Generic Authenticated Data Structures, Formally

Matthias Brun

Department of Computer Science, ETH Zürich, Switzerland
mbrun@student.ethz.ch

Dmitriy Traytel

Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
traytel@inf.ethz.ch

Abstract

Authenticated data structures are a technique for outsourcing data storage and maintenance to an untrusted server. The server is required to produce an efficiently checkable and cryptographically secure proof that it carried out precisely the requested computation. Recently, Miller et al. [10] demonstrated how to support a wide range of such data structures by integrating an authentication construct as a first class citizen in a functional programming language. In this paper, we put this work to the test of formalization in the Isabelle proof assistant. With Isabelle's help, we uncover and repair several mistakes and modify the small-step semantics to perform call-by-value evaluation rather than requiring terms to be in administrative normal form.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Authenticated Data Structures, Verifiable Computation, Isabelle/HOL, Nominal Isabelle

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.10

Supplement Material `https://isa-afp.org/entries/LambdaAuth.html`

Acknowledgements We thank David Basin for supporting this work and Andrew Miller for discussing our counterexamples and proposing a remedy to the issue with type soundness. Joshua Schneider and the anonymous ITP reviewers helped to improve the presentation through numerous comments.

1 Introduction

Consider a client that requests data from a server and trusts the server to answer its request truthfully, making financial or security-critical decisions based on the response. In this common scenario, a malicious actor can profit from causing the server to give incorrect answers to a client's query. Authenticated data structures (ADS) prevent this attack by effectively removing the need for the client to trust the server. To do so, they require the server to accompany all responses to queries with an efficiently verifiable proof that its answer is honest.

Merkle trees [9] are the prototypical example of ADS. They are binary trees that store data in their leaves. Every leaf node is augmented with a hash of the corresponding data and every inner node is augmented with a hash of its child nodes' hashes. An example Merkle tree is shown in Figure 1. The server stores this entire tree, whereas the client only stores the top hash H_0. The client can then query the server for any of the stored data. The server, upon being queried, traverses the tree to find the requested data and returns it along with the hashes needed to reconstruct the root hash. The client can then recompute the root hash to verify that it matches its stored root hash. In our example, querying the server for D_2 would result in it returning D_2 as well as the hashes HD_1 and H_2. The client can then verify that the result of $\mathsf{hash}\ (\mathsf{hash}\ (HD_1 \parallel \mathsf{hash}\ D_2) \parallel H_2)$ matches its stored root hash.

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 10; pp. 10:1–10:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

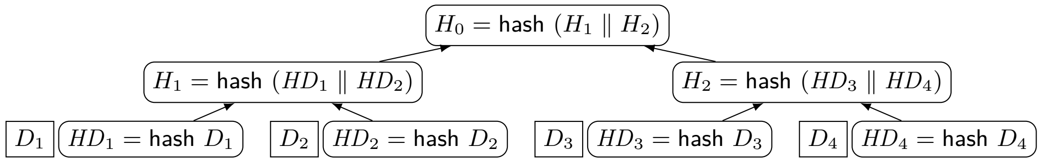

Figure 1 An example Merkle tree.

Early work on ADS [5, 9, 16] has focused on designing particular data structures for this purpose. More recently, Miller et al. [10] have put forward a more general view on the matter. In their paper, titled *Authenticated Data Structures, Generically* (*ADSG*), they introduce λ• (pronounced *lambda auth*), a purely functional language, which supports generic, user-specified ADS. The programs of λ• run in two modes. The server, which hosts the data, computes certain hash values and sends them to the client. The client verifies that the passed hash values are the expected ones. ADSG establishes correctness (verification succeeds if both parties correctly follow the protocol) and security (tricking the client requires discovering a hash collision) for all well-typed λ• programs. Given that ADS are intended to be used in security-critical applications, it is crucial that these correctness and security properties do in fact hold.

We formalized λ• in Isabelle/HOL and proved the claims stated in *ADSG*. During the formalization process, we identified several problems, many of which we rectified with relative ease. Nevertheless, a serious problem prevents us from reaching a fully satisfactory statement and proof of the conventional formulation of λ•'s type soundness.

In addition to finding and correcting mistakes, we also make a modification to the language semantics. "To keep the semantics simple," *ADSG* works with expressions in administrative normal form (ANF) [6]. ANF only supports recursive evaluation in arguments of let expressions and thus requires all other constructs to be applied to values (rather than unevaluated expressions). While this does not make the language any less powerful, the restrictive syntax makes λ• somewhat cumbersome to use, e.g., instead of writing $t\ u$ for expressions t and u one has to write $\mathsf{let}\ f = t\ \mathsf{in}\ \mathsf{let}\ x = u\ \mathsf{in}\ f\ x$. To hide this verbosity from the user, arbitrary expressions are typically translated into ANF in a separate step. However, such a translation would need to correctly handle λ•'s authentication construct. Instead, we extended the semantics to permit recursive argument evaluation for most expressions. We have performed this modification only after finishing the formalization of λ• and proving all the theorems for the ANF semantics. Isabelle allowed us to quickly discover all the ramifications of our changes. Thus, correcting the proofs that were affected by the modification was a matter of a few hours. In the following, we present only the modified semantics that supports recursive evaluation.

