A Lazy, Concurrent Convertibility Checker

NATHANAËLLE COURANT, OCamlPro, France
XAVIER LEROY, Collège de France, PSL University, France

Convertibility checking — determining whether two lambda-terms are equal up to reductions — is a crucial component of proof assistants and dependently-typed languages. Practical implementations often use heuristics to quickly conclude that two terms are convertible, or are not convertible, without reducing them to normal form. However, these heuristics can backfire, triggering huge amounts of unnecessary computation. This paper presents a novel convertibility-checking algorithm that relies crucially on *laziness* and *concurrency*. Laziness is used to share computations, while concurrency is used to explore multiple convertibility subproblems in parallel or via fair interleaving. Unlike heuristics-based approaches, our algorithm always finds an easy solution to the convertibility problem, if one exists. The paper describes the algorithm in process calculus style, discusses its complexity, and reports on its mechanized proof of partial correctness and its lightweight experimental evaluation.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Formal software verification*; • **Theory of computation** → *Type theory*; *Lambda calculus*; *Functional constructs*; *Operational semantics*; *Proof theory*; *Process calculi*; • **Mathematics of computing** → *Lambda calculus*.

Additional Key Words and Phrases: Convertibility, Lazy evaluation, Normalization, Proof assistants, Proof checking, Type checking

ACM Reference Format:
Nathanaëlle Courant and Xavier Leroy. 2026. A Lazy, Concurrent Convertibility Checker. *Proc. ACM Program. Lang.* 10, POPL, Article 53 (January 2026), 27 pages. https://doi.org/10.1145/3776695

1 Introduction

Lambda-terms and related functional notations are widely used in functional programming languages, in higher-order type systems, and in higher-order logics. These terms come with a notion of *reduction*, which expresses elementary steps of computation, such as the famous beta-reduction $(\lambda x.\, t)\ u \longrightarrow t[x := u]$, which expresses the application of a function.

Reductions have two main uses: *evaluation* (reducing a term to a final result) and *conversion* (determining whether two terms are equal up to reductions). Evaluation accounts for the execution of functional programs and their specialization through partial evaluation techniques. Conversion is used in type systems and in logics based on type theory to state that two types or two propositions are identical up to computation. This concept is captured by the well-known typing rule

$$\frac{\Gamma \vdash a : t' \qquad t \approx t'}{\Gamma \vdash a : t}\ \textsc{conv}$$

For example, the two propositions $2 + 2 = 4$ and $4 = 4$ are convertible, since $2 + 2$ reduces to 4; therefore, the trivial proof term `refl 4` (reflexivity of equality) of $4 = 4$ also proves $2 + 2 = 4$; no deduction steps are necessary. This is an instance of proof by reflection, where deduction and proof

Authors' Contact Information: Nathanaëlle Courant, nathanaelle.courant@ocamlpro.com, OCamlPro, Paris, France; Xavier Leroy, xavier.leroy@college-de-france.fr, Collège de France, PSL University, Paris, France.

ACM 2475-1421/2026/1-ART53
https://doi.org/10.1145/3776695

search are replaced by computations performed during type checking [Boutin 1997; Kokke and Swierstra 2015].

Due to the CONV rule, proof assistants and programming languages based on type theory such as Agda, Lean and Rocq verify convertibility of terms at every proof step and type-checking step. Therefore, it is crucial to find algorithms for convertibility checking that are both correct and efficient. Many different reduction sequences can be applied to a term; some sequences lead quickly to the desired result, while others can take much longer or diverge.

To evaluate a term, we have *reduction strategies* such as call by name, call by value and call by need. The performance characteristics and efficient implementation of these strategies are well known. For example, call by need is optimal (in terms of the number of beta-reductions) for weak reduction [Balabonski 2013], and high-performance implementations exist.

In contrast, no good reduction strategy is known for checking whether two terms are convertible. The textbook approach is to reduce both terms to normal form and compare the normal forms for equality. This approach can perform arbitrary amounts of unnecessary computation (see §2 for examples). Convertibility checkers used by proof assistants perform incremental evaluation of the two terms, bringing them to a state where they are either syntactically equal or obviously non-convertible. They use heuristics to determine which evaluation to perform next. As illustrated in §2 and §10, these heuristics are sometimes ineffective, performing unnecessary computation and resulting in proofs that take forever to check and proof tactics that take forever to fail.

This paper presents a novel algorithm for checking convertibility that relies crucially on *laziness* and *concurrency*. Laziness, or more precisely non-strict evaluation, is used to avoid unnecessary computations and to share computations between multiple convertibility subproblems. Concurrency is used to explore multiple convertibility subproblems in parallel or via fair interleaving, stopping them all as soon as one of them returns conclusive evidence. Existing convertibility checkers would explore these subproblems sequentially, in an order chosen by heuristics; they can get stuck exploring the wrong subproblem first. In contrast, our concurrent exploration method will never overlook an easy solution to the convertibility problem, if one exists.

Our convertibility algorithm has been proved sound using the Rocq proof assistant. It can be easily implemented as an abstract machine. This machine has the subterm property [Accattoli and Lago 2012, §3.1], meaning that it can be statically compiled to virtual machine code or native code.

The remainder of this paper is organized as follows. Section 2 gives examples that illustrate the difficulties of convertibility checking. Section 3 introduces the small process calculus that we use to express our algorithms. Section 4 describes call-by-need evaluation (to WHNF) and normalization. Sections 5 and 6, the core of the paper, describe the convertibility algorithm. Its implementation as an abstract machine with explicit scheduling is shown in section 7, and its Rocq proof of soundness is described in section 8. Section 9 analyses the performance of our algorithm. Section 10 reports on a preliminary experimental evaluation, using the Rocq conversion checker as the baseline. Related work is discussed in section 11 and followed by concluding remarks in section 12.

2 Intuitions on Convertibility Checking

Consider the problem of determining whether two expressions e_1 and e_2 are convertible, written $e_1 \approx e_2$, in the sense that they are equal up to integer arithmetic calculations. For example, $6\times4+1 \approx 10 + 14 + 1$ since 6×4 is 24 and $10 + 14$ is also 24.

A simple algorithm for determining whether $e_1 \approx e_2$ is to compute the integer values of e_1 and e_2 and then compare these values for equality. For example,

$6 \times 4 + 1 \approx 10 + 14 + 1$	since $6 \times 4 + 1$ evaluates to 25, as does $10 + 14 + 1$
$6 \times 4 + 1 \not\approx 3 \times 8$	since $6 \times 4 + 1$ evaluates to 25 and 3×8 to 24.

However, this algorithm can perform unnecessary computations. For example, let F be an expensive integer function, of cost $O(2^n)$, say. To determine that

$$F\ 20 \approx F\ 20$$

the simple algorithm computes $F\ 20$ twice. However, it suffices to note that the two sides of the conversion problem are syntactically identical; therefore, their values must be equal, and no computation is needed.

Similarly, assume that expressions include the lists constructors `cons` and `nil`. To determine that

$$\texttt{cons}\ (F\ 20)\ \texttt{nil} \not\approx \texttt{nil}$$

we do not need to compute $F\ 20$ at all. It suffices to observe that the head constructors of the left-hand side (`cons`) and of the right-hand side (`nil`) are different; therefore, no amount of calculation can make them equal.

Often, the two expressions being tested for convertibility are not identical, but "fairly close". Consider:

$$F\ 20 \approx F\ (19 + 1)$$

Again, there is no need to compute both sides. It suffices to show that $20 \approx 19 + 1$, by a simple computation. Then, $F\ 20 \approx F\ (19 + 1)$ follows immediately.

The previous example suggests the following heuristic: to show that two applications of the same function are convertible, first check if the arguments are pairwise compatible:

$$F\ a_1\ \ldots\ a_n \approx F\ b_1\ \ldots\ b_n \quad \text{if} \quad a_i \approx b_i \quad \text{for } i = 1, \ldots, n$$

Only when this first check fails should the definition of F be used to further reduce the two sides of the convertibility test.

Unfortunately, this heuristic is not always profitable. Consider

$$K\ 0\ (F\ 20) \approx K\ 0\ (F\ 21)$$

where K is the familiar combinator $K\ x\ y = x$. The heuristic above causes $F\ 20$ and $F\ 21$ to be computed, which is expensive. However, if we unroll the definition of K first, the convertibility problem becomes $0 \approx 0$, which is trivial, and no computation of F is needed.

In many cases, it is preferable to unroll the definition of a recursive function once rather than fully evaluate an application of the function. For example, let `exp` be the naive exponentiation function

$$\texttt{exp}\ n\ =\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{exp}(n-1) + \texttt{exp}(n-1)$$

Consider the convertibility problem

$$\texttt{exp}\ 40 \approx \texttt{exp}\ 39 + \texttt{exp}\ 39$$

Unrolling the definition of `exp` in the left-hand side and simplifying the `if` allows us to prove convertibility without evaluating `exp` 40 or `exp` 39.

When comparing applications of two different functions, unrolling the definitions of both functions is often the right thing to do, but not always. Consider the mutually-recursive functions

$$\begin{aligned}\texttt{even}\ n\ &=\ \texttt{if}\ n = 0\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{odd}(n-1)\\ \texttt{odd}\ n\ &=\ \texttt{if}\ n = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{even}(n-1)\end{aligned}$$

and the convertibility problem

$$\texttt{odd}\ 999999 \approx \texttt{even}\ 1000000$$

The problem can easily be solved by unrolling the definition of even in the right-hand side, reducing it to odd 999999. However, if we unroll odd in the left hand side and even in the right-hand side simultaneously, we obtain even 999998 ≈ odd 999999, which is still far from a solution.

As the examples above show, there are many different ways to determine if two expressions are convertible. Some ways are faster for certain expressions, but no single way is consistently better than the others. Proof assistants generally use a fixed strategy to determine when and where to perform reductions and unrolling of function definitions. For instance, given $F\ a_1\ \cdots\ a_n \approx F\ b_1\ \cdots\ b_n$, the Rocq proof checker first tries to prove $a_i \approx b_i$ for $i = n, n-1, \ldots, 1$ before unrolling F. Given $F\ a_1\ \cdots\ a_n \approx G\ b_1\ \cdots\ b_m$, it chooses whether to unroll F or G based on numerical priorities, which can be controlled with the `Strategy` command [Rocq Development Team 2025].This is a reasonable strategy. However, any such strategy can go wrong and perform huge amounts of unnecessary computation, which prevent proof checking from completing in a reasonable amount of time [Gross 2021, section 2.6.2].

The approach we propose and develop in this paper is to explore multiple ways to solve a convertibility problem in parallel, instead of trying one way after another based on a fixed strategy. For instance, when presented with the problem $F\ e_1 \approx F\ e_2$, we do not decide whether to begin by solving $e_1 \approx e_2$, or by unrolling F on the left to obtain $e_1' \approx F\ e_2$, or by unrolling F on the right to obtain $F\ e_1 \approx e_2'$. Rather, we set up the three corresponding problems and solve them in parallel, stopping them as soon as $e_1 \approx e_2$ terminates with a "yes", or $e_1' \approx F\ e_2$ or $F\ e_1 \approx e_2'$ terminate with a "yes" or a "no".

In other words, we view solving a convertibility problem as searching a tree of possible proofs of convertibility or non-convertibility. Using concurrency and fair interleaving, our approach performs a breadth-first traversal of the proof search tree to find the simplest possible proof in this proof space. In contrast, existing convertibility checkers perform a mostly depth-first traversal of the proof search tree, using strategies to select which branch to explore first. Sometimes, they go down a very long branch and fail to produce a proof in reasonable time. In the worst case, our breadth-first approach can take time exponential in the length of the shortest proof, but this is still preferable to the existing depth-first approaches, which can take arbitrarily long.

With so many convertibility subproblems being generated and solved in parallel, it is crucial to avoid duplicating computations between subproblems. For instance, if $G\ x = 1 + x$, the problem $G\ (F\ 20) \approx G\ (F\ 19)$ generates two main subproblems: $F\ 20 \approx F\ 19$, to compare the arguments to G, and $1 + F\ 20 \approx 1 + F\ 19$, after unrolling G on both sides. We really want to evaluate $F\ 20$ and $F\ 19$ only once, not twice each. To this end, we systematically use lazy evaluation to share computations within expressions, as in $(\lambda x.\, x + x)\ (F\ 20)$, and between convertibility problems, as in the example above.

Since we are testing the convertibility of lambda-terms, we need a notion of lazy evaluation that extends beyond weak reduction (as in Haskell and other functional languages) to include strong reduction (within the body of a lambda-abstraction). This is presented in §4. Since we interleave the executions of multiple evaluations and multiple convertibility problems, we need a formulation of lazy evaluation that plays well with concurrency, which is presented in the next section.

