Explicit Auditing

Wilmer Ricciotti(✉) and James Cheney

LFCS, School of Informatics, University of Edinburgh,
10 Crichton Street, Edinburgh EH8 9AB, UK
research@wilmer-ricciotti.net, jcheney@inf.ed.ac.uk

Abstract. The Calculus of Audited Units (CAU) is a typed lambda calculus resulting from a computational interpretation of Artemov's Justification Logic under the Curry-Howard isomorphism; it extends the simply typed lambda calculus by providing *audited types*, inhabited by expressions carrying a *trail* of their past computation history. Unlike most other auditing techniques, CAU allows the inspection of trails at runtime as a first-class operation, with applications in security, debugging, and transparency of scientific computation.

An efficient implementation of CAU is challenging: not only do the sizes of trails grow rapidly, but they also need to be normalized after every beta reduction. In this paper, we study how to reduce terms more efficiently in an untyped variant of CAU by means of explicit substitutions and explicit auditing operations, finally deriving a call-by-value abstract machine.

Keywords: Lambda calculus · Justification Logic
Audited computation · Explicit substitutions · Abstract machines

1 Introduction

Transparency is an increasing concern in computer systems: for complex systems, whose desired behavior may be difficult to formally specify, auditing is an important complement to traditional techniques for verification and static analysis for security [2,6,12,16,19,27], program slicing [22,26], and provenance [21,24]. However, formal foundations of auditing as a programming language primitive are not yet well-established: most approaches view auditing as an extra-linguistic operation, rather than a first-class construct. Recently, however, Bavera and Bonelli [14] introduced a calculus in which recording and analyzing audit trails are first-class operations. They proposed a λ-calculus based on a Curry-Howard correspondence with Justification Logic [7–10] called *calculus of audited units*, or **CAU**. In recent work, we developed a simplified form of **CAU** and proved strong normalization [25].

The type system of **CAU** is based on modal logic, following Pfenning and Davies [23]: it provides a type $[\![s]\!]\, A$ of audited units, where s is "evidence", or

An extended version of this paper can be found at https://arxiv.org/abs/1808.00486.

B. Fischer and T. Uustalu (Eds.): ICTAC 2018, LNCS 11187, pp. 376–395, 2018.
https://doi.org/10.1007/978-3-030-02508-3_20

the expression that was evaluated to produce the result of type A. Expressions of this type $!_q M$ contain a value of type A along with a "trail" q explaining how M was obtained by evaluating s. Trails are essentially (skeletons of) proofs of reduction of terms, which can be *inspected* by structural recursion using a special language construct.

To date, most work on foundations of auditing has focused on design, semantics, and correctness properties, and relatively little attention has been paid to efficient execution, while most work on auditing systems has neglected these foundational aspects. Some work on tracing and slicing has investigated the use of "lazy" tracing [22]; however, to the best of our knowledge there is no prior work on how to efficiently evaluate a language such as **CAU** in which auditing is a built-in operation. This is the problem studied in this paper.

A naïve approach to implementing the semantics of **CAU** as given by Bavera and Bonelli runs immediately into the following problem: a **CAU** reduction first performs a *principal contraction* (e.g. beta reduction), which typically introduces a local trail annotation describing the reduction, that can block further beta-reductions. The local trail annotations are then moved up to the nearest enclosing audited unit constructor using one or more *permutation reductions*. For example:

$$\begin{aligned} !_q \mathcal{F}[(\lambda x.M)\ N] &\xrightarrow{\beta} !_q \mathcal{F}[\boldsymbol{\beta} \triangleright M\,\{N/x\}] \\ &\xrightarrow{\tau}\!\!\!\!\rightarrow\ !_{\mathbf{t}(q,\mathcal{Q}[\boldsymbol{\beta}])} \mathcal{F}[M\,\{N/x\}] \end{aligned}$$

where $\mathcal{F}[]$ is a bang-free evaluation context and $\mathcal{Q}[\boldsymbol{\beta}]$ is a subtrail that indicates where in context $\mathcal{F}$ the $\boldsymbol{\beta}$-step was performed. As the size of the term being executed (and distance between an audited unit constructor and the redexes) grows, this evaluation strategy slows down quadratically in the worst case; eagerly materializing the traces likewise imposes additional storage cost.

While some computational overhead seems inevitable to accommodate auditing, both of these costs can in principle be mitigated. Trail permutations are computationally expensive and can often be delayed without any impact on the final outcome. Pushing trails to the closest outer bang does not serve any real purpose: it would be more efficient to keep the trail where it was created and perform normalization only if and when the trail must be inspected (and this operation does not even actually require an actual pushout of trails, because we can reuse term structure to compute the trail structure on-the-fly).

This situation has a well-studied analogue: in the λ-calculus, it is not necessarily efficient to eagerly perform all substitutions as soon as a β-reduction happens. Instead, calculi of *explicit substitutions* such as Abadi et al.'s $\lambda\sigma$ [1] have been developed in which substitutions are explicitly tracked and rewritten. Explicit substitution calculi have been studied extensively as a bridge between the declarative rewriting rules of λ-calculi and efficient implementations. Inspired by this work, we hypothesize that calculi with auditing can be implemented more efficiently by delaying the operations of trail extraction and erasure, using explicit symbolic representations for these operations instead of performing them eagerly.

Particular care must be placed in making sure that the trails we produce still correctly describe the order in which operations were actually performed (e.g. respecting call-by-name or call-by-value reduction): when we perform a principal contraction, pre-existing trail annotations must be recorded as history that happened *before* the contraction, and not after. In the original eager reduction style, this is trivial because we never contract terms containing trails; however, we will show that, thanks to the explicit trail operations, correctness can be achieved even when adopting a lazy normalization of trails.

Contributions. We study an extension of Abadi et al.'s calculus $\lambda\sigma$ [1] with explicit auditing operations. We consider a simplified, untyped variant $\mathbf{CAU}^-$ of the Calculus of Audited Units (Sect. 2); this simplifies our presentation because type information is not needed during execution. We revisit $\lambda\sigma$ in Sect. 3, extend it to include auditing and trail inspection features, and discuss problems with this initial, naïve approach. We address these problems by developing a new calculus $\mathbf{CAU}^-_\sigma$ with explicit versions of the "trail extraction" and "trail erasure" operations (Sect. 4), and we show that it correctly refines $\mathbf{CAU}^-$ (subject to an obvious translation). In Sect. 5, we build on $\mathbf{CAU}^-_\sigma$ to define an abstract machine for audited computation and prove its correctness. Some proofs have been omitted due to space constraints and are included in the extended version of this paper.

2 The Untyped Calculus of Audited Units

The language $\mathbf{CAU}^-$ presented here is an untyped version of the calculi λ^h [14] and Ricciotti and Cheney's λ^{hc} [25] obtained by erasing all typing information and a few other related technicalities: this will allow us to address all the interesting issues related to the reduction of **CAU** terms, but with a much less pedantic syntax. To help us explain the details of the calculus, we adapt some examples from our previous paper [25]; other examples are described by Bavera and Bonelli [14].

Unlike the typed variant of the calculus, we only need one sort of variables, denoted by the letters $x, y, z \ldots$. The syntax of $\mathbf{CAU}^-$ is as follows:

$$\textbf{Terms}\ M, N ::= x \mid \lambda x.M \mid M\ N \mid \mathrm{let}_!(x := M, N) \mid !_q M \mid q \triangleright M \mid \iota(\vartheta)$$
$$\textbf{Trails}\ \ q, q' \ ::= \mathbf{r} \mid \mathbf{t}(q, q') \mid \boldsymbol{\beta} \mid \boldsymbol{\beta}_! \mid \mathbf{ti} \mid \mathbf{lam}(q) \mid \mathbf{app}(q, q') \mid \mathbf{let}_!(q, q') \mid \mathbf{tb}(\zeta)$$

$\mathbf{CAU}^-$ extends the pure lambda calculus with *audited units* $!_q M$ (colloquially, "bang M"), whose purpose is to decorate the term M with a log q of its computation history, called *trail* in our terminology: when M evolves as a result of computation, q will be updated by adding information about the reduction rules that have been applied. The form $!_q M$ is in general not intended for use in source programs: instead, we will write $!\ M$ for $!_{\mathbf{r}} M$, where $\mathbf{r}$ represents the empty execution history (*reflexivity* trail).

Audited units can then be employed in larger terms by means of the "let-bang" operator, which unpacks an audited unit and thus allows us to access its contents. The variable declared by a $\mathrm{let}_!$ is bound in its second argument: in

essence $\mathrm{let}_!(x := !_q M, N)$ will reduce to N, where free occurrences of x have been replaced by M; the trail q will not be discarded, but will be used to produce a new trail explaining this reduction.

The expression form $q \triangleright M$ is an auxiliary, intermediate annotation of M with partial history information which is produced during execution and will eventually stored in the closest surrounding bang.

