RESEARCH-ARTICLE

Closure Conversion, Flat Environments, and the Complexity of Abstract Machines

BENIAMINO ACCATTOLI, Computer Science Laboratory of The Ecole Polytechnique, Palaiseau, Ile-de-France, France

CLÁUDIO BELO LOURENÇO, Huawei Technologies Co., Ltd, United Kingdom, Reading, Berkshire, U.K.

DAN R GHICA, University of Birmingham, Birmingham, West Midlands, U.K.

GIULIO GUERRIERI, University of Sussex, Brighton, East Sussex, U.K.

CLAUDIO SACERDOTI-COEN, University of Bologna, Bologna, BO, Italy

Open Access Support provided by:

Computer Science Laboratory of The Ecole Polytechnique

Huawei Technologies Co., Ltd, United Kingdom

University of Bologna

University of Birmingham

University of Sussex

Published: 10 September 2025

Citation in BibTeX format

PPDP '25: Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming
September 10 - 11, 2025
Rende, Italy

https://doi.org/10.1145/3756907.3756922
ISBN: 9798400720857

Closure Conversion, Flat Environments, and the Complexity of Abstract Machines

Beniamino Accattoli
Inria & LIX, École Polytechnique
Palaiseau, France
beniamino.accattoli@inria.fr

Cláudio Belo Lourenço
Huawei Central Software Institute
Edinburgh, United Kingdom
claudio.lourenco@huawei.com

Dan R. Ghica
Huawei Central Software Institute
Edinburgh, United Kingdom
University of Birmingham
Birmingham, United Kingdom
dan.ghica@huawei.com

Giulio Guerrieri
University of Sussex
Brighton, United Kingdom
g.guerrieri@sussex.ac.uk

Claudio Sacerdoti Coen
Università di Bologna
Bologna, Italy
claudio.sacerdoticoen@unibo.it

Abstract

Closure conversion is a program transformation at work in compilers for functional languages to turn inner functions into global ones, by building *closures* pairing the transformed functions with the *environment* of their free variables. Abstract machines rely on similar and yet different concepts of *closures* and *environments*. We study the relationship between the two approaches. We adopt a simple λ-calculus with tuples as source language and study abstract machines for both the source language and the target of closure conversion. Moreover, we focus on the simple case of flat closures/environments (no sharing of environments). We provide three contributions. Firstly, a new simple proof technique for the correctness of closure conversion, inspired by abstract machines. Secondly, we show how the closure invariants of the target language allow us to design a new way of handling environments in abstract machines, not suffering the shortcomings of other styles.

Thirdly, we study the machines from the point of view of time complexity. We show that closure conversion decreases various dynamic costs while increasing the size of the initial code. Despite these changes, the overall complexity of the machines before and after closure conversion turns out to be the same.

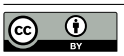
CCS Concepts

• **Theory of computation → Lambda calculus**; **Abstract machines**; **Operational semantics**; **Invariants**; • **Software and its engineering → Compilers**.

Keywords

Lambda calculus, abstract machine, program transformation

ACM Reference Format:
Beniamino Accattoli, Cláudio Belo Lourenço, Dan R. Ghica, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2025. Closure Conversion, Flat Environments, and the Complexity of Abstract Machines. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming (PPDP '25), September 10–11, 2025, Rende, Italy.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3756907.3756922



PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7/25/09
https://doi.org/10.1145/3756907.3756922

1 Introduction

A feature of functional languages, as well as of the λ-calculi on which they are based, is the possibility of defining *inner functions*, that is, functions defined inside the definition of other functions. The tricky point is that inner functions can also use the external variables of their enveloping functions. For instance, the Church numeral $\underline{3} := \lambda x.\lambda y.x(x(xy))$ contains the inner function $\lambda y.x(x(xy))$ that uses the externally defined variable x.

The result of functional programs can be a function. In particular, it can be the instantiation of an inner function. For instance, the result of applying $\underline{3}$ above to a value v is the instantiated function $\lambda y.v(v(vy))$, as one can easily see by doing one β-reduction step.

Closure Conversion. In practice, however, compiled functional programs do not follow β-reduction literally, nor do they produce the code of the instantiated function itself, because the potential duplications of the substitution process would be too costly. The idea is to decompose β-reduction in smaller, micro steps, delaying substitution as much as possible, and computing representations of instantiated functions called *closures*. A closure, roughly, is the pair of the defined inner function (that is, before instantiation) plus the tuple of instantiations for its free variables, called its *environment*.

This is achieved via a program transformation called *closure conversion*, that re-structures the code turning beforehand all inner functions into closures. At compile time, closures pair inner functions with *initial environments* that simply contain the free variables of the functions. Execution shall then dynamically fill up the environments with the actual instantiations.

Compilation and Abstract Machines. Abstract machines are often seen as a technique alternative to compilation and related to the interpretation of programming languages. This is because abstract machines tend to be developed for the source language, before the pipeline of transformations and optimizations of the compiler.

A *first aim* of our paper is to take a step toward closing the gap between compilation and abstract machines, by studying how transformations used by compilers—here closure conversion—induce

invariants exploitable for the design of machines working at further stages of the compilation pipeline, while still being abstract.

Same Terminology, Different Concepts. Similarly to compilation, abstract machines do not follow β-reduction literally, nor do they produce the code of instantiated functions. In particular, some abstract machines use data structures called again *closures* and *environments*. These notions, however, are similar and yet *different*.

In closure conversion, an inner function with free variables is transformed at compile time into a closed function paired with the environment of its free variables. The environment is in general *open* (it is closed by some enveloping converted function, yet locally it is open), and shall be filled/closed only during execution, which provides the instantiations. The aim is to hoist the closed function up to global scope while its environment stays at the call site.

In abstract machines, every piece of code receives an environment, not just functions, and environments and closures have a mutually recursive structure. Additionally, environments are built dynamically, during execution, not in advance, and they are always closed: the pair of a piece of code and its environment is called a *closure* because they read back (or decode) to a closed λ-term. Moreover, there is no hoisting of code.

A *second aim* of this paper is to establish and clarify the relationship between these two approaches. To disambiguate, we keep *closures* for the pairs *(closed function, possibly open environment)* of closure conversion and use *m-closures* for the pairs *(code, closed environment)* used in abstract machines; we further disambiguate calling *bags* the environments of closures, while the environments of m-closures keep their usual names. With respect to this terminology, we study *closure conversion for m-closures*. One of the outcomes of closure conversion turns out to be the elimination of m-closures.

Complexity of Abstract Machines. Finally, a last influence on our study comes from the recent development by Accattoli and coauthors of both a complexity-based theory of abstract machines [2–4, 6, 9, 12, 13, 20] and time and space reasonable cost models for the λ-calculus [7, 8, 10]. Such a line of work has developed fine analyses of tools and techniques for abstract machines, studying how different kinds of environments and forms of sharing impact on the cost of execution of abstract machines. The *third aim* of the paper is to understand how tuples and closure conversions affect environments, forms of sharing, and the cost of execution.

To avoid misunderstandings, our aim is *not* the study of optimized/shared notions of (compiler) closures, their efficient representations, their minimization, or the trade-off between access time and allocation time—in fact, we adopt the basic form of *flat* closure conversion that we apply to *all* functions for *all* their free variables.

Flat Environments. Different data structures for (m-)closures and bags/environments and various closure conversion algorithms can be used. The design space, in particular, is due to chains of nested functions—say, f is nested inside g, in turn, nested inside h—where two consecutive nested functions, f and g, can both use variables, say x, of h. Therefore, one might want the closures for f and g to share the environment entry for x. Perhaps surprisingly, sharing bags between closures can easily break *safety for space* of closure conversion, i.e. might not preserve the space required by the source program, if parts of shared environments survive the lifespan of

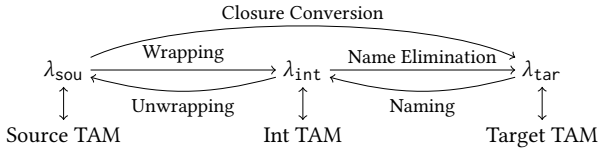


Figure 1: λ-Calculi and abstract machines in the paper.

the associated closures. The simplest approach is using *flat bags* (and flat closure conversion), where no sharing between bags is used; with flat bags, there are two distinct but identical entries for x in the bags of f and g. Flat bags are safe for space [31, 35, 36].

Machine environments can also either be shared or flat (i.e. with no sharing between environments). The distinction, however, is not often made since it does not show up in the abstract specification of the machine, but only when one concretely implements it or studies the complexity of the overhead of the machine (since the two techniques have different costs). For a meaningful comparison with flat closure conversion, for source programs we adopt an abstract machine meant to be implemented using flat environments.

Our aim is to understand how closure conversion compares *asymptotically* with m-closures of abstract machines, in particular with respect to one of the key parameters for time analyses, the size of the initial term. Flat environments are chosen for their simplicity and their safeness for space. More elaborated forms are future work.

Our Setting. We decompose closure conversion in two phases, first going from a source calculus $\lambda_{\mathtt{sou}}$ to an intermediate one $\lambda_{\mathtt{int}}$, and then to a target calculus $\lambda_{\mathtt{tar}}$. We provide each of the three calculi with its own abstract machine. Our setting is summed up in Fig. 1, where *TAM* stands for *tupled abstract machine*.

Our source λ-calculus $\lambda_{\mathtt{sou}}$ is the simplest possible setting accommodating closure conversion, that is, Plotkin's effect-free call-by-value λ-calculus extended with tuples, because tuples are needed to define the conversion. Our setting is untyped, because it is how abstract machines are usually studied, and for the sake of minimality.

The first transformation $\underline{\cdot} : \lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$, dubbed *wrapping*, replaces abstractions $\lambda\widetilde{x}.t$ (where $\widetilde{x}$ is a sequence of variables) with the dedicated construct of *abstract closures* $\underline{\lambda\widetilde{x}.t} := [\![\lambda\widetilde{y}.\lambda\widetilde{x}.\underline{t}\,|\,\vec{\widetilde{y}}]\!]$ where $\widetilde{y}$ is the sequence of free variables of $\lambda\widetilde{x}.t$, and $\vec{\widetilde{y}}$ is the tuple of these same variables, forming the *initial bag* of the closure.

Roughly, the second transformation $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$, deemed *name elimination*, turns the abstraction $\lambda\widetilde{y}$ of many variables into the abstraction of a single variable representing the bag/environment. It is similar to a translation from named variables to de Bruijn indices. In fact, name elimination is only outlined in the paper, the details are given in [5], because they are mostly routine.

The target calculus $\lambda_{\mathtt{tar}}$ is not the low-level target language of a compiler. It is indeed still high-level, because we study only flat closure conversion, not the whole compilation pipeline. We do not model the hoisting of closed functions up to global scope; it is an easy aspect of closure conversion and is usually avoided in its study.

We give three contributions, shaping the paper into three parts.

Contribution 1: A Simple Proof of Correctness. The correctness of closure conversion is often showed by endowing both the source and the target calculus of the transformation with big-step operational semantics and establishing a logical relation between the two

[16, 28, 31, 37]. The reason is that adopting a small-step semantics does not seem to work: closure conversion does not commute with meta-level substitution (that is, the substitution of converted terms is not the conversion of the substituted terms), and thus it does not map β-steps from the source to the target calculus.

Bowman and Ahmed [18] are to our knowledge the only ones adopting a small-step semantics. They neatly get around the non-commutation issue by noticing that the substitution of converted terms is *η-equivalent to* the conversion of the substituted terms. Our proof technique looks at the same issue in a different way, inspired by correctness proofs for abstract machines and independently of η-equivalence (which we do not consider for our calculi, for the sake of minimality). It might be of interest for languages where η-equivalence is not sound (e.g. because of observable effects, as in OCaml, or when η-equivalence does not preserve typability, like in languages with mutability and the value restriction), since in these cases Bowman and Ahmed's proof might not scale up.

Contribution 2: New Kinds of Machine Environments. According to Fernández and Siafakas [23], there are two kinds of abstract machines, those using many (shared or flat) *local environments*, which are defined by mutual induction with m-closures, and those using a single (necessarily flat) *global environment* or *heap* and no m-closures. Each kind has pros and cons, there is no absolute better style of machine environments; see also Accattoli and Barras [4].

For our machine for the source calculus $\lambda_{\mathtt{sou}}$, the Source TAM, we adopt flat local environments and m-closures, so as to allow us to compare closures (with flat bags) and m-closures. The contribution here is that the invariants enforced by flat closure conversion (more precisely, by wrapping) enable a new management of environments, what we dub *stackable environments* and plug into the Int TAM, our machine for the intermediate calculus $\lambda_{\mathtt{int}}$. Stackable environments have the pros of *both* global and local environments, and *none* of their cons, as explained in Sect. 8. They are called *stackable* because the current one can be put on hold—on the stack—when entering a closed function with its new environment, and re-activated when the evaluation of the function is over. But be careful: their stackability is not necessarily an advantage, it is just the way they work; the advantage is the lack of the cons of local and global environments.

Moving to the target calculus $\lambda_{\mathtt{tar}}$ enables a further tweak of environments, adopted by the Target TAM: environments—which usually are *maps* associating variables to values—become *tuples* of values, with no association to variables. This is enabled by *name elimination*, which turns variables into indices referring to the tuple, in a way reminiscent of de Bruijn indices.

Contribution 3: Time Complexity. We study how tuples and flat closure conversion impact the forms of sharing and the time complexity of the machines. Our analyses produce four insights:

(1) *Tuples raise the overhead*: we give a theoretical analysis of the cost of adding tuples to the pure λ-calculus, which, to our knowledge, does not appear anywhere in the literature. We show why tuples require their own form of sharing and that the *creation of tuples* at runtime is *unavoidable*. This is done by adapting *size exploding families* from the study of reasonable time cost models for the λ-calculus [1]. Moreover, tuples raise the dependency on the size $|t|$ of the initial code t of the overhead of the machine. Namely, let the *height* $\mathsf{hg}(t)$ be the maximum number of bound variables of t in the scope of which a sub-term of t is contained: the dependency for flat environments raises from $O(\mathsf{hg}(t))$ to $O(|t| \cdot \mathsf{hg}(t))$.

(2) *Name elimination brings a logarithmic speed-up*: with variable names, flat environments have at best $O(\log(\mathsf{hg}(t)))$ access time, while de Bruijn indices (or our name elimination) enable $O(1)$ access time—this is true both before and after closure conversion. Before conversion, however, the improvement does not lower the overall asymptotic overhead of the machine with respect to the size of the initial term, which is dominated by the other flat environment operations. After closure conversion, instead, it *does* lower the overall dependency of the machine from $O(|t^{\mathsf{cc}}| \cdot \mathsf{hg}(t^{\mathsf{cc}}))$ to $O(|t^{\mathsf{cc}}|)$, where t^{cc} is t after closure conversion.

