathsf{set}(r, \S V); \mathsf{get}(r), \diamond, \emptyset\rangle$$

that the depth of V is preserved. In order to preserve the depth of occurrences that go through the store, this third constraint is needed: (3) for each region r, $\mathsf{get}(r)$ and $\mathsf{set}(r)$ must occur at a fixed depth d_r. We observe that this is the case of program P where $d_r = 0$. Consequently, the reduction terminates as follows

$$\langle \mathsf{set}(r, \S V); \mathsf{get}(r), \diamond, \emptyset\rangle \longrightarrow^* \langle \mathsf{get}(r), \diamond, r \Leftarrow \S V\rangle \longrightarrow^* \langle \S V, \diamond, \emptyset\rangle$$

where the depth of V is still preserved. Stratification on the functional side has been deeply studied by Terui with the Light Affine λ-calculus [20] and extended to regions by Amadio and the second author [15].

Types and Contexts — The syntax of types and contexts is the following:

$$\begin{array}{lr}
\textit{Types} & A, B, C ::= X \mid \mathsf{Unit} \mid A \multimap B \mid\ !A \mid \S A \mid \mu X.A \mid \mathsf{Reg}_r A \\
\textit{Variable contexts} & \Gamma, \Delta ::= x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n) \\
\textit{Region contexts} & R ::= r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n)
\end{array}$$

We have a countable set of type variables $X, X', \ldots$ We distinguish the terminal type Unit, the affine functional type $A \multimap B$, the type $!A$ of terms which reduce on a duplicable value, the type $\S A$ of terms containing values that may have been duplicated, recursive types $\mu X.A$ and the type $\mathsf{Reg}_r A$ of terms which reduce to region r that contains values of type A. Hereby types may depend on regions. Following [15], a region context associates a natural number δ_i to each region r_i of a finite set of regions $\{r_1, \ldots, r_n\}$ that we write $dom(R)$. Writing $r : (\delta, A)$ means that the region r contains values of type A and that gets and sets on r may only happen at a fixed depth depending on δ. A variable context associates each variable with an usage $u \in \{\lambda, \S, !\}$ which constraints the variable to be bound by a λ-abstraction, a $\mathsf{let}\,\S$-expression or a $\mathsf{let}\,!$-expression respectively. In the sequel we write Γ_u for $x_1 : (u, A_1), \ldots, x_n : (u, A_n)$. Writing $x : (u, A)$ means that the variable x ranges on values of type A and can be bound according to u.

Types depend on region names. Therefore, we have to be careful in stating when a type A is well-formed with respect to a region context R, written $R \vdash A$. Informally, the judgment $r_1 : (\delta_1, A_1), \ldots, r_n : (\delta_n, A_n) \vdash B$ is well formed provided that: (1) all the region names occurring in the types $A_1, \ldots, A_n, B$ 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 $A_1, \ldots, A_n, B$ are such that $B = A_i$. The judgment $R \vdash \Gamma$

is well-formed if $R \vdash A$ is well-formed for every $x : (u, A) \in \Gamma$. We invite the reader to check in [1] that these judgements can be easily defined.

Typing Rules — A typing judgment takes the form $R; \Gamma \vdash^{\delta} P : A$ and is indexed by an integer δ. The rules are given in Figure 1. They should entail the following:

- if $x : (\lambda, A) \in \Gamma$ then x occurs at most once and it must be at depth 0 in P,
- if $x : (\S, A) \in \Gamma$ then x occurs at most once and it must be at depth 1 in the scope of a § constructor in P,
- if $x : (!, A) \in \Gamma$ then x occurs arbitrarily many times and it must be at depth 1 in the scope of a † constructor in P,
- if $r : (\delta', A) \in R$ then $\mathsf{get}(r)$ and $\mathsf{set}(r)$ occur at depth $\delta - \delta'$ in P.

$$\mathsf{v}\,\dfrac{R \vdash}{R; x : (\lambda, A) \vdash^{\delta} x : A} \qquad \mathsf{u}\,\dfrac{R \vdash}{R; - \vdash^{\delta} () : \mathsf{Unit}} \qquad \mathsf{r}\,\dfrac{R \vdash \quad r : (\delta, A) \in R}{R; - \vdash^{\delta} r : \mathsf{Reg}_r A}$$

$$\mathsf{c}\,\dfrac{R; \Gamma, x : (!, A), y : (!, A) \vdash^{\delta} M : B}{R; \Gamma, z : (!, A) \vdash^{\delta} M[z/x, z/y] : B} \qquad \mathsf{w}\,\dfrac{R; \Gamma \vdash^{\delta} M : B \quad R \vdash \Gamma, x : (u, A)}{R; \Gamma, x : (u, A) \vdash^{\delta} M : B}$$

$$\mathsf{lam}\,\dfrac{R; \Gamma, x : (\lambda, A) \vdash^{\delta} M : B}{R : \Gamma \vdash^{\delta} \lambda x.M : A \multimap B} \qquad \mathsf{app}\,\dfrac{R; \Gamma \vdash^{\delta} M_1 : A \multimap B \quad R; \Delta \vdash^{\delta} M_2 : A}{R; \Gamma, \Delta \vdash^{\delta} M_1 M_2 : B}$$