Example 1. In $\mathbf{CAU}^-$ we can express history-carrying terms explicitly: for instance, if we use $\bar{n}$ to denote the Church encoding of a natural number n, and *plus* or *fact* for lambda terms computing addition and factorial on said representation, we can write audited units like

$$!_q \bar{2} \qquad !_{q'} \bar{6}$$

where q is a trail representing the history of $\bar{2}$ i.e., for instance, a witness for the computation that produced $\bar{2}$ by reducing $plus\ \bar{1}\ \bar{1}$; likewise, q' might describe how computing $fact\ \bar{3}$ produced $\bar{6}$. Supposing we wish to add these two numbers together, at the same time retaining their history, we will use the $\mathrm{let}_!$ construct to look inside them:

$$\mathrm{let}_!(x := !_q \bar{2}, \mathrm{let}_!(y := !_{q'} \bar{6}, plus\ x\ y)) \longrightarrow\!\!\!\!\rightarrow q'' \triangleright \bar{8}$$

where the final trail q'' is produced by composing q and q'; if this reduction happens inside an external bang, q'' will eventually be captured by it.

Trails, representing sequences of reduction steps, encode the (possibly partial) computation history of a given subterm. The main building blocks of trails are $\boldsymbol{\beta}$ (representing standard beta reduction), $\boldsymbol{\beta_!}$ (contraction of a let-bang redex) and **ti** (denoting the execution of a trail inspection). For every class of terms we have a corresponding congruence trail ($\mathbf{lam}, \mathbf{app}, \mathbf{let}_!, \mathbf{tb}$, the last of which is associated with trail inspections), with the only exception of bangs, which do not need a congruence rule because they capture all the computation happening inside them. The syntax of trails is completed by reflexivity **r** (representing a null computation history, i.e. a term that has not reduced yet) and transitivity **t** (i.e. sequential composition of execution steps). As discussed by our earlier paper [25], we omit Bavera and Bonelli's symmetry trail form.

Example 2. We build a pair of natural numbers using Church's encoding:

$$\begin{aligned} !\ ((\lambda x, y, p.p\ x\ y)\ 2)\ 6 &\rightarrow !_{\mathbf{t}(\mathbf{r},\mathbf{app}(\boldsymbol{\beta},\mathbf{r}))}\ (\lambda y, p.p\ 2\ y)\ 6 \\ &\rightarrow !_{\mathbf{t}(\mathbf{t}(\mathbf{r},\mathbf{app}(\boldsymbol{\beta},\mathbf{r})),\boldsymbol{\beta})}\ \lambda p.p\ 2\ 6 \end{aligned}$$

The trail for the first computation step is obtained by transitivity (trail constructor **t**) from the original trivial trail (**r**, i.e. reflexivity) composed with $\boldsymbol{\beta}$, which describes the reduction of the applied lambda: this subtrail is wrapped in a congruence **app** because the reduction takes place deep inside the left-hand subterm of an application (the other argument of **app** is reflexivity, because no reduction takes place in the right-hand subterm).

The second beta-reduction happens at the top level and is thus not wrapped in a congruence. It is combined with the previous trail by means of transitivity.

The last term form $\iota(\vartheta)$, called *trail inspection*, will perform primitive recursion on the computation history of the current audited unit. The metavariables ϑ and ζ associated with trail inspections are *trail replacements*, i.e. maps associating to each possible trail constructor, respectively, a term or a trail:

$$\vartheta ::= \{M_1/\mathbf{r}, M_2/\mathbf{t}, M_3/\boldsymbol{\beta}, M_4/\boldsymbol{\beta_!}, M_5/\mathbf{ti}, M_6/\mathbf{lam}, M_7/\mathbf{app}, M_8/\mathbf{let}_!, M_9/\mathbf{tb}\}$$
$$\zeta ::= \{q_1/\mathbf{r}, q_2/\mathbf{t}, q_3/\boldsymbol{\beta}, q_4/\boldsymbol{\beta_!}, q_5/\mathbf{ti}, q_6/\mathbf{lam}, q_7/\mathbf{app}, q_8/\mathbf{let}_!, q_9/\mathbf{tb}\}$$

When the trail constructors are irrelevant for a certain ϑ or ζ, we will omit them, using the notations $\{\overrightarrow{M}\}$ or $\{\overrightarrow{q}\}$. These constructs represent (or describe) the nine cases of a structural recursion operator over trails, which we write as $q\vartheta$.

Definition 1. *The operation $q\vartheta$, which produces a term by structural recursion on q applying the inspection branches ϑ, is defined as follows:*

$$\mathbf{r}\vartheta \triangleq \vartheta(\mathbf{r}) \qquad \mathbf{t}(q,q')\vartheta \triangleq \vartheta(\mathbf{t})\ (q\vartheta)\ (q'\vartheta) \qquad \boldsymbol{\beta}\vartheta \triangleq \vartheta(\boldsymbol{\beta}) \qquad \boldsymbol{\beta_!}\vartheta \triangleq \vartheta(\boldsymbol{\beta_!})$$
$$\mathbf{ti}\vartheta \triangleq \vartheta(\mathbf{ti}) \qquad \mathbf{lam}(q)\vartheta \triangleq \vartheta(\mathbf{lam})\ (q\vartheta) \qquad \mathbf{tb}(\{\overrightarrow{q}\})\vartheta \triangleq \vartheta(\mathbf{tb})\ \overrightarrow{(q\vartheta)}$$
$$\mathbf{app}(q,q')\vartheta \triangleq \vartheta(\mathbf{app})\ (q\vartheta)\ (q'\vartheta) \qquad \mathbf{let}_!(q,q')\vartheta \triangleq \vartheta(\mathbf{let}_!)\ (q\vartheta)\ (q'\vartheta)$$

where the sequence $\overrightarrow{(q\vartheta)}$ is obtained from $\overrightarrow{q}$ by pointwise recursion.

Example 3. Trail inspection can be used to count all of the contraction steps in the history of an audited unit, by means of the following trail replacement:

$$\vartheta_+ ::= \{\bar{0}/\mathbf{r}, \mathit{plus}/\mathbf{t}, \bar{1}/\boldsymbol{\beta}, \bar{1}/\boldsymbol{\beta_!}, \bar{1}/\mathbf{ti}, \lambda x.x/\mathbf{lam}, \mathit{plus}/\mathbf{app}, \mathit{plus}/\mathbf{let}_!, \mathit{sum}/\mathbf{tb}\}$$

where *sum* is a variant of *plus* taking nine arguments, as required by the arity of **tb**. For example, we can count the contractions in $q = \mathbf{t}(\mathbf{let}_!(\boldsymbol{\beta}, \mathbf{r}), \boldsymbol{\beta_!})$ as:

$$q\vartheta_+ = \mathit{plus}\ (\mathit{plus}\ \bar{1}\ \bar{0})\ \bar{1}$$

2.1 Reduction

Reduction in $\mathbf{CAU}^-$ includes rules to contract the usual beta redexes (applied lambda abstractions), "beta-bang" redexes, which unpack the bang term appearing as the definiens of a $\mathrm{let}_!$, and trail inspections. These rules, which we call *principal contractions*, are defined as follows:

$$(\lambda x.M)\ N \xrightarrow{\beta} \boldsymbol{\beta} \triangleright M\{N/x\} \qquad \mathrm{let}_!(x := !_q M, N) \xrightarrow{\beta} \boldsymbol{\beta_!} \triangleright N\{q \triangleright M/x\}$$
$$!_q\mathcal{F}[\iota(\vartheta)] \xrightarrow{\beta} !_q\mathcal{F}[\mathbf{ti} \triangleright q\vartheta]$$

Substitution $M\{N/x\}$ is defined in the traditional way, avoiding variable capture. The first contraction is familiar, except for the fact that the reduct $M\{N/x\}$ has been annotated with a $\boldsymbol{\beta}$ trail. The second one deals with unpacking a bang: from $!_q M$ we obtain $q \triangleright M$, which is then substituted for x in the target term N; the resulting term is annotated with a $\boldsymbol{\beta_!}$ trail. The third

contraction defines the result of a trail inspection $\iota(\vartheta)$. Trail inspection will be contracted by capturing the current history, as stored in the nearest enclosing bang, and performing structural recursion on it according to the branches defined by ϑ. The concept of "nearest enclosing bang" is made formal by contexts $\mathcal{F}$ in which the hole cannot appear inside a bang (or *bang-free* contexts, for short):

$$\mathcal{F} ::= \blacksquare \mid \lambda x.\mathcal{F} \mid \mathcal{F}\ M \mid M\ \mathcal{F} \mid \mathrm{let}_!(\mathcal{F}, M) \mid \mathrm{let}_!(M, \mathcal{F}) \mid q \triangleright \mathcal{F} \mid \iota(\{\overrightarrow{M}, \mathcal{F}, \overrightarrow{N}\})$$

The definition of the principal contractions is completed, as usual, by a contextual closure rule stating that they can appear in any context $\mathcal{E}$:

$$\mathcal{E} ::= \blacksquare \mid \lambda x.\mathcal{E} \mid \mathcal{E}\ M \mid M\ \mathcal{E} \mid \mathrm{let}_!(\mathcal{E}, M) \mid \mathrm{let}_!(M, \mathcal{E}) \mid !_q\mathcal{E} \mid q \triangleright \mathcal{E} \mid \iota(\{\overrightarrow{M}, \mathcal{E}, \overrightarrow{N}\})$$

$$\frac{M \xrightarrow{\beta} N}{\mathcal{E}[M] \xrightarrow{\beta} \mathcal{E}[N]}$$

The principal contractions introduce local trail subterms $q' \triangleright M$, which can block other reductions. Furthermore, the rule for trail inspection assumes that the q annotating the enclosing bang really is a complete log of the history of the audited unit; but at the same time, it violates this invariant, because the **ti** trail created after the contraction is not merged with the original history q.