3 Expressing Laziness with a Process Calculus

The following artificial example illustrates the features we need from the metalanguage used to describe our convertibility testing algorithm.

$$\begin{aligned} H(n) \quad = \quad & \texttt{let}\ a = F(n)\ \texttt{and}\ b = G(n)\ \texttt{in} \\ & \texttt{if}\ n < 0\ \texttt{then}\ 1\ \texttt{else}\ a \times a \times b \end{aligned}$$

We would like to avoid unnecessary evaluations of $F(n)$ and $G(n)$ when evaluating $H(n)$. Namely:

Processes:	$P ::= \alpha \,!\, E$	send E on channel α
	$\mid P_1 \parallel P_2$	parallel composition
	$\mid \nu\alpha.\, P$	channel creation
Expressions:	$E ::= x \mid v$	variables, values
	$\mid E?$	receive a value from channel E
	$\mid F\ E_1\ \cdots\ E_n$	function applications
	$\mid C\ E_1\ \cdots\ E_n$	data constructor applications
Values:	$v ::= \alpha \mid C\ v_1\ \cdots\ v_n$	channels, constructors

Structural equivalences:

$$P_1 \parallel P_2 = P_2 \parallel P_1$$
$$P_1 \parallel (P_2 \parallel P_3) = (P_1 \parallel P_2) \parallel P_3$$
$$(\nu\alpha.\, P_1) \parallel P_2 = \nu\alpha.\, (P_1 \parallel P_2) \quad \text{if } \alpha \text{ not free in } P_2$$

Generic reduction rule:

$$\alpha \,!\, v \parallel \Gamma[\alpha?] \longrightarrow \alpha \,!\, v \parallel \Gamma[v]$$

(Plus: specific reduction rules $\alpha \,!\, F(\ldots) \longrightarrow \ldots$ for specific functions F.)

Fig. 1. The simple process calculus used as a metalanguage in this article.

- the bindings of a and b should be *lazy*, so that $F(n)$ and $G(n)$ are not evaluated at all if $n < 0$;
- the computations bound to a and b should be *shared* between multiple uses of these variables, so that $F(n)$ and $G(n)$ are evaluated only once if $n \geq 0$, even though a is used twice;
- the evaluations of $F(n)$ and $G(n)$ should proceed *in parallel*, or by *fair interleaving*, so that $a \times a \times b$ produces 0 as soon as one of $F(n)$ or $G(n)$ returns 0, without waiting for the other computation to terminate.

To support these features, we will use a simple process calculus loosely inspired by the π-calculus [Milner 1999]. As summarized in Fig. 1, we have *processes* P that execute in parallel ($P_1 \parallel P_2$) and communicate values over *channels* ($\alpha, \beta, \gamma, \ldots$). The process $\alpha \,!\, E$ computes the value of expression E and sends it over channel α. In an expression, $\alpha?$ denotes the value read from channel α when it is available. Finally, $\nu\alpha.\, P$ creates a fresh channel name α for the duration of the execution of P.

Using this notation, here is the process that computes $H(n)$ and returns its value on channel γ:

$$\gamma \,!\, H(n) \;=\; \nu\alpha, \beta.\ \alpha \,!\, F(n) \parallel \beta \,!\, G(n) \parallel \gamma \,!\, \texttt{if}\ n < 0\ \texttt{then}\ 1\ \texttt{else}\ \alpha? \times \alpha? \times \beta?$$

The two bound variables a and b are represented by two fresh channels α and β. Their bindings are represented by the two processes $\alpha \,!\, F(n)$ and $\beta \,!\, G(n)$ that run in parallel with the body of H.

The syntax and semantics of the process calculus are summarized in Fig. 1. The crucial part is the value communication rule:

$$\alpha \,!\, v \parallel \Gamma[\alpha?] \longrightarrow \alpha \,!\, v \parallel \Gamma[v]$$

It says that if a sending process $\alpha \,!\, E$ has reduced to $\alpha \,!\, v$, where v is the value of E, any receiver $\alpha?$ in any context Γ can be replaced by the value v. Unlike in the π-calculus, the sending process $\alpha \,!\, v$ remains unchanged, so that all present or future occurrences of $\alpha?$ can also be replaced by v.

Does the encoding of H in terms of processes satisfy the list of requirements above?

- *Sharing* of computations bound to variables is enforced by the communication rule. In $H(n)$ above, $F(n)$ is computed only once, and its value v replaces the two occurrences of $\alpha?$.

- *Parallelism* is inherent in the process calculus encoding of H. The evaluations of $F(n)$ and $G(n)$ can be freely interleaved, and we can give a parallel semantics to the multiplication operator: $0 \times E \longrightarrow 0$ and $E \times 0 \longrightarrow 0$, even if E is blocked on a receive operation.
- *Laziness* is not guaranteed by the process calculus encoding, only *non-strictness*: the evaluations of $F(n)$ and $G(n)$ can start right away, but can also be delayed until the values of $\alpha?$ and $\beta?$ are required to make progress. However, laziness can be enforced by an appropriate *scheduling* of process reductions. Typically, the processes $\alpha\,!\,F(n)$ and $\beta\,!\,G(n)$ should not be reduced until the values of $\alpha?$ and $\beta?$ are needed to make progress in the process that sends on γ, that is, until n was tested nonnegative.

More generally, we can encode ML-style explicit laziness, presented as two expressions: `lazy` E, which produces a thunk that evaluates E on demand, and `force` E, which forces the thunk E and returns its value. We represent thunks by channels; thus, `force` E is simply $E?$, and `lazy` has the following reduction rule:

$$\alpha\,!\,\Delta[\texttt{lazy}\ E] \longrightarrow \nu\beta.\,\alpha\,!\,\Delta[\beta] \parallel \beta\,!\,E$$

where Δ is an expression evaluation context. Without scheduling restrictions, the evaluation of E can start immediately, making `lazy` E behave like a future. To enforce laziness, we need to perform only the reductions that are necessary to produce the final value on the result channel α. Consider the following typical intermediate evaluation state:

$$\alpha\,!\,\Delta_0[\alpha_1?] \parallel \alpha_1\,!\,\Delta_1[\alpha_2?] \parallel \cdots \parallel \alpha_{n-1}\,!\,\Delta_{n-1}[\alpha_n?] \parallel \alpha_n\,!\,E \parallel P$$

where E can reduce and the Δ_i are expression evaluation contexts. There is a chain of computations waiting for values to be sent on channels: α is waiting for α_1, which is waiting for α_2, all the way to α_{n-1}, which is waiting for α_n, which is not waiting on anyone and can make progress (since E can reduce). Therefore, the reduction of E is the one that must be performed at this point. Other reductions in the remaining processes P might be possible, but are not required to progress on the evaluation of α, so they are not performed.

4 Strong Call-by-Need Reduction

We now use our process calculus notation to describe algorithms that reduce lambda-terms to weak head normal form, then to normal form. Our algorithms implement call-by-need strategies, using channels and parallel processes to share the reductions of subterms. However, they make no attempt at sharing subcontexts the way optimal reduction algorithms do.

4.1 The Source Language

While our approach extends to richer functional languages, this paper considers only the pure untyped lambda-calculus, and its extension with defined constants c.

Pure lambda terms: $\quad t, u ::= x \mid \lambda x.t \mid t\ u$

Extended lambda terms: $\quad t, u ::= x \mid \lambda x.t \mid t\ u \mid c$

Defined constants are constants that are bound to terms at top-level. For example, this is written `def` $c := t$ in Lean and `Definition` $c := t.$ in Rocq. Unlike ordinary constants, which are opaque, defined constants can be expanded (replaced by their definitions) at any time during computation.

4.2 Reduction to Weak Head Normal Form

To compute weak head normal forms (WHNF) of pure lambda terms, we use an environment machine inspired by Krivine's machine [Krivine 2007], with the main difference that environments e map variables not to unevaluated thunks, but to channels connected to processes that evaluate these thunks. The machine produces values of the following shape:

Values:	$v ::= \langle x, t, e\rangle$	closure of function $\lambda x.\, t$ by environment e
	$\mid [x\ s]$	free variable x applied to arguments s
Environments:	$e ::= \{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}$	mapping from variables to channels
Stacks:	$s ::= \alpha_1 \cdot \alpha_2 \cdots \alpha_n \cdot \epsilon$	lists of arguments (channels)

To compute the WHNF of term t in environment e, the machine starts in state $\texttt{eval}\ t\ e$ and performs the following transitions.

$$\begin{aligned}
\alpha \,!\, \texttt{eval}\ t\ e &\longrightarrow \alpha \,!\, \texttt{reduce}\ t\ e\ \epsilon \\
\alpha \,!\, \texttt{reduce}\ (t\ u)\ e\ s &\longrightarrow \nu\beta.\, \alpha \,!\, \texttt{reduce}\ t\ e\ (\beta \cdot s) \parallel \beta \,!\, \texttt{eval}\ u\ e \\
\alpha \,!\, \texttt{reduce}\ (\lambda x.t)\ e\ s &\longrightarrow \alpha \,!\, \texttt{apply}\ \langle x, t, e\rangle\ s \\
\alpha \,!\, \texttt{reduce}\ x\ e\ s &\longrightarrow \alpha \,!\, \texttt{apply}\ e(x)?\ s \quad \text{if } x \in \mathrm{Dom}(e) \\
\alpha \,!\, \texttt{reduce}\ x\ e\ s &\longrightarrow \alpha \,!\, \texttt{apply}\ [x]\ s \quad \text{if } x \notin \mathrm{Dom}(e) \\
\alpha \,!\, \texttt{apply}\ v\ \epsilon &\longrightarrow \alpha \,!\, v \\
\alpha \,!\, \texttt{apply}\ \langle x, t, e\rangle\ (\beta \cdot s) &\longrightarrow \alpha \,!\, \texttt{reduce}\ t\ (e + x \mapsto \beta)\ s \\
\alpha \,!\, \texttt{apply}\ [x\ s']\ s &\longrightarrow \alpha \,!\, [x\ (s' \cdot s)]
\end{aligned}$$

In the $\texttt{reduce}\ t\ e\ s$ state, the machine traverses the spine of applications of t, recording arguments on the stack s. If t is an application $t\ u$, the machine sets up a new process $\texttt{eval}\ u\ e$ that reduces u and sends its value on a fresh channel β, then pushes β on the stack and proceeds with the reduction of t. In all other cases, the machine switches to the $\texttt{apply}\ v\ s$ state, where v is the value of t: a closure if t is a function abstraction, a neutral value $[x]$ if t is a free variable x, and the value read from channel $e(x)$ if x is a bound variable.

In the $\texttt{apply}\ v\ s$ state, if s is empty, evaluation is finished and v is returned. If s is not empty and v is a function closure, a β-reduction step is performed and the machine resumes in the reduce state. If v is a constant or a free variable already applied to arguments s', the arguments s are added to s'.

In terms of reduction strategies, this machine implements non-strict evaluation with sharing (of the evaluation of a function argument). The process $\beta \,!\, \texttt{eval}\ u\ e$ that is created when the application $t\ u$ is reduced can start executing immediately or stay idle until the value of a variable x bound to u is needed for the first time. At that time, the machine computes $e(x)?$, that is, $\beta?$, forcing the process $\beta \,!\, \texttt{reduce}\ u\ e\ \epsilon$ to evaluate to $\beta \,!\, v$ for some value v. If the value of x is needed again later, it is obtained from this $\beta \,!\, v$ process; no recomputation is required. Therefore, depending on the scheduling of processes, the machine implements call-by-value, call-by-need, or any strategy "in between", but not call-by-name. Call-by-need can be obtained by restricting the scheduling of processes appropriately, as outlined at the end of §3.

4.3 Reduction under Lambdas and Normalization

Normal forms can be computed by alternating evaluation phases, which produce values (WHNFs), and reification phases, which turn these values into terms in normal forms. This is similar to the "eval" and "reify" phases of normalization by evaluation [Berger et al. 1998] and of type-directed partial evaluation [Danvy 1996]. For example, if evaluation produces a function closure $\langle x, t, e\rangle$, we can apply it to a fresh free variable $[y]$, obtaining a value v, then recursively reify v to a normal-form term t, and finally produce the normal form $\lambda y.\, t$.

A naive implementation of this normalization procedure can duplicate evaluations, however. For example, consider the term $(\lambda f.\, g\ f\ f)\ (\lambda x.\, t)$ where g is a free variable. Evaluation produces the value $[g\ \beta\ \beta]$, where β is a channel that produces a closure for $\lambda x.\, t$. Naive reification will reify each occurrence of this closure independently, causing t to be normalized twice.

As described in [Biernacka et al. 2022] and independently observed by us, this unsharing of function values can be avoided by anticipating the need to reduce in the function body t when creating the closure for the function $\lambda x.\, t$. In our channel-based presentation, this means adding two components to every function closure $\langle x, t, e \rangle$: a fresh free variable y used for normalization, and a channel δ connected to a process that evaluates on demand the application of t to y.