$$!\text{-}\mathsf{prom}\,\dfrac{R; x : (\lambda, A) \vdash^{\delta} V : A}{R; x : (!, A) \vdash^{\delta+1} !V : !A} \qquad \S\text{-}\mathsf{prom}\,\dfrac{R; \Gamma_{\lambda}, \Delta_{\lambda} \vdash^{\delta} M : A}{R; \Gamma_{\S}, \Delta_{!} \vdash^{\delta+1} \S M : \S A}$$

$$\dagger\text{-}\mathsf{elim}\,\dfrac{\begin{array}{c} R; \Gamma \vdash^{\delta} V : \dagger A \\ R; \Delta, x : (\dagger, A) \vdash^{\delta} M : B \end{array}}{R; \Gamma, \Delta \vdash^{\delta} \mathsf{let}\ \dagger x = V\ \mathsf{in}\ M : B} \qquad \mathsf{get}\,\dfrac{R; - \vdash^{\delta} r : \mathsf{Reg}_r A}{R; - \vdash^{\delta} \mathsf{get}(r) : A}$$

$$\mathsf{set}\,\dfrac{\begin{array}{c} R; - \vdash^{\delta} r : \mathsf{Reg}_r A \\ R; \Gamma \vdash^{\delta} V : A \end{array}}{R; \Gamma \vdash^{\delta} \mathsf{set}(r, V) : \mathsf{Unit}} \qquad \mathsf{un/fold}\,\dfrac{R; \Gamma \vdash^{\delta} M : \mu X.A}{R; \Gamma \vdash^{\delta} M : A[\mu X.A/X]}$$

Fig. 1. Typing rules

Here are several crucial remarks on the rules:

- In binary rules, we implicitly require that contexts Γ and Δ are disjoint. There are explicit rules (w) and (c) for the weakening and contraction of variables and we may only contract variables with usage '!'. Therefore, λ-abstractions and $\mathsf{let}\,\S$-expressions can bind at most one occurrence of free variable while $\mathsf{let}\,!$-expressions can bind several occurrences.

- The rule $!\text{-}\mathsf{prom}$ entails that $!V$ may contain at most one occurrence of free variable. This is to rule out a term like $Z = \lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ !(\lambda z.yy)$ whose n-th application $(Z \dots (Z(Z!V)))$ would duplicate 2^n times the value $!V$. To recover some duplication power, the rule $\S\text{-}\mathsf{prom}$ allows a term of the shape $\S M$ to contain many occurrences of free variable. On the other hand, $\mathsf{let}\ \S$-expressions cannot bind many occurrences of free variable.
- It is important that the type $\S A \multimap !A$ is *not* inhabited, otherwise from a value of the shape $\S(\lambda z.yy)$ we can produce a value $!(\lambda z.yy)$ and we loose the subject reduction property for there are two occurrences of y under a bang. Also, the rule $!\text{-}\mathsf{prom}$ must only applies to values so that the program $\lambda x.\mathsf{let}\ !y = x\ \mathsf{in}\ \S\mathsf{set}(r, x); !\mathsf{get}(r)$ cannot be given type $\S A \multimap !A$.
- The depth δ of a judgment is incremented when we construct a modal term. This allows to count the number of nested modalities and to stratify regions by requiring that the depth of a region matches the depth of the judgment in the rule R.
- For space consideration the rule $\mathsf{un/fold}$ can be used upside down.

Definition 2. *We say that a program M is well-typed if a judgment $R; \Gamma \vdash^{\delta} M : A$ can be derived for some R, Γ and δ such that:*

(1) If $r : (\delta_r, B) \in R$ then $B = \S C$.
(2) For every type fixpoint $\mu X.B$ that appears in R, Γ and A, the occurrences of X in B are guarded *by (occur under) a modality $\dagger$.*
(3) Every depth index in the derivation is positive. Note that if it is not the case, we can always find $\delta' > \delta$ and R' such that it is true for $R'; \Gamma \vdash^{\delta'} M : A$.

Remark 2. The above three conditions are required to give a well-founded interpretation. The fact that region types can only be guarded by a paragraph is due to properties of the light monoid (see Lemma 2).

The following progress property can be derived as long as the program does not try to read an empty region.