On the technical side, we used Nominal Isabelle [8, 17] (Section 2) to model the syntax and semantics of λ• (Section 3), which involves several variable binding constructs. Of particular interest is our abstract modeling of a hash function that is compatible with Nominal and can be used in binding-aware definitions (Subsection 3.1). The small-step semantics of λ• is split into three transition relations that correspond to the client's, the server's, and an idealized view of the computation, respectively. Following *ADSG*, we relate programs evaluated under these three views using an inductive predicate (Section 4) and prove that if one of the related programs takes a step, the others can follow, unless a hash collision occurred (Section 5).

Related Work. *ADSG* [10] is our object of study. While our paper aspires to be self-contained with respect to the scope of the formalization, we refer to *ADSG* for the illuminating usages of the λ• language to implement Merkle trees, blockchains, and authenticated red-black trees.

The literature on formal studies of authenticated data structures is sparse, and in all cases focused on specific instances. Examples include the automatic verification of Merkle trees using weak monadic second-order logic on trees [12] and the formalization of blockchains [15] and cryptographic ledgers [20] (based on Merkle trees) in the Coq proof assistant. The two latter works both assume injective hash functions, which we avoid (Subsection 3.1).

A key feature of our formalization is the use of Nominal Isabelle [8, 17, 19], Isabelle's implementation of Nominal logic [7] on top of higher-order logic, to model a syntax involving binding of variables. More precisely, we use Nominal2 [8, 17], the most recent implementation of Nominal Isabelle, which has previously been employed successfully in formalizations of Gödel's incompleteness theorems [13], lazy programming language semantics [3], and rewriting [11].

A frequently used alternative to the Nominal approach of modeling bound variables are de Bruijn indices, i.e., nameless pointers to binding constructors. We chose Nominal because it allows us to work more abstractly, without the need to manipulate pointers. We refer to Urban and Berghofer [18] for a comparison of the two approaches and to Blanchette et al. [2] for an extensive overview of the issue of binding variables in proof assistants and beyond.

2 Nominal Isabelle

The treatment of bound variables in pen and paper proofs is often informal, with renaming of clashing variables being implicitly assumed for most definitions. *ADSG* is no exception in this regard. In a formalization, a more rigorous approach is necessary. *Nominal Logic* [7] is a powerful such approach that is well-supported in Isabelle with the *Nominal* framework [8, 17]. We sketch the most important features of Nominal and refer to Huffman and Urban [8] for a more extensive introduction.

Nominal allows us to closely follow the informal presentation of *ADSG* in the formalization by enforcing the Barendregt convention [1, p. 26]:

> If $M_1, \dots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

A central notion for achieving this flexibility is that of an object's support supp, which corresponds to the set of *atoms* (i.e., variable names) that occur free in it. An atom a outside of the support of x is *fresh* in x, written $a \sharp x \equiv a \notin \mathsf{supp}\ x$. We will use two kinds of atoms: type variables *tvar* and term variables *var*, which are embedded into the type of atoms using the overloaded function atom. We will often see statements of the kind $\mathsf{atom}\ a \sharp x$ in the premises of our definitions, making explicit the requirement that some (type) variable name a does not clash with any of the ones in x. These additional freshness assumptions are typically the only required modifications to an informal lemma's statement.

Nominal Isabelle provides commands for defining binding-aware datatypes, recursive functions, and inductive predicates, along with a proof method for performing binding-aware structural induction. The syntax of λ• (types *ty* and terms *term*), shown in Figure 2, is defined via the **nominal_datatype** command, which requires us to explicitly specify which names are bound in which constructors. For λ•'s terms these are Lam, Rec, and Let, which model lambda abstractions (i.e., $\lambda x.\ t$ is written as $\mathsf{Lam}\ x\ t$), recursive functions, and let expressions, respectively, as well as Mu for recursive types. To define functions on a Nominal datatype we use the **nominal_function** command. The syntax for Nominal function definitions is the same as for normal functions except that freshness assumptions may be added when operating on datatype constructors that bind variables. For example,

nominal_datatype *term* =
  Unit
| Var *var*
| Lam (x :: *var*) (t :: *term*) **binds** x **in** t
| Rec (x :: *var*) (t :: *term*) **binds** x **in** t
| Inj1 *term*
| Inj2 *term*
| Pair *term term*
| Let *term* (x :: *var*) (t :: *term*) **binds** x **in** t
| App *term term*
| Case *term term term*
| Prj1 *term*
| Prj2 *term*
| Roll *term*
| Unroll *term*
| Auth *term*
| Unauth *term*
| Hash *hash*
| Hashed *hash term*

nominal_datatype *ty* =
  One
| Fun *ty ty*
| Sum *ty ty*
| Prod *ty ty*
| Mu (α :: *tvar*) (τ :: *ty*) **binds** α **in** τ
| Alpha *tvar*
| AuthT *ty*

inductive *value* :: *term* ⇒ *bool* **where**
  value Unit
| value (Var x)
| value (Lam x e)
| value (Rec x e)
| value v ⟶ value (Inj1 v)
| value v ⟶ value (Inj2 v)
| value v_1 ∧ value v_2 ⟶ value (Pair v_1 v_2)
| value v ⟶ value (Roll v)
| value (Hash h)
| value v ⟶ value (Hashed h v)

Figure 2 Syntax for terms and types.

the Lam case of the definition for capture-avoiding substitution, written $t[t'/x]$ and read as "in t substitute t' for x," is the following.

$$\text{atom } y \sharp (x, t') \longrightarrow (\text{Lam } y\ t)[t'/x] = \text{Lam } y\ (t[t'/x])$$

Definitions of inductive predicates use similar premises, as can be seen for example in our typing judgment's Lam rule in Figure 4. To enable binding-aware proofs by rule induction, Nominal can be instructed to prove a strong induction rule (after the user discharges a simpler abstract property, which is automatic for most definitions). The strong induction rule guarantees the absence of name clashes with a finite but arbitrary set of atoms.