Values: $v ::= \langle x, t, e, y, \delta \rangle$ enriched function closure
$\mid [x\ s]$

The evaluation rule for function abstractions becomes:

$$\alpha\,!\,\texttt{reduce}\ (\lambda x.t)\ e\ s \longrightarrow \nu\gamma\delta.\ \alpha\,!\,\texttt{apply}\ \langle x, t, e, y, \delta \rangle\ s \parallel \delta\,!\,\texttt{eval}\ t\ (e + x \mapsto \gamma) \parallel \gamma\,!\,[y]$$
$$\text{where } y \text{ is a fresh variable}$$

The extra components of closures are ignored during application:

$$\alpha\,!\,\texttt{apply}\ \langle x, t, e, y, \delta \rangle\ (\beta \cdot s) \longrightarrow \alpha\,!\,\texttt{reduce}\ t\ (e + x \mapsto \beta)\ s$$

With this twist on closures, we can define $\texttt{nf}\ t$, the normalization of a closed term t, and $\texttt{reify}\ v$, the reification of a value v as a lambda-term, by the following rules:

$$\alpha\,!\,\texttt{nf}\ t \longrightarrow \nu\beta.\ \alpha\,!\,\texttt{reify}\ \beta? \parallel \beta\,!\,\texttt{eval}\ t\ \{\,\}$$
$$\texttt{reify}\ \langle x, t, e, y, \delta \rangle \longrightarrow \lambda y.\, \texttt{reify}\ \delta?$$
$$\texttt{reify}\ [x\ \beta_1 \cdots \beta_n] \longrightarrow x\ (\texttt{reify}\ \beta_1?)\ \cdots\ (\texttt{reify}\ \beta_n?)$$

4.4 Defined Constants

We now extend the evaluation and reification approach of sections 4.2 and 4.3 to defined constants. Unlike `let`-bound variables, which have local scope and must therefore be expanded during reduction to WHNF, defined constants have global scope and can remain unexpanded in WHNFs, resulting in values $[c\ s]$ that carry an unexpanded constant c and a possibly empty list of arguments s to which c is applied. This enables faster convertibility testing of two terms, as outlined in §2 and developed in §5. However, when unfolding a defined constant c to reduce a value $[c\ s]$, we must be careful not to duplicate the evaluation of the definition t of c or its application to the arguments s. To this end, we reuse the approach of §4.3: we attach a channel δ to each $[c\ s]$ value and connect this channel to a process that evaluates this application of c on demand.

Values: $v ::= \langle x, t, e, y, \delta \rangle$
$\mid [x\ s]$
$\mid [c\ s]@\delta$ constant c applied to s, with actual value available from δ

Given a set of constant definitions $c_1 := t_1, \ldots, c_n := t_n$, we define a global environment K that maps constants to fresh channels, and a process KP that evaluates the t_i on demand and sends their values to those channels.

$$K = \{c_1 \mapsto \alpha_1; \ldots; c_n \mapsto \alpha_n\}$$
$$KP = \alpha_1\,!\,\texttt{eval}\ t_1\ \{\,\} \parallel \cdots \parallel \alpha_n\,!\,\texttt{eval}\ t_n\ \{\,\}$$

When we evaluate a constant c, we pair it with the channel $K(c)$, thus ensuring that the evaluation of its definition t is properly shared and non-strict:

$$\alpha\,!\,\texttt{reduce}\ c\ e\ s \longrightarrow \alpha\,!\,\texttt{apply}\ ([c]@K(c))\ s$$

New channels and new processes are created when a constant is applied to new arguments:

$$\alpha\,!\,\texttt{apply}\ ([c\ s']@\delta)\ s \longrightarrow \nu\gamma.\ \alpha\,!\,[c\ (s' \cdot s)]@\gamma \parallel \gamma\,!\,\texttt{apply}\ \delta?\ s$$

The normalization procedure of §4.3 is modified as follows:

$$\begin{aligned}
\alpha\,!\,\mathtt{nf}\ t &\longrightarrow \nu\beta.\,\alpha\,!\,\mathtt{reify}\ \beta?\ \|\ \beta\,!\,\mathtt{eval}\ t\ \{\}\ \|\ KP\\
\mathtt{reify}\ \langle x,t,e,y,\delta\rangle &\longrightarrow \lambda y.\,\mathtt{reify}\ \delta?\\
\mathtt{reify}\ [x\ \beta_1\cdots\beta_n] &\longrightarrow x\ (\mathtt{reify}\ \beta_1?)\ \cdots\ (\mathtt{reify}\ \beta_n?)\\
\mathtt{reify}\ ([c\ s]@\delta) &\longrightarrow \mathtt{reify}\ \delta?
\end{aligned}$$

5 Convertibility Testing

5.1 The Basic Algorithm

Just like normalization can be viewed as a combination of evaluation and reification, determining whether two terms t, t' are convertible can be viewed as a combination of evaluation of t and t' and comparison of the resulting values v, v'. For example, if v and v' are applications of free variables x and x', we need to check that $x = x'$ and that the arguments are pairwise convertible.

More precisely, to test the convertibility of t and t' and send the resulting Boolean value to channel α, we start with the following process:

$$\nu\beta\beta'.\,\alpha\,!\,\mathsf{conv}^?\ \beta\ \beta'\ \epsilon\ \|\ \beta\,!\,\mathtt{eval}\ t\ \{\}\ \|\ \beta'\,!\,\mathtt{eval}\ t'\ \{\}\ \|\ KP$$

Here, `eval` is the reduction to WHNF from §4, and $\mathsf{conv}^?\ \beta\ \beta'\ \xi$ is the comparison of the values read from the channels β and β' up to a renaming ξ of free variables (a list of pairs of variables that are considered equal). The function $\mathsf{conv}^?$ and the auxiliary functions conv and conv^* are defined by the following rules:

$$\begin{aligned}
\alpha\,!\,\mathsf{conv}^?\ \beta\ \beta'\ \xi &\longrightarrow \alpha\,!\,\mathsf{conv}\ \beta?\ \beta'?\ \xi\\
\alpha\,!\,\mathsf{conv}\ \langle x,t,e,y,\delta\rangle\ \langle x',t',e',y',\delta'\rangle\ \xi &\longrightarrow \alpha\,!\,\mathsf{conv}^?\ \delta\ \delta'\ ((y,y')\cdot\xi)\\
\alpha\,!\,\mathsf{conv}\ [x\ s]\ [x'\ s']\ \xi &\longrightarrow \alpha\,!\,\mathsf{conv}^*\ s\ s'\ \xi \quad \text{if } (x,x')\in\xi \text{ and } |s|=|s'|\\
\alpha\,!\,\mathsf{conv}\ v_1\ v_2\ \xi &\longrightarrow \alpha\,!\,\mathsf{F} \quad \text{in all other cases}\\
\alpha\,!\,\mathsf{conv}^*\ \beta_1\cdots\beta_n\ \beta'_1\cdots\beta'_n\ \xi &\longrightarrow \nu\gamma_1\ldots\gamma_n.\ \alpha\,!\,\gamma_1?\wedge\ldots\wedge\gamma_n?\\
&\qquad \|\ \gamma_1\,!\,\mathsf{conv}^?\ \beta_1\ \beta'_1\ \xi\ \|\ \cdots\ \|\ \gamma_n\,!\,\mathsf{conv}^?\ \beta_n\ \beta'_n\ \xi
\end{aligned}$$

(We have only shown the cases for pure lambda terms. The cases for defined constants c are discussed in §5.2.)

If the values read from channels β and β' are two closures $\langle x,t,e,y,\delta\rangle$ and $\langle x',t',e',y',\delta'\rangle$, we recursively compare the WHNFs of the corresponding function bodies, which can be read from δ and δ', up to equality of the variables y and y', which we express by adding the pair (y,y') to the current renaming ξ. (The freshness requirements on the variables stored in extended function closures guarantee that y and y' are not already involved in the renaming ξ, as shown in the Rocq proof described in §8.)

If the values read from channels β and β' are two applications of free variables $[x\ s]$ and $[x'\ s']$, we check that the variables x and x' are equal up to the current renaming ξ, that the argument lists s and s' have the same length, and that the arguments are pairwise convertible, as expressed by $\mathsf{conv}^*\ s\ s'\ \xi$. We could have written

$$\alpha\,!\,\mathsf{conv}\ [x\ \beta_1\cdots\beta_n]\ [x'\ \beta'_1\cdots\beta'_n]\ \xi \longrightarrow \alpha\,!\,\mathsf{conv}^?\ \beta_1\ \beta'_1\ \xi\wedge\ldots\wedge\mathsf{conv}^?\ \beta_n\ \beta'_n\ \xi \quad \text{if } (x,x')\in\xi$$

The more convoluted definition above, using auxiliary processes and fresh channels, makes it obvious that the sub-convertibility tests $\mathsf{conv}^?\ \beta_i\ \beta'_i\ \xi$ can run in parallel.

In the definition of conv^*, the $\wedge$ operator is "parallel and": it reduces to false as soon as one of its arguments reduces to false, even if the other argument is blocked reading from a channel.

$$\mathsf{T} \wedge \mathsf{T} \to \mathsf{T} \qquad \mathsf{F} \wedge E \to \mathsf{F} \qquad E \wedge \mathsf{F} \to \mathsf{F}$$

Combined with the non-strictness of `eval` (reduction to WHNF), this gives our convertibility test nice early-failure properties. For example, if t_1 and t_2 are terms that are expensive to compute, and x_1 and x_2 are two different free variables, the test determines that $x_1\ t_1 \not\approx x_2\ t_2$ without ever computing t_1 nor t_2: the two terms reduce to the values $v_1 = [x_1\ \beta_1]$ and $v_2 = [x_2\ \beta_2]$, where β_1 and β_2 are channels connected to processes that evaluate t_1 and t_2, and the comparison $\mathsf{conv}\ v_1\ v_2\ \xi$ returns F immediately, since $(x_1, x_2) \notin \xi$.

Likewise, the test can determine that $x\ x_1\ t_1 \not\approx x\ x_2\ t_2$ without computing t_1 or t_2 in full. Two convertibility subproblems are generated, one corresponding to $x_1 \approx x_2$ and the other to $t_1 \approx t_2$:

$$\alpha\,!\,\gamma_1? \wedge \gamma_2? \parallel \gamma_1\,!\,\mathsf{conv}\ [x_1]\ [x_2]\ \xi \parallel \gamma_2\,!\,\mathsf{conv}^?\ \beta_1\ \beta_2\ \xi \parallel \cdots$$

Assuming fair interleaving, $\mathsf{conv}\ [x_1]\ [x_2]\ \xi$ quickly reduces to F, causing F to be sent on α, while $\mathsf{conv}^?\ \beta_1\ \beta_2\ \xi$ has barely started to evaluate t_1 and t_2

Since reduction to WHNF preserves sharing, our convertibility test can also avoid some repeated evaluations that a more naive algorithm would perform. For example, to determine that $(\lambda x.\, f\ x\ x)\ t \approx (\lambda y.\, f\ y\ y)\ t$, where f is a free variable, it evaluates t only twice, once for the left-hand side occurrence of t and once for the right-hand side occurrence. (Section 6.3 shows one way to also share the convertibility processes, not just the evaluation processes. Section 6.4 shows one way to avoid evaluating t at all.)

5.2 Handling Defined Constants

We now extend the basic convertibility test from §5.1 to support defined constants as introduced in §4.4. As the examples in §2 demonstrate, there is no optimal strategy to handle defined constants in a convertibility test: in general, constants have to be unfolded (replaced by their definitions) to determine convertibility; but in some cases, the test can conclude that two terms are convertible without unfolding constants, treating them as simple names instead, which can avoid unnecessary computations. Our algorithm strives to keep all possibilities open, exploring them in parallel and relying on non-strict evaluation and early-failure and early-success optimizations to shorten this exploration.

Consider again the comparison $\mathsf{conv}\ v\ v'\ \xi$ of two values. If one value is a possibly applied, defined constant $[c\ s]@\delta$ and the other is a different kind of value, there is no choice but to unfold the definition of c and continue reducing to WHNF. The resulting value can simply be read from δ, since the evaluation that produced $[c\ s]@\delta$ anticipated this need (as explained in §4.4).

$$\alpha\,!\,\mathsf{conv}\ [c\ s]@\delta\ v\ \xi \to \alpha\,!\,\mathsf{conv}\ \delta?\ v\ \xi \qquad \text{if } v \text{ is not a constant}$$
$$\alpha\,!\,\mathsf{conv}\ v\ [c\ s]@\delta\ \xi \to \alpha\,!\,\mathsf{conv}\ v\ \delta?\ \xi \qquad \text{if } v \text{ is not a constant}$$

If the values v and v' are the same constant, or more generally the same application of a constant, that is, $[c\ s]@\delta$ and $[c'\ s']@\delta'$ with $\delta = \delta'$, we know that $c = c'$ and $s = s'$ and we can return T immediately.

$$\alpha\,!\,\mathsf{conv}\ [c\ s]@\delta\ [c'\ s']@\delta\ \xi \to \mathsf{T}$$

If the values v and v' are applications of different defined constants $[c\ s]@\delta$ and $[c'\ s']@\delta'$, with $c \neq c'$, it is tempting to unfold both c and c' in one step. However, as exemplified in §2, this can result in missed opportunities to conclude quickly that the two values are convertible. Instead,

we explore the two possibilities (unfold c or unfold c') in parallel, and choose whichever result is obtained first.

$$\begin{aligned}\alpha\ !\ \mathsf{conv}\ [c\ s]@\delta\ [c'\ s']@\delta'\ \xi \to \nu\beta\gamma.\ & \alpha\ !\ \beta?\oplus\gamma? && \text{if } c\neq c' \text{ or } |s|\neq|s'|\\ \|\ & \beta\ !\ \mathsf{conv}\ [c\ s]@\delta\ \delta'?\ \xi\\ \|\ & \gamma\ !\ \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi\end{aligned}$$

The Boolean choice operator $\oplus$ returns whichever of its two arguments terminates first, knowing that they evaluate to the same Boolean value. (Unfolding a constant in the left-hand side or in the right-hand side doesn't change the Boolean value of the convertibility test.)

$$\mathsf{T}\oplus E\to\mathsf{T}\qquad \mathsf{F}\oplus E\to\mathsf{F}\qquad E\oplus\mathsf{T}\to\mathsf{T}\qquad E\oplus\mathsf{F}\to\mathsf{F}$$