(3) *Amortized constant cost of transitions*: in any abstract machine, independently of their implementation, the number of transitions of an execution and the cost of some single transitions depend on the size of the initial term. This is related to the higher-order nature of λ-calculi. Closure conversion impacts on the cost of single transitions (but not on their number): their *amortized* cost becomes constant. The insight is that the non-constant cost of transitions in ordinary abstract machines is related to inner functions.

(4) *Dynamically faster, statically bigger, overall the same*: the previous two points show that closure conversion decreases the dependency of machines on the size of the initial term during execution. The dynamic improvement however is counter-balanced by the fact that $|t^{\mathsf{cc}}|$ is possibly *bigger* than $|t|$, namely $|t^{\mathsf{cc}}| \in O(|t| \cdot \mathsf{hg}(t))$. Therefore, the overall complexity is $O(|t| \cdot \mathsf{hg}(t))$ also after closure conversion.

OCaml Code and Proofs. As additional material on GitHub [33], we provide an OCaml implementation of the Target TAM, the machine for closure converted terms, described in [5, Appendix K].

All proofs are in [5], the long version of this paper.

Terms	t, u, s, r	$::=$	$x \mid \lambda x.t \mid tu$	$(\lambda x.t)v \mapsto_{\beta_v} t\{x \leftarrow v\}$
Values	v, v'	$::=$	$\lambda x.t$	$\dfrac{t \mapsto_{\beta_v} u}{C\langle t\rangle \to_{\beta_v} C\langle u\rangle}$
Ev. Ctxs	C, C'	$::=$	$\langle\cdot\rangle \mid tC \mid Cv$	

Figure 2: The untyped pure call-by-value calculus $\lambda_{\mathtt{cbv}}$

2 Preliminaries: $\lambda_{\mathtt{cbv}}$, a Call-by-Value λ-Calculus

In Fig. 2 we present $\lambda_{\mathtt{cbv}}$, a variant of Plotkin's call-by-value λ-calculus [32] with its β-reduction by value $\to_{\beta_v}$, adopting two specific choices. Firstly, the only values are λ-abstractions. Excluding variables from values differs from [32] but is common in the machine-oriented literature. It does not change the result of evaluation while inducing a faster substitution process, see [14].

Secondly, we adopt a small-step operational semantics, defined via evaluation contexts. Evaluation contexts C are special terms with exactly one occurrence of the *hole* constant $\langle\cdot\rangle$. We write $C\langle t\rangle$ for the term obtained from the evaluation context C by replacing its hole with the term t (possibly capturing some free variables of t).

The small-step rule of β_v-reduction is *weak*, that is, it does not evaluate abstraction bodies (indeed, the production $\lambda x.C$ is *absent*

$$\begin{array}{rrcl} \textsc{Terms} & t,u,s & ::= & x \mid tu \mid \pi_i t \mid \overbrace{\lambda\widetilde{x}.t}^{\lambda\widetilde{x}.t} \mid \overbrace{(\!|t_1,..,t_n|\!)}^{\vec{t}} \quad n\geq 0 \\ \textsc{Values} & v,v' & ::= & \lambda\widetilde{x}.t \mid \vec{v} \\ \textsc{Ev. Ctxs} & C,C' & ::= & \langle\cdot\rangle \mid tC \mid Cv \mid \pi_i C \mid \underbrace{(\!|t_1,..,t_k,C,v_1,..,v_h|\!)}_{(\!|\widetilde{t},C,\widetilde{v}|\!)} \quad k,h\geq 0 \end{array}$$

$$\begin{array}{l|l} (\lambda\widetilde{x}.t)\,\vec{v} \mapsto_{\beta_v} t\{\widetilde{x}\leftarrow\vec{v}\} \text{ if } \|\widetilde{x}\| = \|\vec{v}\| & \dfrac{t \mapsto_a u}{C\langle t\rangle \to_a C\langle u\rangle}\ a\in\{\beta_v,\pi\} \\ \pi_i\,\vec{v} \mapsto_{\pi} v_i \quad \text{if } 1\leq i\leq\|\vec{v}\| & \to_{\mathsf{sou}} := \to_{\beta_v}\cup\to_{\pi} \end{array}$$

Figure 3: The source calculus λ_{sou} extending λ_{cbv} with tuples.

in the definition of evaluation context C in Fig. 2), as it is common in functional programming languages, and *deterministic*, namely proceeding from right to left (as forced by the production Cv)[1].

We identify terms up to α-renaming; $t\{x\leftarrow u\}$ stands for metalevel capture-avoiding substitution of u for the free occurrences of x in t.

The lemma below rests on the closed hypothesis and will be used as a design check for the calculi in the next sections. It is an untyped instantiation of Wright and Felleisen's uniform evaluation property [41]. The term "harmony" is borrowed from [11].

Lemma 2.1 (λ_{cbv} harmony). *If $t\in\lambda_{\mathsf{cbv}}$ is closed, then either t is a value or $t\to_{\beta_v} u$ for some closed $u\in\lambda_{\mathsf{cbv}}$.*

Notations. We set some notations for both calculi and machines. Let $\to$ be a reduction relation. An evaluation sequence $e\colon t\to^* u$ is a possibly empty finite sequence of $\to$-steps from t to u, the length of which is noted $|e|$ (we also write $e\colon t\to^k u$ if $|e|=k$). If moreover $\to_a\subseteq\to$, then $|e|_a$ is the number of a-steps in e.

3 Part 1: The Source Calculus λ_{sou}

In this section, we extend λ_{cbv} with tuples, which are needed to define closure conversion, obtaining λ_{sou}, our source calculus.

Terms. The *source calculus* λ_{sou} defined in Fig. 3 adopts n-ary *tuples* $\vec{t}=(\!|t_1,..,t_n|\!)$, together with projections π_i on the i^{th} element[2]. Abstractions are now on *sequences* of variables $\widetilde{x}=x_1,..,x_n$. With a slight abuse, we also compact $(\!|t_1,..,t_n|\!)$ into $(\!|\vec{t}|\!)$, and write $(\!|\widetilde{t},u,\widetilde{s}|\!)$ (note that replacing sequences with tuples changes the meaning: $(\!|\widetilde{t},u,\widetilde{s}|\!)$ and $(\!|\vec{t},u,\vec{s}|\!)$ are different terms, and we shall need both notations). Both tuples and sequences of variables can be empty, that is, $\lambda.t$ and $(\!|\,|\!)$ are terms of λ_{sou}. *Values* now are abstractions *and* tuples *of values*—tuples of arbitrary terms are not values in general.

Notations and Conventions about Tuples and Sequences. We assume that in every sequence $\widetilde{x}$ all elements are distinct and, for brevity, we abuse notations and consider sequences of variables also as the sets of their elements, writing $x_i\in\widetilde{x}$, or $\mathtt{fv}(t)=\widetilde{x}$, or $\widetilde{x}\cup\widetilde{y}$. We set $\|\vec{t}\|:=n$ if $\vec{t}=(\!|t_1,..,t_n|\!)$ and call it the *length* of the tuple $\vec{t}$ (so, $\|(\!|\,|\!)\|=0$), and similarly for $\|\widetilde{x}\|$. Moreover, if $\widetilde{x}=x_1,..,x_n$ and $\vec{v}=(\!|v_1,..,v_n|\!)$ we then set $t\{\widetilde{x}\leftarrow\vec{v}\}:=t\{x_1\leftarrow v_1,..,x_n\leftarrow v_n\}$ (the *simultaneous* substitution). We write $\widetilde{x}\#\widetilde{y}$ when $\widetilde{x}$ and $\widetilde{y}$ have no element in common, and $\mathtt{fv}(t)\subseteq\widetilde{x}\#\widetilde{y}$ when moreover $\mathtt{fv}(t)\subseteq(\widetilde{x}\cup\widetilde{y})$.

Small-Step Operational Semantics. The β_v-rule can fire only if the argument is a tuple of values of the right length, and similarly for the π-rule. For instance, $(\lambda x.x)(\lambda y.yy)\not\to_{\beta_v}\lambda y.yy$, because one needs a unary tuple around the argument, that is, $(\lambda x.x)(\!|\lambda y.yy|\!)\to_{\beta_v}\lambda y.yy$. Note that evaluation contexts now enter projections and tuples, proceeding right-to-left. As it shall be the case for all calculi in this paper, the operational semantics $\to_{\mathsf{sou}}$ of λ_{sou} is *deterministic*: if $t\to_{\mathsf{sou}}u$ and $t\to_{\mathsf{sou}}s$ then $u=s$ and t is not a value. The proof is a routine induction (see [5, Appendix A]).

[1]The right-to-left order (adopted also in [24, 26]) induces a more natural presentation of the machines, but all our results could be restated using the left-to-right order.

[2]We do not define projections as tuple-unpacking abstractions because it would turn π-steps into β-steps and so blur the cost analysis (that counts β but not π-steps).

$$\begin{array}{rrcl} \textsc{Bags} & b,b' & ::= & \vec{x} \mid \vec{v} \\ \textsc{Terms} & t,u,s,r & ::= & x \mid tu \mid \pi_i t \mid \vec{t} \mid [\![\widetilde{y};\widetilde{x}.t|b]\!] \\ \textsc{Values} & v,v' & ::= & [\![\widetilde{y};\widetilde{x}.t|b]\!] \mid \vec{v} \\ \textsc{Eval ctxs} & C,C' & ::= & \langle\cdot\rangle \mid tC \mid Cv \mid \pi_i C \mid (\!|\widetilde{t},C,\widetilde{v}|\!) \end{array}$$

$$\begin{array}{l|l} [\![\widetilde{y};\widetilde{x}.t|\vec{v_1}]\!]\,\vec{v_2} \mapsto_{\mathtt{i}\beta_v} t\{\widetilde{y};\widetilde{x}\leftarrow\vec{v_1};\vec{v_2}\} \text{ if } \|\widetilde{y}\|=\|\vec{v_1}\| \text{ and } \|\widetilde{x}\|=\|\vec{v_2}\| & \dfrac{t\mapsto_a u}{C\langle t\rangle\to_a C\langle u\rangle}\ a\in\{\mathtt{i}\beta_v,\mathtt{i}\pi\} \\ \pi_i\,\vec{v} \mapsto_{\mathtt{i}\pi} v_i \quad \text{if } 1\leq i\leq\|\vec{v}\| & \to_{\mathtt{int}} := \to_{\mathtt{i}\beta_v}\cup\to_{\mathtt{i}\pi} \end{array}$$

Figure 4: The intermediate calculus $\lambda_{\mathtt{int}}$.

Clashes. In an untyped setting, there might be terms with *clashes*, that is, irreducible badly formed configurations such as $\pi_i(\lambda x.t)$. To exclude clashes without having to have types, we adopt a notion of clash-freeness, which would be ensured by any type system.

Definition 3.1 (Clashes, clash-free terms). A term t is a *clash* if it has shape $C\langle u\rangle$ where u has one of the following forms:

- *Clashing projection*: $u=\pi_i v$ and if $v=\vec{v}$ then $\|\vec{v}\|<i$;
- *Clashing abstraction*: $u=(\lambda\widetilde{x}.s)v$ and if $v=\vec{v}$ then $\|\widetilde{x}\|\neq\|\vec{v}\|$;
- *Clashing tuple*: $u=\vec{r}\,s$.

A term t is *clash-free* when, co-inductively, t is not a clash and if $t\to_{\mathsf{sou}}u$ then u is clash-free.

Note that clashes are normal forms. All the calculi and machines of the paper shall come with their notion of clash and clash-freeness, which shall be taken into account in statements and proofs but the definitions of which shall be omitted (they are in [5]).

Lemma 3.2 (λ_{sou} harmony). *If $t\in\lambda_{\mathsf{sou}}$ is closed and clash-free, then either t is a value or $t\to_{\mathsf{sou}}u$ for some closed and clash-free u.*

4 Part 1: the Intermediate Calculus $\lambda_{\mathtt{int}}$ and the Wrapping Transformation

In this section, we define the intermediate calculus $\lambda_{\mathtt{int}}$ and the wrapping translation from λ_{sou} to $\lambda_{\mathtt{int}}$. We discuss why the natural first attempt to show the correctness of the translation does not work, and solve the issue via a reverse translation from $\lambda_{\mathtt{int}}$ to λ_{sou}.

Terms of $\lambda_{\mathtt{int}}$. In $\lambda_{\mathtt{int}}$, defined in Fig. 4, abstractions $\lambda\widetilde{x}.t$ are replaced by *closures* $[\![\lambda\widetilde{y}.\lambda\widetilde{x}.t|b]\!]$, which are compactly noted $[\![\widetilde{y};\widetilde{x}.t|b]\!]$. The *bag* b of a closure can be of two forms $\vec{z}$ and $\vec{v}$, giving *variable closures* $[\![\widetilde{y};\widetilde{x}.t|\,\vec{z}]\!]$ and *evaluated closures* $[\![\widetilde{y};\widetilde{x}.t|\,\vec{v}]\!]$, which are both values. In a closure $[\![\widetilde{y};\widetilde{x}.t|b]\!]$, $\widetilde{y}$ and $\widetilde{x}$ verify $\widetilde{y}\#\widetilde{x}$ (i.e. no elements in common), and scope over the *body* t of the closure; $\widetilde{y}$ and $\widetilde{x}$ do not scope over b. The idea is that $\mathtt{fv}(\lambda\widetilde{x}.t)\subseteq\widetilde{y}$, so that $\lambda\widetilde{y}.\lambda\widetilde{x}.t$ is closed. The elements of b are meant to replace the variables $\widetilde{y}$ in t.

The rationale behind $\lambda_{\mathtt{int}}$ is understood by looking at the translation from λ_{sou} to $\lambda_{\mathtt{int}}$ in Fig. 5. Basically, every abstraction is closed and paired with the bag of its free variables. Evaluated closures $[\![\widetilde{y};\widetilde{x}.t|\,\vec{v}]\!]$ are not in the image of the translation, unless

Wrapping translation $\lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$

$$\begin{array}{rcl@{\qquad}rcl} \underline{x} & := & x & \underline{tu} & := & \underline{t}\,\underline{u} \\ \underline{\lambda\widetilde{x}.t} & := & [\![\widetilde{y};\widetilde{x}.\underline{t}\,|\,\overrightarrow{y}]\!] \quad \text{if } \mathtt{fv}(\lambda\widetilde{x}.t) = \widetilde{y} & & & \\ \underline{(\!|t_1,..,t_n|\!)} & := & (\!|\underline{t_1},..,\underline{t_n}|\!) & \underline{\pi_i t} & := & \pi_i \underline{t} \end{array}$$

Unwrapping (reverse) translation $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{sou}}$

$$\begin{array}{rcl@{\qquad}rcl} \lceil x \rceil & := & x & \lceil tu \rceil & := & \lceil t \rceil \lceil u \rceil \\ \lceil [\![\widetilde{y};\widetilde{x}.t\,|\,\overrightarrow{y}]\!] \rceil & := & \lambda\widetilde{x}.\lceil t \rceil & \lceil [\![\widetilde{y};\widetilde{x}.t\,|\,\overrightarrow{v}]\!] \rceil & := & \lambda\widetilde{x}.\lceil t \rceil\{\widetilde{y}\leftarrow\overrightarrow{\lceil v \rceil}\} \\ \lceil (\!|t_1,..,t_n|\!) \rceil & := & (\!|\lceil t_1 \rceil,..,\lceil t_n \rceil|\!) & \lceil \pi_i t \rceil & := & \pi_i \lceil t \rceil \end{array}$$

Figure 5: The translations $\underline{\cdot}\colon \lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$ (wrapping) and $\lceil\cdot\rceil\colon \lambda_{\mathtt{int}} \to \lambda_{\mathtt{sou}}$ (unwrapping).