Nominal is designed to support user-defined types as long as all objects have finite support. A particularly useful type for us will be that of finite maps, written (α, β) *fmap*, to model type environments and parallel substitutions. Finite maps are defined as the subtype of functions $\alpha \Rightarrow \beta$ *option* that map all but finitely many arguments to None. Other formalizations use association lists to represent type environments [18]. However, to ensure that any key in the list occurs at most once these require a validity predicate, cluttering the rules and proofs with implementation details. Finite maps nicely complemented our use of Nominal and allowed us to keep the statements of definitions and lemmas very close to those in *ADSG*. We use the syntax $\varnothing$ to denote the empty finite map, $\Gamma[x]$ to denote a lookup of x in the finite map Γ and $\Gamma[x \mapsto a]$ to denote an update to the finite map Γ, assigning a to x.

3 Syntax and Semantics of λ•

We formalize the terms and types for λ• as Nominal datatypes, along with an inductive predicate specifying which terms are considered to be values. These are listed in Figure 2. The terms and types are those of a standard lambda calculus with unit (One), product (Prod), sum (Sum), and recursive types (Mu), and the corresponding term constructors (e.g.,

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{nominal\_function}\ \mathit{shallow}\ ::\ \mathit{term} \Rightarrow \mathit{term}\ (\llparenthesis\_\rrparenthesis)\ \textbf{where}} \\
\ \ \llparenthesis\mathsf{Unit}\rrparenthesis = \mathsf{Unit} & & \mid \llparenthesis\mathsf{Var}\ v\rrparenthesis = \mathsf{Var}\ v & \\
\mid \llparenthesis\mathsf{Lam}\ x\ e\rrparenthesis = \mathsf{Lam}\ x\ \llparenthesis e\rrparenthesis & & \mid \llparenthesis\mathsf{Rec}\ x\ e\rrparenthesis = \mathsf{Rec}\ x\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Inj1}\ e\rrparenthesis = \mathsf{Inj1}\ \llparenthesis e\rrparenthesis & & \mid \llparenthesis\mathsf{Inj2}\ e\rrparenthesis = \mathsf{Inj2}\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Pair}\ e_1\ e_2\rrparenthesis = \mathsf{Pair}\ \llparenthesis e_1\rrparenthesis\ \llparenthesis e_2\rrparenthesis & & \mid \llparenthesis\mathsf{Roll}\ e\rrparenthesis = \mathsf{Roll}\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Let}\ e_1\ x\ e_2\rrparenthesis = \mathsf{Let}\ \llparenthesis e_1\rrparenthesis\ x\ \llparenthesis e_2\rrparenthesis & & \mid \llparenthesis\mathsf{App}\ e_1\ e_2\rrparenthesis = \mathsf{App}\ \llparenthesis e_1\rrparenthesis\ \llparenthesis e_2\rrparenthesis & \\
\mid \llparenthesis\mathsf{Case}\ e\ e_1\ e_2\rrparenthesis = \mathsf{Case}\ \llparenthesis e\rrparenthesis\ \llparenthesis e_1\rrparenthesis\ \llparenthesis e_2\rrparenthesis & & \mid \llparenthesis\mathsf{Prj1}\ e\rrparenthesis = \mathsf{Prj1}\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Prj2}\ e\rrparenthesis = \mathsf{Prj2}\ \llparenthesis e\rrparenthesis & & \mid \llparenthesis\mathsf{Unroll}\ e\rrparenthesis = \mathsf{Unroll}\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Auth}\ e\rrparenthesis = \mathsf{Auth}\ \llparenthesis e\rrparenthesis & & \mid \llparenthesis\mathsf{Unauth}\ e\rrparenthesis = \mathsf{Unauth}\ \llparenthesis e\rrparenthesis & \\
\mid \llparenthesis\mathsf{Hash}\ h\rrparenthesis = \mathsf{Hash}\ h & & \mid \llparenthesis\mathsf{Hashed}\ h\ e\rrparenthesis = \mathsf{Hash}\ h &
\end{array}
$$

Figure 3 The shallow projection.

Roll, the constructor of recursive types) and their inverses (e.g., Unroll, the destructor of recursive types) [14]. They also include the non-standard AuthT type constructor, Auth and Unauth term constructors, and auxiliary constructors Hashed consisting of a hash-value pair and Hash consisting of just a hash. We postpone the discussion of hash values and the type *hash* and introduce a few auxiliary functions first. Also the precise meaning of the Auth and Unauth constructors will become clear once we formally define the small-step semantics. Intuitively, Auth signals the client and server to compute a hash value, while Unauth signals the server to output a value to the client and the client to verify the hash of this value.