Proposition 1 (Progress). *If $R; \Gamma \vdash^{\delta} M : A$ then $\langle C, \diamond, \emptyset \rangle \longrightarrow^{*} \langle V, \diamond, S \rangle$ and $R; \Gamma \vdash^{\delta} V : A$ and every assigned value in S can be typed.*

Remark 3. A program whose state is partionned into a fixed number of regions can simulate a program with a statically unknown number of dynamic allocations. In fact, there is a typed translation from the language with dynamic locations to the language with regions. Let us consider a small example where for the sake simplicity we do not care about the multiplicity of variables. Here is a program that generates two references à la ML with the same value V:

$$((\lambda f.\lambda x.fx; fx)(\lambda y.\mathsf{ref}\ y))V$$

A single region r can be used to abstract both references by assigning the type $\mathsf{Reg}_r A$ to the subterm $(\mathsf{ref}\ y)$. It then suffices to translate $(\mathsf{ref}\ y)$ into $(\mathsf{set}(r, y); r)$ and observe that the translated program reduces to the configuration

$$\langle r, \diamond, (r \Leftarrow V) \uplus (r \Leftarrow V) \rangle$$

Regions simulate references as long as the values written to regions do not overwrite the previous ones. This is the case in our abstract machine, but also, reading a region amounts to consume a value from the region while the values stored in references should be persistent. We note that it is enough to duplicate, use and rewrite the value to the store to simulate this phenomenon.

The goal of the next section is to prove the following theorem

Theorem 1 (Polynomial termination). *There exists a family of polynomials $\{P_d\}_{d\in\mathbb{N}}$ such that if M is well-typed then $\langle M, \diamond, \diamond\rangle$ terminates in at most $P_{d(M)}(size(M))$ steps.*

3 "Indexed" Quantitative Realizability

We now present a biorthogonality-based interpretation of $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$. Apart from the use of biorthogonality, this interpretation has two particularities:

- First, the realizability model is *quantitative*. A type is interpreted by a set of *weighted realizers* (that is a program together with a store and a quantity bounding its normalization time). This allows to prove complexity properties of programs.
- Secondly, the semantics is *indexed* (or *stratified*), meaning that we interpret a type by a family of sets indexed by $\mathbb{N}$. Moreover the interpretation of a type is defined by double induction, first on the *index* n, and secondly on the *size* of the type. This allows to interpret recursive types and references.

It is worth noticing that while our interpretation is similar to the so-called "step-indexed" models, the meaning of indexes is not (directly) related to the number of computation steps but to the depth of terms (and so our model could be described as a "depth-indexed" model). It is the quantitative part which is used to keep track of the number of computational steps.

3.1 The Light Monoid

The realizability model is parametrized by a *quantitative monoid*, whose elements represent an information about the amount of time needed by a program to terminate.

Definition 3. *A* quantitative monoid *is a structure* $(\mathcal{M}, +, \boldsymbol{0}, \boldsymbol{1}, \leq, \|.\|)$ *where:*

- $(\mathcal{M}, +, \boldsymbol{0}, \leq)$ *is a preordered monoid.*
- $\|.\| : \mathcal{M} \longrightarrow \mathbb{N}$ *is a function such that:*
  - *for every* $p, q \in \mathcal{M}$*, we have* $\|p\| + \|q\| \leq \|p + q\|$*.*
  - *Morever,* $\|.\|$ *is compatible with* $\leq$*.*
- $\boldsymbol{1} \in \mathcal{M}$ *is such that* $1 \leq \|\boldsymbol{1}\|$*.*

Example 2. A simple instance of a quantitative monoid is given by the set $\mathbb{N}$ of positive integers, endowed with the usual addition on integers, the elements 0 and 1, and the operation $\|.\|$ defined by $\|n\| = n$.

From now on, we will often denote a quantitative monoid by its carrier $\mathcal{M}$, and we use lower-case consonnes letters $p, q, m, v, \dots$ to denote its elements. Moreover, $\mathbf{n}$ denotes the element of $\mathcal{M}$ defined as $\underbrace{\mathbf{1} + \cdots + \mathbf{1}}_{n \text{ times}}$.

Remark 4. Here are some intuitions about the previous definition.

- The operation $+$ is used to obtain the resource consumption resulting of the interaction of two programs.
- The elements of $\mathcal{M}$ are abstract quantities, so given such an abstract quantity $p \in \mathcal{M}$, $\|p\|$ provides the *concrete* quantity associated to it.
- The inequality $\forall p, q, \|p\| + \|q\| \leq \|p + q\|$ informally represents the fact that the amount of resource consumed by the interaction of two programs is potentially more important than the total amount of resource used by the two programs alone.

Definition 4. *Given a quantitative monoid, we say that a function* $f : \mathcal{M} \longrightarrow \mathcal{M}$ *is* sensible *if whenever* $p \in \mathcal{M}$ *we have* $f(p) \leq f(p + \mathbf{1})$ *and* $\|f(p)\| \neq \|f(p + \mathbf{1})\|$. *The set of sensible functions on* $\mathcal{M}$ *is denoted by* $\mathcal{M}[.]$.

We now define the notion of *light monoid*, which will be used to describe the execution time of $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ programs.

Definition 5. *We call* light monoid *a quantitative monoid* $\mathcal{M}$ *equipped with three sensible functions* $!, \S, F : \mathcal{M} \longrightarrow \mathcal{M}$ *such that for every* $p, q \in \mathcal{M}$, *the following properties hold:*

- *There is some* p' *such that* $p \leq p'$ *and* $\S p' \leq !p$
- $\S(p + q) \leq \S p + \S q$
- *There are* p', q' *such that* $p \leq p'$ *and* $q \leq q'$, *that enjoy* $\S p' + \S q' \leq \S(p + q)$
- $!p + !p \leq !p + \mathbf{1}$
- $!(p + q) \leq F(p) + !q$

Those inequations will be needed to prove that our realizability interpretation is sound with respect to the typing rules involving the modalities ! and §. Such a light monoid exists, as witnessed by the following example and property.