$\mathtt{fv}(\lambda\widetilde{x}.t) = \emptyset$, which gives the closure $[\![;\widetilde{x}.t\,|\,(\!|\,|\!)]\!]$ that is both a variable and an evaluated closure. Evaluated closures are generated by the reduction rules, discussed after defining well-formedness.

Definition 4.1. A closure $[\![\widetilde{y};\widetilde{x}.t|b]\!]$ is *well-formed* if $\mathtt{fv}(t) \subseteq \widetilde{y}\#\widetilde{x}$, $\|b\| = \|\widetilde{y}\|$ and if b is a variable bag then $b = (\!|\widetilde{y}|\!)$. Terms $t \in \lambda_{\mathtt{int}}$ and evaluation contexts $C \in \lambda_{\mathtt{int}}$ are *well-formed* if all their closures are well-formed, and *prime* if moreover they are variable closures.

Operational Semantics. The intermediate variant $\to_{\mathtt{i}\beta_v}$ of the β_v-rule involves a well-formed evaluated closure $[\![\widetilde{y};\widetilde{x}.t|\overrightarrow{v_1}]\!]$ and an argument $\overrightarrow{v_2}$ of the same length as $\widetilde{x}$, and amounts to substitute the bag $\overrightarrow{v_1}$ on $\widetilde{y}$ and the argument $\overrightarrow{v_2}$ on $\widetilde{x}$. Substitution $t\{\widetilde{y};\widetilde{x}\leftarrow\overrightarrow{v_1};\overrightarrow{v_2}\}$ is defined as expected (see [5, Appendix B]) by performing $\{\widetilde{y}\leftarrow\overrightarrow{v_1}\}$ and $\{\widetilde{x}\leftarrow\overrightarrow{v_2}\}$ *simultaneously* and requires that $\mathtt{fv}(t) \subseteq \widetilde{y}\#\widetilde{x}$. Well-formed terms are stable by substitution, when defined, and the reduct of $\to_{\mathtt{i}\beta_v}$ is a (well-formed) term of $\lambda_{\mathtt{int}}$ because (by well-formedness) closures close their bodies, so the closing substitution generated by the step turns variable bags into value bags. Tuple projection is as in $\lambda_{\mathtt{sou}}$. Note that evaluation contexts do not enter closures. See [5, Appendix B] for the definition of clash(-freeness) for $\lambda_{\mathtt{int}}$. A term $t \in \lambda_{\mathtt{int}}$ is *good* if it is well-formed and clash-free.

The intermediate calculus $\lambda_{\mathtt{int}}$ is deterministic (if $t \to_{\mathtt{int}} u$ and $t \to_{\mathtt{int}} s$ then $u = s$ and t is not a value) and harmonic.

Lemma 4.2 (Harmony of $\lambda_{\mathtt{int}}$). *Let $t \in \lambda_{\mathtt{int}}$ be closed and good. Then either t is a value or $t \to_{\mathtt{int}} u$ for some closed good $u \in \lambda_{\mathtt{int}}$.*

Translation From Source to Intermediate, or Wrapping. The wrapping translation $\underline{\cdot}$ from $\lambda_{\mathtt{sou}}$ to $\lambda_{\mathtt{int}}$ takes a (possibly open) term $t \in \lambda_{\mathtt{sou}}$ and returns a term $\underline{t} \in \lambda_{\mathtt{int}}$, and it is defined in Fig. 5. As already mentioned, it turns abstractions into closures by closing them and pairing them with the bag of their free variables. It is extended to evaluation contexts C as expected, setting $\underline{\langle\cdot\rangle} := \langle\cdot\rangle$.

Lemma 4.3 (Properties of the translation $\lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$).

(1) Values*: if $v \in \lambda_{\mathtt{sou}}$ then $\underline{v}$ is a value of $\lambda_{\mathtt{int}}$.*
(2) Terms*: if $t \in \lambda_{\mathtt{sou}}$ then $\underline{t} \in \lambda_{\mathtt{int}}$ is well-formed and prime.*
(3) Contexts*: if $C \in \lambda_{\mathtt{sou}}$ then $\underline{C}$ is an evaluation context of $\lambda_{\mathtt{int}}$.*

Problem: Wrapping and Substitution Do not Commute. As source values and contexts are translated to their intermediate analogous, one may think that the translation preserves reduction steps: if $t \to_{\mathtt{sou}} u$ then $\underline{t} \to_{\mathtt{int}} \underline{u}$. But this is *false*, because the translation does not commute with substitution, as also discussed in [18, 38]. Indeed, in general $\underline{t}\{x\leftarrow\underline{v}\} \neq \underline{t\{x\leftarrow v\}}$: take $t := \lambda y.yx$ and a closed value v, we get the terms below, where $[\![x;y.yx|(\!|\underline{v}|\!)]\!] \neq [\![;y.y\underline{v}|(\!|\,|\!)]\!]$.

$$\begin{array}{rcl@{\qquad}rcl} \underline{t}\{x\leftarrow\underline{v}\} & = & [\![x;y.yx|(\!|\underline{x}|\!)]\!]\{x\leftarrow\underline{v}\} & = & [\![x;y.yx|(\!|\underline{v}|\!)]\!] \\ \underline{t\{x\leftarrow v\}} & = & \underline{\lambda y.yv} & = & [\![;y.y\underline{v}|(\!|\,|\!)]\!] \end{array}$$

The point being that $t\{x\leftarrow v\}$ is *closed* and so its wrapping is different from first wrapping t, which is instead *open*, and then closing the wrapped term $\underline{t}$ using $\{x\leftarrow\underline{v}\}$. Not only $\underline{t}\{x\leftarrow\underline{v}\} \neq \underline{t\{x\leftarrow v\}}$, they are not even related by $\to_{\mathtt{int}}$, or by the equational theory generated by $\to_{\mathtt{int}}$ (but they can be shown to be contextually equivalent).

The issue is that the translation targets prime terms (Lemma 4.3.2) but $\to_{\mathtt{int}}$ creates evaluated closures (which are not in the image of translation), i.e., *the reduct of a prime term (of $\lambda_{\mathtt{int}}$) may not be prime.* Take $t := \lambda y.yx$ again: $(\lambda x.t)(\!|v|\!) \to_{\mathtt{sou}} t\{x\leftarrow v\} = \lambda y.yv$ and $\underline{(\lambda x.t)(\!|v|\!)} = [\![;x.[\![x;y.yx|(\!|x|\!)]\!]\,|\,(\!|\,|\!)]\!](\!|\underline{v}|\!)$ is prime, but $\underline{(\lambda x.t)(\!|v|\!)} \to_{\mathtt{int}}$-reduces to the non-prime $[\![x;y.yx|(\!|\underline{v}|\!)]\!] = \underline{t}\{x\leftarrow\underline{v}\} \neq \underline{t\{x\leftarrow v\}}$.

Reverse Translation. The literature usually overcomes this problem by switching to a different approach, adopting a big-step semantics and a logical relation proof technique. One of our contributions is to show a direct solution, as done also by Bowman and Ahmed [18], but in a different way. The idea comes from the correctness of abstract machines, which is proved by projecting the machine on the calculus, rather than the calculus on the machine. Therefore, we define a reverse *unwrapping* translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$ and show that it smoothly preserves reduction steps.

The *unwrapping translation* $\lceil t \rceil$ of a well formed term $t \in \lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$ is defined in Fig. 5. It amounts to substitute the bag b for the sequence $\widetilde{y}$ of variables of a closure $[\![\widetilde{y};\widetilde{x}.t|b]\!]$. In contrast with wrapping, now unwrapping and substitution commute.

Proposition 4.4 (Commutation of substitution and the reverse translation). *If $t, \overrightarrow{v_1}, \overrightarrow{v_2} \in \lambda_{\mathtt{int}}$ and $\mathtt{fv}(t) \subseteq \widetilde{x}\#\widetilde{y}$, then $\lceil t\{\widetilde{x};\widetilde{y}\leftarrow\overrightarrow{v_1};\overrightarrow{v_2}\} \rceil = \lceil t \rceil\{\widetilde{x}\leftarrow\overrightarrow{\lceil v_1 \rceil}\}\{\widetilde{y}\leftarrow\overrightarrow{\lceil v_2 \rceil}\}$.*

Strong Bisimulation. From the commutation property (Prop. 4.4), it easily follows that the reverse translation projects and reflects rewrite steps and—if terms are closed—also normal forms. As a consequence, it is a termination-preserving strong bisimulation, possibly the strongest form of *correctness* for a program transformation.

Theorem 4.5 (Source-intermediate termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{int}}$ be closed and well-formed.*

(1) Projection*: for $a \in \{\beta_v, \pi\}$, if $t \to_{\mathtt{i}a} u$ then $\lceil t \rceil \to_a \lceil u \rceil$.*
(2) Halt*: t is $\to_{\mathtt{int}}$-normal if and only if $\lceil t \rceil$ is $\to_{\mathtt{sou}}$-normal.*
(3) Reflection*: for $a \in \{\beta_v, \pi\}$, if $\lceil t \rceil \to_a u$ then there exists $s \in \lambda_{\mathtt{int}}$ such that $t \to_{\mathtt{i}a} s$ and $\lceil s \rceil = u$.*
(4) Inverse*: if u is a source term then $\lceil \underline{u} \rceil = u$.*

Projection can be extended to sequences of steps. The reflection and inverse properties ensure that reflections of consecutive steps from the source can be composed, bypassing the problem with the wrapping translation, see the corollary below. Its proof only depends on the abstract properties of our notion of bisimulation, thus similar corollaries hold for all bisimulations in this paper.

Corollary 4.6 (Preservation of reduction steps). *If $t \to^k_{\mathtt{sou}} u$ then there exists $s \in \lambda_{\mathtt{int}}$ such that $\underline{t} \to^k_{\mathtt{int}} s$ and $\lceil s \rceil = u$.*

Proof. By induction on k. If $k = 0$ then $t = u$; taking $s := \underline{t}$, the inverse property gives $\lceil s \rceil = \lceil \underline{t} \rceil = t = u$. For $k > 0$, one has

$$
\begin{array}{rrcl}
\textsc{Projected vars} & p, p' & ::= & \pi_i \mathtt{w} \mid \pi_i \mathtt{s} \\
\textsc{Terms} & t, u, s, r & ::= & p \mid \pi_i t \mid \vec{t} \mid t\, u \mid [\![t|b]\!]_{n,m} \\
\textsc{Bags} & b, b' & ::= & \vec{p} \mid \vec{v} \\
\textsc{Values} & v, v' & ::= & [\![t|b]\!]_{n,m} \mid \vec{v} \\
\textsc{Eval Contexts} & C, C' & ::= & \langle\cdot\rangle \mid t\,C \mid C\,\vec{v} \mid \pi_i C \mid (\!|\widetilde{t}, C, \widetilde{v}|\!)
\end{array}
$$

Name elimination translation $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$

$$
\begin{array}{lll}
y_i\,{}^{\widetilde{y},\widetilde{x}} := \pi_i \mathtt{w} & (\!|t_1,..,t_n|\!)^{\widetilde{y},\widetilde{x}} := (\!|\, t_1\,{}^{\widetilde{y},\widetilde{x}}, .., \, t_n\,{}^{\widetilde{y},\widetilde{x}}\,|\!) \\
\pi_i t\,{}^{\widetilde{y},\widetilde{x}} := \pi_i\, t\,{}^{\widetilde{y},\widetilde{x}} & [\![\widetilde{z};\widetilde{w}.t|b]\!]^{\widetilde{y},\widetilde{x}} := [\![\, t\,{}^{\widetilde{z},\widetilde{w}} \mid b\,{}^{\widetilde{y},\widetilde{x}}\,]\!] \\
x_i\,{}^{\widetilde{y},\widetilde{x}} := \pi_i \mathtt{s} & t\,u\,{}^{\widetilde{y},\widetilde{x}} := t\,{}^{\widetilde{y},\widetilde{x}}\; u\,{}^{\widetilde{y},\widetilde{x}}
\end{array}
$$

Naming (reverse) translation $\lambda_{\mathtt{tar}} \to \lambda_{\mathtt{int}}$

$$
\begin{array}{lll}
\pi_i \mathtt{w}\,{}_{\widetilde{y},\widetilde{x}} := y_i & (t\,u)_{\widetilde{y},\widetilde{x}} := t_{\widetilde{y},\widetilde{x}}\; u_{\widetilde{y},\widetilde{x}} \\
\pi_i \mathtt{s}\,{}_{\widetilde{y},\widetilde{x}} := x_i & (\!|t_1,..,t_n|\!)_{\widetilde{y},\widetilde{x}} := (\!|\, t_1\,{}_{\widetilde{y},\widetilde{x}}, .., \, t_n\,{}_{\widetilde{y},\widetilde{x}}\,|\!) \\
(\pi_i t)_{\widetilde{y},\widetilde{x}} := \pi_i\, t\,{}_{\widetilde{y},\widetilde{x}} & [\![t|b]\!]_{n,m\,\widetilde{y},\widetilde{x}} := [\![\widetilde{z};\widetilde{w}.\, t_{\widetilde{z},\widetilde{w}} \mid b_{\widetilde{y},\widetilde{x}}]\!]
\end{array}
$$

with $\widetilde{z}\#\widetilde{w}$ fresh, $\|\widetilde{z}\| = n$, $\|\widetilde{w}\| = m$

Figure 6: The target calculus $\lambda_{\mathtt{tar}}$, and the translations $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$ (name elimination) and $\lambda_{\mathtt{tar}} \to \lambda_{\mathtt{int}}$ (naming).

$t \to_{\mathtt{sou}}^{k-1} u' \to_{\mathtt{sou}} u$. By *i.h.*, $\underline{t} \to_{\mathtt{int}}^{k-1} s'$ with $\lceil s' \rceil = u'$, so $\lceil s' \rceil \to_{\mathtt{sou}} u$; by reflection $s' \to_{\mathtt{int}} s$ for some s such that $\lceil s \rceil = u$. □

About the proof of the various points of Theorem 4.5, the closed hypothesis is used for the left-to-right direction of the halt property, which in turn is used to prove the reflection property. Projection is used instead in the right-to-left direction of the halt property. The inverse property is straightforward.