Substitution on terms and on types uses the syntax $t[u/x]$ for both. The definitions are standard, with simple, structural recursion on the non-standard constructs:

$$
\begin{array}{llll}
(\mathsf{Auth}\ t)[u/x] = \mathsf{Auth}\ (t[u/x]) & & (\mathsf{Unauth}\ t)[u/x] = \mathsf{Unauth}\ (t[u/x]) \\
(\mathsf{Hash}\ h)[u/x] = \mathsf{Hash}\ h & & (\mathsf{Hashed}\ h\ t)[u/x] = \mathsf{Hashed}\ h\ (t[u/x])
\end{array}
$$

Furthermore, we define a parallel substitution function $\mathsf{psubst} :: \mathit{term} \Rightarrow (\mathit{var}, \mathit{term})\ \mathit{fmap} \Rightarrow \mathit{term}$. It replaces all variables by terms assigned by the finite map given as its second argument:

$$\mathsf{psubst}\ (\mathsf{Var}\ y)\ \Delta = (\textbf{case}\ \Delta[y]\ \textbf{of}\ \mathsf{Some}\ t \Rightarrow t \mid \mathsf{None} \Rightarrow \mathsf{Var}\ y)$$

For all other cases it is structurally recursive.

A *closed* term is one with empty support or, equivalently, $\mathsf{closed}\ t = (\forall x :: \mathit{var}.\ \mathsf{atom}\ x \sharp\ t)$.

ADSG also introduces the *shallow projection* function, written $\llparenthesis\_\rrparenthesis$, whose formal definition is given in Figure 3. It replaces all Hashed h v subterms in a given term with Hash h.

3.1 Modeling the Hash Function

The security of λ● relies on a collision-resistant hash function. *ADSG* provides a useful modeling trick, which permits us to omit the formalization of this assumption or collision-resistance in general. In our formalization, we use very mild assumptions on how the hash function may behave. Our security statement is then a disjunction between the statements "everything worked out as planned" and "a hash collision has occurred." Clearly, if we use a collision-resistant hash function, the second disjunct will be violated with high probability. (This meta-argument is not captured in our formal modeling.)

We start by introducing a new type: **typedecl** *hash*. The only property we require of this type is that it does not contain any atoms, which we obtain by instantiating the *pure* type class. Doing so allows us to make use of the following lemma with $\alpha = \mathit{hash}$.

▶ **Lemma 1** (No atoms occur in pure types).

$$\mathsf{atom}\ x \sharp\ (t :: \alpha :: \mathit{pure})$$

Because our desired hash function hash :: *term* ⇒ *hash* will be used in inductive predicates involving the *term* type, such as the small-step semantics, Nominal requires it to be *equivariant*, i.e., satisfy the strong property $\forall p.\ p \bullet \mathsf{hash}\ t = \mathsf{hash}\ (p \bullet t)$ for all terms t. Here, p is a permutation, i.e., a variable renaming, and $\bullet$ denotes its application to a an arbitrary object. (The application to the object's variables is defined by instantiating a type class, which is automatic for Nominal datatypes.) Since a hash contains no free variables, applying a permutation to it is the identity function. Clearly then, equivariance can *only* hold if permuting free variables does not change the hash – a counterintuitive requirement for a hash function, which we want to avoid.

For closed terms t the above property holds for any function hash. Moreover, it turns out that we will only apply hash to closed terms. Nominal, however, is blind to this fact and still requires us to prove equivariance for all terms. These two observations lead to the following solution. We declare a hash function using Isabelle's **consts** command, which introduces a new constant symbol without providing any specification of the constant beyond its type.

consts hash_term :: *term* ⇒ *hash*

This function is not necessarily equivariant. (We can neither prove nor disprove this.) Equivariance is established by composing hash_term with the function collapse_frees :: *term* ⇒ *term*, which maps all free variables of a term to a single fixed variable (definition omitted).

definition hash :: *term* ⇒ *hash* **where** hash = hash_term ∘ collapse_frees

The function hash is equivariant ($\forall p.\ p \bullet \mathsf{hash}\ t = \mathsf{hash}\ (p \bullet t)$) and equal to hash_term on closed terms ($\mathsf{closed}\ t \longrightarrow \mathsf{hash}\ t = \mathsf{hash\_term}\ t$), because $\mathsf{collapse\_frees}\ t = t$ on closed terms t. Whenever we make use of the hash function hash, we ensure that its argument is closed.

3.2 Typing Judgement

The typing judgment $\Gamma \vdash e : \tau$, read "given the type environment $\Gamma :: (var,\ ty)\ fmap$ the term e is well-typed and has type τ," for $\lambda\bullet$ is defined in Figure 4. The rules are standard except for the last two, which allow the introduction and elimination of authenticated types $\mathsf{AuthT}\ \tau$ via the Auth and Unauth constructors. In other words, these two rules fix the following types for the authentication constructors: $\mathsf{Auth} :: \tau \Rightarrow \mathsf{AuthT}\ \tau$ and $\mathsf{Unauth} :: \mathsf{AuthT}\ \tau \Rightarrow \tau$.