Example 3. We define the structure $(\mathcal{M}_l, +, \mathbf{0}, \mathbf{1}, \leq, \|.\|)$ where

- $\mathcal{M}$ is the set of triples $(n, m, f) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$ where f is a polynomial.
- $(n, m, f) + (l, k, g) = (n + l, max(m, k), max(f, g))$.
- $\mathbf{0} = (0, 0, x \mapsto 0)$
- $\mathbf{1} = (1, 0, x \mapsto x)$
- $(n, m, f) \leq (l, k, g)$ iff $n \leq l \wedge n + m \leq l + k \wedge f \leq g$
- If $(n, m, f) \in \mathcal{M}$, $\|(n, m, f)\| = n\, f(m + n)$.

Then $\mathcal{M}_l$ is a quantitative monoid. Moreover, we can define the three following operations $!, \S, F$ on $\mathcal{M}_l$:

- $\S = (n, m, f) \mapsto (n/m, m, x \mapsto x^2 f(x^2))$
- $! = (n, m, f) \mapsto (1, n + m, x \mapsto x^3 f(x^3))$
- $F = (n, m, f) \mapsto (1 + n + m, m, x \mapsto x^3 f(x^3))$

Property 1. The three operations $!, \S$ and F endow $\mathcal{M}_l$ with a structure of light monoid.

Notice that in the monoid $\mathcal{M}_l$, the operations $!, \S$ and F make the degree of the third component of any element of $\mathcal{M}_l$ grow.

3.2 Orthogonality

The main technical tool used to define our model is orthogonality. Whereas it is usually defined as a relation between a program and an environment, in our work it is a relation between weighted programs and weighted environments. From now on, $\mathcal{M}$ denotes a light monoid.

Definition 6. – *A* weighted term *is a tuple* (M, p) *where* M *is a term and* p *an element of* $\mathcal{M}$*. The set of weighted terms is denoted by* $\Lambda_{\mathcal{M}}$*.*
– *A* weighted stack *is a pair* (E, e) *where* E *is a stack and* e *an element of* $\mathcal{M}[.]$*. The set of weighted stacks is denoted by* $\Pi_{\mathcal{M}}$*.*

We choose a *pole* $\perp\!\!\!\perp \subseteq \mathsf{Conf} \times \mathcal{M}$ as the set of bounded-time terminating weighted configurations:

$$\perp\!\!\!\perp = \{(\langle M, E, S\rangle, p) \mid \langle M, E, S\rangle \Downarrow^n \wedge n \leq \|p\|\}$$

In orthogonality-based models, fixing a pole, also called *observable*, corresponds to choosing a notion of *correct* computation.

Proposition 2. *This pole satisfies some important properties:*

1. *($\leq$-saturation) If* $(\langle M, E, S\rangle, p) \in \perp\!\!\!\perp$ *and* $p \leq q$ *then* $(\langle M, E, S\rangle, q) \in \perp\!\!\!\perp$*.*
2. *($\longrightarrow$-saturation) If* $(\langle M, E, S\rangle, p) \in \perp\!\!\!\perp$ *and* $\langle M', E, S'\rangle \longrightarrow \langle M, E, S\rangle$ *then* $(\langle M', E', S'\rangle, p + \boldsymbol{1}) \in \perp\!\!\!\perp$*.*

The pole induces a notion of orthogonality. In contrast with usual models, since we need to deal with references, the orthogonality relation is parametrized by a set $\mathcal{S}$ of stores.

Definition 7. *The* orthogonality relation $\perp_{\mathcal{S}} \subseteq \Lambda_{\mathcal{M}} \times \Pi_{\mathcal{M}}$ *is defined as:*

$$(M, p) \perp_{\mathcal{S}} (E, e) \textit{ iff } \forall (S, s) \in \mathcal{S}, (\langle M, E, S\rangle, e(p + s)) \in \perp\!\!\!\perp$$

This orthogonality relation lifts to sets of weighted terms and weighted stacks. If $X \subseteq \Lambda_{\mathcal{M}}$ *(resp* $X \subseteq \Pi_{\mathcal{M}}$*),*

$$X^{\perp_{\mathcal{S}}} = \{\ (E, e) \in \Pi_{\mathcal{M}} \mid \forall (M, p) \in X, (M, p) \perp_{\mathcal{S}} (E, e)\ \}$$

$$\textit{(resp. } X^{\perp_{\mathcal{S}}} = \{\ (M, p) \in \Lambda_{\mathcal{M}} \mid \forall (E, e) \in X, (M, p) \perp_{\mathcal{S}} (E, e)\ \}\textit{)}$$

The operation $(.)^{\perp_{\mathcal{S}}}$ satisfies the usual orthogonality properties.

