Certifying and reasoning on cost annotations of functional programs

Roberto M. Amadio, Yann Regis-Gianas

Certifying and reasoning on cost annotations of functional programs

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

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

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

1 Introduction

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

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

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

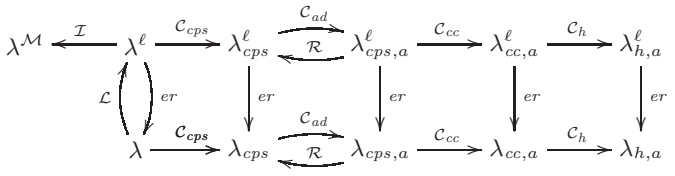

Table1. The compilation chain with its labelling and instrumentation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 The compilation chain: commutation and simulation

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

2.1 Conventions

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

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

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

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

Syntax

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

Reduction Rules

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

Label erasure

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

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

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

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

2.2 The source language

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

2.3 Compilation to CPS form

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

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

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

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

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

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

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

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

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

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

Syntax CPS terms

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

Reduction rules

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

CPS translation

$$
\begin{array}{ll}
\psi(x) & = x \\
\psi(\lambda x^{+}.M) & = \lambda x^{+}, k.(M : k) \\
\psi(V_1, \ldots, V_n) & = (\psi(V_1), \ldots, \psi(V_n))
\end{array}
$$

$$
\begin{array}{ll}
V : k & = @(k, \psi(V)) \\
V : (\lambda x.M) & = [\psi(V)/x]M \\
@(M_0, \ldots, M_n) : K & = M_0 : \lambda x_0. \ldots (M_n : \lambda x_n.@(x_0, \ldots, x_n, K)) \\
\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : K & = M_1 : \lambda x.(M_2 : K) \\
(M_1, \ldots, M_n) : K & = M_1 : \lambda x_1. \ldots (M_n : \lambda x_n.(x_1, \ldots, x_n) : K\ ) \\
\pi_i(M) : K & = M : \lambda x.\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ y : K \\
(\ell > M) : K & = \ell > (M : K) \\
(M > \ell) : K & = M : (\lambda x.\ell > (x : K))
\end{array}
$$

$$
\mathcal{C}_{cps}(M) = M : \lambda x.@(halt, x), \qquad halt \text{ fresh}
$$

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

2.4 Transformation in administrative CPS form

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

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

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

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

SYNTAX

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

REDUCTION RULES

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

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

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

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

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

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

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

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

2.5 Closure conversion

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

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

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

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

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

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

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

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

Table5. Transformations in administrative CPS form and readback

2.6 Hoisting

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

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

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

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

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

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

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

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

Closure Conversion

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

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

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

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

Table6. Closure conversion on administrative CPS terms

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

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

2.7 Composed commutation and simulation properties

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

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

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

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

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

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

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

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

Syntactic restrictions on $\lambda^{\ell}_{cps,a}$ after hoisting
All function definitions are at top level.

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

Specification of the hoisting transformation

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

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

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

Table7. Hoisting transformation

3 Reasoning on the cost annotations

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

3.1 Initial labelling

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

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

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

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

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

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

Table8. A sound and precise labelling of the source code

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

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

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

3.2 Instrumentation

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

3.3 Higher-order Hoare Logic

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

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

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

Table9. Instrumentation of labelled λ-calculus.

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

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

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

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

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

Syntax

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

Logical reflection of types

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

Logical reflection of values

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

Table10. The surface language.

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

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

3.4 Prototype implementation

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

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

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

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

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

3.5 Example

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

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

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

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

4 Conclusion

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

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

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

Table11. An higher-order function and its specification.

References

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

A Examples

This section collects some examples.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or hoist and then reduce:

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

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

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

B Proofs

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

B.1 Proof of proposition 1 [CPS commutation]

The proof takes the following steps:

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

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

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

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

   We proceed by induction on M.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

B.2 Proof of proposition 2 [CPS simulation]

The proof takes the following steps.

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

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

We proceed by induction on M.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

B.3 Proof of proposition 3 [AD commutation]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

B.4 Proof of proposition 4 [AD simulation]

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

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

Then we prove the property by case analysis.

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

B.5 Proof of proposition 5 [CC commutation]

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

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

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

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

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

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

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

$\ell > M$ We have:

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

□

B.6 Proof of proposition 6 [CC simulation]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

(h_1)

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

(h_2)

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

(h_3)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.8 Proof of proposition 8 [hoisting commutation]

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

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

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

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

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

B.9 Proof of proposition 9 [hoisting simulation]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

– The reduction rule is

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

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

(h_1) There are two critical cases.
1. The let-definition that defines a function of the hoisting transformation coincides with the let-definition of x. In this case M is actually a restricted term T. The diagram is closed in one step.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

(h_2) Again there are two critical situations.
1. The top level let-definition of the hoisting transformation coincides with the let-definition of the variable x in the reduction. This is the case illustrated by the example 5. If we reduce first then we have to apply the hoisting transformation twice (again using preservation under name substitution). After this we have to commute the let-definitions and finally collapse two identical ones.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

(h_3) There are two critical cases.
1. The function let-definition in the hoisting transformation coincides with the let-definition of the variable x in the reduction. We close the diagram in one step.
2. The path π' determines a subterm of M. If we reduce first then we have to apply the hoisting transformation twice to close the diagram using the fact that these transformations are preserved by name substitution.

– The reduction rule is

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

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

(h_1) There are two critical cases.
1. The first let-definition in the hoisting transformation coincides with the let-definition of the tuple in the reduction. We close the diagram in one step
2. The first let-definition in the hoisting transformation coincides with the projection in the reduction. If we reduce first then there is no need to apply a hoisting transformation to close the diagram because the projection disappears.

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

(h_3) Same argument as in the previous case.

– The reduction rule is

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

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

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

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

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

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

B.11 Proof of proposition 10 [labelling properties]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\pi_j(M)$ We have:

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

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

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

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

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