For these reasons, we only want to perform principal contractions on terms not containing local trails: after each principal contraction, we apply the following rewrite rules, called *permutation reductions*, to ensure that the local trail is moved to the nearest enclosing bang:

$$\begin{array}{rr}
\mathbf{r} \triangleright M \xrightarrow{\tau} M & q \triangleright (q' \triangleright M) \xrightarrow{\tau} \mathbf{t}(q, q') \triangleright M \\
!_q(q' \triangleright M) \xrightarrow{\tau} !_{\mathbf{t}(q,q')}M & \lambda x.(q \triangleright M) \xrightarrow{\tau} \mathbf{lam}(q) \triangleright \lambda x.M \\
(q \triangleright M)\ N \xrightarrow{\tau} \mathbf{app}(q, \mathbf{r}) \triangleright M\ N & M\ (q \triangleright N) \xrightarrow{\tau} \mathbf{app}(\mathbf{r}, q) \triangleright M\ N \\
\multicolumn{2}{c}{\mathrm{let}_!(x := q \triangleright M, N) \xrightarrow{\tau} \mathbf{let}_!(q, \mathbf{r}) \triangleright \mathrm{let}_!(x := M, N)} \\
\multicolumn{2}{c}{\mathrm{let}_!(x := M, q \triangleright N) \xrightarrow{\tau} \mathbf{let}_!(\mathbf{r}, q) \triangleright \mathrm{let}_!(x := M, N)} \\
\multicolumn{2}{c}{\iota(\{M_1, \ldots, q \triangleright M_i, \ldots, M_9\}) \xrightarrow{\tau} \mathbf{tb}(\{\mathbf{r}, \ldots, q, \ldots, \mathbf{r}\}) \triangleright \iota(\{M_1, \ldots, M_9\})}
\end{array}$$

Moreover, the following rules are added to the $\xrightarrow{\tau}$ relation to ensure confluence:

$$\begin{array}{rrr}
\mathbf{t}(q, \mathbf{r}) \xrightarrow{\tau} q & \mathbf{t}(\mathbf{r}, q) \xrightarrow{\tau} q & \mathbf{tb}(\{\overrightarrow{\mathbf{r}}\}) \xrightarrow{\tau} \mathbf{r} \\
\mathbf{app}(\mathbf{r}, \mathbf{r}) \xrightarrow{\tau} \mathbf{r} & \mathbf{lam}(\mathbf{r}) \xrightarrow{\tau} \mathbf{r} & \mathbf{let}_!(\mathbf{r}, \mathbf{r}) \xrightarrow{\tau} \mathbf{r} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{t}(q_1, q_2), q_3) \xrightarrow{\tau} \mathbf{t}(q_1, \mathbf{t}(q_2, q_3))} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{lam}(q), \mathbf{lam}(q')) \xrightarrow{\tau} \mathbf{lam}(\mathbf{t}(q, q'))} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{lam}(q_1), \mathbf{t}(\mathbf{lam}(q_1'), q)) \xrightarrow{\tau} \mathbf{t}(\mathbf{lam}(\mathbf{t}(q_1, q_1')), q)} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{app}(q_1, q_2), \mathbf{app}(q_1', q_2')) \xrightarrow{\tau} \mathbf{app}(\mathbf{t}(q_1, q_1'), \mathbf{t}(q_2, q_2'))} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{app}(q_1, q_2), \mathbf{t}(\mathbf{app}(q_1', q_2')), q) \xrightarrow{\tau} \mathbf{t}(\mathbf{app}(\mathbf{t}(q_1, q_1'), \mathbf{t}(q_2, q_2')), q)} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{let}_!(q_1, q_2), \mathbf{let}_!(q_1', q_2')) \xrightarrow{\tau} \mathbf{let}_!(\mathbf{t}(q_1, q_1'), \mathbf{t}(q_2, q_2'))} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{let}_!(q_1, q_2), \mathbf{t}(\mathbf{let}_!(q_1', q_2')), q) \xrightarrow{\tau} \mathbf{t}(\mathbf{let}_!(\mathbf{t}(q_1, q_1'), \mathbf{t}(q_2, q_2')), q)} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{tb}(\overrightarrow{q_1}), \mathbf{tb}(\overrightarrow{q_2})) \xrightarrow{\tau} \mathbf{tb}(\overrightarrow{\mathbf{t}(q_1, q_2)})} \\
\multicolumn{3}{c}{\mathbf{t}(\mathbf{tb}(\overrightarrow{q_1}), \mathbf{t}(\mathbf{tb}(\overrightarrow{q_2}), q)) \xrightarrow{\tau} \mathbf{t}(\mathbf{tb}(\overrightarrow{\mathbf{t}(q_1, q_2)}), q)}
\end{array}$$

As usual, $\xrightarrow{\tau}$ is completed by a contextual closure rule. We prove

Lemma 1 ([14]). *$\xrightarrow{\tau}$ is terminating and confluent.*

When a binary relation $\xrightarrow{\mathcal{R}}$ on terms is terminating and confluent, we will write $\mathcal{R}(M)$ for the unique $\mathcal{R}$-normal form of M. Since principal contractions must be performed on τ-normal terms, it is convenient to merge contraction and τ-normalization in a single operation, which we will denote by $\xrightarrow{\mathbf{CAU}^-}$:

$$\frac{M \xrightarrow{\beta} N}{M \xrightarrow{\mathbf{CAU}^-} \tau(N)}$$

Example 4. We take again the term from Example 1 and reduce the outer $\mathrm{let}_!$ as follows:

$$\begin{aligned}
&! \ \mathrm{let}_!(x := !_q 2, \mathrm{let}_!(y := !_{q'} 6, \mathit{plus}\ x\ y)) \\
&\xrightarrow{\beta} \ ! \ (\boldsymbol{\beta_!} \triangleright \mathrm{let}_!(y := !_{q'} 6, \mathit{plus}\ (q \triangleright 2)\ y)) \\
&\xrightarrow{\tau} \!\!\!\!\rightarrow \ !_{\mathbf{t}(\beta_!, \mathbf{let}_!(\mathbf{r}, \mathbf{app}(\mathbf{app}(\mathbf{r}, q), \mathbf{r})))}\, \mathrm{let}_!(y := !_{q'} 6, \mathit{plus}\ 2\ y)
\end{aligned}$$

This $\mathrm{let}_!$-reduction substitutes $q \triangleright 2$ for x; a $\boldsymbol{\beta_!}$ trail is produced immediately inside the bang, in the same position as the redex. Then, we τ-normalize the resulting term, which results in the two trails being combined and used to annotate the enclosing bang.

3 Naïve Explicit Substitutions

We seek to adapt the existing abstract machines for the efficient normalization of lambda terms to $\mathbf{CAU}^-$. Generally speaking, most abstract machines act on nameless terms, using de Bruijn's indices [15], thus avoiding the need to perform renaming to avoid variable capture when substituting a term into another.

Moreover, since a substitution $M\{N/x\}$ requires to scan the whole term M and is thus *not* a constant time operation, it is usually not executed immediately in an eager way. The abstract machine actually manipulates *closures*, or pairs of a term M and an environment s declaring lazy substitutions for each of the free variables in M: this allows s to be applied in an incremental way, while scanning the term M in search for a redex. In the $\lambda\sigma$-calculus of Abadi et al. [1], lazy substitutions and closures are manipulated explicitly, providing an elegant bridge between the classical λ-calculus and its concrete implementation in abstract machines. Their calculus expresses beta reduction as the rule

$$(\lambda.M)\ N \longrightarrow M[N]$$

where $\lambda.M$ is a nameless abstraction *à la de Bruijn*, and $[N]$ is a (suspended) *explicit substitution* mapping the variable corresponding to the first dangling index in M to N, and all the other variables to themselves. Terms in the form

$M[s]$, representing closures, are syntactically part of $\lambda\sigma$, as opposed to substitutions $M\{N/x\}$, which are meta-operations that *compute* a term. In this section we formulate a first attempt at adding explicit substitutions to $\mathbf{CAU}^-$. We will not prove any formal result for the moment, as our purpose is to elicit the difficulties of such a task. An immediate adaptation of $\lambda\sigma$-like explicit substitutions yields the following syntax:

$$\begin{array}{ll} \textbf{Terms} & M, N ::= 1 \mid \lambda.M \mid M\ N \mid \mathrm{let}_!(M, N) \mid !_q M \mid q \triangleright M \mid \iota(\vartheta) \mid M[s] \\ \textbf{Substitutions} & s, t \quad ::= \langle\rangle \mid \uparrow \mid s \circ t \mid M \cdot s \end{array}$$

where 1 is the first de Bruijn index, the nameless λ binds the first free index of its argument, and similarly the nameless $\mathrm{let}_!$ binds the first free index of its second argument. Substitutions include the identity (or empty) substitution $\langle\rangle$, lift $\uparrow$ (which reinterprets all free indices n as their successor $n+1$), the composition $s \circ t$ (equivalent to the sequencing of s and t) and finally $M \cdot s$ (indicating a substitution that will replace the first free index with M, and other indices n with their predecessor $n-1$ under substitution s). Trails are unchanged.