Finally, if the two values being compared are applications $[c\ s]@\delta$ and $[c\ s']@\delta'$ of the same defined constant c to the same number of arguments, a third possibility arises: just compare the arguments pairwise and return T if they are pairwise convertible, as in the case of free variables or abstract constants.

$$\begin{aligned}\alpha\ !\ \mathsf{conv}\ [c\ s]@\delta\ [c\ s']@\delta'\ \xi \to \nu\beta\gamma\eta.\ & \alpha\ !\ \eta?\ \overrightarrow{\oplus}\ (\beta?\oplus\gamma?) && \text{if } |s|=|s'|\\ \|\ & \beta\ !\ \mathsf{conv}\ [c\ s]@\delta\ \delta'?\ \xi\\ \|\ & \gamma\ !\ \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi\\ \|\ & \eta\ !\ \mathsf{conv}^*\ s\ s'\ \xi\end{aligned}$$

The "biased choice" operator $E_1\ \overrightarrow{\oplus}\ E_2$ returns the Boolean value of E_2 under the assumption that E_1 implies E_2. Therefore, if E_1 terminates early with value T, $E_1\ \overrightarrow{\oplus}\ E_2$ can return T without waiting for E_2 to terminate; and if E_2 terminates early, its value can be returned immediately, without waiting for E_1 to terminate.

$$\mathsf{T}\ \overrightarrow{\oplus}\ E\to\mathsf{T}\qquad E\ \overrightarrow{\oplus}\ \mathsf{T}\to\mathsf{T}\qquad E\ \overrightarrow{\oplus}\ \mathsf{F}\to\mathsf{F}$$

Here, we use a combination $E_1\ \overrightarrow{\oplus}\ (E_2\oplus E_3)$ of biased choice and regular choice, where E_1 is "the argument lists s and s' are pairwise convertible", E_2 is "$c\ s\approx c\ s'$ after unrolling c in the left-hand side", and E_3 is "$c\ s\approx c\ s'$ after unrolling c in the right-hand side". As soon as one of E_1, E_2 or E_3 returns T, we know that $c\ s$ and $c\ s'$ are convertible and can immediately return T. As soon as one of E_2 or E_3 returns F, we know that $c\ s$ and $c\ s'$ are not convertible and can return F. If E_1 returns F, we know that the arguments are not convertible but cannot conclude anything about the convertibility of $c\ s$ and $c\ s'$. (Consider $c=\lambda x.\,\lambda y.\,x$ and s, s' differing in their second elements.)

6 Extensions

6.1 Avoiding Redundant Unfolding of Constants

The progressive unfolding of defined constants, on either the left-hand side or the right-hand side but not both sides simultaneously, naturally leads to duplicated conv tests. For example, if $c\neq c'$ are defined constants, and assuming that $\delta?$ reduces to v and $\delta'?$ to v',

$$\begin{aligned}\alpha\ !\ \mathsf{conv}\ [c]@\delta\ [c']@\delta'\ \xi &\to^+ \nu\beta\gamma.\ \alpha\ !\ \beta?\oplus\gamma?\ \|\ \beta\ !\ \mathsf{conv}\ [c]@\delta\ v'\ \xi\ \|\ \gamma\ !\ \mathsf{conv}\ v\ [c']@\delta'\ \xi\\ &\to^+ \nu\beta\gamma.\ \alpha\ !\ \beta?\oplus\gamma?\ \|\ \beta\ !\ \mathsf{conv}\ v\ v'\ \xi\ \|\ \gamma\ !\ \mathsf{conv}\ v\ v'\ \xi\end{aligned}$$

We have two identical processes $\mathsf{conv}\ v\ v'$ that run concurrently. Evaluations within v and v' will be shared, and one processes will become unneeded as soon as the other terminates. Nonetheless, some convertibility testing work is duplicated, and the number of convertibility processes can increase exponentially.

To avoid this, we introduce *applied frozen constants* as a new type of values:

$$\begin{array}{lll} \text{Values:} & v ::= \langle x, t, e, y, \delta\rangle \mid [x\ s] \mid [c\ s]@\delta & \\ & \quad \mid [c\ s] & \text{frozen constant } c \text{ applied to } s \end{array}$$

Frozen constants cannot be unfolded. During evaluation, they behave like free variables:

$$\alpha\ !\ \mathsf{apply}\ [c\ s]\ s' \to \alpha\ !\ [c\ (s \cdot s')]$$

Consequently, the only way for a value v to be convertible with a frozen constant $[c\ s]$ is for v to reduce (possibly by unfolding) to the same constant c applied to some arguments s', with the arguments s and s' being pairwise convertible.

When comparing two applied defined constants $[c\ s]@\delta$ and $[c'\ s']@\delta'$, we still create two parallel processes, one that unfolds c and another that unfolds c'. However, in the process that unfolds c', we freeze c, replacing $[c\ s]@\delta$ with $[c\ s]$. This way, further unfoldings of c will only take place in one of the processes, but not in both.

$$\begin{array}{rll} \alpha\ !\ \mathsf{conv}\ [c\ s]@\delta\ [c'\ s']@\delta'\ \xi \to \nu\beta\gamma.\ \alpha\ !\ \beta?\ \overrightarrow{\oplus}\ \gamma? & & \text{if } c \neq c' \text{ or } |s| \neq |s'| \\ \qquad \| \ \beta\ !\ \mathsf{conv}\ [c\ s]\ \delta'?\ \xi & & \\ \qquad \| \ \gamma\ !\ \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi & & \\ \alpha\ !\ \mathsf{conv}\ [c\ s]@\delta\ [c\ s']@\delta'\ \xi \to \nu\beta\gamma\eta.\ \alpha\ !\ \eta?\ \overrightarrow{\oplus}\ (\beta?\ \overrightarrow{\oplus}\ \gamma?) & & \text{if } |s| = |s'| \\ \qquad \| \ \beta\ !\ \mathsf{conv}\ [c\ s]\ \delta'?\ \xi & & \\ \qquad \| \ \gamma\ !\ \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi & & \\ \qquad \| \ \eta\ !\ \mathsf{conv}^*\ s\ s'\ \xi & & \end{array}$$

Note the use of a biased choice in $\beta?\ \overrightarrow{\oplus}\ \gamma?$: the β process, which is the one that freezes c, may return F even though the two values are convertible; only the γ process returns an authoritative result.

When they appear as arguments in a convertibility test, applications of frozen constants $[c\ s]$ are treated almost like applications of free variables $[x\ s]$. However, a special case is needed when comparing an application of a frozen constant with an application of the same constant that is not frozen.

$$\begin{array}{rll} \alpha\ !\ \mathsf{conv}\ [c\ s]\ [c\ s']\ \xi \to \alpha\ !\ \mathsf{conv}^*\ s\ s'\ \xi & & \text{if } |s| = |s'| \\ \alpha\ !\ \mathsf{conv}\ [c\ s]\ [c\ s']@\delta'\ \xi \to \nu\beta\gamma.\ \alpha\ !\ \beta? \vee \gamma? & & \text{if } |s| = |s'| \\ \qquad \| \ \beta\ !\ \mathsf{conv}^*\ s\ s'\ \xi & & \\ \qquad \| \ \gamma\ !\ \mathsf{conv}\ [c\ s]\ \delta'?\ \xi & & \\ \alpha\ !\ \mathsf{conv}\ [c\ s]@\delta\ [c\ s']\ \xi \to \nu\beta\gamma.\ \alpha\ !\ \beta? \vee \gamma? & & \text{if } |s| = |s'| \\ \qquad \| \ \beta\ !\ \mathsf{conv}^*\ s\ s'\ \xi & & \\ \qquad \| \ \gamma\ !\ \mathsf{conv}\ \delta?\ [c'\ s]\ \xi & & \end{array}$$

6.2 Handling η-Conversion

Several proof assistants consider terms equal up to η-conversion of functions: $\lambda x.\, M\ x \approx M$. Our algorithm can be extended to handle η-conversion as well. The key observation is that, in the presence of η-conversion, in order to prove $\lambda x.\, M \approx N$, it suffices to show $M[x := y] \approx N\ y$ where y is a fresh variable. However, this should only be attempted when it is safe and profitable to do so. For example, if N is a lambda-abstraction, this approach is not profitable; and if N is a pair (N_1, N_2) (in an extension of our lambda-calculus with pairs), this approach creates a term $(N_1, N_2)\ y$ that goes wrong during evaluation.

The first safe and profitable case is the comparison of a function value with a neutral value, that is, an applied free variable $[z\ s]$ or an applied frozen constant $[c\ s]$. Then, we can safely apply the

neutral value to a fresh variable.

$$\alpha \,!\, \mathsf{conv}\ \langle x, t, e, y, \delta\rangle\ [z\ s]\ \xi \longrightarrow \nu\beta.\, \alpha \,!\, \mathsf{conv}\ \delta?\ [z\ (s \cdot \beta)]\ ((y, y') \cdot \xi) \parallel \beta \,!\, [y'] \quad y' \text{ fresh}$$

(We omit three similar rules: one with $[c\ s]$ instead of $[z\ s]$ and two with the function value and the neutral value swapped.)

When one side is an abstraction and the other is a defined constant, we explore two possibilities in parallel: unfolding the constant or applying it to a fresh free variable. In the latter case, we must prevent further unfolding of the constant by using the frozen constant mechanism of §6.1. Unfolding the constant after applying it to a fresh free variable can be unsafe (*e.g.* if the constant expands to a pair) and is not profitable. The corresponding rule is as follows:

$$\begin{aligned}\alpha \,!\, \mathsf{conv}\ \langle x, t, e, y, \delta\rangle\ [c\ s]@\delta'\ \xi \longrightarrow \nu\beta\gamma\eta.\ & \alpha \,!\, \eta? \overrightarrow{\oplus}\ \gamma? && y' \text{ fresh}\\ & \parallel \eta \,!\, \mathsf{conv}\ \delta?\ [c\ (s \cdot \beta)]\ ((y, y') \cdot \xi)\\ & \parallel \gamma \,!\, \mathsf{conv}\ \langle x, t, e, y, \delta\rangle\ \delta'?\ \xi\\ & \parallel \beta \,!\, [y']\end{aligned}$$

(Plus a similar rule for $\mathsf{conv}\ [c\ s]@\delta'\ \langle x, t, e, y, \delta\rangle\ \xi$.)

6.3 Sharing Convertibility Processes

As mentioned at the end of §5.1, our convertibility checker benefits from our lazy evaluator's ability to share the repeated evaluations of sub-terms. However, the convertibility tests themselves are not shared and can easily be duplicated.

Consider the test $(\lambda x.\, f\ x\ x)\ t \approx (\lambda y.\, f\ y\ y)\ u$ where f is a free variable. After the initial evaluation steps, we end up comparing two inert values $[f\ \beta\ \beta]$ and $[f\ \gamma\ \gamma]$:

$$\alpha \,!\, \mathsf{conv}\ [f\ \beta\ \beta]\ [f\ \gamma\ \gamma]\ \xi \parallel \beta \,!\, \mathtt{eval}\ t\ \{\,\} \parallel \gamma \,!\, \mathtt{eval}\ u\ \{\,\}$$

This process further reduces to

$$\alpha \,!\, \delta? \wedge \eta? \parallel \delta \,!\, \mathsf{conv}^?\ \beta\ \gamma\ \xi \parallel \eta \,!\, \mathsf{conv}^?\ \beta\ \gamma\ \xi \parallel \beta \,!\, \mathtt{eval}\ t\ \{\,\} \parallel \gamma \,!\, \mathtt{eval}\ u\ \{\,\}$$

The evaluations of t and u remain shared, but the convertibility test $\mathsf{conv}^?\ \beta\ \gamma\ \xi$ is duplicated. In some cases involving recursive functions, this can result in exponential duplication of convertibility tests (see the `perfect` example in §10).

This problem could be avoided by re-sharing $\mathsf{conv}^?\ \beta\ \gamma\ \xi$ processes as they are created, using hash-consing on the channels β and γ. On the example above, this re-sharing would result in

$$\alpha \,!\, \delta? \wedge \delta? \parallel \delta \,!\, \mathsf{conv}^?\ \beta\ \gamma\ \xi \parallel \beta \,!\, \mathtt{eval}\ t\ \{\,\} \parallel \gamma \,!\, \mathtt{eval}\ u\ \{\,\}$$

with a single $\mathsf{conv}^?\ \beta\ \gamma\ \xi$ process whose output is used twice.

Re-sharing convertibility processes also solves the issue with redundant unfolding of constants described in §6.1, without the need to introduce frozen constant values and treat them specially. However, frozen constant values have other uses, *e.g.* to handle η-conversion, and they are cheaper to implement than process re-sharing.

6.4 Sharing Identical Source Subterms

Evaluation processes $\mathtt{eval}\ t\ e$ carefully avoid duplicating computations when performing a beta-reduction or unfolding a constant. Consequently, $(\lambda x.\, x + x)\ t \approx 0$ evaluates t only once, and $1 + c \approx c + 1$ evaluates the definition of c only once. However, multiple occurrences of the same source subterm t are evaluated independently. For example, $t + t \approx 0$ evaluates t twice, and so does $1 + t \approx t + 1$.