In addition to this typing judgment, we define an alternative, weaker typing judgment $\Gamma \vdash_W e : \tau$, which is not present in *ADSG*. This version replaces the last two rules with the ones in Figure 5, which do not introduce authenticated types, i.e., fixing $\mathsf{Auth} :: \tau \Rightarrow \tau$ and $\mathsf{Unauth} :: \tau \Rightarrow \tau$. This modification is motivated by an ambiguity in *ADSG*, which we will encounter when discussing type soundness. We use the unqualified *well-typed* to mean well-typed according to the original typing judgment and *weakly well-typed* to mean well-typed according to the modified rules.

Neither well-typed nor weakly well-typed terms may contain the Hashed and Hash term constructors, as there is no rule for them. These auxiliary constructors will arise only as the result of some computations and are not meant to be used as a language construct by the end-users of $\lambda\bullet$. Thus, the use of these constructors loosely resembles the use of memory locations as an auxiliary language construct in lambda calculi with references [14, Chapter 13].

$$\frac{}{\Gamma \vdash \mathsf{Unit} : \mathsf{One}} \qquad \frac{\Gamma[x] = \mathsf{Some}\ \tau}{\Gamma \vdash \mathsf{Var}\ x : \tau} \qquad \frac{\mathsf{atom}\ x \sharp \Gamma \qquad \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \mathsf{Lam}\ x\ e : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp (\Gamma, e_1) \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Let}\ e_1\ x\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e : \mathsf{Fun}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e' : \tau_1}{\Gamma \vdash \mathsf{App}\ e\ e' : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp \Gamma \qquad \mathsf{atom}\ y \sharp (\Gamma, x) \qquad \Gamma[x \mapsto \mathsf{Fun}\ \tau_1\ \tau_2] \vdash \mathsf{Lam}\ y\ e : \mathsf{Fun}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e) : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{Inj1}\ e : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{Inj2}\ e : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj1}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj2}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e : \mathsf{Sum}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_1 : \mathsf{Fun}\ \tau_1\ \tau \qquad \Gamma \vdash e_2 : \mathsf{Fun}\ \tau_2\ \tau}{\Gamma \vdash \mathsf{Case}\ e\ e_1\ e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Pair}\ e_1\ e_2 : \mathsf{Prod}\ \tau_1\ \tau_2}$$

$$\frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}{\Gamma \vdash \mathsf{Roll}\ e : \mathsf{Mu}\ \alpha\ \tau} \qquad \frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e : \mathsf{Mu}\ \alpha\ \tau}{\Gamma \vdash \mathsf{Unroll}\ e : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{Auth}\ e : \mathsf{AuthT}\ \tau} \qquad \frac{\Gamma \vdash e : \mathsf{AuthT}\ \tau}{\Gamma \vdash \mathsf{Unauth}\ e : \tau}$$

Figure 4 The typing judgment.

$$\frac{\Gamma \vdash_W e : \tau}{\Gamma \vdash_W \mathsf{Auth}\ e : \tau} \qquad \frac{\Gamma \vdash_W e : \tau}{\Gamma \vdash_W \mathsf{Unauth}\ e : \tau}$$

Figure 5 Alternative, weaker typing rules.

3.3 Operational Small-Step Semantics

Figure 6 defines the small-step semantics as the inductive predicate $\langle \pi_1, e_1 \rangle\ m{\rightarrow}\ \langle \pi_2, e_2 \rangle$, meaning "the expression e_1 in combination with the *proof stream* π_1 can take a step in *mode* m to yield the expression e_2 and the proof stream π_2." A proof stream is simply a list of $\lambda\bullet$-expressions; the infix operator @ appends lists. The mode, which is a parameter of the semantics, can be one of three values:

datatype *mode* = I | P | V

The three modes I, P, and V are read as *ideal*, *prover*, and *verifier*, respectively. The ideal mode represents the unauthenticated evaluation. The authenticated evaluation proceeds with the prover mode running on the server, while the verifier mode runs on the client. Most rules are those of a standard lambda-calculus; they are shared for all three modes. Only the last six rules of $\langle \pi_1, e_1 \rangle\ m{\rightarrow}\ \langle \pi_2, e_2 \rangle$ for Auth and Unauth depend on the mode.

In the ideal mode, Auth and Unauth are simply removed, i.e., semantically they are identity functions. Upon encountering Auth v, both the prover and the verifier compute the hash of v's shallow projection. The prover uses the hash to generate the hash-value-pair Hashed (hash $(\!|v|\!)$) v, wheras the verifier generates just the hash Hash (hash v). The rules thus enforce that the Hashed constructor only ever arises in the prover mode and the Hash constructor only in the verifier mode. Thus, the shallow projection can be omitted for the verifier. The Unauth rules are the most interesting ones, as they establish the communication of the prover and the verifier via the proof stream. Unauth can only ever be applied to

$$\frac{\langle \pi,\ e_1\rangle\ m\!\rightarrow \langle \pi',\ e_1'\rangle}{\langle \pi,\ \mathsf{App}\ e_1\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{App}\ e_1'\ e_2\rangle} \qquad \frac{\mathsf{value}\ v_1 \quad \langle \pi,\ e_2\rangle\ m\!\rightarrow \langle \pi',\ e_2'\rangle}{\langle \pi,\ \mathsf{App}\ v_1\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{App}\ v_1\ e_2'\rangle}$$