We write $M[N_1 \cdots N_k]$ as syntactic sugar for $M[N_1 \cdots N_k \cdot \langle\rangle]$. Then, $\mathbf{CAU}^-$ reductions can be expressed as follows:

$$(\lambda.M)\ N \xrightarrow{\beta} \boldsymbol{\beta} \triangleright M[N] \qquad \mathrm{let}_!(!_q M, N) \xrightarrow{\beta} \boldsymbol{\beta_!} \triangleright N[q \triangleright M]$$
$$!_q \mathcal{F}[\iota(\vartheta)] \xrightarrow{\beta} !_q \mathcal{F}[\mathbf{ti} \triangleright q\vartheta]$$

(trail inspection, which does not use substitutions, is unchanged). The idea is that explicit substitutions make reduction more efficient because their evaluation does not need to be performed all at once, but can be delayed, partially or completely; delayed explicit substitutions applied to the same term can be merged, so that the term does not need to be scanned twice. The evaluation of explicit substitution can be defined by the following σ-rules:

$$\begin{array}{rlrl} 1[\langle\rangle] & \xrightarrow{\sigma} 1 & \langle\rangle \circ s & \xrightarrow{\sigma} s \\ 1[M \cdot s] & \xrightarrow{\sigma} M & \uparrow \circ \langle\rangle & \xrightarrow{\sigma} \uparrow \\ (\lambda M)[s] & \xrightarrow{\sigma} \lambda(M[1 \cdot (s \circ \uparrow)]) & \uparrow \circ (M \cdot s) & \xrightarrow{\sigma} s \\ (M\ N)[s] & \xrightarrow{\sigma} M[s]\ N[s] & (M \cdot s) \circ t & \xrightarrow{\sigma} M[t] \cdot (s \circ t) \\ (!_q M)[s] & \xrightarrow{\sigma} !_q(M[s]) & (s_1 \circ s_2) \circ s_3 & \xrightarrow{\sigma} s_1 \circ (s_2 \circ s_3) \\ \mathrm{let}_!(M, N)[s] & \xrightarrow{\sigma} \mathrm{let}_!(M, N[1 \cdot (s \circ \uparrow)]) & (q \triangleright M)[s] & \xrightarrow{\sigma} q \triangleright (M[s]) \\ \iota(\{\overrightarrow{M}\})[s] & \xrightarrow{\sigma} \iota(\{\overrightarrow{M[s]}\}) & M[s][t] & \xrightarrow{\sigma} M[s \circ t] \end{array}$$

These rules are a relatively minor adaptation from those of $\lambda\sigma$: as in that language, σ-normal forms do not contain explicit substitutions, save for the case of the index 1, which may be lifted multiple times, e.g.:

$$1[\uparrow^n] = 1[\underbrace{\uparrow \circ \cdots \circ \uparrow}_{n \text{ times}}]$$

If we take $1[\uparrow^n]$ to represent the de Bruijn index $n+1$, as in $\lambda\sigma$, σ-normal terms coincide with a nameless representation of $\mathbf{CAU}^-$.

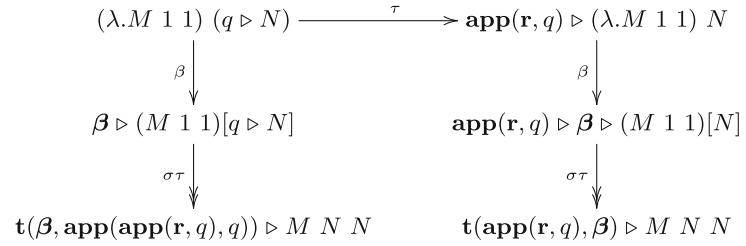

Fig. 1. Non-joinable reduction in $\mathbf{CAU}^-$ with naïve explicit substitutions

The σ-rules are deferrable, in that we can perform β-reductions even if a term is not in σ-normal form. We would like to treat the τ-rules in the same way, perhaps performing τ-normalization only before trail inspection; however, we can see that changing the order of τ-rules destroys confluence even when β-redexes are triggered in the same order. Consider for example the reductions in Fig. 1: performing a τ-step before the beta-reduction, as in the right branch, yields the expected result. If instead we delay the τ-step, the trail q decorating N is duplicated by beta reduction; furthermore, the order of q and $\boldsymbol{\beta}$ gets mixed up: even though q records computation that happened (once) *before* $\boldsymbol{\beta}$, the final trail asserts that q happened (twice) *after* $\boldsymbol{\beta}$.[1] As expected, the two trails (and consequently the terms they decorate) are not joinable.

The example shows that β-reduction on terms whose trails have not been normalized is *anachronistic*. If we separated the trails stored in a term from the underlying, trail-less term, we might be able to define a *catachronistic*, or time-honoring version of β-reduction. For instance, if we write $\lfloor M \rfloor$ for trail-erasure and $\lceil M \rceil$ for the trail-extraction of a term M, catachronistic beta reduction could be written as follows:

$$(\lambda.M)\ N \xrightarrow{\beta} \mathbf{t}(\lceil (\lambda.M)\ N \rceil, \boldsymbol{\beta}) \triangleright \lfloor M \rfloor\ [\lfloor N \rfloor]$$

$$\mathrm{let}_!(!_q M, N) \xrightarrow{\beta} \mathbf{t}(\lceil \mathrm{let}_!(!_q M, N) \rceil, \boldsymbol{\beta_!}) \triangleright \lfloor N \rfloor\ [q \triangleright M]$$

$$!_q \mathcal{F}[\iota(\vartheta)] \xrightarrow{\beta} !_q \mathcal{F}[\mathbf{ti} \triangleright q'\vartheta] \qquad (\text{where } q' = \tau(\mathbf{t}(q, \lceil \mathcal{F}[\iota(\vartheta)] \rceil)))$$

We could easily define trail erasure and extraction as operations on pure $\mathbf{CAU}^-$ terms (without explicit substitutions), but the cost of eagerly computing their result would be proportional to the size of the input term; furthermore, the extension to explicit substitutions would not be straightforward. Instead, in the next section, we will describe an extended language to manipulate trail projections explicitly.

[1] Although the right branch describes an unfaithful account of history, it is still a coherent one: we will explain this in more detail in the conclusions.

4 The Calculus $\mathbf{CAU}^{-}_{\sigma}$

We define the untyped Calculus of Audited Units with explicit substitutions, or $\mathbf{CAU}^{-}_{\sigma}$, as the following extension of the syntax of $\mathbf{CAU}^{-}$ presented in Sect. 2:

$$M, N ::= 1 \mid \lambda.M \mid M\ N \mid \mathrm{let}_!(M, N) \mid !_q M \mid q \triangleright M \mid \iota(\vartheta) \mid M[s] \mid \lfloor M \rfloor$$
$$q, q' ::= \mathbf{r} \mid \mathbf{t}(q, q') \mid \boldsymbol{\beta} \mid \boldsymbol{\beta_!} \mid \mathbf{ti} \mid \mathbf{lam}(q) \mid \mathbf{app}(q, q') \mid \mathbf{let}_!(q, q') \mid \mathbf{tb}(\zeta) \mid \lceil M \rceil$$
$$s, t ::= \langle\rangle \mid \uparrow \mid M \cdot s \mid s \circ t$$

$\mathbf{CAU}^{-}_{\sigma}$ builds on the observations about explicit substitutions we made in the previous section: in addition to closures $M[s]$, it provides syntactic trail erasures denoted by $\lfloor M \rfloor$; dually, the syntax of trails is extended with the explicit trail-extraction of a term, written $\lceil M \rceil$. In the naïve presentation, we gave a satisfactory set of σ-rules defining the semantics of explicit substitutions, which we keep as part of $\mathbf{CAU}^{-}_{\sigma}$. To express the semantics of explicit projections, we provide in Fig. 2 rules stating that $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ commute with most term constructors (but not with !) and are blocked by explicit substitutions. These rules are completed by congruence rules asserting that they can be used in any subterm or subtrail of a given term or trail.

$$\begin{array}{rcl@{\qquad}rcl}
\lfloor 1 \rfloor & \xrightarrow{\sigma} & 1 & \lceil 1 \rceil & \xrightarrow{\sigma} & \mathbf{r} \\
\lfloor 1[\uparrow^n] \rfloor & \xrightarrow{\sigma} & 1[\uparrow^n] & \lceil 1[\uparrow^n] \rceil & \xrightarrow{\sigma} & \mathbf{r} \\
\lfloor \lambda.M \rfloor & \xrightarrow{\sigma} & \lambda.\lfloor M \rfloor & \lceil \lambda.M \rceil & \xrightarrow{\sigma} & \mathbf{lam}(\lceil M \rceil) \\
\lfloor M\ N \rfloor & \xrightarrow{\sigma} & \lfloor M \rfloor\ \lfloor N \rfloor & \lceil M\ N \rceil & \xrightarrow{\sigma} & \mathbf{app}(\lceil M \rceil, \lceil N \rceil) \\
\lfloor !_q M \rfloor & \xrightarrow{\sigma} & !_q M & \lceil !_q M \rceil & \xrightarrow{\sigma} & \mathbf{r} \\
\lfloor \mathrm{let}_!(M, N) \rfloor & \xrightarrow{\sigma} & \mathrm{let}_!(\lfloor M \rfloor, \lfloor N \rfloor) & \lceil \mathrm{let}_!(M, N) \rceil & \xrightarrow{\sigma} & \mathbf{let}_!(\lceil M \rceil, \lceil N \rceil) \\
\lfloor q \triangleright M \rfloor & \xrightarrow{\sigma} & \lfloor M \rfloor & \lceil q \triangleright M \rceil & \xrightarrow{\sigma} & \mathbf{t}(q, \lceil M \rceil) \\
\left\lfloor \iota(\{\overrightarrow{M}\}) \right\rfloor & \xrightarrow{\sigma} & \iota(\{\overrightarrow{\lfloor M \rfloor}\}) & \left\lceil \iota(\{\overrightarrow{M}\}) \right\rceil & \xrightarrow{\sigma} & \mathbf{tb}(\{\overrightarrow{\lceil M \rceil}\})
\end{array}$$

Fig. 2. σ-reduction for explicit trail projections

The τ rules from Sect. 2 are added to $\mathbf{CAU}^{-}_{\sigma}$ with the obvious adaptations. We prove that σ and τ, together, yield a terminating and confluent rewriting system.