To share more evaluations, we can `let`-bind some subterms of the source terms before starting the convertibility test. For example, $t + t$ can be replaced by $\mathtt{let}\ x = t\ \mathtt{in}\ x + x$. To share subterms

that occur on both sides of a convertibility test, we need `let` bindings that cover both sides, leading to generalized convertibility problems of the following form:

$$\texttt{let}\ x_1 = t_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = t_n\ \texttt{in}\ t \approx t'$$

For example, $1+t \approx t+1$ can be replaced by $\texttt{let}\ x = t\ \texttt{in}\ 1+x \approx x+1$. More generally, $C[t] \approx C'[t]$ can be replaced by $\texttt{let}\ x = t\ \texttt{in}\ C[x] \approx C'[x]$ where x is fresh, provided that t does not depend on variables that are bound by C or C'. This transformation includes as a special case the lifting of maximal free expressions described by Peyton Jones [1987, chapter 15] to implement full laziness [Balabonski 2012].

One way to implement the transformation outlined above is to perform hash-consing on the closed subterms of the original convertibility problem $t \approx t'$. This yields (in linear time) a DAG where multiple occurrences of the same closed subterm are shared. Then, we transcribe (in linear time) this DAG as a set of `let` bindings to obtain a generalized convertibility problem of the form shown above.

To evaluate this generalized convertibility problem, we set up the following process:

$$\begin{aligned}\nu\beta\beta'\gamma_1 \ldots \gamma_n.\ & \alpha\ !\ \mathsf{conv}^?\ \beta\ \beta'\ \epsilon \\ & \parallel \beta\ !\ \texttt{eval}\ t\ \{x_1 \mapsto \gamma_1, \ldots, x_n \mapsto \gamma_n\} \parallel \beta'\ !\ \texttt{eval}\ t'\ \{x_1 \mapsto \gamma_1, \ldots, x_n \mapsto \gamma_n\} \\ & \parallel \gamma_1\ !\ \texttt{eval}\ t_1\ \{\,\} \parallel \ldots \parallel \gamma_n\ !\ \texttt{eval}\ t_n\ \{x_1 \mapsto \gamma_1, \ldots, x_{n-1} \mapsto \gamma_{n-1}\}\end{aligned}$$

Since it is now possible for the same channel to appear on both sides of a convertibility sub-problem, we add a new early-success case:

$$\alpha\ !\ \mathsf{conv}^?\ \beta\ \beta'\ \xi \longrightarrow \mathsf{T} \quad \text{if } \beta = \beta'$$

For example, $c\ t \approx c\ t$, encoded as $\texttt{let}\ x = t\ \texttt{in}\ c\ x \approx c\ x$, triggers the new case above when comparing the two argument stacks for c, causing T to be returned without evaluating t at all.

7 An Explicitly-Scheduled Abstract Machine

The evaluation and convertibility testing functions of §4 and §5 do not enforce laziness: they allow evaluations to start reducing as soon as the corresponding processes are created, before we know that their results are needed. We now make scheduling explicit in these functions, specifying which processes should be reduced at each step, so that processes are not executed before their results are needed, and the executions of active processes are interleaved fairly.

The explicitly-scheduled rules are shown in figures 2 and 3. They can be viewed as the transitions of an abstract machine whose states are triples $P\ \left[W, Q\right]$ or, equivalently, as reduction rules for processes P annotated with scheduling information W, Q. In both cases,

- P is the parallel composition of a set of elementary processes $\alpha_1\ !\ E_1 \parallel \cdots \parallel \alpha_n\ !\ E_n$, identified by their channels α_i.
- W is a map from channels to sets of channels. It records which processes are waiting on the result of other processes: $W(\alpha) = \{\beta_1, \ldots, \beta_n\}$ means that the processes $\beta_1, \ldots, \beta_n$ are blocked waiting for the process α to send a value.
- Q is a queue of channels representing the active processes: the processes whose results are needed to advance the resolution of the current convertibility problem. Inactive processes are never scheduled for execution. However, a process can alternate between the "inactive" and "active" states as the convertibility problem progresses. There are no duplicates in this queue, ensuring fairness between the active processes.

Many of the abstract machine transitions simply perform round-robin execution of the active processes. They have the following shape:

$$\alpha\ !\ E \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow P' \parallel P\ \left[W, Q \cdot \alpha\right]$$

Initial state when testing convertibility of t_1 and t_2: ($\alpha \mapsto \{*\}$ means that α is always needed)

$$\alpha \,!\, \mathsf{conv}^?\ \beta\ \gamma\ \epsilon \parallel \beta \,!\, \mathtt{eval}\ t_1\ \epsilon \parallel \gamma \,!\, \mathtt{eval}\ t_2\ \epsilon \parallel KP\ \left[(\alpha \mapsto \{*\}), \alpha \cdot \epsilon\right]$$

Transitions for convertibility processes:

$$\alpha \,!\, \mathsf{conv}^?\ \beta\ \beta\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{T} \parallel P\ \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\alpha \,!\, \mathsf{conv}^?\ \beta\ \beta'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ \beta?\ \beta'?\ \xi \parallel P\ \left[W, Q \cdot \alpha\right] \quad \text{if } \beta \neq \beta'$$

Obtaining the values to be compared:

$$\alpha \,!\, \mathsf{conv}\ \beta?\ E'\ \xi \parallel \beta \,!\, v \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ v\ E'\ \xi \parallel \beta \,!\, v \parallel P\ \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathsf{conv}\ E\ \beta'?\ \xi \parallel \beta' \,!\, v' \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ E\ v'\ \xi \parallel \beta' \,!\, v' \parallel P\ \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathsf{conv}\ \beta?\ \beta'?\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ \beta?\ \beta'?\ \xi \parallel P\ \left[\mathbf{need}(\alpha, \beta, \mathbf{need}(\alpha, \beta', W, Q))\right]$$

$$\alpha \,!\, \mathsf{conv}\ \beta?\ v'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ \beta?\ v'\ \xi \parallel P\ \left[\mathbf{need}(\alpha, \beta, W, Q)\right]$$

$$\alpha \,!\, \mathsf{conv}\ v\ \beta'?\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ v\ \beta'?\ \xi \parallel P\ \left[\mathbf{need}(\alpha, \beta', W, Q)\right]$$

Comparing two values:

$$\alpha \,!\, \mathsf{conv}\ \langle x, t, e, y, \delta\rangle\ \langle x', t', e', y', \delta'\rangle\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}^?\ \delta\ \delta'\ ((y, y') \cdot \xi) \parallel P\ \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathsf{conv}\ [x\ s]\ [x'\ s']\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}^*\ s\ s'\ \xi \parallel P\ \left[W, Q \cdot \alpha\right] \quad \text{if } (x, x') \in \xi \text{ and } |s| = |s'|$$

$$\alpha \,!\, \mathsf{conv}\ [c\ s]@\delta\ [c'\ s']@\delta\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{T} \parallel P\ \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\alpha \,!\, \mathsf{conv}\ [c\ s]@\delta\ [c'\ s']@\delta'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \nu\beta\gamma.\ \alpha \,!\, \beta? \oplus \gamma? \parallel \beta \,!\, \mathsf{conv}\ [c\ s]@\delta\ \delta'?\ \xi \parallel \gamma \,!\, \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi \parallel P\ \left[W[\beta \mapsto \{\alpha\}, \gamma \mapsto \{\alpha\}], Q \cdot \beta \cdot \gamma\right] \quad \text{if } c \neq c' \text{ or } |s| \neq |s'|$$

$$\alpha \,!\, \mathsf{conv}\ [c\ s]@\delta\ [c\ s']@\delta'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \nu\beta\gamma\eta\zeta.\ \alpha \,!\, \eta? \overrightarrow{\oplus}\ \zeta? \parallel \zeta \,!\, \beta? \oplus \gamma? \parallel \beta \,!\, \mathsf{conv}\ [c\ s]@\delta\ \delta'?\ \xi \parallel \gamma \,!\, \mathsf{conv}\ \delta?\ [c'\ s']@\delta'\ \xi \parallel \eta \,!\, \mathsf{conv}^*\ s\ s'\ \xi \parallel P\ \left[W[\eta \mapsto \{\alpha\}, \zeta \mapsto \{\alpha\}, \beta \mapsto \{\zeta\}, \gamma \mapsto \{\zeta\}], Q \cdot \eta \cdot \beta \cdot \gamma\right] \quad \text{if } |s| = |s'|$$

$$\alpha \,!\, \mathsf{conv}\ [c\ s]@\delta\ v'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ \delta?\ v'\ \xi \parallel P\ \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathsf{conv}\ v\ [c'\ s']@\delta'\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{conv}\ v\ \delta'?\ \xi \parallel P\ \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathsf{conv}\ v_1\ v_2\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{F} \parallel P\ \left[\mathbf{finish}(\alpha, W, Q)\right] \quad \text{in all other cases}$$

Fig. 2. The explicitly-scheduled abstract machine, part 1.

Comparing two argument stacks:

$$\alpha \,!\, \mathsf{conv}^* \;\epsilon\; \epsilon\; \xi \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{T} \parallel P \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\begin{aligned}\alpha \,!\, \mathsf{conv}^* \;(\beta \cdot s)\; (\beta' \cdot s')\; \xi \parallel P \left[w, \alpha \cdot Q\right] \to \nu\gamma\eta.\;\; & \alpha \,!\, \gamma? \wedge \eta? \\ & \parallel \gamma \,!\, \mathsf{conv}^?\; \beta\; \beta'\; \xi \\ & \parallel \eta \,!\, \mathsf{conv}^*\; s\; s'\; \xi \parallel P \\ & \left[W[\gamma \mapsto \{\alpha\}, \eta \mapsto \{\alpha\}], Q \cdot \gamma \cdot \eta\right]\end{aligned}$$

Transitions for evaluation processes:

$$\alpha \,!\, \mathtt{eval}\; t\; e \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{reduce}\; t\; e\; \epsilon \parallel P \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathtt{reduce}\; (t\; u)\; e\; s \parallel P \left[W, \alpha \cdot Q\right] \to \nu\beta.\; \alpha \,!\, \mathtt{reduce}\; t\; e\; (\beta \cdot s) \parallel \beta \,!\, \mathtt{eval}\; u\; e \parallel P \left[W, Q \cdot \alpha\right]$$

$$\begin{aligned}\alpha \,!\, \mathtt{reduce}\; (\lambda x.t)\; e\; s \parallel P \left[W, \alpha \cdot Q\right] \to \nu\gamma\delta.\;\; & \alpha \,!\, \mathtt{apply}\; \langle x, t, e, y, \delta\rangle\; s \\ & \parallel \delta \,!\, \mathtt{eval}\; t\; (e + x \mapsto \gamma) \\ & \parallel \gamma \,!\, [y] \parallel P \\ & \left[W, Q \cdot \alpha\right] \\ & \text{where } y \text{ is a fresh variable}\end{aligned}$$

$$\alpha \,!\, \mathtt{reduce}\; x\; e\; s \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{apply}\; e(x)?\; s \parallel P \left[W, Q \cdot \alpha\right] \quad \text{if } x \in \mathrm{Dom}(e)$$

$$\alpha \,!\, \mathtt{reduce}\; x\; e\; s \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{apply}\; [x]\; s \parallel P \left[W, Q \cdot \alpha\right] \quad \text{if } x \notin \mathrm{Dom}(e)$$

$$\alpha \,!\, \mathtt{reduce}\; c\; e\; s \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{apply}\; ([c]@K(c))\; s \parallel P \left[W, Q \cdot \alpha\right]$$

Obtaining the value to be applied:

$$\alpha \,!\, \mathtt{apply}\; \beta?\; s \parallel \beta \,!\, v \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{apply}\; v\; s \parallel \beta \,!\, v \parallel P \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathtt{apply}\; \beta?\; s \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{apply}\; \beta?\; s \parallel P \left[\mathbf{need}(\alpha, \beta, W, Q)\right]$$

Applying a value to a stack:

$$\alpha \,!\, \mathtt{apply}\; v\; \epsilon \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, v \parallel P \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\alpha \,!\, \mathtt{apply}\; \langle x, t, e, y, \delta\rangle\; (\beta \cdot s) \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathtt{reduce}\; t\; (e + x \mapsto \beta)\; s \parallel P \left[W, Q \cdot \alpha\right]$$

$$\alpha \,!\, \mathtt{apply}\; [x\; s'] \; s \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, [x\; (s' \cdot s)] \parallel P \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\alpha \,!\, \mathtt{apply}\; ([c\; s']@\delta)\; s \parallel P \left[W, \alpha \cdot Q\right] \to \nu\gamma.\; \alpha \,!\, [c\; (s' \cdot s)]@\gamma \parallel \gamma \,!\, \mathtt{apply}\; \delta?\; s \parallel P \left[\mathbf{finish}(\alpha, W, Q)\right]$$

Transitions for Boolean connectors (symmetrical rules for $\wedge$ and $\oplus$ omitted):

$$\alpha \,!\, \beta? \oplus \gamma? \parallel \beta \,!\, v \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, v \parallel \beta \,!\, v \parallel P \left[\mathbf{finish}(\alpha, \mathbf{unneed}(\alpha, \gamma, W, Q))\right]$$

$$\alpha \,!\, \beta? \parallel \beta \,!\, v \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, v \parallel \beta \,!\, v \parallel P \left[\mathbf{finish}(\alpha, W, Q)\right]$$

$$\alpha \,!\, \beta? \wedge \gamma? \parallel \beta \,!\, \mathsf{F} \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{F} \parallel \beta \,!\, \mathsf{F} \parallel P \left[\mathbf{finish}(\alpha, \mathbf{unneed}(\alpha, \gamma, W, Q))\right]$$

$$\alpha \,!\, \beta? \wedge \gamma? \parallel \beta \,!\, \mathsf{T} \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \gamma? \parallel \beta \,!\, \mathsf{T} \parallel P \left[\mathbf{need}(\alpha, \gamma, W, Q)\right]$$

$$\alpha \,!\, \beta? \overrightarrow{\oplus} \gamma? \parallel \beta \,!\, \mathsf{T} \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \mathsf{T} \parallel \beta \,!\, \mathsf{T} \parallel P \left[\mathbf{finish}(\alpha, \mathbf{unneed}(\alpha, \gamma, W, Q))\right]$$

$$\alpha \,!\, \beta? \overrightarrow{\oplus} \gamma? \parallel \beta \,!\, \mathsf{F} \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, \gamma? \parallel \beta \,!\, \mathsf{F} \parallel P \left[\mathbf{need}(\alpha, \gamma, W, Q)\right]$$

$$\alpha \,!\, \beta? \overrightarrow{\oplus} \gamma? \parallel \gamma \,!\, v \parallel P \left[W, \alpha \cdot Q\right] \to \alpha \,!\, v \parallel \gamma \,!\, v \parallel P \left[\mathbf{finish}(\alpha, \mathbf{unneed}(\alpha, \beta, W, Q))\right]$$

Fig. 3. The explicitly-scheduled abstract machine, part 2.

where $\alpha \,!\, E \longrightarrow P'$ is one of the reduction steps for the evaluation or convertibility functions of §4 and §5. The reduction step is performed because the process α is active and in head position in the queue Q. The process is then moved to the end of Q. The other processes P and the wait map W are unchanged. The net effect of these transitions is to step through the executions of the active processes in round-robin manner.