5 Part 1: Outline of the Target Calculus $\lambda_{\mathtt{tar}}$ and of Name Elimination

In this section, we quickly outline the *target calculus* $\lambda_{\mathtt{tar}}$ and the translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{tar}}$, dubbed *name elimination*, which, when composed with the wrapping translation of Section 4, provides the *closure conversion* transformation. The unsurprising details (definitions, statements, and proofs) are given in [5, Appendix C]. The ideas behind $\lambda_{\mathtt{tar}}$ and name elimination, defined in Fig. 6, are:

- *Variable binders*: replacing the sequences $\widetilde{y}$ and $\widetilde{x}$ of abstracted variables in closures $[\![\widetilde{y};\widetilde{x}.t|b]\!]$ with the special variables $\mathtt{w}$ and $\mathtt{s}$ (short for *wrapped* and *source*), standing for the tuples $\vec{v_1}$ and $\vec{v_2}$ meant to be substituted on $\widetilde{y}$ and $\widetilde{x}$;
- *Variable occurrences*: replacing every occurrence of a variable y_i or x_j in t with the *projected variables* $\pi_i\mathtt{w}$ or $\pi_j\mathtt{s}$, which are two special compound terms as $\mathtt{w}$ and $\mathtt{s}$ cannot appear without being paired with a projection.

We use p (for *projected variable*) to refer to either $\pi_i\mathtt{w}$ or $\pi_j\mathtt{s}$. The transformation is similar to switching to de Bruijn indices, except that—because of the already wrapped setting—the index simply refers to the composite binder of the closure, rather than to the nested binders above the variable occurrence. It also slightly differs from standard closure conversion: the latter would eliminate the $\widetilde{y}$ names but usually not the $\widetilde{x}$ ones. We eliminate both, as the total elimination of names shall induce a logarithmic speed-up for the abstract machine associated with $\lambda_{\mathtt{tar}}$, see Sect. 12.

The terminology *name elimination* refers to the fact that, after this transformation, a term uses only two variables, $\mathtt{w}$ and $\mathtt{s}$. As all closures would then have shape $[\![\mathtt{w};\mathtt{s}.t|b]\!]$, we simplify the notation and just write $[\![t|b]\!]$, with the implicit assumption that every closure now binds $\mathtt{w}$ and $\mathtt{s}$ in t. To define a reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ dubbed *naming* in Fig. 6, closures $[\![t|b]\!]_{n,m}$ are annotated with two natural numbers $n, m \in \mathbb{N}$, which record the length of the replaced sequences of variables $\widetilde{y}$ and $\widetilde{x}$. To simplify the notation, however, we shall omit these annotations when not relevant. The *body* of a closure $[\![t|b]\!]$ is t, and—as in $\lambda_{\mathtt{int}}$—closures are either variable closures, if $b = \widetilde{p}$, or evaluated closures, if $b = \vec{v}$. A term $t \in \lambda_{\mathtt{tar}}$ is *closed* if $\pi_i\mathtt{w}$ and $\pi_i\mathtt{s}$ do not occur out of closure bodies.

Name elimination is parametric in two lists $\widetilde{y}$ and $\widetilde{x}$ of abstracted variables, which intuitively are those of the closest enclosing closure being translated. In particular the two parametric lists change when the translation crosses the boundary of a closure in the $[\![\widetilde{z};\widetilde{w}.t|b]\!]^{\widetilde{y},\widetilde{x}}$ clause in Fig. 6. On closed terms, the translation is meant to be applied with empty parameter lists (as $t^{\epsilon,\epsilon}$).

Reduction $\to_{\mathtt{tar}} := \to_{\mathtt{t}\beta_v} \cup \to_{\mathtt{t}\pi}$ is defined in [5, Appendix C].

Results. We prove that name elimination, its reverse naming translation (from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$), and the composed naming-unwrapping reverse translation (from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{sou}}$) are termination-preserving strong bisimulations as in Theorem 4.5. The technical development is rather smooth and follows the structure of Section 4 without the subtleties related to the commutation with substitution.

More precisely, in [5, Appendix C], we show that the name elimination translation is a termination-preserving strong bisimulation between $\lambda_{\mathtt{int}}$ and $\lambda_{\mathtt{tar}}$. The two obtained strong bisimulation theorems—namely Theorem 4.5 and the latter concerning name elimination—do not compose, because the first one uses the reverse translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$, while the second one uses the direct translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{tar}}$. To solve the issue, we use the reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ dubbed *naming* in Fig. 6.

Unsurprisingly, naming is a termination-preserved strong bisimulation from the target calculus $\lambda_{\mathtt{tar}}$ to the intermediate one $\lambda_{\mathtt{int}}$.

Theorem 5.1 (Target-intermediate termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{tar}}$ be closed.*

1. Projection*: for $a \in \{\beta_v, \pi\}$, if $t \to_{\mathtt{t}a} u$ then $t_{\epsilon,\epsilon} \to_{\mathtt{i}a} u_{\epsilon,\epsilon}$.*
2. Halt*: t is $\to_{\mathtt{tar}}$-normal (resp. a value, resp. a clash) if and only if $t_{\epsilon,\epsilon}$ is $\to_{\mathtt{tar}}$-normal (resp. a value, resp. a clash).*
3. Reflection*: for $a \in \{\beta_v, \pi\}$, if $t_{\epsilon,\epsilon} \to_{\mathtt{i}a} u$ then there exists $s \in \lambda_{\mathtt{tar}}$ such that $t \to_{\mathtt{t}a} s$ and $s_{\epsilon,\epsilon} = u$.*
4. Inverse*: let $u \in \lambda_{\mathtt{int}}$, if $\mathtt{fv}(u) \subseteq \widetilde{y}\#\widetilde{x}$ then $u^{\widetilde{y},\widetilde{x}}_{\widetilde{y},\widetilde{x}} =_\alpha u$.*

The following theorem states that the reverse transformation $\lceil t_{\epsilon,\epsilon} \rceil$ of closure conversion (obtained by composing naming and unwrapping) is a termination-preserving strong bisimulation. It is our final *correctness* result for closure conversion. The proof is obtained by simply composing the two bisimulations results for unwrapping (Theorem 4.5) and naming (Theorem 5.1).

Theorem 5.2 (Target-source termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{tar}}$ be closed.*

1. Projection*: for $a \in \{\beta_v, \pi\}$, if $t \to_{\mathtt{t}a} u$ then $\lceil t_{\epsilon,\epsilon} \rceil \to_a \lceil u_{\epsilon,\epsilon} \rceil$.*
2. Halt*: t is $\to_{\mathtt{tar}}$-normal if and only if $\lceil t_{\epsilon,\epsilon} \rceil$ is $\to_{\mathtt{sou}}$-normal.*

(3) Reflection: *for* $a \in \{\beta_v, \pi\}$, *if* $\lceil t \rceil_{\epsilon,\epsilon} \to_a u$ *then there exists* $s \in \lambda_{\mathtt{tar}}$ *such that* $t \to_{ta} s$ *and* $\lceil s \rceil_{\epsilon,\epsilon} = u$.

(4) Inverse: *if* $u \in \lambda_{\mathtt{sou}}$ *then* $\lceil \underline{u}^{\epsilon,\epsilon} \rceil_{\epsilon,\epsilon} = u$.

6 Part 2 Preliminaries: Abstract Machines

This section starts the second part of the paper. Here, we introduce the terminology and the form of implementation theorem that we adopt for our abstract machines, along the lines of Accattoli and co-authors [2, 6, 13], here adapted to handle clashes.

Abstract Machines Glossary. An *abstract machine* for a strategy (i.e. a deterministic reduction) $\to_{\mathtt{str}}$ of a calculus λ_{cal} is a quadruple $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ where $(\mathtt{States}, \rightsquigarrow)$ is a deterministic labeled transition system with transitions $\rightsquigarrow$ partitioned into *principal transitions* $\rightsquigarrow_{pr}$, corresponding to the steps of the strategy and labeled with the labels of the rewrite rules in λ_{cal} (here variants of β_v and π steps), and *overhead transitions* $\rightsquigarrow_{oh}$, that take care of the various tasks of the machine (searching, substituting, and α-renaming), together with two functions:

- *Initialization* $\cdot^\circ : \lambda_{cal} \to \mathtt{States}$ turns λ_{cal}-terms into states;
- *Read-back* $\underline{\cdot} : \mathtt{States} \to \lambda_{cal}$ turns states into λ_{cal}-terms and satisfies the constraint $\underline{t^\circ} = t$ for every λ_{cal}-term t.

A state $q \in \mathtt{States}$ is composed by the *active term* t, and some data structures. A state q is *initial* for t if $t^\circ = q$. A state is *final* if no transitions apply; final states are partitioned into *successful* and *clash* states. A *run* r is a possibly empty sequence of transitions. For runs, we use notations such as $|r|$ as for evaluation sequences (Sect. 2). An *initial run* (*from* t) is a run from an initial state t°. A state q is *reachable* if it is the target state of an initial run.

Abstract machines manipulate *pre-terms*, that is, terms without implicit α-renaming, even if for simplicity we keep calling them terms. In that setting, we write t^α in a state q for a *fresh renaming* of t, i.e. t^α is α-equivalent to t but all of its bound variables are fresh (with respect to those in t and in the other components of q).

Implementation Theorem, Abstractly. We now define when a machine *implements* the strategy $\to_{\mathtt{str}}$ of a calculus λ_{cal}, abstracting and generalizing the setting of the previous sections.

Definition 6.1 (Machine implementation). A machine $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ *implements the strategy* $\to_{\mathtt{str}}$ of a calculus λ_{cal} when given a $t \in \lambda_{cal}$ the following holds:

(1) *Runs to evaluations*: for any $\mathtt{M}$-run $r : t^\circ \rightsquigarrow^* q$ there exists a $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} \underline{q}$. Additionally, if q is a successful state then $\underline{q}$ is a clash-free $\to_{\mathtt{str}}$-normal form.

(2) *Evaluations to runs*: for every $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} u$ there exists a $\mathtt{M}$-run $r : t^\circ \rightsquigarrow^* q$ such that $\underline{q} = u$. Additionally, if u is a clash-free $\to_{\mathtt{str}}$-normal form then there exists a successful state q' such that $q \rightsquigarrow^*_{oh} q'$.

(3) *Principal matching*: in both previous points the number $|e|_l$ of steps with label l in the evaluation e is exactly the number $|r|_l$ of principal l transitions in r, i.e. $|e|_l = |r|_l$.

Next, we give sufficient conditions that a machine and a (deterministic) strategy have to satisfy in order for the former to implement the latter, what we call *an implementation system.*

M-Closures $c ::= (t, E)$ Local Envs $E ::= \epsilon \mid [x \leftarrow c] : E'$ Stacks $S ::= \epsilon \mid \bullet c : S \mid \circ c : S$

M-Clos.		Stack	Trans.	M-Clos.	Stack
$t\,u$	E	S	$\rightsquigarrow_{sea_1}$	$u \mid E$	$\circ(t, E) : S$
$\lambda x.t$	E	$\circ c : S$	$\rightsquigarrow_{sea_2}$	c	$\bullet(\lambda x.t, E) : S$
$\lambda x.t$	E	$\bullet c : S$	$\rightsquigarrow_{\beta_v}$	$t \mid [x \leftarrow c] : E$	S
x	E	S	$\rightsquigarrow_{sub}$	$E(x)$	S

Figure 7: The local abstract machine (LAM) for $\lambda_{\mathtt{cbv}}$.

Definition 6.2 (Implementation system). A machine $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ and a strategy $\to_{\mathtt{str}}$ form an *implementation system* if:

(1) *Overhead transparency*: $q \rightsquigarrow_{oh} q'$ implies $\underline{q} = \underline{q'}$;

(2) *Principal projection*: $q \rightsquigarrow_{pr} q'$ implies $\underline{q} \to_{\mathtt{str}} \underline{q'}$ and the two have the same label;

(3) *Overhead termination*: $\rightsquigarrow_{oh}$ terminates;

(4) *Halt*: $\mathtt{M}$ successful states read back to clash-free $\to_{\mathtt{str}}$-normal forms, and clash states to clashes of λ_{cal}.

Via a simple lemma for the *evaluation to runs* part (in [5, Appendix D]), we obtain the following abstract implementation theorem.

Theorem 6.3 (Sufficient condition for implementations). *Let* $\mathtt{M}$ *be a machine and* $\to_{\mathtt{str}}$ *be a strategy forming an implementation system. Then,* $\mathtt{M}$ *implements* $\to_{\mathtt{str}}$ *(in the sense of Definition 6.1).*

Local Environments. We overview one the two main forms of environments for abstract machines, the *local* one (as opposed to *global*), for the archetypal call-by-value calculus $\lambda_{\mathtt{cbv}}$ of Sect. 2, to compare it later with the Source TAM for $\lambda_{\mathtt{sou}}$. The terminology local/global is due to [23], and the two techniques are analyzed e.g. (for call-by-name and call-by-need) in [4] but they are folklore.

The *local abstract machine* (LAM), a machine with local environments for $\lambda_{\mathtt{cbv}}$, is defined in Fig. 7. It is a right-to-left variant of the CEK machine [22]. It uses a *stack* to search for β_v-redexes, filling it with entries that encode an evaluation context of $\lambda_{\mathtt{cbv}}$. The substitutions triggered by the encountered β_v-redexes are delayed and stored in environments. There is one principal transition $\rightsquigarrow_{\beta_v}$, of label β_v, and three overhead transitions $\rightsquigarrow_{sea_1}$, $\rightsquigarrow_{sea_2}$, and $\rightsquigarrow_{sub}$ realizing the search for β_v-redexes and substitution. Contrary to the global approach, the LAM has many environments E, paired with terms so as to form *m-closures* c. In fact, a local environment E is itself a list of pairs of variables and m-closures: m-closures and environments are defined by mutual induction. A *state* is a triple (t, E, S) where (t, E) is the m-closure of an active term t and a local environment E, and S is a stack. Initial states have shape (t, ϵ, ϵ).

M-Closures are called in this way because when evaluating a closed term an invariant ensures that $\mathtt{fv}(t) \subseteq \mathtt{dom}(E)$ for any m-closure (t, E) in a reachable state (including the active one), where the domain $\mathtt{dom}(E)$ of an environment E is the set of variables on which it has a substitution. The use of m-closures allows one to avoid α-renaming (and copying code) in transition $\rightsquigarrow_{sub}$, at the price of using many environments, thus using space anyway. The duplication of E in transition $\rightsquigarrow_{sea_1}$ is where different implementation approaches (shared *vs* flat) play a main role. With shared environments (whose simplest implementation is as linked lists), the duplication only duplicates the pointer to the environment, not the whole environment. With flat environments (whose simplest

implementation is as arrays), duplication is an actual duplication of the array; we shall further discuss the duplication of flat environments when discussing the complexity of abstract machines.