Theorem 1. $(\xrightarrow{\sigma} \cup \xrightarrow{\tau})$ *is terminating and confluent.*

Proof. Tools like AProVE [17] are able to prove termination automatically. Local confluence can be proved easily by considering all possible pairs of rules: full confluence follows as a corollary of these two results.

4.1 Beta Reduction

We replace the definition of β-reduction by the following lazy rules that use trail-extraction and trail-erasure to ensure that the correct trails are eventually produced:

$$(\lambda.M)\ N \xrightarrow{\text{Beta}} \mathbf{t}(\mathbf{app}(\mathbf{lam}(\lceil M \rceil), \lceil N \rceil), \boldsymbol{\beta}) \triangleright \lfloor M \rfloor\, [\lfloor N \rfloor]$$
$$\text{let}_!(!_q M, N) \xrightarrow{\text{Beta}} \mathbf{t}(\mathbf{let}_!(\mathbf{r}, \lceil N \rceil), \boldsymbol{\beta}_!) \triangleright \lfloor N \rfloor\, [q \triangleright M]$$
$$!_q \mathcal{F}[\iota(\vartheta)] \xrightarrow{\text{Beta}} !_q \mathcal{F}[\mathbf{ti} \triangleright q'\vartheta] \qquad (\text{where } q' = \sigma\tau(\mathbf{t}(q, \lceil \mathcal{F}[\iota(\vartheta)] \rceil)))$$

where $\mathcal{F}$ specifies that the reduction cannot take place within a bang, a substitution, or a trail erasure:

$$\mathcal{F} ::= \blacksquare \mid \lambda.\mathcal{F} \mid (\mathcal{F}\ N) \mid (M\ \mathcal{F}) \mid \text{let}_!(\mathcal{F}, N) \mid \text{let}_!(M, \mathcal{F}) \mid q \triangleright \mathcal{F} \mid \iota(\overrightarrow{M}, \mathcal{F}, \overrightarrow{N}) \mid \mathcal{F}[s]$$

As usual, the relation is extended to inner subterms by means of congruence rules. However, we need to be careful: we cannot reduce within a trail-erasure, because if we did, the newly created trail would be erroneously erased:

$$\begin{aligned}
\text{wrong: } & \lfloor (\lambda.M)\ N \rfloor \\
& \xrightarrow{\text{Beta}} \lfloor \mathbf{t}(\mathbf{app}(\mathbf{lam}(\lceil M \rceil), \lceil N \rceil), \boldsymbol{\beta}) \triangleright \lfloor M \rfloor\, [\lfloor N \rfloor] \rfloor \\
& \xrightarrow{\sigma} \lfloor \lfloor M \rfloor\, [\lfloor N \rfloor] \rfloor \\
\text{correct: } & \lfloor (\lambda.M)\ N \rfloor \\
& \xrightarrow{\sigma} (\lambda.\, \lfloor M \rfloor)\ \lfloor N \rfloor \\
& \xrightarrow{\text{Beta}} \mathbf{t}(\mathbf{app}(\mathbf{lam}(\lceil \lfloor M \rfloor \rceil), \lceil \lfloor N \rfloor \rceil), \boldsymbol{\beta}) \triangleright \lfloor M \rfloor\, [\lfloor N \rfloor]
\end{aligned}$$

This is why we express the congruence rule by means of contexts $\mathcal{E}_\sigma$ such that holes cannot appear within erasures (the definition also employs substitution contexts $\mathcal{S}_\sigma$ to allow reduction within substitutions):

$$\frac{M \xrightarrow{\text{Beta}} N}{\mathcal{E}_\sigma[M] \xrightarrow{\text{Beta}} \mathcal{E}_\sigma[N]}$$

Formally, evaluation contexts are defined as follows:

Definition 2 (evaluation context)

$$\begin{aligned}
\mathcal{E}_\sigma ::= {} & \blacksquare \mid \lambda.\mathcal{E}_\sigma \mid (\mathcal{E}_\sigma\ N) \mid (M\ \mathcal{E}_\sigma) \mid \text{let}_!(\mathcal{E}_\sigma, N) \mid \text{let}_!(M, \mathcal{E}_\sigma) \mid !_q \mathcal{E}_\sigma \mid q \triangleright \mathcal{E}_\sigma \\
& \mid \iota(\{\overrightarrow{M}, \mathcal{E}_\sigma, \overrightarrow{N}\}) \mid \mathcal{E}_\sigma[s] \mid M[\mathcal{S}_\sigma] \\
\mathcal{S}_\sigma ::= {} & \mathcal{S}_\sigma \circ t \mid s \circ \mathcal{S}_\sigma \mid \mathcal{E}_\sigma \cdot s \mid M \cdot \mathcal{S}_\sigma
\end{aligned}$$

We denote $\sigma\tau$-equivalence (the reflexive, symmetric, and transitive closure of $\xrightarrow{\sigma\tau}$) by means of $\xleftrightarrow{\sigma\tau}$. As we will prove, $\sigma\tau$-equivalent $\mathbf{CAU}^-_\sigma$ terms can be interpreted as the same $\mathbf{CAU}^-$ term: for this reason, we define reduction in $\mathbf{CAU}^-_\sigma$ as the union of $\xrightarrow{\text{Beta}}$ and $\xleftrightarrow{\sigma\tau}$:

$$\xrightarrow{\mathbf{CAU}^-_\sigma} := \xrightarrow{\text{Beta}} \cup \xleftrightarrow{\sigma\tau} \tag{1}$$

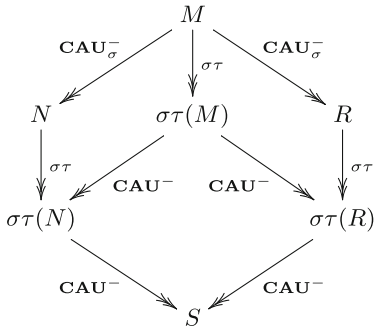

Fig. 3. Relativized confluence for $\mathbf{CAU}^-_\sigma$.

4.2 Properties of the Rewriting System

The main results we prove concern the relationship between $\mathbf{CAU}^-$ and $\mathbf{CAU}^-_\sigma$: firstly, every $\mathbf{CAU}^-$ reduction must still be a legal reduction within $\mathbf{CAU}^-_\sigma$; in addition, it should be possible to interpret every $\mathbf{CAU}^-_\sigma$ reduction as a $\mathbf{CAU}^-$ reduction over suitable $\sigma\tau$-normal terms.

Theorem 2. *If* $M \xrightarrow{\boldsymbol{CAU^-}}\!\!\!\!\!\rightarrow N$, *then* $M \xrightarrow{\boldsymbol{CAU^-_\sigma}}\!\!\!\!\!\rightarrow N$.

Theorem 3. *If* $M \xrightarrow{\boldsymbol{CAU^-_\sigma}}\!\!\!\!\!\rightarrow N$, *then* $\sigma\tau(M) \xrightarrow{\boldsymbol{CAU^-}}\!\!\!\!\!\rightarrow \sigma\tau(N)$.

Although $\mathbf{CAU}^-_\sigma$, just like $\mathbf{CAU}^-$, is *not* confluent (different reduction strategies produce different trails, and trail inspection can be used to compute on them, yielding different terms as well), the previous results allow us to use Hardin's interpretation technique [18] to prove a *relativized* confluence theorem:

Theorem 4. *If* $M \xrightarrow{\boldsymbol{CAU^-_\sigma}}\!\!\!\!\!\rightarrow N$ *and* $M \xrightarrow{\boldsymbol{CAU^-_\sigma}}\!\!\!\!\!\rightarrow R$, *and furthermore* $\sigma\tau(N)$ *and* $\sigma\tau(R)$ *are joinable in* $\boldsymbol{CAU^-}$, *then* N *and* R *are joinable in* $\boldsymbol{CAU^-_\sigma}$.

Proof. See Fig. 3.