When a new process is created, it can be either in the inactive state or in the active state. For example, in the rule that reduces a function application

$$\alpha \,!\, \texttt{reduce}\ (t\ u)\ e\ s \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \nu\beta.\, \alpha \,!\, \texttt{reduce}\ t\ e\ (\beta \cdot s) \parallel \beta \,!\, \texttt{eval}\ u\ e \parallel P\ \left[W, Q \cdot \alpha\right]$$

the process $\beta \,!\, \texttt{eval}\ u\ e$ that computes the value of the function argument u is initially inactive, since we do not need its value right away. It will become active when the value $\beta?$ is needed. In contrast, the rule that tests the convertibility of two nonempty stacks,

$$\begin{aligned}\alpha \,!\, \texttt{conv}^*\ (\beta \cdot s)\ (\beta' \cdot s')\ \xi \parallel P\ \left[Q, \alpha \cdot W\right] \longrightarrow \nu\gamma\eta.\ & \alpha \,!\, \gamma? \wedge \eta? \\ & \parallel \gamma \,!\, \texttt{conv}^?\ \beta\ \beta'\ \xi \\ & \parallel \eta \,!\, \texttt{conv}^*\ s\ s'\ \xi \parallel P \\ & \left[W[\gamma \mapsto \{\alpha\}, \eta \mapsto \{\alpha\}], Q \cdot \gamma \cdot \eta\right]\end{aligned}$$

creates two new convertibility processes, γ and η, which need to start executing right away, while α need to wait for them to produce Boolean values. Therefore, γ and η are added to Q, and α is recorded (in W) as waiting both on γ and on η.

When an active process reduces to a value, all the processes waiting for this value must be restarted. This is performed by the **finish** operation:

$$\textbf{finish}(\alpha, W, Q) = (W[\alpha \mapsto \emptyset], Q \cdot (W(\alpha) \setminus Q))$$

A typical use is the application of a value to an empty stack:

$$\alpha \,!\, \texttt{apply}\ v\ \epsilon \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \alpha \,!\, v \parallel P\ \left[\textbf{finish}(\alpha, W, Q)\right]$$

The effect of **finish** is to restart all the processes that were blocked while reading from α, adding them to the queue of active processes. Then, $W(\alpha)$ is set to $\emptyset$ since no process remains waiting on α. The process α is not added back to Q, since it has terminated and does not need to be scheduled ever again.

Symmetrically, the $\textbf{need}(\alpha, \beta, W, Q)$ operation suspends the process α until the process β has produced a value.

$$\begin{aligned}\textbf{need}(\alpha, \beta, W, Q) &= (W[\beta \mapsto W(\beta) \cup \{\alpha\}], Q) && \text{if } W(\beta) \neq \emptyset \\ \textbf{need}(\alpha, \beta, W, Q) &= (W[\beta \mapsto \{\alpha\}], Q \cdot \beta) && \text{if } W(\beta) = \emptyset\end{aligned}$$

A typical use of **need** is the rule for applying a value read from a channel:

$$\alpha \,!\, \texttt{apply}\ \beta?\ s \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \alpha \,!\, \texttt{apply}\ \beta?\ s \parallel P\ \left[\textbf{need}(\alpha, \beta, W, Q)\right] \tag{i}$$

The process $\alpha \,!\, \texttt{apply}\ \beta?\ s$ needs to receive a value from channel β before it can proceed. Therefore, α becomes inactive, β becomes active if it was not already, and a dependency of α on β is added to the map W.

Note that the **need** operation must not be performed if the desired value is already available. To ensure this, for each rule like (i) above, we have a companion rule:

$$\alpha \,!\, \texttt{apply}\ \beta?\ s \parallel \beta \,!\, v \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \alpha \,!\, \texttt{apply}\ v\ s \parallel \beta \,!\, v \parallel P\ \left[W, Q \cdot \alpha\right]) \tag{ii}$$

where the value v produced by process β is directly transferred to process α. (The convention we follow for the abstract machine is that transitions are determined by the first rule that matches. Since rule (ii) appears before rule (i) in figure 3, (ii) takes precedence over (i).)

A process may need to wait for several processes to produce their values. This is the case in the rule that obtains two values to be compared:

$$\alpha \,!\, \mathsf{conv}\ \beta?\ \beta'?\ \xi \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \alpha \,!\, \mathsf{conv}\ \beta?\ \beta'?\ \xi \parallel P\ \left[\mathbf{need}(\alpha, \beta, \mathbf{need}(\alpha, \beta', W, Q))\right]$$

Here, both β and β' need to be restarted, and α must be marked as waiting on both β and β'.

Finally, when evaluating Boolean connectors, the value of an active process may become unneeded. For example, if we are evaluating $\alpha \,!\, \beta? \wedge \gamma?$ and the process β produces F, we no longer need the value of γ. Therefore, we update the scheduling state to reflect this fact before producing F on α:

$$\alpha \,!\, \beta? \wedge \gamma? \parallel \beta \,!\, \mathsf{F} \parallel P\ \left[W, \alpha \cdot Q\right] \longrightarrow \alpha \,!\, \mathsf{F} \parallel \beta \,!\, \mathsf{F} \parallel P\ \left[\mathbf{finish}(\alpha, \mathbf{unneed}(\alpha, \gamma, W, Q))\right]$$

The **unneed** operation is defined recursively as

$$\begin{aligned}
\mathbf{unneed}(\alpha, \beta, W, Q) &= (W[\beta \mapsto W(\beta) \setminus \{\alpha\}], Q) \qquad \text{if } W(\beta) \neq \{\alpha\} \\
\mathbf{unneed}(\alpha, \beta, W, Q) &= \mathbf{unneed}(\beta, \gamma_1, \ldots, \mathbf{unneed}(\beta, \gamma_n, W[\beta \mapsto \emptyset], Q \setminus \{\beta\})) \\
&\qquad \text{if } W(\beta) = \{\alpha\} \text{ and } \{\gamma \mid \beta \in W(\gamma)\} = \{\gamma_1, \ldots, \gamma_n\}
\end{aligned}$$

The dependency of α on β is removed from W. Moreover, if α was the only process waiting on β to produce a value, the execution of β is stopped by removing β from the queue Q of active processes, and we recursively call $\mathbf{unneed}(\beta, \gamma)$ on all processes γ that β was waiting for. This is similar to removing a reference in a reference counting system. The **unneed** operation can be implemented efficiently by using a doubly linked list for Q and storing W as a pair of maps in both directions.

8 Rocq Proof

We formally verified the partial correctness of our concurrent convertibility test, using the Rocq interactive theorem prover. The formalization includes the main algorithm presented in §5, without the extensions shown in §6. It also incorporates extensions to the core λ-calculus to handle data constructors and pattern-matching. The proof only proves partial correctness; that is, if the algorithm produces a result, then the result is correct. It does not prove termination. The proof does not formalize scheduling either, because it is unnecessary for proving correctness.

The reduction part of the algorithm is formulated in the style of §4, as transitions over sets of concurrent threads, leaving scheduling unspecified. Since the sharing of convertibility processes shown in §6.3 is not included, all convertibility processes of §5 are replaced by a tree of Boolean operations. The leaves of this tree are convertibility tests between two channels.

The development is included as an artifact for this paper. It sums up to about 10000 lines of Rocq in total, which makes it a moderately-sized proof. Here is the final theorem:

```
Lemma all_correct :
  forall defs t1 t2 st b,
    defs_wf defs ->
    closed_at t1 0 -> closed_at t2 0 ->
    dvar_below (length defs) t1 -> dvar_below (length defs) t2 ->
    star step (init_conv defs t1 t2) (cthread_done b, st) ->
    reflect (convertible (betaiota defs) t1 t2) b.
```

It expresses that, given well-formed constant definitions `defs` and two closed terms `t1` and `t2` that reference only constants defined in `defs`, if we start the convertibility abstract machine in the initial state corresponding to `defs`, `t1` and `t2`, and if it stops after a finite number of transitions on a final `cthread_done` state carrying the Boolean result `b`, then `b` is `true` if and only if `t1` and `t2` are convertible. The `betaiota` relation, which should actually be called `betadelta`, is the union of beta

$$\frac{t \to_{\beta\delta} t' \qquad t' \approx u : B}{t \approx u : B}\ \textsc{red-l} \qquad \frac{u \to_{\beta\delta} u' \qquad t \approx u' : B}{t \approx u : B}\ \textsc{red-r}$$

$$\frac{t \approx u[y := x] : B}{\lambda x.\, t \approx \lambda y.\, u : B}\ \textsc{lam} \qquad \frac{}{\lambda x.\, t \approx y\ u_1\ \cdots\ u_n : \mathsf{F}}\ \textsc{lam-var} \qquad \frac{}{x\ t_1\ \cdots\ t_n \approx \lambda y.\, u : \mathsf{F}}\ \textsc{var-lam}$$

$$\frac{t_1 \approx u_1 : \mathsf{T} \quad \ldots \quad t_n \approx u_n : \mathsf{T}}{x\ t_1\ \cdots\ t_n \approx x\ u_1\ \cdots\ u_n : \mathsf{T}}\ \textsc{var-1} \qquad \frac{x \neq y}{x\ t_1\ \cdots\ t_n \approx y\ u_1\ \cdots\ u_m : \mathsf{F}}\ \textsc{var-2}$$

$$\frac{t_i \approx u_i : \mathsf{F}}{x\ t_1\ \cdots\ t_n \approx x\ u_1\ \cdots\ u_n : \mathsf{F}}\ \textsc{var-3} \qquad \frac{t_1 \approx u_1 : \mathsf{T} \quad \ldots \quad t_n \approx u_n : \mathsf{T}}{c\ t_1\ \cdots\ t_n \approx c\ u_1\ \cdots\ u_n : \mathsf{T}}\ \textsc{const}$$

Fig. 4. The inference rules for the convertibility judgment $t \approx u : B$

reduction and unrolling of defined constants. Note that this does not guarantee termination nor the absence of errors or deadlocks: we have not proved that a `cthread_done` state will be reached. However, this guarantees that once this state is reached, then the result thus obtained is correct.

Unsurprisingly, one of the more delicate aspects of the Rocq development is the representation of variables in terms. We use de Bruijn indices for the inputs of the convertibility test (such as the terms `t1` and `t2` above) and named variables for evaluated terms. De Bruijn indices are easier to work with, but named variables allow for sharing without the need for explicit weakenings. This approach necessitates generating fresh variable names in the state of the reduction, and proving numerous invariants justifying that the generated variables are indeed fresh.

One limitation of the Rocq proof is that, for a reduction step from a configuration representing a term t to a configuration representing a term t', it only shows that t and t' are convertible ($t \approx t'$) but not that t reduces to t' ($t \to^+ t'$). Knowing that $t \approx t'$ is enough to prove the partial correctness of the convertibility checking algorithm. However, we would need to know that $t \to^+ t'$ in order to prove that convertibility always terminates when given two strongly normalizing terms. We previously had a proof of $t \to^+ t'$ for an earlier, simpler version of our call-by-need evaluator, but the proof was so complex and difficult to extend that we switched to the simpler proof of $t \approx t'$.

9 Performance Analysis

It is not obvious how to characterize the performance of an algorithm for convertibility checking. There is no useful upper bound as a function of the size n of the input terms: the convertibility problem is TOWER-complete even when restricted to simply-typed terms [Statman 1979] [Nguyên 2024]. Condoluci [2020] gives an $O(mn)$ complexity bound, where m is the number of reductions needed to fully normalize the input terms and n is their size. Since our algorithm avoids computing normal forms as much as it can, we would prefer a bound that does not involve m.