$$\frac{\mathsf{value}\ v \quad \mathsf{atom}\ x \sharp (v, \pi)}{\langle \pi,\ \mathsf{App}\ (\mathsf{Lam}\ x\ e)\ v\rangle\ m\!\rightarrow \langle \pi,\ e[v/x]\rangle} \qquad \frac{\mathsf{value}\ v \quad \mathsf{atom}\ x \sharp (v, \pi) \quad e' = e[\mathsf{Rec}\ x\ e/x]}{\langle \pi,\ \mathsf{App}\ (\mathsf{Rec}\ x\ e)\ v\rangle\ m\!\rightarrow \langle \pi,\ \mathsf{App}\ e'\ v\rangle}$$

$$\frac{\mathsf{value}\ v \quad \mathsf{atom}\ x \sharp (v, \pi)}{\langle \pi,\ \mathsf{Let}\ v\ x\ e\rangle\ m\!\rightarrow \langle \pi,\ e[v/x]\rangle} \qquad \frac{\mathsf{atom}\ x \sharp (e_1, e_1', \pi, \pi') \quad \langle \pi,\ e_1\rangle\ m\!\rightarrow \langle \pi',\ e_1'\rangle}{\langle \pi,\ \mathsf{Let}\ e_1\ x\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Let}\ e_1'\ x\ e_2\rangle}$$

$$\frac{\langle \pi,\ e_1\rangle\ m\!\rightarrow \langle \pi',\ e_1'\rangle}{\langle \pi,\ \mathsf{Pair}\ e_1\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Pair}\ e_1'\ e_2\rangle} \qquad \frac{\mathsf{value}\ v_1 \quad \langle \pi,\ e_2\rangle\ m\!\rightarrow \langle \pi',\ e_2'\rangle}{\langle \pi,\ \mathsf{Pair}\ v_1\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Pair}\ v_1\ e_2'\rangle}$$

$$\frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Prj1}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Prj1}\ e'\rangle} \qquad \frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Prj2}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Prj2}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v_1 \quad \mathsf{value}\ v_2}{\langle \pi,\ \mathsf{Prj1}\ (\mathsf{Pair}\ v_1\ v_2)\rangle\ m\!\rightarrow \langle \pi,\ v_1\rangle} \qquad \frac{\mathsf{value}\ v_1 \quad \mathsf{value}\ v_2}{\langle \pi,\ \mathsf{Prj2}\ (\mathsf{Pair}\ v_1\ v_2)\rangle\ m\!\rightarrow \langle \pi,\ v_2\rangle}$$

$$\frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Inj1}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Inj1}\ e'\rangle} \qquad \frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Inj2}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Inj2}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Case}\ (\mathsf{Inj1}\ v)\ e_1\ e_2\rangle\ m\!\rightarrow \langle \pi,\ \mathsf{App}\ e_1\ v\rangle} \qquad \frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Case}\ (\mathsf{Inj2}\ v)\ e_1\ e_2\rangle\ m\!\rightarrow \langle \pi,\ \mathsf{App}\ e_2\ v\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Unroll}\ (\mathsf{Roll}\ v)\rangle\ m\!\rightarrow \langle \pi,\ v\rangle} \qquad \frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Case}\ e\ e_1\ e_2\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Case}\ e'\ e_1\ e_2\rangle}$$

$$\frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Unroll}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Unroll}\ e'\rangle} \qquad \frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Roll}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Roll}\ e'\rangle}$$

$$\frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Auth}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Auth}\ e'\rangle} \qquad \frac{\langle \pi,\ e\rangle\ m\!\rightarrow \langle \pi',\ e'\rangle}{\langle \pi,\ \mathsf{Unauth}\ e\rangle\ m\!\rightarrow \langle \pi',\ \mathsf{Unauth}\ e'\rangle}$$

$$\frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Auth}\ v\rangle\ \mathsf{I}\!\rightarrow \langle \pi,\ v\rangle} \qquad \frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Unauth}\ v\rangle\ \mathsf{I}\!\rightarrow \langle \pi,\ v\rangle}$$

$$\frac{\mathsf{closed}\ (\!|v|\!) \quad \mathsf{value}\ v}{\langle \pi,\ \mathsf{Auth}\ v\rangle\ \mathsf{P}\!\rightarrow \langle \pi,\ \mathsf{Hashed}\ (\mathsf{hash}\ (\!|v|\!))\ v\rangle} \qquad \frac{\mathsf{value}\ v}{\langle \pi,\ \mathsf{Unauth}\ (\mathsf{Hashed}\ h\ v)\rangle\ \mathsf{P}\!\rightarrow \langle \pi\ @\ [(\!|v|\!)],\ v\rangle}$$

$$\frac{\mathsf{closed}\ v \quad \mathsf{value}\ v}{\langle \pi,\ \mathsf{Auth}\ v\rangle\ \mathsf{V}\!\rightarrow \langle \pi,\ \mathsf{Hash}\ (\mathsf{hash}\ v)\rangle} \qquad \frac{\mathsf{closed}\ s_0 \quad \mathsf{hash}\ s_0 = h}{\langle s_0 \# \pi,\ \mathsf{Unauth}\ (\mathsf{Hash}\ h)\rangle\ \mathsf{V}\!\rightarrow \langle \pi,\ s_0\rangle}$$

$$\frac{}{\langle \pi,\ e\rangle\ m\!\rightarrow_0 \langle \pi,\ e\rangle}$$

$$\frac{\langle \pi_1,\ e_1\rangle\ m\!\rightarrow_i \langle \pi_2,\ e_2\rangle \quad \langle \pi_2,\ e_2\rangle\ m\!\rightarrow \langle \pi_3,\ e_3\rangle}{\langle \pi_1,\ e_1\rangle\ m\!\rightarrow_{i+1} \langle \pi_3,\ e_3\rangle}$$