While the proof of Theorem 2 is not overly different from the similar proof for the $\lambda\sigma$-calculus, Theorem 3 is more interesting. The main challenge is to prove that whenever $M \xrightarrow{\text{Beta}} N$, we have $\sigma\tau(M) \xrightarrow{\mathbf{CAU}^-}\!\!\!\!\!\rightarrow \sigma\tau(N)$. However, when proceeding by induction on $M \xrightarrow{\text{Beta}} N$, the terms $\sigma\tau(M)$ and $\sigma\tau(N)$ are too normalized to provide us with a good enough induction hypothesis: in particular, we would want them to be in the form $q \triangleright R$ even when q is reflexivity. We call terms in this quasi-normal form *focused*, and prove the theorem by reasoning on them. The details of the proof are discussed in the extended version.

5 A Call-by-Value Abstract Machine

In this section, we derive an abstract machine implementing a weak call-by-value strategy. More precisely, the machine will consider subterms shaped like $q \triangleright \lfloor \overline{M}[e] \rfloor$, where $\overline{M}$ is a pure $\mathbf{CAU}^-$ term with no explicit operators, and e is an *environment*, i.e. an explicit substitution containing only values. In the tradition of lazy abstract machines, values are *closures* (typically pairing a lambda and an environment binding its free variables); in our case, the most natural notion of closure also involves trail erasures and bangs:

Closures	C	$::= \lfloor (\lambda \overline{M})[e] \rfloor \mid !_q C$
Values	V, W	$::= q \triangleright C$
Environments	e	$::= \langle\rangle \mid V \cdot e$

According to this definition, the most general case of closure is a telescope of bangs, each equipped with a complete history, terminated at the innermost level by a lambda abstraction applied to an environment and enclosed in an erasure.

$$!_{q_1} \cdots !_{q_n} \lfloor (\lambda \overline{M})[e] \rfloor$$

The environment e contains values with dangling trails, which may be captured by bangs contained in $\overline{M}$; however, the erasure makes sure that none of these trails may reach the external bangs; thus, along with giving meaning to free variables contained in lambdas, closures serve the additional purpose of making sure the history described by the $q_1, \ldots, q_n$ is complete for each bang.

The machine we describe is a variant of the SECD machine. To simplify the description, the code and environment are not separate elements of the machine state, but they are combined, together with a trail, as the top item of the stack. Another major difference is that a code κ can be not only a normal term without explicit operations, but also be a fragment of abstract syntax tree. The stack π is a list of tuples containing a trail, a code, and an environment, and represents the subterm currently being evaluated (the top of the stack) and the unevaluated context, i.e. subterms whose evaluation has been deferred (the remainder of the stack). As a pleasant side-effect of allowing fragments of the AST into the stack, we never need to set aside the current stack into the dump: D is just a list of values representing the evaluated context (i.e. the subterms whose evaluation has already been completed).

Codes	κ	$::= \overline{M} \mid @ \mid ! \mid \mathrm{let}_!(\overline{M}) \mid \iota$
Tuples	τ	$::= (q \mid \kappa \mid e)$
Stack	π	$::= \overrightarrow{\tau}$
Dumps	D	$::= \overrightarrow{V}$
Configurations	ς	$::= (\pi, D)$

The AST fragments allowed in codes include application nodes @, bang nodes !, incomplete let bindings $\mathrm{let}_!(\overline{M})$, and inspection nodes ι. A tuple $(q \mid \overline{M} \mid e)$ in which the code happens to be a term can be easily interpreted as $q \triangleright \lfloor \overline{M}[e] \rfloor$; however, tuples whose code is an AST fragment only make sense within a certain

machine state. The machine state is described by a configuration ς consisting of a stack and a dump.

$$(\epsilon, \epsilon)\ \mathbf{ctx} \qquad (\epsilon, V :: \epsilon)\ \mathbf{tm}$$

$$\frac{(\pi, D)\ \mathbf{ctx}}{((q|\overline{M}|e) :: (q'|@|\langle\rangle) :: \pi, D)\ \mathbf{ctx}} \qquad \frac{(\pi, D)\ \mathbf{ctx}}{((q|\overline{M}|e) :: \pi, D)\ \mathbf{tm}}$$

$$\frac{(\pi, D)\ \mathbf{ctx}}{((q|@|\langle\rangle) :: \pi, V :: D)\ \mathbf{ctx}} \qquad \frac{(\pi, D)\ \mathbf{ctx}}{((q|@|\langle\rangle) :: \pi, W :: V :: D)\ \mathbf{tm}}$$

$$\frac{(\pi, D)\ \mathbf{ctx}}{((q|\,\mathrm{let}_!(\overline{M})|e) :: \pi, D)\ \mathbf{ctx}} \qquad \frac{(\pi, D)\ \mathbf{ctx}}{((q|\,\mathrm{let}_!(\overline{M})|e) :: \pi, V :: D)\ \mathbf{tm}}$$

$$\frac{(\pi, D)\ \mathbf{ctx}}{((q|!|\langle\rangle) :: \pi, D)\ \mathbf{ctx}} \qquad \frac{(\pi, D)\ \mathbf{ctx}}{((q|!|\langle\rangle) :: \pi, V :: D)\ \mathbf{tm}}$$

$$\frac{(\pi, D)\ \mathbf{ctx}}{\left(\begin{array}{l}\overrightarrow{(q_i|\overline{M_i}|e_i)_{i=k+1,\ldots,9}} :: \\ \quad (q'|\iota|\langle\rangle) :: \pi, \\ \overrightarrow{V_{\{j=1,\ldots,k-1\}}} :: D\end{array}\right)\ \mathbf{ctx}} \qquad \frac{(\pi, D)\ \mathbf{ctx}}{((q|\iota|\langle\rangle) :: \pi, \overrightarrow{V_9} :: D)\ \mathbf{ctx}}$$

Fig. 4. Term and context configurations

A meaningful state cannot contain just any stack and dump, but must have a certain internal coherence, which we express by means of the two judgments in Fig. 4: in particular, the machine state must be a *term configuration*; this notion is defined by the judgment ς **tm**, which employs a separate notion of *context configuration*, described by the judgment ς **ctx**.

We can define the denotation of configurations by recursion on their well-formedness judgment:

Definition 3

1. *The denotation of a context configuration is defined as follows:*

$$\begin{aligned}
\underline{(\epsilon, \epsilon)} &\triangleq \blacksquare \\
\underline{((q|\overline{M}|e) :: (q'|@|\langle\rangle) :: \pi, D)} &\triangleq \underline{(\pi, D)}[q' \triangleright (\blacksquare\ (q \triangleright \lfloor \overline{M}[e] \rfloor))] \\
\underline{((q|@|\langle\rangle) :: \pi, V :: D)} &\triangleq \underline{(\pi, D)}[q \triangleright (V\ \blacksquare)] \\
\underline{((q|\,\mathrm{let}_!(\overline{M})|e) :: \pi, D)} &\triangleq \underline{(\pi, D)}[q \triangleright \mathrm{let}_!(\blacksquare, \lfloor \overline{M}[1 \cdot (e \circ \uparrow)] \rfloor)] \\
\underline{((q|!|\langle\rangle) :: \pi, D)} &\triangleq \underline{(\pi, D)}[q \triangleright\ !\blacksquare] \\
\underline{(\overrightarrow{(q_i|\overline{M_i}|e_i)} :: (q'|\iota|\langle\rangle) :: \pi, \overrightarrow{V_j} :: D)} &\triangleq \underline{(\pi, D)}[q' \triangleright \iota(\overrightarrow{V_j}, \blacksquare, \overrightarrow{(q_i \triangleright \lfloor \overline{M_i}[e_i] \rfloor)})]
\end{aligned}$$

where in the last line $i + j + 1 = 9$.

	source		↦ target	
1	$(q\|\overline{M}\ \overline{N}\|e) :: \pi$	D	$(\mathbf{r}\|\overline{M}\|e) :: (\mathbf{r}\|\overline{N}\|e) :: (q\|@\|\langle\rangle) :: \pi$	D
2	$(q\|@\|\langle\rangle) :: \pi$	$(q' \triangleright C) :: (q'' \triangleright \lfloor(\lambda\overline{M})[e]\rfloor) :: D$	$(q; \mathbf{app}(q'', q'); \boldsymbol{\beta}\|\overline{M}\|(\mathbf{r} \triangleright C) \cdot e) :: \pi$	D
3	$(q\|\lambda\overline{M}\|e) :: \pi$	D	π	$(q \triangleright \lfloor(\lambda\overline{M})[e]\rfloor) :: D$
4	$(q\|\,\mathrm{let}_!(\overline{M}, \overline{N})\|e) :: \pi$	D	$(\mathbf{r}\|\overline{M}\|e) :: (q\|\,\mathrm{let}_!(\overline{N})\|e) :: \pi$	D
5	$(q\|\,\mathrm{let}_!(\overline{N})\|e) :: \pi$	$(q' \triangleright !V) :: D$	$(q; \mathbf{let}_!(q', \mathbf{r}); \boldsymbol{\beta}_!; q_{\overline{N},e,V}\|\overline{N}\|V \cdot e) :: \pi$	D
6	$(q\|!_{q'}\overline{M}\|e) :: \pi$	D	$(q'; \lceil\overline{M}[e]\rceil\ \|\overline{M}\|e) :: (q\|!\|\langle\rangle) :: \pi$	D
7	$(q\|!\|\langle\rangle) :: \pi$	$V :: D$	π	$(q \triangleright !V) :: D$
8	$(q\|\iota(\overrightarrow{M_9})\|e) :: \pi$	D	$\overrightarrow{(\mathbf{r}\|\overline{M_i}\|e)_{i=1,\ldots,9}} :: (q\|\iota\|\langle\rangle) :: \pi$	D
9	$(q\|\iota\|\langle\rangle) :: \pi$	$\overrightarrow{(q_i \triangleright C_i)_{i=1,\ldots,9}} :: D$	$(q; \mathbf{tb}(\overrightarrow{q_i}); \mathbf{ti}\|J_{q,\overrightarrow{q_i},\pi,D}\|[(\mathbf{r} \triangleright \overrightarrow{C_i})]) :: \pi$	D
10	$(q\|n\|e) :: \pi$	D	π	$(q \triangleright e(n)) :: D$