7 Part 2: the Source TAM for λ_{sou}

Here we present a machine with local environments, the *source tupled abstract machine* (Source TAM) for the source calculus λ_{sou}. We adopt local rather than global environments to have m-closures, to then show that wrapping removes their need, in the next section.

The Source TAM is defined in Fig. 8. M-Closures carry a flag $\blacklozenge \in \{\circ, \bullet\}$, where $\circ$ stands for *non-(completely)-evaluated* and $\bullet$ for *evaluated*. In λ_{sou}, values have a tree structure, the leaves of which are abstractions. The evaluated m-closures $\bullet c$ of the Source TAM have a similar tree structure, plus local environments. A *state* is a couple $(\blacklozenge c \mid S)$, where $\blacklozenge c$ is a m-closure and S is a stack. The active m-closure is also flagged, naturally inducing a partition of transitions in two blocks. Stack entries are flagged m-closures, π_i and *partially evaluated tuples* $((\widetilde{t}, \downarrow, \widetilde{\bullet c}), E)$ ($\widetilde{t}$ are non-evaluated terms).

The *initialization* of t is the *initial state* $t^\circ := (\circ(t, \epsilon) \mid \epsilon)$. *Successful states* are $(\bullet c \mid \epsilon)$, i.e., an evaluated m-closure and an empty stack. Clash and clash-free states are defined in [5, Appendix E].

Transitions. The union of all the transitions of the Source TAM is noted $\leadsto_{\text{STAM}}$. The principal transitions are $\leadsto_{\bullet\beta_v}$ and $\leadsto_{\bullet\pi}$, of label β_v and π, all the other transitions are overhead ones. If $\widetilde{x} = x_1, .., x_n$ and $\widetilde{\bullet c} = (\!|\bullet c_1, .., \bullet c_n|\!)$, we set $[\widetilde{x} \leftarrow \widetilde{\bullet c}] := [x_1 \leftarrow \bullet c_1] .. [x_n \leftarrow \bullet c_n]$; this notation is used in transition $\leadsto_{\bullet\beta_v}$. In presence of tuples, there are additional overhead transitions. If the active m-closure is:

- $\circ(\pi_i t, E)$ then $\leadsto_{\circ sea_2}$ triggers the evaluation of t and the projection π_i goes on the stack (to trigger transition $\leadsto_{\bullet\pi}$);
- $\circ(\lambda\widetilde{x}.t, E)$ then $\leadsto_{\circ sea_5}$ flips the flag to $\bullet$ (weak evaluation);
- $\circ((\!|)\!|, E)$ then $\leadsto_{\circ sea_4}$ changes the flag to $\bullet$, discarding E;
- $\circ((\!|. ., t_n|\!), E)$ then $\leadsto_{\circ sea_3}$ evaluates its elements right-to-left, adding a partially evaluated tuple to the stack $((\!|. ., \downarrow|\!), E)$.
- $\bullet c$ then the behavior depends on the first element of the stack. If it is a partially evaluated tuple $((\!|. ., t, \downarrow, . .|\!), E)$ then $\leadsto_{\bullet sea_6}$ swaps $\bullet c$ and the next element t in the tuple (duplicating E), similarly to $\leadsto_{\bullet sea_1}$. If the tuple on the stack is $((\!|\downarrow, . .|\!), E)$ then $\leadsto_{\bullet sea_3}$ plugs $\bullet c$ on $\downarrow$ forming a new evaluated m-closure.

The name of transitions $\leadsto_{\bullet sea_1}$ and $\leadsto_{\bullet sea_3}$ stresses that they do the dual job of $\leadsto_{\circ sea_1}$ and $\leadsto_{\circ sea_3}$—such a duality shall be exploited in the complexity analysis of Sect. 11. Transition $\leadsto_{\bullet sea_6}$ has no dual (it is not named $\leadsto_{\bullet sea_2}$ because it is not the dual of $\leadsto_{\circ sea_2}$).

Invariant and Read Back. Here is an invariant of the Source TAM.

Lemma 7.1 (m-closure invariant). *Let q be a Source TAM reachable state and $\blacklozenge c = \blacklozenge(t, E)$ be a m-closure or $((\!|. ., t, . ., \downarrow, \widetilde{\bullet c}|\!), E)$ be a stack entry in q. Then $\mathsf{fv}(t) \subseteq \mathsf{dom}(E)$.*

The read-back $\underline{\cdot}$ of the Source TAM to λ_{sou} is defined in Fig. 8. M-Closures and states read back to terms, stacks read back to evaluation contexts. In the read back of partially evaluated m-closures $((\!|\widetilde{t}, \downarrow, \widetilde{\bullet c}|\!), E)$ on the stack, E spreads on the non-evaluated terms $\widetilde{t}$ (the m-closures in $\widetilde{\bullet c}$ have their own environments on their leaves).

Lemma 7.2 (Read-back properties).

(1) *$\underline{\bullet c}$ is a value of λ_{sou} for every $\bullet c$ of the Source TAM.*
(2) *$\underline{S}$ is an evaluation context of λ_{sou} for any Source TAM stack S.*

Implementation Theorem. According to the recipe in Sect. 6, we now prove the properties inducing the implementation theorem. The read-back to evaluation contexts (Lemma 7.2) is used for principal projection. The m-closure invariant (Lemma 7.1) is used in the proof of the halt property, to prove that the machine is never stuck on the left-hand side of a $\leadsto_{\circ sub}$ transition (see [5, Appendix E]). Overhead termination is proved via a measure, developed in Sect. 11, which gives a bound on the number of overhead transitions.

Theorem 7.3. *The Source TAM and λ_{sou} form an implementation system (as in Def. 6.2), hence the Source TAM implements $\to_{\mathsf{sou}}$.*

8 Part 2: the Int TAM for $\lambda_{\mathtt{int}}$

Here we present the *intermediate tupled abstract machine* (Int TAM) for the intermediate calculus $\lambda_{\mathtt{int}}$. Its feature is a new way of handling environments, resting on the strong properties of $\lambda_{\mathtt{int}}$.

The Int TAM is defined in Fig. 9. The flags will be explained after an overview of the new aspects of the machine. Mostly, the Int TAM behaves as the Source TAM by just having removed the structure of m-closures. The principal transitions are $\leadsto_{\bullet\beta_v}$ and $\leadsto_{\bullet\pi}$, of label β_v and π, all the other transitions are overhead. There are two new aspects: transition $\leadsto_{\circ sub_c}$ that evaluates variable bags and the use of *stackable environments* rather than *local* environments and m-closures. The machine is akin to those with global environments, except that no α-renaming is needed, as in the local approach.

Evaluating Bags. In the Int TAM, abstractions $\lambda\widetilde{x}.t$ are replaced by non-evaluated prime closures $\circ[\![\widetilde{y}; \widetilde{x}.t | (\!|\widetilde{y}|\!)]\!]$. The new transition $\leadsto_{\circ sub_c}$ (replacing $\leadsto_{\circ sea_5}$) substitutes on all variables in $\widetilde{y}$ in one shot, producing the evaluated closure $\bullet[\![\widetilde{y}; \widetilde{x}.t | E(\widetilde{y})]\!]$ where $E(\widetilde{y}) = (\!|E(y_1), .., E(y_n)|\!)$ if $\widetilde{y} = y_1, .., y_n$, and E is the (global) environment.

Stackable Environments. Transitions $\leadsto_{\bullet\beta_v}$ and the new $\leadsto_{\bullet sea_7}$ encapsulate a second new aspect, *stackable environments*. Indeed:

- *Closure bodies and environments*: when the machine encounters the analogous of a $\to_{\mathtt{i}\beta_v}$-redex $[\![\widetilde{y}; \widetilde{x}.t | \overrightarrow{v_1}]\!]\ \overrightarrow{v_2}$, the new entries $[\widetilde{y} \leftarrow \overrightarrow{\bullet v_1}]\ [\widetilde{x} \leftarrow \overrightarrow{\bullet v_2}]$ of the environment created by transition $\leadsto_{\bullet\beta_v}$ in the machine are all that is needed to evaluate t, because the free variables of t are all among $\widetilde{y}$ and $\widetilde{x}$. Thus, the environment E that is active before firing the redex is useless to evaluate t, and can be removed after transition $\leadsto_{\bullet\beta_v}$.
- *Stackability*: E is not garbage collected but pushed on the new *activation stack*, along with the ordinary stack S (now called *constructor stack*) which holds non-evaluated terms with variables in $\mathsf{dom}(E)$. This is still done by transition $\leadsto_{\bullet\beta_v}$.
- *Popping*: when the body t of the closure has been evaluated, the focus is on a value $\bullet v$ and the constructor stack is empty. The activation stack has the pair (S, E) that was active before firing the β_v-redex. The machine throws away the current environment $E' := [\widetilde{y} \leftarrow \overrightarrow{\bullet v_1}][\widetilde{x} \leftarrow \overrightarrow{\bullet v_2}]$, since $\widetilde{y}$ and $\widetilde{x}$ have no occurrences out of t, and reactivates the pair (S, E), to keep evaluating terms in S. This is done by the new transition $\leadsto_{\bullet sea_7}$.

Flags. The Int TAM evaluates (well-formed) terms of $\lambda_{\mathtt{int}}$ decorated (only on top) with a flag $\blacklozenge \in \{\circ, \bullet\}$: $\circ t$ denotes that t has not been evaluated yet, while $\bullet t$ denotes that t has been evaluated. The results of evaluation are values, thus the $\bullet$ flag shall be associated to

Eval. m-clos.
$$\bullet c ::= \bullet(\lambda\widetilde{x}.t, E) \mid \overbrace{(\!|\bullet c_1, .., \bullet c_n|\!)}^{\vec{\bullet c}} \quad n \geq 0$$
Flagged m-clos.
$$\circledcirc c ::= \bullet c \mid \circ(t, E)$$
Local envs
$$E, E' ::= \epsilon \mid [x \leftarrow \bullet c] : E$$
St. entries
$$S_{en} ::= \circledcirc c \mid \pi_i \mid ((\widetilde{t}, \downarrow, \widetilde{\bullet c}|\!|), E)$$
Stacks
$$S, S' ::= \epsilon \mid S_{en} : S$$
States
$$q, q' ::= (\circledcirc c \mid S)$$

m-closure			Stack	Trans.	m-closure			Stack	
$\circ$	tu	E	S	$\leadsto_{\circ sea_1}$	$\circ$	u	E	$\circ(t, E) : S$	
$\circ$	$\pi_i t$	E	S	$\leadsto_{\circ sea_2}$	$\circ$	t	E	$\pi_i : S$	
$\circ$	$(\!\|., t_n\|\!)$	E	S	$\leadsto_{\circ sea_3}$	$\circ$	t_n	E	$((\!\|., \downarrow\|\!), E) : S$	
$\circ$	$(\!\|\|\!)$	E	S	$\leadsto_{\circ sea_4}$	$\bullet$	$(\!\|\|\!)$	ϵ	S	
$\circ$	$\lambda\widetilde{x}.t$	E	S	$\leadsto_{\circ sea_5}$	$\bullet$	$\lambda\widetilde{x}.t$	E	S	
$\circ$	x	E	S	$\leadsto_{\circ sub}$		$E(x)$		S	
	$\bullet c$		$\circ(t, E) : S$	$\leadsto_{\bullet sea_1}$	$\circ$	t	E	$\bullet c : S$	
	$\bullet c$		$((\!\|., t, \downarrow, ..\|\!), E) : S$	$\leadsto_{\bullet sea_6}$	$\circ$	t	E	$((\!\|., \downarrow, \bullet c, ..\|\!), E) : S$	
	$\bullet c$		$((\!\|\downarrow, ..\|\!), E) : S$	$\leadsto_{\bullet sea_3}$		$(\!\|\bullet c, ..\|\!)$		S	
$\bullet$	$\lambda\widetilde{x}.t$	E	$\vec{\bullet c} : S$	$\leadsto_{\bullet \beta_v}$	$\circ$	t	$[\vec{x} \leftarrow \vec{\bullet c}] : E$	S	(∗)
	$\vec{\bullet c}$		$\pi_i : S$	$\leadsto_{\bullet \pi}$		$\bullet c_i$		S	(#)

Side conditions: (∗) if $\|\widetilde{x}\| = \|\vec{\bullet c}\|$ (#) if $1 \leq i \leq \|\vec{\bullet c}\|$

States read back $\underline{\ }$ (to terms of λ_{sou})
$$\underline{(\circledcirc c \mid S)} := \underline{S}\langle \underline{\circledcirc c} \rangle$$

M-Closures read-back $\underline{\ }$ (to terms of λ_{sou})
$$\underline{\circledcirc(t, \epsilon)} := t \qquad \underline{(\!|\bullet c_1, .., \bullet c_n|\!)} := (\!|\underline{\bullet c_1}, .., \underline{\bullet c_n}|\!)$$
$$\underline{\circledcirc(t, [x \leftarrow \bullet c] : E)} := \underline{(t\{x \leftarrow \bullet \underline{c}\}, E)}$$

Stacks read-back $\underline{\ }$ (to evaluation contexts of λ_{sou})
$$\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{\circ(t, E) : S} := \underline{S}\langle \circ(t, E)\langle\cdot\rangle\rangle \qquad \underline{\bullet c : S} := \underline{S}\langle\langle\cdot\rangle \underline{\bullet c}\rangle$$
$$\underline{\pi_i : S} := \underline{S}\langle \pi_i\langle\cdot\rangle\rangle \qquad \underline{((\!|t_1, .., t_n, \downarrow, \widetilde{\bullet c}|\!), E) : S} := \underline{S}\langle(\!|\underline{(t_1, E)}, .., \underline{(t_n, E)}, \langle\cdot\rangle, \underline{\widetilde{\bullet c}}|\!)\rangle$$

Figure 8: The source tupled abstract machine (Source TAM).