Lemma 1. *Suppose $X, Y \subseteq \Lambda_{\mathcal{M}}$ or $X, Y \subseteq \Pi_{\mathcal{M}}$:*

1. *$X \subseteq Y$ implies $Y^{\perp_{\mathcal{S}}} \subseteq X^{\perp_{\mathcal{S}}}$*
2. *$X \subseteq X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$*
3. *$X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}\perp_{\mathcal{S}}} = X^{\perp_{\mathcal{S}}}$*

Definition 8. *If X is a set of weighted realizers, we define its* $\leq$-closure $\overline{X} = \{ (M, p) \mid \exists q \leq p, (M, q) \in X \}$.

Remark 5. Notice that for any $\mathcal{S}$, we have $\overline{X} \subseteq X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$.

We say that a set $X \subseteq \Lambda_{\mathcal{M}}$ is a $\mathcal{S}$-behavior if $X = X^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$. Finally, we can define the set of $\mathcal{S}$-reducibility candidates. To do that, we first need to extend the language of terms with a new constant

$$M \quad ::= \ldots \quad | \quad \maltese$$

This constant comes with no particular reduction rule. It can be seen as a special variable considered as a closed term and is in a sense the dual of the empty stack. Notice that none of the previous constructions are modified. Moreover, at the end of the day, because we only consider typable terms (that do not contain any $\maltese$), $\maltese$ is only a technical intermediate.

Definition 9. *The set of* $\mathcal{S}$-reducibility candidates*, denoted by $\mathsf{CR}_{\mathcal{S}}$ is the set of $\mathcal{S}$-behaviors X such that $(\maltese, \boldsymbol{0}) \in X \subseteq \{(\diamond, x \mapsto x)\}^{\perp_{\mathcal{S}}}$*

Remark 6. If $(M, p) \in X$ where X is a $\mathcal{S}$-reducibility candidate and if $(\diamond, \mathbf{0}) \in \mathcal{S}$, then $\langle M, \diamond, \diamond \rangle$ terminates in at most $\|p\|$ steps. In fact our notion of reducibility candidate extends the usual notion in the non-quantitative case.

Finally, suppose R is a set of regions and suppose $\mathcal{S}_R$ is a set of stores whose domain is restricted to a R. We say that :

$\mathcal{S}_R \sqsubseteq \mathcal{S}' \Leftrightarrow \mathcal{S}'$ contains $\mathcal{S}_R$ and if $(S, s) \in \mathcal{S}'$ and if we write $S = S^{\delta} \uplus S''$, then there is a decomposition $s = s' + s''$ such that $(S^{\delta}, s') \in \mathcal{S}_R$, $dom(S'') = \{ r_i \mid \delta_i > \delta \}$ and moreover, if $(S_R, s_R) \in \mathcal{S}_R$ then $(S'' \uplus S_R, s'' + s_R) \in \mathcal{S}'$.

Remark 7. This quite involved definition will permit to the interpretation of a type to enjoy properties similar to the one called *extension/restriction* in [1]. In other words, given a store, it gives a way to say what substore can be removed safely and what stores can be added to it safely.

3.3 Interpretation of $\lambda_{\mathbf{LAL}}^{\mathsf{Reg},\mu}$

Using the orthogonality machinery previously defined, we can give an interpretation of $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ types as reducibility candidates. Suppose R is the following region context:

$$R = r_1 : (\delta_1, \S A_1), \ldots, r_n : (\delta_n, \S A_n)$$

We define three indexed sets: the *interpretation* $|R|_\delta$ of the region context R, the *pre-interpretation* $\|R \vdash A\|_\delta$ of a type A and its *interpretation* $|R \vdash A|_\delta^{\mathcal{S}}$ with respect to a set of stores $\mathcal{S}$. These three notions are defined by mutual induction, first on the index δ, and then on the size of the type A.

$$\begin{aligned}|R|_{=\delta} = \{\ (S, \sum_{\delta_i=\delta}\sum_{1\leq j\leq k_i} \S q_j^i) \mid & \ dom(S) = \{\ r_i \mid r_i : (\delta_i, \S A_i) \in R \wedge \delta_i = \delta\ \} \\ \wedge & \quad \forall r_i \in dom(S), S(r_i) = \{\S V_1^i, \S V_2^i, \ldots, \S V_{k_i}^i\} \\ \wedge & \quad \forall j \in [1, k_i], (V_j^i, q_j^i) \in \|R \vdash A_i\|_{\delta_i - 1}\ \}\end{aligned}$$

$$|R|_{\delta+1} = \{\ (S_1 \uplus S_\delta, s_1 + \S s_\delta) \mid (S_1, s_1) \in |R|_{=\delta+1}, (S_\delta, s_\delta) \in |R|_\delta\ \}$$

For convenience, we start the indexing of the interpretation at -1 instead of 0.

