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 16, 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 $M \rightarrow N$. We write $\xrightarrow{*}$ for the reflexive and transitive closure of $\rightarrow$. We also define $M \stackrel{\alpha}{\Rightarrow} N$ 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

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

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:*

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

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

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

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

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

Reduction Rules

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

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

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 $\mathcal{C}_{cc}(M) \stackrel{\alpha}{\Rightarrow} \mathcal{C}_{cc}(M')$.*

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

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

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:*

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

or hoist and then reduce:

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

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 $M \leadsto N$ then there is a N' such that $N \xrightarrow{\ell} N'$ and $M'\ (\leadsto^* \circ S_h)\ N'$. 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* $N \stackrel{\alpha}{\Rightarrow} N'$ *and* $M'\ \mathcal{R}_C\ N'$.

$$\begin{array}{c}\text{SYNTAX FOR } \lambda^{\ell}_{h}\\ \text{Syntactic restrictions on } \lambda^{\ell}_{cps,vn} \text{ after hoisting}\\ \text{All function definitions are at top level.}\end{array}$$

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

SPECIFICATION OF THE HOISTING TRANSFORMATION

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

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

$$\begin{array}{lll}
(h_1) & D[\mathsf{let}\ x = C\ \mathsf{in}\ \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M] \rightsquigarrow & \\
 & D[\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = C\ \mathsf{in}\ M] & \text{if } x \notin \mathsf{fv}(\lambda z^+.T)\\[1em]
(h_2) & D[\mathsf{let}\ x = (\lambda w^+.\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M)\ \mathsf{in}\ N] \rightsquigarrow & \\
 & D[\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \mathsf{let}\ x = \lambda w^+.M\ \mathsf{in}\ N] & \text{if } \{w^+\} \cap \mathsf{fv}(\lambda z^+.T) = \emptyset\\[1em]
(h_3) & D[\ell > \mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ M] \rightsquigarrow & \\
 & D[\mathsf{let}\ y = \lambda z^+.T\ \mathsf{in}\ \ell > M] &
\end{array}$$

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

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

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:

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

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

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

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 $\lceil \mathcal{I}(A) \rceil \rightarrow \mathsf{prop}$, 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** $\{\ \ldots\}$, 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 l1 and l2 as input, and it is defined, as usual, by induction on 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** l1 **with** Nil $\Rightarrow \bullet$ | Cons(x, xs) $\Rightarrow \bullet$", ℓ_{nil} dominates "Nil", ℓ_{cons} dominates "Cons(x, $\bullet$)", and ℓ_{rec} dominates "concat(xs, 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.*, costof(ℓ_{nil}) $= 2$, costof(ℓ_{rec}) $= 6$, costof(ℓ_{cons}) $= 5$ and costof(ℓ_{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(ℓ_match ) + costof(ℓ_nil)
          + (costof(ℓ_rec) + costof(ℓ_match) + costof(ℓ_cons)) × length (l1)
}
=
   ℓ_match >
   match l1 with
   | Nil → ℓ_nil > l2
   | Cons (x, xs) → ℓ_cons > Cons (x, concat (xs, l2) > ℓ_rec)
   end
```

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

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

```
routine x_19 (c_20, x_7)
ℓ_rec:
 k_2 ← proj 1 c_20 ;
 x ← proj 2 c_20 ;
 x_14 ← make_int 1 ;
 x_15 ← make_tuple (x_14, x, x_7) ;
 x_22 ← proj 0 k_2 ;
 call x_22 (k_2, x_15)