$$q_{\overline{N},e,V} \triangleq \lceil\lfloor\overline{N}[1 \cdot (e \circ \uparrow)]\rfloor\,[V]\rceil$$
$$J_{q,\overrightarrow{q_i},\pi,D} \triangleq \mathcal{I}((q; \mathbf{tb}(\overrightarrow{q_i})), \pi, D)$$
$$e(n) \triangleq \begin{cases} C & \text{if } e = (q \triangleright C) \cdot e' \text{ and } n = 1 \\ e'(m) & \text{if } e = V \cdot e' \text{ and } n = m + 1 \end{cases}$$

Fig. 5. Call-by-value abstract machine

2. The denotation of a term configuration is defined as follows:

$$\begin{aligned}
\mathcal{T}(\epsilon, V :: \epsilon) &\triangleq V \\
\mathcal{T}((q|\overline{M}|e) :: \pi, D) &\triangleq \underline{(\pi, D)}[q \triangleright \lfloor\overline{M}[e]\rfloor] \\
\mathcal{T}((q|@|\langle\rangle) :: \pi, W :: V :: D) &\triangleq \underline{(\pi, D)}[q \triangleright (V\ W)] \\
\mathcal{T}((q|\,\mathrm{let}_!(\overline{M})|e) :: \pi, V :: D) &\triangleq \underline{(\pi, D)}[q \triangleright \mathrm{let}_!(V, \lfloor\overline{M}[1 \cdot (e \circ \uparrow)]\rfloor)] \\
\mathcal{T}((q|!|\langle\rangle) :: \pi, V :: D) &\triangleq \underline{(\pi, D)}[q \triangleright !V] \\
\mathcal{T}((q|\iota|\langle\rangle) :: \pi, \overrightarrow{V_9} :: D) &\triangleq \underline{(\pi, D)}[q \triangleright \iota(\overrightarrow{V_9})]
\end{aligned}$$

We see immediately that the denotation of a term configuration is a $\mathbf{CAU}^-_\sigma$ term, while that of a context configuration is a $\mathbf{CAU}^-_\sigma$ context (Definition 2).

The call-by-value abstract machine for $\mathbf{CAU}^-$ is shown in Fig. 5: in this definition we use semi-colons as a compact notation for sequences of transitivity trails. The evaluation of a pure, closed term $\overline{M}$, starts with an empty dump and a stack made of a single tuple $(\mathbf{r}, \overline{M}, \langle\rangle)$: this is a term configuration denoting $\mathbf{r} \triangleright \lfloor\overline{M}[\langle\rangle]\rfloor$, which is $\sigma\tau$-equivalent to $\overline{M}$. Final states are in the form $\epsilon, V :: \epsilon$, which simply denotes the value V. When evaluating certain erroneous terms (e.g. $(!\ M)\ V$, where function application is used on a term that is not a function), the machine may get stuck in a non-final state; these terms are rejected by the

$$\begin{aligned}
\mathcal{I}(q_\vartheta, (q'|!|\epsilon) :: \pi, D) &= \sigma\tau(q_\vartheta) \\
\mathcal{I}(q_\vartheta, (q'|\overline{M}|e) :: (q''|@|\epsilon) :: \pi, D) &= \mathcal{I}((q''; \mathbf{app}(q_\vartheta, q')), \pi, D) \\
\mathcal{I}(q_\vartheta, (q'|@|\langle\rangle) :: \pi, (q'' \triangleright C) :: D) &= \mathcal{I}((q', \mathbf{app}(q'', q_\vartheta)), \pi, D) \\
\mathcal{I}(q_\vartheta, (q'|\,\mathrm{let}_!(\overline{M})|e) :: \pi, D) &= \mathcal{I}((q'; \mathbf{let}_!(q_\vartheta, \mathbf{r})), \pi, D) \\
\mathcal{I}(q_\vartheta, \overrightarrow{(q_i|M_i|e_i)} :: (q'|\iota|\langle\rangle) :: \pi, \overrightarrow{(q_j \triangleright C_j)} :: D) &= \mathcal{I}((q'; \mathbf{tb}(\overrightarrow{q_j}, q_\vartheta, \overrightarrow{q_i})), \pi, D)
\end{aligned}$$

Fig. 6. Materialization of trails for inspection

typed **CAU**. The advantage of our machine, compared to a naive evaluation strategy, is that in our case all the principal reductions can be performed in constant time, except for trail inspection which must examine a full trail, and thus will always require a time proportional to the size of the trail.

Let us now examine the transition rules briefly. Rules 1–3 and 10 closely match the "Split CEK" machine [3] (a simplified presentation of the SECD machine), save for the use of the @ code to represent application nodes, while in the Split CEK machine they are expressed implicitly by the stack structure.

Rule 1 evaluates an application by decomposing it, placing two new tuples on the stack for the subterms, along with a third tuple for the application node; the topmost trail remains at the application node level, and two reflexivity trails are created for the subterms; the environment is propagated to the subterm tuples.

The idea is that when the machine reaches a state in which the term at the top of the stack is a value (e.g. a lambda abstraction, as in rule 3), the value is moved to the dump, and evaluation continues on the rest of the stack. Thus when in rule 2 we evaluate an application node, the dump will contain two items resulting from the evaluation of the two subterms of the application; for the application to be meaningful, the left-hand subterm must have evaluated to a term of the form $\lambda\overline{M}$, whereas the form of the right-hand subterm is not important: the evaluation will then continue as usual on $\overline{M}$ under an extended environment; the new trail will be obtained by combining the three trails from the application node and its subexpressions, followed by a $\boldsymbol{\beta}$ trail representing beta reduction.

The evaluation of $\mathrm{let}_!$ works similarly to that of applications; however, a term $\mathrm{let}_!(\overline{M}, \overline{N})$ is split intro $\overline{M}$ and $\mathrm{let}_!(\overline{N})$ (rule 4), so that $\overline{N}$ is never evaluated independently from the corresponding $\mathrm{let}_!$ node. When in rule 5 we evaluate the $\mathrm{let}_!(\overline{N})$ node, the dump will contain a value corresponding to the evaluation of $\overline{M}$ (which must have resulted in a value of the form $!V$): we then proceed to evaluate $\overline{N}$ in an environment extended with V; this step corresponds to a principal contraction, so we update the trail accordingly, by adding $\boldsymbol{\beta_!}$; additionally, we need to take into account the trails from V after substitution into $\overline{N}$: we do this by extending the trail with $\lceil \lfloor \overline{N}[1 \cdot (e\circ \uparrow)] \rfloor [V] \rceil$.

Bangs are managed by rules 6 and 7. To evaluate $!_{q'}\overline{M}$, we split it into $\overline{M}$ and a ! node, placing the corresponding tuples on top of the stack; the original external trail q remains with the ! node, whereas the internal trail q' is placed in the tuple with $\overline{M}$; the environment e is propagated to the body of the bang but, since it may contain trails, we need to extend q' with the trails resulting from substitution into $\overline{M}$. When in rule 7 we evaluate the ! node, the top of the dump contains the value V resulting from the evaluation of its body: we update the dump by combining V with the bang and proceed to evaluate the rest of the stack.

The evaluation of trail inspections (rules 8 and 9) follows the same principle as that of applications, with the obvious differences due to the fact that inspections have nine subterms. The principal contraction happens in rule 9, which assumes that the inspection branches have been evaluated to $q_1 \triangleright C_1, \ldots, q_9 \triangleright C_9$ and put

on the dump: at this point we have to reconstruct and normalize the inspection trail and apply the inspection branches. To reconstruct the inspection trail, we combine q and the $\overrightarrow{q_i}$ into the trail for the current subterm $(q; \mathbf{tb}(\overrightarrow{q_i}))$; then we must collect the trails in the context of the current bang, which are scattered in the stack and dump: this is performed by the auxiliary operator $\mathcal{I}$ of Fig. 6, defined by recursion on the well-formedness of the context configuration π, D; the definition is partial, as it lacks the case for ϵ, ϵ, corresponding to an inspection appearing outside all bangs: such terms are considered "inspection-locked" and cannot be reduced. Due to the operator $\mathcal{I}$, rule 9 is the only rule that cannot be performed in constant time.