$$\|R \vdash A\|_{-1} = \overline{\{(\maltese, \mathbf{0})\}}$$

For $\delta \geq 0$, we define the pre-interpretation as:

$$\begin{aligned}\|R \vdash \mathsf{Unit}\|_\delta &= \overline{\{((), \mathbf{0})\}} \\ \|R \vdash \mathsf{Reg}_r A\|_\delta &= \overline{\{(r, \mathbf{0})\}} \\ \|R \vdash A \multimap B\|_\delta &= \overline{\{\ (\lambda x.M, p) \mid \forall (V, v) \in \|R \vdash A\|_\delta, \forall \mathcal{S}, |R|_\delta \sqsubseteq \mathcal{S}, (M[V/x], p+v) \in |R \vdash B|_\delta^{\mathcal{S}}\ \}} \\ \|R \vdash \S A\|_\delta &= \overline{\{\ (\S V, \S v) \mid (V, v) \in \|R \vdash A\|_{\delta-1}\ \}} \\ \|R \vdash !A\|_\delta &= \overline{\{\ (!V, !v) \mid (V, v) \in \|R \vdash A\|_{\delta-1}\ \}} \\ \|R \vdash \mu X.A\|_\delta &= \|R \vdash A[\mu X.A/X]\|_\delta\end{aligned}$$

The interpretation of a type with respect to a set $\mathcal{S}$ is just defined as the bi-orthogonal of the pre-interpretation:

$$|R \vdash A|_\delta^{\mathcal{S}} = \|R \vdash A\|_\delta^{\perp_{\mathcal{S}}\perp_{\mathcal{S}}}$$

Remark 8. Because of the presence of type fixpoints and regions, there are several circularities that could appear in the definition of $\|R \vdash A\|_\delta$. Yet, the interpretation is well defined for the following reasons:

– The type fixpoints $\mu X.A$ we consider are such that every occurrence of X in A is *guarded* by a modality ! or §. But these modalities make the index of the interpretation decrease by one. Hence, $\|R \vdash \mu X.A\|_{\delta+1}$ is well defined as soon as $\|R \vdash \mu X.A\|_\delta$ is.
– To define $\|R \vdash A\|_{\delta+1}$, we need $|R|_{\delta+1}$ to be already defined. But here again, in R each type is guarded by a modality §. This implies that to define $|R|_{\delta+1}$, we only need to know each $\|R \vdash A_i\|_\delta$.

An important point is that the interpretation of a formula A with respect to a region context R and to an index $\delta \in \mathbb{N}$ is a $|R|_\delta$-reducibility candidate (it will be used to prove bounded-time termination).

Proposition 3. *For all $\delta \in \mathbb{N}$ we have $|R \vdash A|_\delta^{|R|_\delta} \in \mathsf{CR}_{|R|_\delta}$.*

Table 1. Inferring a bound from a $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ typing judgment

$$\mathsf{v}\,\frac{}{\vdash^{\delta} x : \mathbf{0}} \qquad \mathsf{r}\,\frac{}{\vdash^{\delta} r : \mathbf{0}} \qquad \mathsf{u}\,\frac{}{\vdash^{\delta} () : \mathbf{0}} \qquad \mathsf{get}\,\frac{}{\vdash^{\delta} \mathsf{get}(r) : \mathbf{5}}$$

$$\mathsf{set}\,\frac{\vdash^{\delta} V : [\![V]\!]}{\vdash^{\delta+1} \mathsf{set}(r,\S V) : \S[\![V]\!] + \mathbf{1}} \qquad \mathsf{fold}\,\frac{\vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta} M : [\![M]\!]} \qquad \mathsf{unfold}\,\frac{\vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta} M : [\![M]\!]}$$

$$\mathsf{c}\,\frac{x : !, y : ! \vdash^{\delta} M : [\![M]\!]}{z : ! \vdash^{\delta} M[z/x, z/y] : [\![M]\!] + \mathbf{1}} \qquad \mathsf{w}\,\frac{\vdash^{\delta} M : [\![M]\!]}{x : \delta \vdash^{\delta} M : [\![M]\!]}$$

$$\mathsf{lam}\,\frac{\vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta} \lambda x.M : [\![M]\!]} \qquad \mathsf{app}\,\frac{\vdash^{\delta} M_1 : [\![M_1]\!] \quad \vdash^{\delta} M_2 : [\![M_2]\!]}{\vdash^{\delta} M_1 M_2 : [\![M_1]\!] + [\![M_2]\!] + \mathbf{3}}$$

$$\S\text{-}\mathsf{prom}\,\frac{\vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta+1} \S M : \S[\![M]\!] + \mathbf{4}} \qquad \S\text{-}\mathsf{elim}\,\frac{\vdash^{\delta} V : [\![V]\!] \quad \Gamma \vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta} \mathsf{let}\ \S x = V\ \mathsf{in}\ M : [\![M]\!] + [\![V]\!] + \mathbf{3}}$$

$$!\text{-}\mathsf{prom}\,\frac{\vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta+1} !M : F([\![M]\!])} \qquad !\text{-}\mathsf{elim}\,\frac{\vdash^{\delta} V : [\![V]\!] \quad \Gamma \vdash^{\delta} M : [\![M]\!]}{\vdash^{\delta} \mathsf{let}\ !x = V\ \mathsf{in}\ M : [\![M]\!] + [\![V]\!] + \mathbf{3}}$$

3.4 Adequacy and Bounded-Time Termination

We now prove the soundness of our model with respect to $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ and as a corollary the bounded-time termination theorem. In Table 1 is described how to infer an element of $\mathcal{M}$ from a $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ typing judgment: the notation $[\![M]\!]$ corresponds to the element of $\mathcal{M}$ already inferred from the typing judgment of $[\![M]\!]$, and each rule corresponds to the way $[\![M]\!]$ is built.