To this end, we take a step back from the details of the algorithm and view convertibility checking as a *proof search* problem. Given two terms t and u, we aim to derive the judgment $t \approx u : B$ where the Boolean B is T if t and u are convertible and F otherwise. The inference rules for this judgment are given in figure 4. Rules RED-L and RED-R correspond to performing one reduction step $\to_{\beta\delta}$ in t or in u, either beta-reduction (β) or unrolling of a defined constant (δ). The other rules follow the structure of the two terms t and u. Rule CONST is the "shortcut" for proving that two applications of a defined constant c are convertible without unrolling c.

A goal $t \approx u : B$ generally admits multiple proofs, but some proofs are smaller than others. For example, $c\ x \approx c\ x : \mathsf{T}$ has a proof of size 2 (rules CONST and VAR-1) and other, longer proofs obtained by first unrolling c on both sides (rules RED-L and RED-R).

The algorithm of §7 can be viewed as a *breadth-first search* of the tree of possible proofs. For example, given the problem $c\ t_1\ \cdots\ t_n \approx c\ u_1\ \cdots\ u_n : B$, the algorithm searches in parallel for three kinds of possible proofs, those ending with rule CONST, those ending with rule RED-L, and those ending with rule RED-R. In contrast, other convertibility checkers, such as Rocq's, perform a *depth-first search* for a proof of convertibility, guided by heuristics.

The first author proved that if there exists a proof of $t \approx u : B$ of size s, the algorithm of §7 terminates in time $O((k+1)^{2s})$, where $k \geq 1$ is the maximal arity of variable and constant applications in the original terms t, u [Courant 2024, chapter 12]. In other words, our algorithm is exponential in the size s of the smallest proof of (non-)convertibility. In contrast, convertibility checkers based on depth-first search can take time unbounded by any function of s, since they can perform an arbitrarily large amount of computation before finding a proof.

The complexity argument above needs to be made more precise: the size s of the smallest convertibility proof depends crucially on the inference rules and the reduction strategy used. While the inference rules in figure 4 are somewhat canonical, the strategy used by $\rightarrow_{\beta\delta}$ reductions in rules RED-L and RED-R has a huge impact on the size s of convertibility proofs. For instance, using weak call-by-name or weak call-by-value can result in convertibility proofs that are exponentially bigger (or worse) than those obtained using weak call-by-need; and using optimal reduction [Lamping 1990] could lead to even smaller proofs.

To clarify this dependency on the reduction strategy used, the complexity argument of Courant [2024, chapter 12] is formulated in terms of an *effective reduction structure*, which is an abstract presentation of graph reduction. Therefore, the complexity argument is independent of the details of our call-by-need evaluator, and only relies on the sharing properties of graph reduction.

The size s of a convertibility proof can be decomposed as $s = r + f + b$, where r is the number of reduction steps, f the number of "forced" convertibility steps (those where only one rule applies to the current goal), and b the number of "branching" convertibility steps (those where several rules apply to the current goal). Obviously, the branching convertibility steps are those responsible for the exponential overhead of our algorithm. We conjecture that there exist scheduling strategies for which our algorithm has a complexity bound of $O((r+f)K^b)$ for some constant $K \geq 2$, instead of $O(K^{r+f+b})$ as in the analysis above. The idea is to use non-uniform scheduling of processes, where each process has a a share of CPU time, with all shares summing to 1. Each process is scheduled with a frequency proportional to its share. A convertibility process that performs a branching step would divide its share among the processes that it creates. The effect of this scheduling policy would be to slow down the exponential explosion in the number of processes caused by branching steps, giving more time to reduction and convertibility processes created earlier.

10 Experimental Evaluation

For the experimental evaluation, we used two OCaml implementations of our convertibility checker, which corresponds to the version verified in Rocq extended with inductive types and fixed points. The implementation named "Full" in the following follows the algorithm of §7 and implements the sharing of convertibility processes described in §6.3. The implementation named "Simple" in the following uses an earlier version of our algorithm that does not share convertibility processes and uses a heuristic to determine which side to unfold first when encountering different head constants, instead of trying both unfoldings in parallel. Variables in the input terms are represented by the type `string`, which comes with additional costs compared to Rocq's internal de Bruijn

Table 1. Timings, in seconds, for the examples of convertibility problems given in the text for Rocq and our two OCaml implementations, Simple and Full. Speedups are relative to Rocq and are expressed as base-10 logarithms, *i.e.* decimal orders of magnitude. Higher is faster. See the main text for the description of the test cases and the explanation of the two results given for the last test.

Test case	Rocq	Simple		Full	
	time	*time*	*speedup*	*time*	*speedup*
`exp2 15` $\approx$ `exp2 (14 + 1)`	3×10^{-5}	5×10^{-5}	-0.22	6×10^{-3}	-2.3
`zero (exp2 15)` $\approx$ `zero (exp2 16)`	0.14	5×10^{-6}	+4.4	2×10^{-5}	+3.8
`ldepth (perfect 15 L)` $\approx$ `ldepth2 (perfect 15 L)`	9×10^{-5}	2×10^{-4}	-0.35	5×10^{-5}	0.26
`perfect 15 L` $\approx$ `perfect 14 (N L L)`	0.018	0.013	+0.14	9×10^{-5}	+2.3
`(exp2 15, false)` $\not\approx$ `(exp2 16, true)`	4×10^{-6}	6×10^{-6}	-0.18	8×10^{-6}	-0.30
`(false, exp2 15)` $\not\approx$ `(true, exp2 16)`	0.61	1×10^{-6}	+5.8	8×10^{-6}	+4.9
`pair1 (exp2 15)` $\approx$ `(false, exp2 15)`	3×10^{-5}	7×10^{-5}	-0.37	2×10^{-4}	-0.82
`pair2 (exp2 15)` $\approx$ `(exp2 15, false)`	0.078	5×10^{-5}	+3.2	2×10^{-4}	+2.6
$\underbrace{\texttt{f4}(\cdots(\texttt{f4}\ (\texttt{f4}\ 0))\cdots)}_{\text{30 applications of f4}} \approx \underbrace{\texttt{f4}(\cdots(\texttt{f4}\ (\texttt{f4}\ 0))\cdots)}_{\text{30 applications of f4}}$	2×10^{-5}	$\begin{cases} 6 \times 10^{-5} \\ 0.18 \end{cases}$	$\begin{cases} -0.5 \\ -4.0 \end{cases}$	0.15	-3.9

indices. Neither implementation performs the pre-sharing of subterms described in §6.4. On the Rocq side, we instrumented Rocq's convertibility checker so that it prints the time taken. (This is much more precise than just relying on Rocq's `Time` command, which also accounts for other aspects of type checking.) We used Rocq 8.15.2, extended with these changes to the convertibility checker. Moreover, both Rocq and our implementation were compiled by OCaml 4.12.1, and the measurements were performed on a Intel Core i7-1165G7 2.80GHz CPU and 2x 16GiB SODIMM DDR4 Synchronous 3200 MHz (0.3 ns) RAM, running Linux 5.15.74 with NixOS 22.05.

The test cases we used are described and commented below, while the time measurements are shown in table 1. On all the test cases, the timings are quite small, because larger inputs would cause stack overflows, and only one digit is significant. However, these rough measurements are already sufficient to spot nonlinear behaviors.

The test cases use the following defined constants:

```
Fixpoint exp2 n := match n with O => 1 | S n => 2 * exp2 n end.
Definition zero (n : nat) := 0.

Inductive tree := L : tree | N : tree -> tree -> tree.
Fixpoint perfect n t := match n with O => t | S n => perfect n (N t t) end.
Fixpoint ldepth t := match t with L => 0 | N t1 t2 => S (ldepth t1) end.
Fixpoint ldepth2 t := match t with L => 0 | N t1 t2 => ldepth2 t + 1 end.

Definition pair1 n := (is_zero n, n).   Definition pair2 n := (n, is_zero n).

Definition f0 (n : nat) := n.           Definition f1 n := f0 (f0 n).
Definition f2 n := f1 (f1 n).           Definition f3 n := f2 (f2 n).
Definition f4 n := f3 (f3 n).
```

The first two tests, `exp2 15` $\approx$ `exp2 (14 + 1)` and `zero (exp2 15)` $\approx$ `zero (exp2 16)`, focus on the heuristic used when the two head constants are the same. With Rocq, the first test is

fast but the second one is slow. In both cases, Rocq attempts to prove the convertibility of the arguments before unfolding the definition of the constant. This is a good strategy for `exp2 15` ≈ `exp2 (14+1)`, as it allows Rocq to prove convertibility without expanding `exp2`. However, in the case of `zero (exp2 15)` ≈ `zero (exp2 16)`, Rocq tries and fails to prove the convertibility of `exp2 15` and `exp2 16`, which costs a lot of work. Expanding `zero` would have immediately proved convertibility.

Here, our two convertibility checkers get the best of both worlds by doing the work in parallel, and both checks are fast. Our Full checker performs worse on the first test: replacing 15 by n and varying n, we experimentally measured $O(n^{2.8})$ complexity instead of the expected $O(n)$. This is caused by the large amount of unfolding opportunities in the branch where we unfold `exp2`, causing an explosion in the number of processes. This issue could be alleviated by the non-uniform scheduling policy outlined at the end of §9.

Next, we will consider terms whose size is exponential in the size of their memory representation, because there is a lot of sharing within the term itself. The function `perfect` takes an argument `n` and a tree `t` and generates a tree with 2^n copies of `t`. However, the evaluation only takes time linear in `n` to evaluate, as the subtrees are shared. The definitions `ldepth` and `ldepth2` compute the length of the leftmost branch of their argument, in linear time for `ldepth`, and quadratic time for `ldepth2`.

Testing the convertibility of `ldepth (perfect 15 L)` and `ldepth2 (perfect 15 L)` is fast both in Rocq and with our convertibility checkers because terms are shared; therefore, the computation of both sides takes only quadratic time. However, Rocq is slow to check the convertibility of `perfect 15 L` and `perfect 14 (N L L)`, because after expanding `perfect`, it has to prove the convertibility of the exact same terms multiple times. For the same reason, the Simple version of our convertibility check is slow, but the Full version, which performs more sharing, is fast.

Another interesting test concerns the order in which the arguments of constructors (or identical defined constants) are compared. We consider two very similar tests of non-convertibility, `(exp2 15, false)` ≠ `(exp2 16, true)` and `(false, exp2 15)` ≠ `(true, exp2 16)`. With Rocq, the first test is almost instantaneous: Rocq starts by comparing `true` and `false`, since Rocq evaluates arguments from right to left. They are different, so the test stops immediately. However, the second test is much slower: Rocq starts by comparing `exp2 15` and `exp2 16`, which fails after a long time. With our convertibility checkers, both tests are equally fast: we test the convertibility of the arguments in parallel, so we immediately detect that `false` and `true` are not convertible, and return this result.

Another peculiarity of Rocq is that once a constant is unfolded, it remains unfolded for future tests, preventing us from benefiting from the optimization with folded constants. The next two tests, `pair1 (exp2 15)` ≈ `(false, exp2 15)` and `pair2 (exp2 15)` ≈ `(exp2 15, false)` demonstrate the problems this can cause. Again, the two tests are identical except for the order of arguments. In the first test, Rocq first compares `exp2 15` and `exp2 15`, which is almost instantaneous thanks to the folded constant optimization. Then, it compares `is_zero (exp2 15)` with `false`, which takes only linear time, thanks to Rocq's laziness. However, in the second test, Rocq first compares `is_zero (exp2 15)` with `false`, forcing it to unfold `exp2` to prove convertibility. Once this is done, it compares `exp2 15` with a version of `exp2 15` that has already been partially computed and where `exp2` has been unfolded. At this point, it has no way but to expand `exp2` on the other side, and the time taken is exponential. With our convertibility checker, both tests are fast. Indeed, when we unfold a constant, we also keep the original folded value, allowing us to still benefit from the folded constant optimization if we encounter it again.

Of course, this comparison wouldn't be honest if we didn't also show a shortcoming of our own convertibility checker. In the final example, we have an identical term on both sides, but it is deeply nested. Rocq is almost instantaneous there by repeatedly applying the folded constant optimization, but since our checker explores what happens both when unfolding and when not unfolding, it is

much slower. The Full version is always slow. The Simple version uses heuristics to choose which side to unfold when encountering two different head constants, and the speed depends heavily on the unfolding order. If we choose to always unfold the older constant first, we obtain the result quickly: when we unfold `f4` on one side, then we will repeatedly unfold `f3`, `f2`, `f1` and then `f0` on that side until that side has only `f4`, preventing the folded constant optimisation from applying and spawning new processes until that point. This severely limit the number of total convertibility processes that are created, thus allowing the code to run quite fast, albeit slower than Rocq. On the other hand, if we always unfold the newer constant first (which is often the best choice in Rocq), when we unfold `f4` on one side, we will match this by unfolding `f4` on the other side next, making `f3` appear as the head constant on both sides, making the folded constant optimisation apply again, and so on with `f2`, `f1` and `f0`, creating in total a very large number of convertibility processes, and thus making the code run very slowly.

However, such examples seem to be quite pathological, and we think they should not happen in practice. Besides, we have a guaranteed complexity of our convertibility test in terms of the shortest existing convertibility proof, which looks like a desirable property that Rocq does not have.