Figure 6 The small-step semantics of $\lambda\bullet$.

expressions of type AuthT. Values of this type are always Hashed h v' and Hash h (for some h, v') in the prover and verifier modes, respectively. The prover appends the shallow projection of v' to the proof stream and continues to evaluate v'. The shallow projection ensures that any hash-value pairs within v' discard the value, keeping just the hash. The verifier consumes the first element of its input proof stream to verify that this value's hash is equal to the hash of its argument. Only if the check succeeds, the evaluation may proceed.

The rules demonstrate that the evaluation in all three modes is structurally identical but a compiler would have to substitute a different function for the Auth and Unauth functions for the prover and verifier modes. In this semantics any given expression can first be executed in mode P by the prover, generating a proof stream, and then in mode V by the verifier, consuming a proof stream. The execution in mode I does not modify or depend on the proof stream at all. The last two rules lift the single-step evaluation to multiple steps, while at the same time counting the number of taken steps.

The three Auth and Unauth rules that require hash computation all have a premise that ensures that hashes are only computed on closed terms. The small-step semantics given in *ADSG* is not restricted in this way. But the restriction is unproblematic: even though our semantics allows the prover and the verifier to evaluate strictly fewer expressions, we will show later that they can still simulate any ideal computation that starts with a closed formula.

Above, we have stated informally that the prover generates the proof stream and the verifier consumes the proof stream. We can formalize this notion in the following two lemmas that will be necessary for the correctness and security proofs.

▶ **Lemma 2** (Execution in mode P generates the proof stream).

$$\langle \pi_1,\ e_P \rangle\ \mathsf{P}{\rightarrow}_i\ \langle \pi_2,\ e'_P \rangle \longrightarrow \exists \pi.\ \pi_2 = \pi_1\ @\ \pi$$

▶ **Lemma 3** (Execution in mode V consumes the proof stream).

$$\langle \pi_1,\ e_V \rangle\ \mathsf{V}{\rightarrow}_i\ \langle \pi_2,\ e'_V \rangle \longrightarrow \exists \pi.\ \pi_1 = \pi\ @\ \pi_2$$

Furthermore, we can show that in mode P we are allowed to add (or remove) a prefix to (from) the proof stream.

▶ **Lemma 4** (Add/remove prefix of prover proof stream).

$$\langle \pi,\ e_P \rangle\ \mathsf{P}{\rightarrow}_i\ \langle \pi',\ e'_P \rangle \longleftrightarrow \langle X\ @\ \pi,\ e_P \rangle\ \mathsf{P}{\rightarrow}_i\ \langle X\ @\ \pi',\ e'_P \rangle$$

In mode V we can modify the proof stream by adding or removing a suffix.

▶ **Lemma 5** (Add/remove suffix of verifier proof stream).

$$\langle \pi,\ e_V \rangle\ \mathsf{V}{\rightarrow}_i\ \langle \pi',\ e'_V \rangle \longleftrightarrow \langle \pi\ @\ X,\ e_V \rangle\ \mathsf{V}{\rightarrow}_i\ \langle \pi'\ @\ X,\ e'_V \rangle$$

In mode I we do not touch the proof stream at all, so we will not need to prepend, append or remove data from them during proofs. However, we do want to prove that the proof stream does not change during evaluation.

▶ **Lemma 6** (Ideal execution does not modify the proof stream).

$$\langle \pi,\ e \rangle\ \mathsf{I}{\rightarrow}_i\ \langle \pi',\ e' \rangle \longrightarrow \pi = \pi'$$

3.4 Freshness Lemmas

In Section 2, we emphasized the importance of freshness when working with Nominal. In many instances, we have to show in our proofs that a certain variable is fresh with respect to some term, proof stream, or type environment. In this section, we discuss some of the more interesting freshness lemmas we needed to prove. One of the most useful lemmas is the following, relating freshness in a typing environment with freshness in terms. We show the lemma for the weak typing judgment, but similar statements hold for the strong typing judgment and for agreement, which will be introduced in Section 4.

▶ **Lemma 7** (Freshness in environment implies freshness in terms).

$$\mathsf{atom}\ x \sharp \Gamma \wedge \Gamma \vdash_W e : \tau \longrightarrow \mathsf{atom}\ x \sharp e$$

Proof. The proof is by induction on $\Gamma \vdash_W e : \tau$, with the only interesting case being the one for $\mathsf{Var}\ x$. Since $\mathsf{Var}\ x$ can only be well-typed if the type environment assigns a type to x, it is easy to show that a being fresh in Γ implies $a \neq x$. Hence, $\mathsf{atom}\ a \sharp \mathsf{Var}\ x$. ◀

For the small-step semantics we have lemmas showing that evaluation preserves freshness in some object, for example in the term when evaluating in mode P.