Definition 10. *We use the notations $\overline{V}$, $\overline{p}$ and $\overline{y}$ to denote respectively a list of values $[V_1, \ldots, V_n]$, a list $[p_1, \ldots, p_n]$ of elements of $\mathcal{M}$ and a list of variables $[y_1, \ldots, y_n]$. If M is a term, we denote by $M[\overline{V}/\overline{y}]$ the term $M[V_1/y_1, \ldots, V_n/y_n]$. If $\overline{p}$ is a list of elements of $\mathcal{M}$ and $\dagger \in \{!, \S\}$, we denote by $\dagger\overline{p}$ the list $[\dagger p_1, \ldots, \dagger p_n]$. We also define $\sum \overline{p}$ to be the sum $\sum_{1 \le i \le n} p_i$.*

If A is a type then we define λA as A itself. Suppose $\Gamma = x_1 : (e_1, A_1), \ldots, x_n : (e_n, A_n)$. Then the notation $(\overline{V}, \overline{p}) \Vdash^{\delta} \Gamma$ stands for $(W_i, p_i) \in \|R \vdash e_i A_i\|_{\delta}$ for $1 \le i \le n$ with $W_i = V_i$ if $e_i = \lambda$ and $W_i = \dagger V_i$ if $e_i = \dagger$.

Example 4. If we have $(\overline{V}, \overline{p}) \Vdash^{\delta} (x_1 : (\lambda, A_1), x_2 : (\S, A_2), x_3 : (!, A_3))$ then $\overline{V} = [V_1, V_2, M_3]$ and $\overline{p} = [p_1, \S p_2, !p_3]$ such that $(V_1, p_1) \in \|R \vdash A_1\|_{\delta}$, $(\S V_2, \S p_2) \in \|R \vdash \S A_2\|_{\delta}$ and $(!V_3, !p_3) \in \|R \vdash !A_3\|_{\delta}$.

Theorem 2 (Adequacy). *Suppose that $R; \Gamma \vdash^{\delta} M : C$. Let $(\overline{V}, \overline{p}) \Vdash^{\delta} \Gamma$, Then, for any $\mathcal{S}$ such that $|R|_{\delta} \sqsubseteq \mathcal{S}$,*

$$(M[\overline{V}/\overline{x}], [\![M]\!] + \sum \overline{p}) \in |R \vdash C|_{\delta}^{\mathcal{S}}$$

Moreover, if M is a value, then we have $(M[\overline{V}/\overline{x}], [\![M]\!] + \sum \overline{p}) \in \|R \vdash C\|_{\delta}$.

Proof. The proof is done by induction on the typing judgment.

One of the inductive cases here is particularly interesting, namely the §-promotion rule. It requires to prove the following lemma.

Lemma 2 (§-prom). *Suppose that for any $\mathcal{S}$ such that $|R|_\delta \sqsubseteq \mathcal{S}$, $(M, m) \in |R \vdash A|^{\mathcal{S}}_{\delta}$ holds. Then for any $\mathcal{S}$ such that $|R|_{\delta+1} \sqsubseteq \mathcal{S}$, we have $(\S M, \S m + \lightning) \in |R \vdash \S A|^{\mathcal{S}}_{\delta+1}$.*

This case is very important, since it justifies many definitions.

- Its proof crucially relies on the fact that in the definition of the region context interpretation $|R|_\delta$, each value is guarded by a modality § and not by a modality !. Indeed, it requires the monoidality property, which is true for § but not for !: $\forall p, q \in \mathcal{M}, \S(p+q) \leq \S p + \S q$.
- It also relies on the fact that we can consider any set of store $\mathcal{S}$ such that $|R|_\delta \sqsubseteq \mathcal{S}$, which is also built-in in our interpretation of the linear arrow $\multimap$.

As a corollary of the adequacy theorem, we obtain the announced bounded-time termination theorem for $\lambda^{\mathsf{Reg},\mu}_{\mathrm{LAL}}$ programs. As we have proved adequacy for *any* choice of a light monoid, we now consider a particular one, that is the light monoid defined in Example 3.

Proof (Termination theorem (Theorem 1)). This theorem is proved using adequacy together with Property 3. Indeed, we know that $\langle M, \diamond, \diamond \rangle$ terminates in at most $\|[\![M]\!]\|$ steps. Now, because we use the light monoid $\mathcal{M}_l$ of Example 3, it is easy to see that only the promotion rules for § and ! make the value of $\|[\![M]\!]\|$ increase significantly: the degree of the third component of $[\![M]\!]$ (which is a polynomial) is bounded by a function of the depth of M. A similar argument is made more precise in [11], for instance.