routine x_16 (c_17, l1, l2, k_2)
ℓ_match:
 switch l1
 0 : ℓ_nil :
     x_18 ← proj 0 k_2 ;
     call x_18 (k_2, l2)
 1 : ℓ_cons :
     xs ← proj 2 l1 ;
     x ← proj 1 l1 ;
     x_13 ← make_tuple (x_19, k_2, x) ;
     x_21 ← proj 0 c_17 ;
     call x_21 (c_17, xs, l2, x_13)
```

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

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

Typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash N : B \quad \Gamma \vdash M : A}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}$$

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

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

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

Restricted syntax CPS types, R type of results

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

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:

$$
\begin{array}{lcll}
\mathcal{C}_{cc}(@(x,y^+)) & = & \begin{array}{l}\mathsf{let}\ x = \pi_1(x)\ \mathsf{in} \\ \mathsf{let}\ (c,e) = x\ \mathsf{in}\ @(c,e,y^+)\end{array} & \begin{array}{l}(\leftarrow \text{EXISTENTIAL ELIMINATION}) \\ \end{array} \\[2ex]
\mathcal{C}_{cc}(\mathsf{let}\ x = C\ \mathsf{in}\ M) & = & \begin{array}{l}\mathsf{let}\ c = \lambda e, x^+.\mathsf{let}\ (z_1,\dots,z_k) = e\ \mathsf{in}\ \mathcal{C}_{cc}(N)\ \mathsf{in} \\ \mathsf{let}\ e = (z_1,\dots,z_k)\ \mathsf{in} \\ \mathsf{let}\ x = (c,e)\ \mathsf{in} \\ \mathsf{let}\ x = (x)\ \mathsf{in} \\ \mathcal{C}_{cc}(M)\end{array} & \begin{array}{l} \\ \\ \\ (\leftarrow \text{EXISTENTIAL INTRODUCTION}) \\ (\text{if } C = \lambda x^+.N, \mathsf{fv}(C) = \{z_1,\dots,z_k\})\end{array}
\end{array}
$$

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

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

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

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

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

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:

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

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':

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

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 $r, r', \ldots$ and a syntactic category of *effects* e with generic elements $e, e', \ldots$ An effect is a finite set of region variables. We keep denoting *types* with $A, B, \ldots$ 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

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

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

REGION NOT-DISPOSED IN A HEAP CONTEXT

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

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

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 $Coh(H,L)$ 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 $NDis(r,H)$ 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

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

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

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

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

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):*

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

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 $\lambda^{\ell,r}_{h,vn}$.

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

Unfortunately this strategy leads to a memory error as:

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

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 $\lambda^{\ell,r}_{h,vn}$.*

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

This time the reduction leads to a normal termination:

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

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:*

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

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

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

Typing rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $P \to P'$ 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:

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

$\pi_i(M)$ 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:

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

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

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

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

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

□

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:

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

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:

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

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

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

$\square$

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:

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

$\mathsf{let}\ x = \pi_i(V)\ \mathsf{in}\ M, V \neq id$ 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:

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

$\ell > M$ We have:

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

□

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 $M \equiv E[@(x, y^+)] \rightarrow E[[y^+/x^+]M]$ 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:

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

- 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. $\square$

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 $\mathcal{C}_h(er(M)) \equiv er(N) \equiv er(\mathcal{C}_h(M))$. □

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. $\square$

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 $\lambda x^+.\ell > \mathcal{L}_1(M) \in W_1$.

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

$\pi_j(M)$ 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 $\mathsf{let}\ x = \mathcal{L}_0(M)\ \mathsf{in}\ \mathcal{L}_i(N) \in W_i$.

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

$@(M_1, \dots, M_n)$ **and** $i = 1$ Same argument as in the previous case to conclude that $@(\mathcal{L}_0(M_1), \dots, \mathcal{L}_0(M_n)) \in W_1$.

(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.

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

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

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

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

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

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

$\pi_j(M)$ We have:

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

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. $\Box$

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 $S=[V_1/x_1,\ldots,V_n/x_n]$ denote a substitution in the λ^ℓ-calculus. Then let $\psi(S)$ be the substitution $[\psi(V_1)/x_1,\ldots,\psi(V_n)/x_n]$. 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 $\Lambda = \Lambda_0 \cdots \Lambda_{n+1}$. $\Box$

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 $\mathcal{C}_{cps}(M) \equiv M \mid \lambda x.@(halt, x)$. Then we obtain the desired statement from the property 3 above observing that if $\Gamma \vdash M : A$ and $\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A), x : \mathcal{C}_{cps}(A) \vdash @(halt, x) : R$ then $\mathcal{C}_{cps}(\Gamma), halt : \neg\mathcal{C}_{cps}(A) \vdash \mathcal{C}_{cps}(M) : R$.

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 $\Gamma' \equiv \Gamma_1, y : \times(A_1, \ldots, A_n), \Gamma_2$, $1 \leq i \leq n$, $\Gamma, \Gamma', x : A_i \vdash^{vn} M'$, 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:

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

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

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

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

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

$ren(F[T])$	$= ren(F)[\mathsf{let}\ \mathsf{all}(r)\ \mathsf{in}\ ren(T)]$	(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 $\Gamma \vdash^{rg} \mathsf{let\ all}(r)\ \mathsf{in}\ T : e$ is derived from $\Gamma \vdash^{rg} T : e \cup \{r\}$, $r \notin e \cup \mathsf{frv}(\Gamma)$, and σ is injective on $\mathsf{frv}(\mathsf{let\ all}(r)\ \mathsf{in}\ T) \cup e$. 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 $\sigma\Gamma \vdash^{rg} \sigma(\mathsf{let\ all}(r)\ \mathsf{in}\ T) : \sigma e$.

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:

$$[x/y]rer(T) \equiv rer([x/y]T) \ .$$

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 $rer(P)$ as $rer(F)[rer(H)[rer(\Delta)]]$, where $rer(F)[rer(H)]$ is an evaluation context. If $rer(P) \xrightarrow{\alpha} Q$ 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

$$\begin{array}{rcl} rer(P) & \equiv & rer(F_1)[\mathsf{let}\ x = \lambda z^+.rer(T)\ \mathsf{in}\ rer(F_2)[rer(H)[@(x, y^+)]] \\ & \to & rer(F_1)[\mathsf{let}\ x = \lambda z^+.rer(T)\ \mathsf{in}\ rer(F_2)[rer(H)[[y^+/z^+]rer(T)]] \ . \end{array}$$

Since P is typable, the heap context is coherent and then P can simulate the reduction above as follows:

$$P \to F[H[[r^*/r_1^*, y^+/z^+]T]]$$

noticing that $rer([r^*/r_1^*, y^+/z^+]T) \equiv [y^+/z^+]rer(T)$ (initial remark and invariance of the region erasure function under region substitutions).