Stackable envs $E, E' ::= \epsilon \mid [x \leftarrow \bullet v] : E$
Constructor stacks $S, S' ::= \epsilon \mid \circ t : S \mid \bullet v : S \mid \pi_i : S \mid (\!|\widetilde{\circ t}, \downarrow, \widetilde{\bullet v}|\!) : S$
Activation stacks $A, A' ::= \epsilon \mid (S, E) : A$
States $q, q' ::= (\circledcirc t \mid S \mid E \mid A)$

Focus	Cons. stack	Env	Act. stack	trans.	Focus	Cons. stack	Env	Act. stack	
$\circ(tu)$	S	E	A	$\leadsto_{\circ sea_1}$	$\circ u$	$\circ t : S$	E	A	
$\circ \pi_i t$	S	E	A	$\leadsto_{\circ sea_2}$	$\circ t$	$\pi_i : S$	E	A	
$\circ(\!\|t_1, .., t_n\|\!)$	S	E	A	$\leadsto_{\circ sea_3}$	$\circ t_n$	$(\!\|\circ t_1, .., \downarrow\|\!) : S$	E	A	
$\circ(\!\|\|\!)$	S	E	A	$\leadsto_{\circ sea_4}$	$\bullet(\!\|\|\!)$	S	E	A	
$\circ[\![\widetilde{y}; \widetilde{x}.t \mid (\!\|\widetilde{y}\|\!)]\!]$	S	E	A	$\leadsto_{\circ sub_c}$	$\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid E(\widetilde{y})]\!]$	S	E	A	
$\circ x$	S	E	A	$\leadsto_{\circ sub_v}$	$E(x)$	S	E	A	
$\bullet v$	$\circ t : S$	E	A	$\leadsto_{\bullet sea_1}$	$\circ t$	$\bullet v : S$	E	A	
$\bullet v$	$(\!\|., \circ t, \downarrow, ..\|\!) : S$	E	A	$\leadsto_{\bullet sea_6}$	$\circ t$	$(\!\|., \downarrow, \bullet v, ..\|\!) : S$	E	A	
$\bullet v$	$(\!\|\downarrow, ..\|\!) : S$	E	A	$\leadsto_{\bullet sea_3}$	$\bullet(\!\|\bullet v, ..\|\!)$	S	E	A	
$\bullet \vec{v}$	$\pi_i : S$	E	A	$\leadsto_{\bullet \pi}$	$\bullet v_i$	S	E	A	if $1 \leq i \leq \|\vec{v}\|$
$\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v_1}]\!]$	$\vec{\bullet v_2} : S$	E	A	$\leadsto_{\bullet \beta_v}$	$\circ t$	ϵ	$[\widetilde{y} \leftarrow \vec{\bullet v_1}][\widetilde{x} \leftarrow \vec{\bullet v_2}]$	$(S, E) : A$	if $\|\widetilde{y}\| = \|\vec{\bullet v_1}\|$, $\|\widetilde{x}\| = \|\vec{\bullet v_2}\|$
$\bullet v$	ϵ	E	$(S, E') : A$	$\leadsto_{\bullet sea_7}$	$\bullet v$	S	E'	A	

Read back $\underline{\ }$ to λ_{int}

Flagged terms $\underline{\circ t} := t \qquad \underline{\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v}]\!]} := [\![\widetilde{y}; \widetilde{x}.t \mid \underline{\vec{\bullet v}}]\!] \qquad \underline{\bullet(\!|\bullet v_1, \ldots, \bullet v_n|\!)} := (\!|\underline{\bullet v_1}, \ldots, \underline{\bullet v_n}|\!)$

Constructor stacks $\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{\pi_i : S} := \underline{S}\langle \pi_i\langle\cdot\rangle\rangle \qquad \underline{\bullet v : S} := \underline{S}\langle\langle\cdot\rangle \underline{\bullet v}\rangle \qquad \underline{\circ t : S} := \underline{S}\langle t\langle\cdot\rangle\rangle \qquad \underline{(\!|\widetilde{\circ t}, \downarrow, \widetilde{\bullet v}|\!) : S} := \underline{S}\langle(\!|\widetilde{t}, \langle\cdot\rangle, \underline{\widetilde{\bullet v}}|\!)\rangle$

Act. stacks $\underline{\epsilon} := \langle\cdot\rangle \quad \underline{(S, E) : A} := \underline{A}\langle \underline{S}\sigma_E\rangle$ | States $\underline{(\circledcirc t \mid S \mid E \mid A)} := \underline{A}\langle \underline{S}\sigma_E\langle \underline{\circledcirc t}\sigma_E\rangle\rangle$ | Env-induced subst. $\sigma_{[\vec{x} \leftarrow \vec{\bullet v}]} := \{\widetilde{x}; \epsilon \leftarrow \underline{\vec{\bullet v}}; (\!|\|\!)\}$

Figure 9: The intermediate tupled abstract machine (Int TAM).

values only, and an invariant shall ensure that every evaluated value $\bullet v$ in a reachable state is closed. In $\circ t$, t has no flags, and non-flagged sub-terms are implicitly considered as flagged with $\circ$. Every evaluated value carries a $\bullet$ flag, so in $\bullet v$ there are in general other $\bullet$ flags (when it is a tuple or the bag of the closure). Moreover, evaluated closures shall have shape $\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v}]\!]$, that is, they have an additional (redundant) $\circ$ flag on their body (it shall be used in section Sect. 12 for the complexity analysis). The machine is started on *prime* terms of λ_{int} (because wrapped terms of λ_{sou} are prime, Lemma 4.3), thus all closures in an initial state have shape $[\![\widetilde{y}; \widetilde{x}.t \mid (\!|\widetilde{y}|\!)]\!]$. In fact, all non-evaluated closures $\circ[\![\widetilde{y}; \widetilde{x}.t \mid b]\!]$ in reachable states shall always be prime (that is, such that $b = (\!|\widetilde{y}|\!)$): we prove this invariant and we avoid assuming the general shape of non-evaluated closures, which would require additional and never used transitions.

The *initialization* t° of $t \in \lambda_{\mathsf{int}}$ is given by the *initial state* $(\circ t \mid \epsilon \mid \epsilon \mid \epsilon)$ where t is closed and prime. *Successful states* have shape $(\bullet v \mid \epsilon \mid E \mid \epsilon)$. Clash states are defined in [5, Appendix F].

Invariants and Read Back. Here are the invariants of the Int TAM.

Lemma 8.1 (Invariants). *Let $q = (\circledcirc t \mid S \mid E \mid A)$ be a Int TAM reachable state.*

1. Well-formedness*: all closures in q are well-formed.*
2. Closed values*: every value $\bullet v$ in q is closed.*
3. Closure*: $\mathtt{fv}(t) \cup \mathtt{fv}(S) \subseteq \mathtt{dom}(E)$ and $\mathtt{fv}(S') \subseteq \mathtt{dom}(E')$ for every entry (S', E') of the activation stack A.*

The read back $\underline{\ }$ is defined in Fig. 9. Flagged terms $\circledcirc t$ and states q read back to terms of λ_{int}, the constructor and activation stacks S and A read back to evaluation contexts of λ_{int}. The read back of A and q is based on a notion of meta-level *environment-induced (simultaneous) substitution* σ_E, defined in Fig. 9 and applied to terms (as read back of flagged terms) and evaluation contexts (as read back of stacks); meta-level substitutions are extended to evaluation contexts as expected, the definition is in [5, Appendix F].

The implementation theorem is proved following the schema used for the Source TAM, see [5, Appendix F]. Overhead transparency for the additional transition $\leadsto_{\bullet sea_7}$ relies on the closed values invariant (Lemma 8.1).

Tupled envs
$E, E' ::= \overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}$

Tupled envs lookup ($\mathtt{v} \in \{\mathtt{w}, \mathtt{s}\}$)
$(\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}})(\pi_i \mathtt{v}) := (\overrightarrow{\bullet v_{\mathtt{v}}})_i$ if $1 \le i \le \|\overrightarrow{\bullet v_{\mathtt{v}}}\|$

Focus	Co.	En	Ac.	trans.	Focus	Co.	En	Act. st.
$\circ[\![t \mid \overrightarrow{p}]\!]$	S	E	A	$\rightsquigarrow_{\circ sub_c}$	$\bullet[\![\circ t \mid \overrightarrow{E(p)}]\!]$	S	E	A
$\circ p$	S	E	A	$\rightsquigarrow_{\circ sub_v}$	$E(p)$	S	E	A
$\bullet[\![\circ t \mid \overrightarrow{\bullet v_1}]\!]$	$\overrightarrow{\bullet v_2} : S$	E	A	$\rightsquigarrow_{\bullet\beta_v}$	$\circ t$	ϵ	$\overrightarrow{\bullet v_1}; \overrightarrow{\bullet v_2}$	$(S, E) : A$

Side conditions: if look-up is defined in $\rightsquigarrow_{\circ sub_c}$ and $\rightsquigarrow_{\circ sub_v}$.

Read back $\underline{\cdot}$ to $\lambda_{\mathtt{tar}}$

Flagged terms $\underline{\bullet[\![\circ t \mid \overrightarrow{\bullet v}]\!]} := [\![t \mid \overrightarrow{\bullet \underline{v}}]\!]$

Env-induced subst. $\underline{\sigma_{\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}}} := \{\!\{\mathtt{w}; \mathtt{s} \leftarrow \overrightarrow{\bullet \underline{v}_{\mathtt{w}}}; \overrightarrow{\bullet \underline{v}_{\mathtt{s}}}\}\!\}$

Figure 10: The target tupled abstract machine (Target TAM): changes with respect to the Int TAM.

Lemma 8.2 (Read back properties).

(1) Values: $\underline{\bullet v}$ *is a value of* $\lambda_{\mathtt{int}}$ *for every* $\bullet v$ *of the Int TAM.*
(2) Evaluation contexts: $\underline{S}$ *and* $\underline{A}$ *are evaluation contexts of* $\lambda_{\mathtt{int}}$ *for every Int TAM constructor and activation stacks* S *and* A.

Theorem 8.3. *The Int TAM and* $\lambda_{\mathtt{int}}$ *form an implementation system on prime terms (as in Def. 6.2), thus the Int TAM implements* $\rightarrow_{\mathtt{int}}$ *on prime terms.*

9 Part 2: the Target TAM for $\lambda_{\mathtt{tar}}$

Here we present the *target tupled abstract machine* (Target TAM) for the target calculus $\lambda_{\mathtt{int}}$, a minor variant of the Int TAM. Beyond the elimination of variable names, its key feature is the use of *tupled environment*, that is, a pair of tuples as data structures for environments, instead of a list of explicit substitution entries $[x \leftarrow \bullet v]$.

The Target TAM is defined in Fig. 10, by giving the only ingredients of the Int TAM that are redefined for the Target TAM, leaving everything else unchanged but for the fact that, when considering the omitted transitions of the Int TAM as transitions of the Target TAM, the symbol E for environment refers to the new notion of environment adopted here (the omitted transitions do not touch the environment). Here the closures $[\![t|b]\!]_{n,m}$ of $\lambda_{\mathtt{tar}}$ are written $[\![t|b]\!]$ (and are then decorated with flags as for the Int TAM) because n and m play a role only for the reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ studied in [5, Appendix C], while here they are irrelevant.

The elimination of names enables the use of a *tupled (stackable) environment*: the environment is now a pair of tuples $\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}$, where $\overrightarrow{\bullet v_{\mathtt{w}}}$ provides values for the wrapped projected variables $\pi_i \mathtt{w}$, and $\overrightarrow{\bullet v_{\mathtt{s}}}$ for the source ones $\pi_i \mathtt{s}$, with no need to associate the values of these tuples to variable names via entries of the form $[x \leftarrow \bullet v]$ (as it was the case for the Int TAM). The change is relevant, as the data structure for environments changes from a *map* to a *tuple*, removing the need (and the cost) of creating a map in transition $\rightsquigarrow_{\bullet\beta_v}$ and inducing a logarithmic speed-up, as we shall see.

The look up into tupled environments is defined in Fig. 10 and is the only new notion needed in the new transitions. The implementation theorem is proved following the same schema used for the Source TAM and the Int TAM, the details are in [5, Appendix G].

Theorem 9.1. *The Target TAM and* $\lambda_{\mathtt{tar}}$ *form an implementation system on prime terms (as in Def. 6.2), thus the Target TAM implements* $\rightarrow_{\mathtt{tar}}$ *on prime terms.*

Actual Implementation of the Target TAM. We provide an OCaml implementation of the Target TAM on GitHub [33], described in [5, Appendix K]. The textual interface asks for a term of the source calculus $\lambda_{\mathtt{sou}}$, which is translated to the target calculus $\lambda_{\mathtt{tar}}$ by applying first wrapping and then name elimination, thus passing through the intermediate calculus $\lambda_{\mathtt{int}}$, as described above. The obtained $\lambda_{\mathtt{tar}}$-term is then reduced by the Target TAM until a normal form is reached, if any, and the final $\lambda_{\mathtt{tar}}$-term is extracted. The machine state is printed after every step, in ASCII art.

The implementation is not particularly optimized and it does not have a graphical user interface. It is designed to stay as close as possible to the definitions given in the paper, and to provide evidence supporting the assumptions of the cost analysis of Sect. 12. In particular, we use OCaml arrays for tuples, variables in abstractions, and bags in closures, for achieving $O(1)$ access times.

10 Part 3 Preliminaries: Sharing, Size Explosion, and the Complexity of Abstract Machines

This section starts the third part of the paper, about the time complexity analysis of abstract machines. Here, we quickly overview the size explosion problem of the λ-calculus as the theoretical motivation for the use of sharing in implementations, as well as the structure of the study of the overhead of abstract machines.

Size Explosion. A well-known issue of the λ-calculus is the existence of families of terms whose size grows *exponentially* with the number of β-steps. They are usually built exploiting some variant of the duplicator $\delta := \lambda x.xx$. We give an example in $\lambda_{\mathtt{cbv}}$. Define:

Variant of δ	Size explod. family		Exploded results	
$\pi := \lambda x.\lambda y.yxx$	$t_0 := \mathtt{I}$	$t_{n+1} := \pi t_n$	$u_0 := \mathtt{I}$	$u_{n+1} := \lambda y.yu_nu_n$

Proposition 10.1 (Size Explosion in $\lambda_{\mathtt{cbv}}$). *Let* $n \in \mathbb{N}$. *Then* $t_n \rightarrow_{\beta_v}^{n} u_n$, *moreover* $|t_n| = O(n)$, $|u_n| = \Omega(2^n)$, *and* u_n *is a value.*

The proof is in [5, Appendix H]. Size explosion has been extensively analyzed in the study of reasonable cost models—see [1] for an introduction—because it suggests that the number n of $\rightarrow_{\beta_v}$ steps is not a reasonable time measure for the execution of λ-terms: for size exploding families, indeed, it does not even account for the time to write down the normal form, which is of size $\Omega(2^n)$.

It is tempting to circumvent the problem by tweaking the calculus, with types, by changing the evaluation strategy, using CPS, and so on. None of these tweaks works, size explosion can always be adapted: it is an inherent feature of higher-order computations [1].

Sharing for Functions. A solution nonetheless exists: it amounts to *share* sub-terms to avoid their blind duplication during evaluation. For size explosion in $\lambda_{\mathtt{cbv}}$, it is enough to add a simple form of *sub-term sharing* by delaying meta-level substitution and avoiding substituting under abstractions. For instance, evaluating the size exploding term t_n above in a variant of $\lambda_{\mathtt{cbv}}$ where sub-term sharing is implemented via explicit substitutions, gives the following normal form, of size *linear* (rather than exponential) in n:

$$\lambda y_1.y_1x_1x_1[x_1 \leftarrow \lambda y_2.y_2x_2x_2] \dots [x_{n-1} \leftarrow \lambda y_n.y_nx_nx_n][x_n \leftarrow \mathtt{I}].$$

The explosion re-appears if one unfolds that normal form to an ordinary λ-term, but it is now encapsulated in the unfolding.