11 Related Work

11.1 Convertibility Checking

The most advanced algorithms for convertibility testing can be found in the implementations of Agda, Lean, Rocq and other dependently-typed frameworks. However, these algorithms are undocumented and difficult to reconstruct from source code. The `smalltt` project by András Kovács is a small, readable implementation of elaboration for dependent types that includes a convertibility checker based on normalization by evaluation.

Among the published work on this topic, the one closest to ours is the MetaRocq (formerly MetaCoq) project, which contains a verified convertibility checker as part of its verified type-checker for the core Rocq language [Sozeau et al. 2020, 2025]. Unlike ours, their checker is proved to terminate when given two well-typed terms as inputs, under the assumption that all well-typed terms are strongly normalizing. The MetaRocq checker handles defined constants with a fixed strategy (e.g. for $c\ t \approx c\ t'$, it always tries $t \approx t'$ before unfolding c) and performs reductions using a Krivine-style machine and a call-by-name strategy, without any support for sharing.

Abel et al. [2018] describe another impressive verification of the metatheory of a dependently-typed language, including a constructive proof that convertibility is decidable. It uses typed reduction and convertibility relations, which facilitate the proof of termination. Adjedj et al. [2024] extend this approach to a verified algorithm for convertibility checking. The reduction strategy is not specified, and no provision is made for sharing reductions. Earlier work [Abel et al. 2008] used normalization by evaluation instead of typed reductions, and is therefore restricted to determining $\beta\eta$-convertibility, while [Abel et al. 2018] handles both β-convertibility and $\beta\eta$-convertibility. Lennon-Bertrand [2025] compares and relates the typed approach to convertibility checking used in the aforementioned work with the untyped approach that we use in this paper.

The idea that convertibility testing can be performed incrementally by alternating between evaluation to WHNF and comparison of the resulting values goes back at least to Coquand [1996]. An early implementation of this approach is described by Grégoire and Leroy [2002]. However, their compiled implementation of WHNF evaluation uses call by value and unrolls constants eagerly, resulting in unnecessary computation.

All the earlier work described above, like our own work, relies on interleaved evaluations and comparisons of values. Condoluci [2020] goes back to the more traditional approach based on

normalization of the two terms followed by equality testing of their normal forms, but uses a clever normalization algorithm that exploits sharing in a call-by-value strategy [Accattoli et al. 2021] and a clever equality test that takes sharing into account and runs in time linear in the size of the shared representation of the two normal forms [Condoluci et al. 2019].

11.2 Strong Call-by-Need Reduction

Call-by-need strategies for strong reduction (evaluation under lambda-abstractions) are difficult to define formally and to implement correctly. Balabonski et al. [2017] develop λ_c, a strong call-by-need calculus that uses explicit substitutions to represent sharing. To enforce laziness, they need to delay reducing under a lambda-abstraction until all applications of that abstraction have been reduced. Our enriched function closures $\langle x, t, e, y, \delta \rangle$ support applying a lambda-abstraction and reducing within its body in any order, which simplifies the presentation.

Balabonski et al. [2021] extend λ_c with the ability to reduce under lambda-abstractions before applying them if it can be determined that the normal form of the lambda-abstraction will be needed. They also present an abstract machine that implements these reductions efficiently. Their approach can perform fewer β-reductions than ours in some cases. However, their abstract machine lacks the subterm property and therefore cannot be statically compiled to virtual machine code or native code.

Biernacka et al. [2022] develop a call-by-need normalizer by applying memoization techniques to a call-by-name normalization-by-evaluation function derived from the KN machine of Crégut [2007]. Applying a mechanized CPS transformation to this normalizer, they obtain the RKNL machine, a simple and efficient abstract machine for strong call-by-need evaluation. While developed independently, our approach to call-by-need normalization described in §4 is essentially isomorphic to the RKNL machine.

11.3 Semantics of Laziness

The first formal presentations of call by need and more generally of lazy evaluation used graph reduction; see Peyton Jones [1987, part II] for a survey. Launchbury [1993] gave a big-step semantics for lazy evaluation using terms and an explicit store for memoization. Ariola et al. [1995] and Ariola and Felleisen [1997] give small-step semantics using `let` bindings or distinguished β-redexes to express sharing and laziness of evaluations. Our process-based notation for lazy / non-strict computations is essentially isomorphic to their `let`-based notation, with parallel processes $\alpha\,!\,t \parallel C[\alpha?]$ playing the role of $\texttt{let}\ x = t\ \texttt{in}\ C[x]$ bindings in Ariola et al. [1995]. We were also inspired by the encoding of the call-by-need weak lambda-calculus in the asynchronous pi-calculus of Sangiorgi [2019], with the difference that Sangiorgi relies on an explicit handshake to delay evaluations until needed, while we rely on an external scheduler.

12 Conclusions and Further Work

We hope this work sparks renewed interest in convertibility checking, which is a difficult problem that is central to the implementation of type- and proof-checkers. The lazy, concurrent convertibility checking algorithm described in this paper is novel in several ways: it does not rely on heuristics, it always finds the simplest proof of (non-)convertibility, and its complexity is bounded as a function of the size of the simplest proof. Admittedly, the bound is exponential, but this is an improvement over heuristics-based sequential algorithms, whose complexity is unbounded in the size of the simplest proof.

This paper focuses on the lambda-calculus with constants. However, the ideas presented here have been extended to a richer language that includes inductive types with pattern-matching and structural recursion [Courant 2024].

As mentioned at the end of §9, the worst-case bound of our algorithm, as well as its actual performance on some examples shown in §10, could probably be improved by a more sophisticated scheduling of processes that allocates unequal amounts of time to different processes. Additionally, constant factors could also be reduced by using more clever imperative data structures for scheduling and by compiling evaluation processes to virtual machine code or even to native code. However, we are skeptical that hardware parallelism can be used to significantly speed up our algorithm, given the slow progress in the area of parallel graph reduction since the 1980s.

The formal proof of §8 needs more work: to prove that the convertibility checker cannot go wrong or deadlock when given two type-safe terms as input, and that it terminates when given two strongly normalizing terms as input. The termination proof sounds challenging, especially if we stick to untyped reductions. Typed reductions in the context of a normalizing type system might provide a simpler proof. However, this would make the convertibility checker specific to a given type system.

Our convertibility checker can easily be instrumented to generate a trace of the nonobvious unrolling decisions it made. Using this trace, (non-)convertibility can then be rechecked by a simpler, purely sequential algorithm. This approach could facilitate the integration of our convertibility checker into an existing proof checker. Additionally, convertibility traces can be cached to improve proof checking times when some proof terms are checked multiple times.

Throughout this work, we have emphasized the importance of sharing in order to avoid repeated evaluations. However, we only considered the sharing of sub-terms via lazy evaluation. Other graph reduction techniques support the sharing of more than just lambda-terms, such as Lamping's optimal reduction algorithm [Lamping 1990] [Gonthier et al. 1992] [Asperti et al. 1996] and the atomic lambda-calculus [Sherratt et al. 2020]. It would be interesting to study the usability of these advanced graph reduction techniques in the context of convertibility checking.

Data-Availability Statement

The Rocq development described in §8 and the benchmarks described in §10 are available at https://doi.org/10.5281/zenodo.17347533 (for reproduction) and at https://github.com/Ekdohibs/efficient-convertibility/ (for reuse).

References

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2008. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5133)*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer, 29–56. https://doi.org/10.1007/978-3-540-70594-9_4

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. https://doi.org/10.1145/3158111

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. https://doi.org/10.1109/LICS52264.2021.9470630

Beniamino Accattoli and Ugo Dal Lago. 2012. On the Invariance of the Unitary Cost Model for Head Reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan (LIPIcs, Vol. 15)*, Ashish Tiwari (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22–37. https://doi.org/10.4230/LIPICS.RTA.2012.22

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. https://doi.org/10.1145/3636501.3636951

Zena M. Ariola and Matthias Felleisen. 1997. The call-by-need lambda calculus. *J. Funct. Program.* 7, 3 (1997), 265–301. https://doi.org/10.1017/S0956796897002724

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The Call-by-Need Lambda Calculus. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 233–246. https://doi.org/10.1145/199448.199507

Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. 1996. The Bologna Optimal Higher-Order Machine. *J. Funct. Program.* 6, 6 (1996), 763–810. https://doi.org/10.1017/S0956796800001994

Thibaut Balabonski. 2012. A unified approach to fully lazy sharing. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 469–480. https://doi.org/10.1145/2103656.2103713

Thibaut Balabonski. 2013. Weak optimality, and the meaning of sharing. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 263–274. https://doi.org/10.1145/2500365.2500606

Thibaut Balabonski, Pablo Barenbaum, Eduardo Bonelli, and Delia Kesner. 2017. Foundations of strong call by need. *Proc. ACM Program. Lang.* 1, ICFP (2017), 20:1–20:29. https://doi.org/10.1145/3110264

Thibaut Balabonski, Antoine Lanco, and Guillaume Melquiond. 2021. A Strong Call-By-Need Calculus. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:22. https://doi.org/10.4230/LIPICS.FSCD.2021.9

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations, ESPRIT Working Group 8533, NADA - New Hardware Design Methods, Survey Chapters (Lecture Notes in Computer Science, Vol. 1546)*, Bernhard Möller and John V. Tucker (Eds.). Springer, 117–137. https://doi.org/10.1007/3-540-49254-2_4

Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022. A simple and efficient implementation of strong call by need by an abstract machine. *Proc. ACM Program. Lang.* 6, ICFP (2022), 109–136. https://doi.org/10.1145/3549822

Samuel Boutin. 1997. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1281)*, Martín Abadi and Takayasu Ito (Eds.). Springer, 515–529. https://doi.org/10.1007/BFB0014565

Andrea Condoluci. 2020. *Beta-Conversion, Efficiently.* Ph. D. Dissertation. Università di Bologna. https://amsdottorato.unibo.it/id/eprint/9444/

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 Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 9:1–9:14. https://doi.org/10.1145/3354166.3354174

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Sci. Comput. Program.* 26, 1-3 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Nathanaëlle Courant. 2024. *Towards an efficient and formally-verified convertibility checker.* Ph. D. Dissertation. Université Paris Cité. https://hal.science/tel-04884688

Nathanaëlle Courant. 2025. *Artifact for "A Lazy, Concurrent Convertibility Checker".* https://doi.org/10.5281/zenodo.17347533

Pierre Crégut. 2007. Strongly reducing variants of the Krivine abstract machine. *High. Order Symb. Comput.* 20, 3 (2007), 209–230. https://doi.org/10.1007/S10990-007-9015-Z

Olivier Danvy. 1996. Type-Directed Partial Evaluation. 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, 242–257. https://doi.org/10.1145/237721.237784

Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. 1992. The Geometry of Optimal Lambda Reduction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 15–26. https://doi.org/10.1145/143165.143172

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. https://doi.org/10.1145/581478.581501

Jason S. Gross. 2021. *Performance Engineering of Proof-Based Software Systems at Scale.* Ph. D. Dissertation. MIT, EECS. https://hdl.handle.net/1721.1/130763

Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda - Programming Proof Search Using Reflection. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9129)*, Ralf Hinze and Janis Voigtländer (Eds.). Springer, 276–301. https://doi.org/10.1007/978-3-319-19797-5_14

Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *High. Order Symb. Comput.* 20, 3 (2007), 199–207. https://doi.org/10.1007/S10990-007-9018-9

John Lamping. 1990. An Algorithm for Optimal Lambda Calculus Reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, Frances E. Allen (Ed.). ACM Press, 16–30. https://doi.org/10.1145/96709.96711

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154. https://doi.org/10.1145/158511.158618

Meven Lennon-Bertrand. 2025. What Does It Take to Certify a Conversion Checker?. In *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, July 14-20, 2025, Birmingham, UK (LIPIcs, Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:23. https://doi.org/10.4230/LIPICS.FSCD.2025.27

Robin Milner. 1999. *Communicating and mobile systems – the π-calculus.* Cambridge University Press.

Lê Thành Dũng Nguyên. 2024. Simply typed convertibility is TOWER-complete even for safe lambda-terms. *Logical Methods in Computer Science* Volume 20, Issue 3, Article 21 (Sep 2024). https://doi.org/10.46298/lmcs-20(3:21)2024

Simon L. Peyton Jones. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall. https://simon.peytonjones.org/slpj-book-1987/

Rocq Development Team. 2025. The Rocq Prover Reference Manual, release 9.0.0. https://rocq-prover.org/doc/V9.0.0/refman/index.html

Davide Sangiorgi. 2019. Asynchronous π-calculus at Work: The Call-by-Need Strategy. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday (Lecture Notes in Computer Science, Vol. 11760)*, Mário S. Alvim, Kostas Chatzikokolakis, Carlos Olarte, and Frank Valencia (Eds.). Springer, 33–49. https://doi.org/10.1007/978-3-030-31175-9_3

David Sherratt, Willem Heijltjes, Tom Gundersen, and Michel Parigot. 2020. Spinal Atomic Lambda-Calculus. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 582–601. https://doi.org/10.1007/978-3-030-45231-5_30

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. https://doi.org/10.1007/S10817-019-09540-0

Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1 (2025), 8:1–8:74. https://doi.org/10.1145/3706056

Richard Statman. 1979. The Typed lambda-Calculus is not Elementary Recursive. *Theor. Comput. Sci.* 9 (1979), 73–81. https://doi.org/10.1016/0304-3975(79)90007-0

Received 2025-07-10; accepted 2025-11-06