$\mathcal{I}$ returns a $\sigma\tau$-normalized trail, which we need to apply to the branches $C_1, \ldots, C_9$; from the implementation point of view, this operation is analogous to a substitution replacing the trail nodes $(\mathbf{r}, \mathbf{t}, \boldsymbol{\beta}, \mathbf{app}, \mathbf{lam}, \ldots)$ with the respective M_i. Suppose that trails are represented as nested applications of dangling de Bruijn indices from 1 to 9 (e.g. the trail $\mathbf{app}(\mathbf{r}, \boldsymbol{\beta})$ can be represented as (1 2 3) for $\mathbf{app} = 1$, $\mathbf{r} = 2$ and $\boldsymbol{\beta} = 3$); then trail inspection reduction amounts to the evaluation of a trail in an environment composed of the trail inspection branches. To sum it up, rule 9 produces a state in which the current tuple contains:

- a trail $(q; \mathbf{tb}(\overrightarrow{q_i}); \mathbf{ti})$ (combining the trail of the inspection node, the trails of the branches, and the trail $\mathbf{ti}$ denoting trail inspection
- the $\sigma\tau$-reduced inspection "trail" (operationally, an open term with nine dangling indices) which results from $\mathcal{I}((q; \mathbf{tb}(\overrightarrow{q_i})), \pi, D)$
- an environment $[\overrightarrow{(\mathbf{r} \triangleright C_i)}]$ which implements trail inspection by substituting the inspection branches for the dangling indices in the trail.

The machine is completed by rule 10, which evaluates de Bruijn indices by looking them up in the environment. Notice that the lookup operation $e(n)$, defined when the de Bruijn index n is closed by the environment e, simply returns the n-th closure in e, but *not* the associated trail; the invariants of our machine ensure that this trail is considered elsewhere (particularly in rules 5 and 6).

The following theorem states that the machine correctly implements reduction.

Theorem 5. *For all valid* ς, $\varsigma \mapsto \varsigma'$ *implies* $\mathcal{T}(\varsigma) \xrightarrow{\boldsymbol{CAU}^-_\sigma} \mathcal{T}(\varsigma')$.

6 Conclusions and Future Directions

The calculus $\mathbf{CAU}^-_\sigma$ which we introduced in this paper provides a finer-grained view over the reduction of history-carrying terms, and proved an effective tool in the study of the smarter evaluation techniques which we implemented in an abstract machine. $\mathbf{CAU}^-_\sigma$ is not limited to the call-by-value strategy used by our machine, and in future work we plan to further our investigation of efficient auditing to call-by-name and call-by-need. Another intriguing direction we are exploring is to combine our approach with recent advances in explicit

substitutions, such as the linear substitution calculus of Accattoli and Kesner [5], and apply the *distillation* technique of Accattoli et al. [3]

In our discussion, we showed that the original definition of beta-reduction, when applied to terms that are not in trail-normal form, creates temporally unsound trails. We might wonder whether these anachronistic trails carry any meaning: let us take, as an example, the reduction on the left branch of Fig. 1:

$$(\lambda.M\ 1\ 1)\ (q \triangleright N) \longrightarrow\!\!\!\!\!\rightarrow \mathbf{t}(\boldsymbol{\beta}, \mathbf{app}(\mathbf{app}(\mathbf{r}, q), q)) \triangleright M\ N\ N$$

We know that q is the trace left behind by the reduction that led to N from the original term, say R:

$$R \longrightarrow q \triangleright N$$

We can see that the anachronistic trail is actually consistent with the reduction of $(\lambda.M\ 1\ 1)\ R$ under a leftmost-outermost strategy:

$$\begin{aligned}(\lambda.M\ 1\ 1)\ R &\longrightarrow \boldsymbol{\beta} \triangleright M\ R\ R \longrightarrow\!\!\!\!\!\rightarrow \boldsymbol{\beta} \triangleright M\ (q \triangleright N)\ (q \triangleright N) \\ &\longrightarrow\!\!\!\!\!\rightarrow \mathbf{t}(\boldsymbol{\beta}, \mathbf{app}(\mathbf{app}(\mathbf{r}, q), q)) \triangleright M\ N\ N\end{aligned}$$

Under the anachronistic reduction, q acts as the witness of an original inner redex. Through substitution within M, we get evidence that the contraction of an inner redex can be swapped with a subsequent head reduction: this is a key result in the proof of standardization that is usually obtained using the notion of *residual* ([13], Lemma 11.4.5). Based on this remark, we conjecture that trails might be used to provide a more insightful proof: it would thus be interesting to see how trails relate to recent advancements in standardization [4,11,20,28].

Acknowledgments. Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Cheney was also supported by ERC Consolidator Grant Skye (grant number 682315). We are grateful to James McKinna and the anonymous reviewers for comments.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. J. Funct. Program. **1**(4), 375–416 (1991). https://doi.org/10.1017/s0956796800000186
2. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of Network and Distributed System Security Symposium, NDSS 2003, San Diego, CA. The Internet Society (2003) http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/7.pdf
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: Proceedings of 19th ACM SIGPLAN Conference on Functional Programming, ICFP 2014, Gothenburg, September 2014, pp. 363–376. ACM Press, New York (2014). https://doi.org/10.1145/2628136.2628154

4. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: Proceedings of 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, January 2014, pp. 659–670. ACM Press, New York (2014). https://doi.org/10.1145/2535838.2535886
5. Accattoli, B., Kesner, D.: The structural λ-calculus. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 381–395. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15205-4_30
6. Amir-Mohammadian, S., Chong, S., Skalka, C.: Correct audit logging: theory and practice. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 139–162. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_8
7. Artemov, S.: Justification logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 1–4. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87803-2_1
8. Artemov, S.: The logic of justification. Rev. Symb. Log. **1**(4), 477–513 (2008). https://doi.org/10.1017/s1755020308090060
9. Artemov, S.N.: Explicit provability and constructive semantics. Bull. Symb. Log. **7**(1), 1–36 (2001). https://doi.org/10.2307/2687821
10. Artemov, S., Bonelli, E.: The intensional lambda calculus. In: Artemov, S.N., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 12–25. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72734-7_2
11. Asperti, A., Levy, J.J.: The cost of usage in the λ-calculus. In: Proceedings of 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, June 2013, pp. 293–300. IEEE CS Press, Washington, DC (2013). https://doi.org/10.1109/lics.2013.35
12. Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 27–48. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_2
13. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematic, vol. 103, 2nd edn. North-Holland, Amsterdam (1984). https://www.sciencedirect.com/science/bookseries/0049-237X/103
14. Bavera, F., Bonelli, E.: Justification logic and audited computation. J. Log. Comput. **28**(5), 909–934 (2018). https://doi.org/10.1093/logcom/exv037
15. de Bruijn, N.: Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. Indagationes Math. **34**(5), 381–392 (1972). https://doi.org/10.1016/1385-7258(72)90034-0
16. Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: theory, implementation and applications. In: Proceedings of 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, IL, October 2011, pp. 151–162. ACM Press, New York (2011). https://doi.org/10.1145/2046707.2046726
17. Giesl, J., et al.: Proving Termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_13
18. Hardin, T.: Confluence results for the pure strong categorical combinatory logic CCL: λ-calculi as subsystems of CCL. Theor. Comput. Sci. **65**(3), 291–342 (1989). https://doi.org/10.1016/0304-3975(89)90105-9

19. Jia, L., et al.: AURA: a programming language for authorization and audit. In: Proceedings of 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Victoria, BC, September 2008, pp. 27–38. ACM Press, New York (2008). https://doi.org/10.1145/1411204.1411212
20. Kashima, R.: A proof of the standardization theorem in lambda-calculus. Technical report, Research Reports on Mathematical and Computing Science, Tokyo Institute of Technology (2000)
21. Moreau, L.: The foundations for provenance on the web. Found. Trends Web Sci. **2**(2–3), 99–241 (2010). https://doi.org/10.1561/1800000010
22. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional programs that explain their work. In: Proceedings of 17th ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, September 2002, pp. 365–376. ACM Press, New York (2012). https://doi.org/10.1145/2364527.2364579
23. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Math. Struct. Comput. Sci. **11**(4), 511–540 (2001). https://doi.org/10.1017/s0960129501003322
24. Ricciotti, W.: A core calculus for provenance inspection. In: Proceedings of 19th International Symposium on Principles and Practice of Declarative Programming, PPDP 2017, Namur, October 2017, pp. 187–198. ACM Press, New York (2017). https://doi.org/10.1145/3131851.3131871
25. Ricciotti, W., Cheney, J.: Strongly normalizing audited computation. In: Goranko, V., Dam, M. (eds.) Proceedings of 26th EACSL Annual Conference, CSL 2017, Stockholm, August 2017. Leibniz International Proceedings in Informatics, vol. 82, Article no. 36. Schloss Dagstuhl Publishing, Saarbrücken/Wadern (2017). https://doi.org/10.4230/lipics.csl.2017.36
26. Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: Imperative functional programs that explain their work. Proc. ACM Program. Lang. **1**(ICFP), Article no. 14 (2017). https://doi.org/10.1145/3110258
27. Vaughan, J.A., Jia, L., Mazurak, K., Zdancewic, S.: Evidence-based audit. In: Proceedings of 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, PA, June 2008, pp. 177–191. IEEE CS Press, Washington, DC (2008). https://doi.org/10.1109/csf.2008.24
28. Xi, H.: Upper bounds for standardizations and an application. J. Symb. Log. **64**(1), 291–303 (1999). https://doi.org/10.2307/2586765