Abstract machines of the previous sections have environments to implement sub-term sharing and avoid the size explosion due to β_v, giving hope for a time complexity lower than exponential.

Parameters for the Time Complexity Analysis of Abstract Machines. Given a strategy $\to_{\mathtt{str}}$ of a calculus λ_{cal} and an abstract machine $\mathtt{M}$ implementing $\to_{\mathtt{str}}$ (see Definition 6.1), the time complexity of $\mathtt{M}$ is obtained by estimating the cost—when concretely implemented on random access machines (RAMs)—of a run $r_e : t_0^\circ \rightsquigarrow^* q$ implementing an arbitrary evaluation sequence $e : t_0 \to^n_{\mathtt{str}} t_n$ (thus having $\underline{q} = t_n$) as a function of two parameters:

(1) *Code size*: the size $|t_0|$ of the initial term t_0;
(2) *Number of $\to_{\mathtt{str}}$-steps/β-steps*: the number n of $\to_{\mathtt{str}}$-steps in e. If λ_{cal} has other rules other than β/β_v, the parameter is often just the number of β/β_v steps, which is usually considered the relevant time cost model.

Recipe for Time Complexity. The way the time complexity of an abstract machine is established tends to follow the same schema:

(1) *Number of overhead transitions*: bounding the number of overhead transitions as a function of $|t_0|$ and n (which by the principal matching property of implementations—see Definition 6.1—is enough to bound the length of r_e);
(2) *Cost of single transitions*: bounding the cost of single transitions, which is typically constant or depends only on $|t_0|$;
(3) *Total cost*: inferring the total cost of a run r by multiplying the number of steps of each kind of transition for their cost, and summing all the obtained costs.

The key tool for such an analysis is the *sub-term property*, an *invariant* of abstract machines stating that some of the terms in a reachable state are sub-terms of the initial term. This allows one to develop bounds with respect to the size $|t_0|$ of the initial term t_0.

Time Complexity of the LAM. The LAM verifies a sub-term invariant, and its time complexity follows a well-known schema in the literature, closely inspected by Accattoli and Barras [4] for call-by-name and call-by-need, and that smoothly adapts to CbV. Consider an evaluation $e\colon t_0 \to^n_{\beta_v} t_n$ in λ_{cbv}. About the first point of the recipe, the bound on overhead transitions is $O((n+1)\cdot|t_0|)$, that is, bilinear. Additionally, if one takes *complete* evaluations, that is, for which $t_n = v$ is a value, then the bound lowers to $O(n)$. Such an independence from the initial term is due to the fact that whether a term is a value can be checked in $O(1)$ in λ_{cbv}, by simply checking whether the top-most constructor is an abstraction.

For the second point of the recipe, the cost of single transitions of the LAM depends on the data structures used for local environments, as discussed by Accattoli and Barras [4]. With flat environments, the cost of manipulating them is $O(|t_0|)$ (because of the duplication of E in $\rightsquigarrow_{sea_1}$), giving a total cost of $O(n\cdot|t_0|)$ for complete runs. With shared environments, the best structures manipulate them in $O(\log|t_0|)$, giving a total cost of $O(n\cdot\log|t_0|)$ for complete runs. Thus, shared local environments are faster, but they are optimized for time and inefficient with respect to space, as they prevent some garbage collection to take place. Here, we take as reference *flat* local environment, which induce the same overall $O(n\cdot|t_0|)$ overhead as global environments and enable a better management of space (not discussed here, see [10] instead).

Notions of Flatness for Local Environment. Let us be precise on a subtle point about local environments and their flatness. Local environments are defined by mutual recursion with m-closures, and various notions of sharing and flatness are possible, as one can share environments, or m-closures, both, or none of them. Sharing both is essentially the same as sharing only environments.

Sharing m-closures rather than environments is what we above called *flat environments*. For instance, if $E = [x_1{\leftarrow}\bullet c_1] : [x_2{\leftarrow}\bullet c_2] : \epsilon$ then $\bullet c_1$ and $\bullet c_2$ are shared so that the concrete representation of E is $E = [x_1{\leftarrow}p_1] : [x_2{\leftarrow}p_2] : \epsilon$ where p_1 and p_2 are pointers to $\bullet c_1$ and $\bullet c_2$. Copying E then means copying $[x_1{\leftarrow}p_1] : [x_2{\leftarrow}p_2] : \epsilon$, *without* recursively copying the structure of $\bullet c_1$ and $\bullet c_2$.

The *super flat environments* obtained by removing sharing for *both* environments and m-closures are studied in [10], where it is shown that the overhead of abstract machines becomes *exponential*. This is the reason why we do not consider them here.

No Sharing for Stackable Environments. An interesting aspect of the new stackable environments is that they need no sharing. Indeed, the Int TAM and the Target TAM never duplicate their stackable environments. In particular, when they discard the current environment E in transition $\rightsquigarrow_{\bullet sea_7}$, it can be collected.

De Bruijn Indices Do Not Change the Overhead. It is well-known that, representing flat environments as arrays and variables with de Bruijn indices, one can look-up environments in $O(1)$ rather than in $O(\log|t_0|)$, which is the best that one can do with (some ordered domain of) names. However, the $O(|t_0|)$ cost of copying flat environments in transition $\rightsquigarrow_{sea_1}$ dominates, so that turning to de Bruijn indices does not change the overall $O(n\cdot|t_0|)$ overhead.

Size Exploding Tuples. In λ_{sou}, there is a new form of size explosion, due to tuples, requiring another form of sharing. To our knowledge, the view provided here is novel. Set $\mathtt{I} := \lambda z.z$ and:

Variant of δ	Size explod. family		Exploded results	
$\tau := \lambda x.(\!|x,x|\!)$	$s_0 := \mathtt{I}$	$s_{n+1} := \tau(\!|s_n|\!)$	$r_0 := \mathtt{I}$	$r_{n+1} := (\!|r_n, r_n|\!)$

Proposition 10.2 (Size explosion of tuples). *Let $n \in \mathbb{N}$. Then $s_n \to^n_{\beta_v} r_n$, moreover $|s_n| = O(n)$, $|r_n| = \Omega(2^n)$, and r_n is a value.*

Sharing for Tuples. To avoid the size explosion of tuples, another form of sharing is used. The idea is the same as for functions: forms of size explosion are circumvented by forms of sharing designed to limit the substitution process. The key point is that tuples should never be copied, only *pointers* to them should be copied, thus representing r_n above using *linear* (rather than exponential) space in n, as follows (where the p_i are *pointers* and $[p_i{\leftarrow}v]$ are *heap entries*):

$$(\!|p_1, p_1|\!)\,[p_1{\leftarrow}(\!|p_2, p_2|\!)]\ldots[p_n{\leftarrow}(\!|p_n, p_n|\!)]\,[p_n{\leftarrow}\mathtt{I}]$$

The abstract machines of the previous sections have environments for sub-term sharing (needed for β_v) but they do *not explicitly* handle tuple sharing. The reason is practical: explicitly handling tuple sharing would require a treatment of *pointers* and a *heap* (i.e., a further global environment) and more technicalities. Our machines, however, are meant to be concretely implemented with tuple sharing, as in our OCaml implementation of the Target TAM [33].

11 Part 3: Complexity of the Source TAM, or, Tuples Raise the Overhead

Here, we develop the time complexity analysis of the Source TAM, stressing the novelty of tuples (see [5, Appendix I] for proofs).

The Source TAM verifies the following *sub-term invariant.*

Size of λ_{sou} terms

$$|x| := 1 \qquad |\lambda\widetilde{x}.t| := |t| + \|\widetilde{x}\| + 1 \qquad |tu| := |t| + |u| + 1$$
$$|\pi_i t| := |t| + 1 \qquad |(\!|t_1, \dots, t_n|\!)| := n + \textstyle\sum_{i=1}^{n} |t_i|$$

Overhead measure for the Source TAM

$$\text{Con. stack entries } S_{en} \quad |\bullet c|_{oh} := 0 \quad |\circ(t,E)|_{oh} := |t| \quad |\pi_i|_{oh} := 0$$
$$|((\!|\circ t_1, .., \circ t_n, \downarrow, \overrightarrow{\bullet c}|\!), E)|_{oh} := n + \textstyle\sum_{i=1}^{n} |t_i|_{oh}$$
$$\text{Stacks} \quad |\epsilon|_{oh} := 0 \quad |S_{en} : S|_{oh} := |S_{en}|_{oh} + |S|_{oh}$$
$$\text{States} \quad |(\bullet c \mid S)|_{oh} := |\bullet c|_{oh} + |S|_{oh}$$

Figure 11: Size $|\cdot|$ of λ_{sou} terms and overhead measure $|\cdot|_{oh}$.

Lemma 11.1 (Sub-term invariant). *Let q be a Source TAM reachable state from the initial state t°.*

1. *u is a sub-term of t for every m-closure of shape $\circ(u, E)$ or $((\!|., u, .., \downarrow, \overrightarrow{\bullet c}|\!), E)$ in q.*
2. *$\lambda\widetilde{x}.u$ is a sub-term of t for every m-closure $\bullet(\lambda\widetilde{x}.u, E)$ in q.*

Tuples are Not Sub-Terms. Point 2 only concerns evaluated m-closures containing abstractions, and *not* evaluated tuples. Consider $t := (\lambda x.\lambda y.(\!|x, y|\!))(\!|\mathtt{I}|\!)(\!|\delta|\!) \to_{\beta_v} (\lambda y.(\!|\mathtt{I}, y|\!))(\!|\delta|\!) \to_{\beta_v} (\!|\mathtt{I}, \delta|\!)$ in λ_{sou} and note that $(\!|\mathtt{I}, \delta|\!)$ is *not* a sub-term of t. The run of the Source TAM on t produces a m-closure $((\!|\bullet\mathtt{I}, \bullet\delta|\!), E)$ for some E. The leaves of the tree-structure of an evaluated tuple are abstractions ($\mathtt{I}$ and δ in the example), which are initial sub-terms, but they might be arranged in ways that were not present in the initial term.

A consequence of this fact is that when the Source TAM starts evaluating a non-empty tuple $(\!|., t_n|\!)$ with transition $\rightsquigarrow_{\circ sea_3}$, by adding $(\!|., \downarrow|\!)$ to the stack, it has to *allocate a new tuple* on the heap for $(\!|., \downarrow|\!)$ (which has a cost, discussed below). This never happens in absence of tuples, that is, in the LAM of Fig. 7. More precisely, the LAM does not copy any code, but it has to allocate new pointers to local environments, when they are extended by $\rightsquigarrow_{\beta_v}$.

Step 1 of the Recipe: Number of Transitions and Overhead Measure. For establishing a bound on overhead transitions, we first factor some of them ($\rightsquigarrow_{\bullet sea_{1,3}}$) out by simply noticing that they are enabled and thus bound by some others ($\rightsquigarrow_{\circ sea_{1,3}}$). Actually, the same is true also for the principal transition $\rightsquigarrow_{\bullet\pi}$, bounded by $\rightsquigarrow_{\circ sea_2}$. We set $\rightsquigarrow_{a,b} := \rightsquigarrow_a \cup \rightsquigarrow_b$ and $|r|_{a,b} := |r|_a + |r|_b$ for every run r.

Lemma 11.2 (Transition match). *Let r be a Source TAM run. Then $|r|_{\pi,\bullet sea_{1,3}} \leq |r|_{\circ sea_{1,2,3}}$.*

For the other transitions, we use a measure $|\cdot|_{oh}$, defined in Fig. 11 together with the size $|t|$ of λ_{sou} terms, which we use to derive a bilinear bound on overhead/projection transitions (Prop. 11.4).

Lemma 11.3 (Overhead measure properties). *Let $t^\circ \rightsquigarrow^*_{\text{STAM}} q$ a Source TAM run and $q \rightsquigarrow_a q'$.*

1. *if $a = \bullet\beta_v$ then $|q'|_{oh} \leq |q|_{oh} + |t|$;*
2. *if $a \in \{\circ sub, \circ sea_{1-5}, \bullet sea_6\}$ then $|q'|_{oh} < |q|_{oh}$;*
3. *if $a \in \{\bullet sea_{1,3}, \bullet\pi\}$ then $|q'|_{oh} = |q|_{oh}$.*

Proposition 11.4 (Bilinear number of transitions). *Let $t \in \lambda_{\mathsf{sou}}$ be closed. If $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ then $|r| \in O\big((|r|_{\beta_v}+1)\cdot|t|\big)$.*

Tuples Raise the Overhead. Prop. 11.4 shows that projection transitions are also bi-linear. In Section 10, we mentioned that, without tuples, the bound improves to $O(|r|_{\beta_v})$ if one considers complete runs (that is, runs ending on values). With tuples, there is no such improvement. Indeed, even just checking that the initial term is actually a value v takes time $O(|v|)$ with tuples: if v is a tree of tuples, the Source TAM has to visit the tree and check that all the leaves are abstractions; in absence of tuples the check instead costs $O(1)$.

Step 2 of the Recipe: Cost of Single Transitions. To obtain fine bounds with respect to the initial term, we introduce the notions of width and height of a term t, both bounded by the size $|t|$ of t.

Definition 11.5 (Width, height). The *width* $\mathtt{wd}(t) \in \mathbb{N}$ of $t \in \lambda_{\mathsf{sou}}$ is the maximum length of a tuple or of a sequence of variables in t. The *height* $\mathtt{hg}(t) \in \mathbb{N}$ of $t \in \lambda_{\mathsf{sou}}$ is the maximum number of bound variables of t in the scope of which a sub-term of t is contained.

As discussed after Lemma 11.1, $\rightsquigarrow_{\circ sea_3}$ has to allocate a new tuple, thus its cost seems to depend on $\mathtt{wd}(t)$, and similarly for $\rightsquigarrow_{\bullet\beta_v}$. However, the price related to tuples can be considered as absorbed by the cost of search (by changing the multiplicative constant), since the new tuple of $\rightsquigarrow_{\circ sea_3}$ is then traversed, if the run is long enough, and the one of $\rightsquigarrow_{\bullet\beta_v}$ was traversed before the transition. Therefore, if we consider *complete* runs (i.e. ending on final states), $\rightsquigarrow_{\circ sea_3}$ and $\rightsquigarrow_{\bullet\beta_v}$ have *amortized* cost independent of $\mathtt{wd}(t)$.