4 Related Work

Approximation Modality — In a series of two papers, Nakano introduced a normalizing intuitionistic type system that features recursive types, which are guarded by a modality $\bullet$ (the approximation modality). Nakano also defines an indexed realizability semantics for this type system. The modality § plays in our work almost the same role as $\bullet$: it makes the index increase. We claim that when we forget the quantitative part of our model, we obtain a model for a language with guarded references, that can be extended to handle control operators, based on a fragment of Nakano's type system: the only difference is that the $\bullet$ modality does not enjoy digging anymore (in presence of control operators, this principle would break normalization).

Stratified Semantics for Light Logics — Several semantics for the "light" logics have been proposed, beginning with fibered phase models [16], a truth-value semantics for **LLL**. We can also mention stratified coherent spaces [4].

These two models are indexed, like ours, but while the indexing is used to achieve completeness with respect to the logic, we use it to interpret fixpoints and references.

Reactive Programming — In [18], Krishnaswami & al. have proposed a type system for a discrete-time reactive programming language that bounds the size of the data flow graph produced by programs. It is based on linear types and a Nakano-style approximation modality, thus bounding space consumption and allowing recursive definitions at the same time. They provide a denotational semantics based on both ultrametric semantics and length spaces. These latter, introduced by Hofmann [8] constitute the starting point of the quantitative realizability presented here.

5 Research Directions

We see several possible directions we plan to explore.

Control Operators — Since we use a biorthogonality-based model, it is natural to extend the language with control operators. Adding the call-cc operator can be done, but it requires to add a modality type ? for *duplicable contexts*. This involves some technical subtleties in the quantitative part, like the symmetrization of the notion of $\mathcal{M}$-contexts. Indeed, in our framework, a $\mathcal{M}$-context can be used to *promote* a weight associated to a term, but with this new ? type, a weight associated to a term would need to be able to promote a weight associated to a stack.

Multithreading — In the original work of Amadio and Madet [15], the language features regions but also multithreading. It is possible to add it to $\lambda_{\mathrm{LAL}}^{\mathsf{Reg},\mu}$ but so far, it seems difficult to adapt the quantitative framework for this extension. It may be possible to adapt the notion of *saturated store* presented in [1], but with a boundedness requirement on it. We plan to explore this direction in the future.

References

1. Amadio, R.M.: On Stratified Regions. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 210–225. Springer, Heidelberg (2009)
2. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(5), 657–683 (2001)
3. Asperti, A.: Light affine logic. In: Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science, pp. 300–308 (1998)
4. Baillot, P.: Stratified coherence spaces: a denotational semantics for light linear logic. Theoretical Computer Science 318(1), 29–55 (2004)
5. Brunel, A.: Quantitative classical realizability (submitted, 2012)

6. Dal Lago, U., Martini, S., Sangiorgi, D.: Light logics and higher-order processes. Electronic Proceedings in Theoretical Computer Science 41 (2010)
7. Girard, J.-Y.: Light Linear Logic. In: Leivant, D. (ed.) LCC 1994. LNCS, vol. 960, pp. 145–176. Springer, Heidelberg (1995)
8. Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Information and Computation 183(1), 57–85 (2003)
9. Krivine, J-L.: Realizability in classical logic. Course notes of a series of lectures given in the University of Marseille (May 2004) (last revision: July 2005), Panoramas et syntheses, Société Mathéematique de France (2005)
10. Lafont, Y.: Soft linear logic and polynomial time. Theoretical Computer Science 318(1-2), 163–180 (2004)
11. Dal Lago, U., Hofmann, M.: Bounded Linear Logic, Revisited. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 80–94. Springer, Heidelberg (2009)
12. Dal Lago, U., Hofmann, M.: A semantic proof of polytime soundness of light affine logic. Theory of Computing Systems 46, 673–689 (2010)
13. Dal Lago, U., Hofmann, M.: Realizability models and implicit complexity. Theoretical Computer Science 412(20), 2029–2047 (2011), Girard's Festschrift
14. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 47–57. ACM, New York (1988)
15. Madet, A., Amadio, R.M.: An Elementary Affine λ-Calculus with Multithreading and Side Effects. In: Ong, L. (ed.) TLCA 2011. LNCS, vol. 6690, pp. 138–152. Springer, Heidelberg (2011)
16. Okada, M., Kanovich, M.I., Scedrov, A.: Phase semantics for light linear logic. Theoretical Computer Science 294(3), 525–549 (2003)
17. Nakano, H.: A modality for recursion. In: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, pp. 255–266. IEEE (2000)
18. Benton, N., Krishnaswami, N.R., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 45–58. ACM (2012)
19. Baillot, P., Gaboardi, M., Mogbil, V.: A PolyTime Functional Language from Light Linear Logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 104–124. Springer, Heidelberg (2010)
20. Terui, K.: Light affine lambda calculus and polynomial time strong normalization. Archive for Mathematical Logic 46(3-4), 253–280 (2007)