Transitions $\bullet sea_1$, $\bullet sea_3$, and $\bullet sea_6$ duplicate E, the length of which is bounded by $\mathtt{hg}(t)$. With flat environments, this costs $O(\mathtt{hg}(t))$. Transition $\rightsquigarrow_{\circ sub}$ has to look-up the environment, the cost of which is $O(\mathtt{hg}(t))$. De Bruijn indices or an ordered domain of names might improve the cost of look-up, but at no overall advantage, because of the dominating cost of duplicating environments for $\bullet sea_1$, $\bullet sea_3$, and $\bullet sea_6$. All other transitions have constant cost, assuming that accessing the i-th component of a tuple (needed for $\rightsquigarrow_{\bullet\pi}$) takes constant time. The next proposition sums it up.

Proposition 11.6 (Cost of single transitions). *Let $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ be a complete Source TAM run. A transition $\rightsquigarrow_a$ of r costs $O(\mathtt{hg}(t))$ if $a \in \{\bullet sea_1, \bullet sea_3, \bullet sea_6, \circ sub\}$, and $O(1)$ otherwise.*

Step 3 of the Recipe: Total Complexity. By simply multiplying the number of single transitions (Proposition 11.4) for their cost (Proposition 11.6), we obtain the complexity of the Source TAM.

Theorem 11.7. *Let $t \in \lambda_{\mathsf{sou}}$ be closed and $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ be a complete Source TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v} + 1)\cdot|t|\cdot\mathtt{hg}(t)\big)$.*

If one flattens $\mathtt{hg}(t)$ as $|t|$, the complexity of the Source TAM is $O\big((|r|_{\beta_v} + 1)\cdot|t|^2\big)$, that is, quadratic in $|t|$, while in absence of tuples—that is, for the LAM—it is linear in $|t|$.

12 Part 3: Complexity of the Target TAM, or, Closure Conversion Preserves the Overhead

In this section, we adapt the time complexity analysis of the Source TAM to an analysis of the Target TAM (skipping the less efficient Int TAM), and then connect source and target by considering the impact of closure conversion on the given analysis.

The Target TAM has a sub-term invariant, expressed compactly thanks to our flags. In particular the part about abstractions of the sub-term invariant for the Source TAM (Lemma 11.1) is here captured by having flagged the body of unevaluated closures with $\circ$.

Lemma 12.1 (Sub-term invariant). *Let q be a Target TAM reachable state from the initial state t°. Then u is a sub-term of t for every non-evaluated term $\circ u$ in q.*

Step 1: Number of (Overhead) Transitions. The bound on the number of (overhead) transitions is obtained following the same reasoning used for the Source TAM. The new transition $\leadsto_{\bullet sea_7}$ is part of the transitions that are factored out, since each $\leadsto_{\bullet sea_7}$ transition is enabled by a $\leadsto_{\bullet \beta_v}$ transition, which adds an entry to the activation stack. We also use an overhead measure (in [5, Appendix J]) which is a direct adaptation to the Target TAM of the one given for the Source TAM. Note indeed that the measure ignores environments, which are the main difference between the two machines.

Proposition 12.2 (Bilinear number of transitions). *Let $t \in \lambda_{\mathtt{tar}}$ be closed. If $r\colon t^\circ \leadsto^*_{\mathrm{TTAM}} q$ then $|r| \in O\big((|r|_{\beta_v}+1)\cdot|t|\big)$.*

Step 2: Cost of Single Transitions. For all transitions of the Target TAM but $\leadsto_{\circ sub_v}$, $\leadsto_{\circ sub_c}$, and $\leadsto_{\bullet sea_7}$ the cost is the same. For $\leadsto_{\circ sub_v}$, the cost is now $O(1)$ since tupled environments have $O(1)$ access time via indices (in the Int TAM its cost is instead $O(\mathsf{hg}(t))$).

For the new transition $\leadsto_{\circ sub_c}$, $O(\mathsf{wd}(t))$ look-ups in the environment are needed. Because of tupled environments, each look-up costs $O(1)$. Thus, the cost seems to depend on $\mathsf{wd}(t)$, but—reasoning as for $\leadsto_{\bullet \beta_v}$ and $\leadsto_{\circ sea_3}$ in Section 11—one can amortize it with the cost of search in complete runs. That is, we shall consider $\leadsto_{\circ sub_c}$ to have constant cost. The new transition $\leadsto_{\bullet sea_7}$ has constant cost as well. Summing up, we get one of the insights mentioned in the introduction: the amortized cost of all single transitions is $O(1)$.

Proposition 12.3 (Cost of single transitions). *Let $r\colon t^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Every transition of r costs $O(1)$.*

Step 3: Total Complexity. Multiplying the number of single transitions (Prop. 12.2) for their cost (Prop. 12.3), we obtain the time complexity of the Target TAM, which seem better than for the Source TAM. After the theorem we discuss why it is not necessarily so.

Theorem 12.4. *Let $t \in \lambda_{\mathtt{tar}}$ be closed and $r\colon t^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v}+1)\cdot|t|\big)$.*

Factoring in the Size Growth of Wrapping. To complete the analysis, consider that the term $t \in \lambda_{\mathtt{tar}}$ on which the Target TAM is run is meant to be the closure conversion (that is, wrapping + name elimination) of a term $u \in \lambda_{\mathtt{sou}}$. While name elimination does not affect the size of terms, wrapping $\underline{\cdot}$ does (proof in [5, Appendix J]).

Lemma 12.5 (Wrapping size growth bound).

1. *If $t \in \lambda_{\mathtt{sou}}$ then $|\underline{t}| \in O(\mathsf{hg}(t)\cdot|t|)$.*
2. *There are families of terms $\{t_n\}_{n\in\mathbb{N}}$ for which $\mathsf{hg}(t_n) = \Theta(|t_n|)$, so that $|\underline{t_n}| \in \Theta(|t_n|^2)$.*

Proof. Point 1: the size increment in $\underline{t}$ is due to the bags introduced by wrapping abstractions, and it is proportional to the number of free variables in the body of the abstraction, bounded by $\mathsf{hg}(t)$; so $|\underline{t}| \in O(\mathsf{hg}(t)\cdot|t|)$. Point 2: take $t_n := \lambda x_1..\lambda x_n.x_1x_2\ldots x_n \neq \lambda x_1,..,x_n.x_1x_2\ldots x_n$ (note n abstractions in t_n, not just 1). □

We can now instantiate the bounds for running the Target TAM on the closure conversion of a $\lambda_{\mathtt{sou}}$ term, obtained by substituting the bounds in Lemma 12.5.2 in Theorem 12.4. We end up obtaining the same complexity as for the Source TAM (Theorem 11.7), despite the bound here being the outcome of a different reasoning.

Theorem 12.6. *Let $t \in \lambda_{\mathtt{sou}}$ be closed and $r\colon (\underline{t}^{\epsilon,\epsilon})^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v}+1)\cdot|t|\cdot\mathsf{hg}(t)\big)$.*

13 Related Work and Conclusions

Related work. The seminal work of Minamide et al. [28] uses existential types to type closure conversion, and other works study the effect of closure conversion on types, as well as conversion towards typed target languages [16, 18, 29]. This line of work explores the use of types in compilation which can span the entire compiler stack, from λ-calculus to (typed) assembly [30].

Appel and co-authors have studied closure conversion, its efficient variants, and its *space safety* [17, 31, 35, 36]. Unlike them, we address a direct style λ-calculus, without relying on a CPS. We are however inspired by [31] in studying flat environments, also known as *flat closures*. Optimizations of flat closures are studied in [25].

Closure conversion of only some abstractions, and of only some of their free variables is studied by Wand and Steckler [39]. Closure conversion has also been studied in relation to graphical languages [34], non-strict languages [37], formalizations [19, 40], extended to mutable state by Mates et al. [27] and to a type-preserving transformation for dependent types by Bowman and Ahmed [18].

Many of the cited works study various aspects of the efficiency of closure conversion. As said in the introduction, our concerns here are orthogonal, as we are rather interested in the asymptotic overhead of the machines with respect to flat closure conversion.

Sullivan et al. [38] study a call-by-push-value λ-calculus where converted and non-converted functions live together, considering also an abstract machine. They do not study the complexity of the machine, nor the new notions of environments studied here.

For abstract machines, we follow Accattoli and co-authors, see for instance [2, 6, 13]. Another framework is [21].

Work orthogonal to our concerns is the *derivation* of abstract machines using closure conversion as a step in the process, along with CPS transformation and defunctionalization [15].

Conclusions. We study the relationship between closure conversion and abstract machines in probably the simplest possible setting, an extension with tuples of Plotkin's untyped call-by-value λ-calculus, and with respect to the simple notion of flat environments. Our starting point is to decompose closure conversion in two subtransformations, dubbed *wrapping* and *name elimination*, turning the source calculus into a target calculus, via an intermediate one.

Each calculus is then paired with a variant of the *tupled abstract machine* (TAM). The Source TAM has machine-closures and local environments, while the Int TAM and the Target TAM have forms of converted closures. Moreover, they exploit the invariants enforced by the transformations, adopting new, better behaved forms of environments, namely *stackable* and *tupled environments*.

We give proof of correctness—under the form of termination-preserving strong bisimulations—for wrapping and name elimination, as well as implementation theorems for every machine with respect to its associated calculus. In particular, the proof technique for the correctness of closure conversion is new and simple.

Lastly, we study the time complexity of the abstract machines, showing that flat closure conversion reshuffles the costs, lowering

the dependency on the initial term, while at the same time increasing the size of the initial term, ending up with the same complexity.

Various directions for future work are possible, e.g. extending the source calculus with recursive let and pattern matching; or refining the abstract machines by making explicit the sharing for tuples.

References

[1] Beniamino Accattoli. 2019. A Fresh Look at the lambda-Calculus (Invited Talk). In *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. 1:1–1:20. doi:10.4230/LIPIcs.FSCD.2019.1

[2] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 363–376. doi:10.1145/2628136.2628154

[3] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2015. A Strong Distillery. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 231–250. doi:10.1007/978-3-319-26529-2_13

[4] Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017)*. 4–16. http://doi.acm.org/10.1145/3131851.3131855

[5] Beniamino Accattoli, Cláudio Belo Lourenço, Dan Ghica, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2025. Closure Conversion, Flat Environments, and the Complexity of Abstract Machines (Long Version). arXiv:2507.15843 [cs.PL] https://arxiv.org/abs/2507.15843

[6] Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019. Crumbling Abstract Machines. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 4:1–4:15. doi:10.1145/3354166.3354169

[7] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. 2021. Strong Call-by-Value is Reasonable, Implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–14. doi:10.1109/LICS52264.2021.9470630

[8] Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta-Reduction is Invariant, Indeed. *Logical Methods in Computer Science* 12, 1 (2016). doi:10.2168/LMCS-12(1:4)2016

[9] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2021. The (In)Efficiency of Interaction. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. doi:10.1145/3434332

[10] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Reasonable Space for the λ-Calculus, Logarithmically. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 47:1–47:13. doi:10.1145/3531130.3533362

[11] Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 206–226. doi:10.1007/978-3-319-47958-3_12

[12] Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. In *Proceedings of the 7th IPM International Conference on Fundamentals of Software Engineering (FSEN 2017), Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. 1–19. doi:10.1007/978-3-319-68972-2_1

[13] Beniamino Accattoli and Giulio Guerrieri. 2019. Abstract machines for Open Call-by-Value. *Sci. Comput. Program.* 184 (2019). doi:10.1016/j.scico.2019.03.002

[14] Beniamino Accattoli and Claudio Sacerdoti Coen. 2017. On the value of variables. *Inf. Comput.* 255 (2017), 224–242. doi:10.1016/j.ic.2017.01.003

[15] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. doi:10.1145/888251.888254

[16] Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 157–168. doi:10.1145/1411204.1411227

[17] Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 293–302. doi:10.1145/75277.75303

[18] William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 797–811. doi:10.1145/3192366.3192372

[19] Adam Chlipala. 2010. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 93–106. doi:10.1145/1706299.1706312

[20] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP 2019)*. 9:1–9:14. doi:10.1145/3354166.3354174

[21] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. https://mitpress.mit.edu/9780262062756/semantics-engineering-with-plt-redex/

[22] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

[23] Maribel Fernández and Nikolaos Siafakas. 2008. New Developments in Environment Machines. In *Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming, WRS@RTA 2008, Hagenberg, Austria, July 14, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 237)*, Aart Middeldorp (Ed.). Elsevier, 57–73. doi:10.1016/J.ENTCS.2009.03.035

[24] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 235–246. doi:10.1145/581478.581501

[25] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing closures in O(0) time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, Olivier Danvy (Ed.). ACM, 30–35. doi:10.1145/2661103.2661106

[26] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA. http://gallium.inria.fr/~xleroy/publi/ZINC.pdf

[27] Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 16:1–16:15. doi:10.1145/3354166.3354181

[28] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 271–283. doi:10.1145/237721.237791

[29] J. Gregory Morrisett and Robert Harper. 1997. Typed Closure Conversion for Recursively-Defined Functions. In *Second Workshop on Higher-Order Operational Techniques in Semantics, HOOTS 1997, Stanford, CA, USA, December 8-12, 1997 (Electronic Notes in Theoretical Computer Science, Vol. 10)*, Andrew D. Gordon, Andrew M. Pitts, and Carolyn L. Talcott (Eds.). Elsevier, 230–241. doi:10.1016/S1571-0661(05)80702-9

[30] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. doi:10.1145/319301.319345

[31] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. doi:10.1145/3341687

[32] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. doi:10.1016/0304-3975(75)90017-1

[33] Claudio Sacerdoti Coen. 2025. Artifact for the Closure Conversion and Abstract Machines paper. https://github.com/sacerdot/closure-conversion-machine/releases/tag/PPDP25.

[34] Ralf Schweimeier and Alan Jeffrey. 1999. A Categorical and Graphical Treatment of Closure Conversion. In *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999 (Electronic Notes in Theoretical Computer Science, Vol. 20)*, Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov (Eds.). Elsevier, 481–511. doi:10.1016/S1571-0661(04)80090-2

[35] Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, Robert R. Kessler (Ed.). ACM, 150–161. doi:10.1145/182409.156783

[36] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 129–161. doi:10.1145/345099.345125

[37] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2021. Strictly capturing non-strict closures. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, Sam Lindley and Torben Æ. Mogensen (Eds.). ACM, 74–89. doi:10.1145/3441296.3441398

[38] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2023. Closure Conversion in Little Pieces. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, Santiago Escobar and Vasco T. Vasconcelos (Eds.). ACM, 10:1–10:13. doi:10.1145/3610612.3610622

[39] Mitchell Wand and Paul Steckler. 1994. Selective and Lightweight Closure Conversion. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 435–445. doi:10.1145/174675.178044

[40] Yuting Wang and Gopalan Nadathur. 2016. A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 752–779. doi:10.1007/978-3-662-49498-1_29

[41] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/INCO.1994.1093