▶ **Lemma 8** (Prover evaluation preserves freshness in terms).

$$\mathsf{atom}\ x \sharp e \wedge \langle \pi, e\rangle \ \mathsf{P}\!\!\rightarrow \langle \pi', e'\rangle \longrightarrow \mathsf{atom}\ x \sharp e'$$

For the proof stream this only holds if the atom is fresh in both the term and the proof stream.

▶ **Lemma 9** (Prover evaluation preserves freshness in proof streams).

$$\mathsf{atom}\ x \sharp e \wedge \mathsf{atom}\ x \sharp \pi \wedge \langle \pi, e\rangle \ \mathsf{P}\!\!\rightarrow \langle \pi', e'\rangle \longrightarrow \mathsf{atom}\ x \sharp \pi'$$

3.5 Type Soundness

Now that we have defined the typing judgment and the small-step semantics of λ•, we turn our attention to type soundness for the execution in mode I. We proceed by proving the standard progress and preservation lemmas.

▶ **Lemma 10** (Progress).

$$\varnothing \vdash_W e : \tau \longrightarrow \mathsf{value}\ e \vee (\exists e'. \langle [], e\rangle \ \mathsf{I}\!\!\rightarrow \langle [], e'\rangle)$$

▶ **Lemma 11** (Preservation).

$$\langle [], e\rangle \ \mathsf{I}\!\!\rightarrow \langle [], e'\rangle \wedge \varnothing \vdash_W e : \tau \longrightarrow \varnothing \vdash_W e' : \tau$$

Using Lemma 10 and Lemma 11, type soundness for weakly well-typed terms follows easily.

▶ **Lemma 12** (Type Soundness).

$$\varnothing \vdash_W e : \tau \longrightarrow \mathsf{value}\ e \vee (\exists e'.\ \langle [], e\rangle \ \mathsf{I}\!\!\rightarrow \langle [], e'\rangle \wedge \varnothing \vdash_W e' : \tau)$$

nominal_function *erase* :: *ty* ⇒ *ty* **where**
  erase One = One
| erase (Fun τ_1 τ_2) = Fun (erase τ_1) (erase τ_2)
| erase (Sum τ_1 τ_2) = Sum (erase τ_1) (erase τ_2)
| erase (Prod τ_1 τ_2) = Prod (erase τ_1) (erase τ_2)
| erase (Mu α τ) = Mu α (erase τ)
| erase (Alpha α) = Alpha α
| erase (AuthT τ) = erase τ

Figure 7 The erase function.

There are two differences in our Lemma 12 compared to *ADSG*'s type soundness statement (Lemma 1). First, *ADSG* formulates the lemma for an arbitrary environment Γ (and consequently for terms that may contain free variables) in the judgment – an oversight which trivially invalidates the lemma: for example, Prj1 (Var x) is not a value and cannot take a step.

The second difference is that we formulate type soundness using the weak typing judgment. Type soundness does not hold for the original set of typing rules. Consider, for example, the well-typed expression Auth Unit of type AuthT One. Since it is not a value it must take a step. However, the resulting expression Unit has the different type One, violating type soundness (namely the preservation property). *ADSG* notes that "for mode I, authenticated values of type $\bullet\tau$ [i.e., AuthT τ] are merely values of type τ." This remark seems to imply that $\forall\tau$. AuthT $\tau \equiv \tau$, a property that is essential to a successful type soundness proof. Our weak typing judgment simulates syntactic equality of authenticated types by simply omitting them and allowing the introduction of the Auth and Unauth constructors without a change of types. However, although this interpretation is necessary for type soundness, it is undesirable. The main purpose of authenticated types is to ensure that Unauth can only be applied to expressions to which Auth has been applied previously. This disallows terms such as Unauth Unit, whose semantics is well-defined in the ideal execution mode but not in the prover and verifier modes. In the weakened typing judgment such terms are considered well-typed.

Since type soundness does not hold for the strong typing judgment, we show the weaker property that well-typed terms are also weakly well-typed after removing any AuthT annotations from its type and type environment. For this purpose we define the function erase (Figure 7), which erases all AuthT annotations in a type but leaves it otherwise unchanged. Using erase we can state and prove the relationship between the weak and the strong typing judgment. The function fmmap :: $(\beta \Rightarrow \gamma) \Rightarrow (\alpha, \beta)$ *fmap* $\Rightarrow (\alpha, \gamma)$ *fmap* is the canonical map function for the type of finite maps.

▶ **Lemma 13** (Well-typedness implies weak well-typedness).

$$\Gamma \vdash e : \tau \longrightarrow \text{fmmap erase } \Gamma \vdash_W e : \text{erase } \tau$$

4 Agreement

When introducing the small-step semantics we have discussed the intended interpretation of the mode. Any expression can be evaluated in mode I, performing a simple unauthenticated computation; in mode P, performing the computation and generating the proof stream; or in mode V, performing the computation and verifying the proof stream. Even though the three

$$\frac{}{\Gamma \vdash \mathsf{Unit},\ \mathsf{Unit},\ \mathsf{Unit} : \mathsf{One}} \qquad \frac{\mathsf{atom}\ x \sharp \Gamma \qquad \Gamma[x \mapsto \tau_1] \vdash e,\ e_P,\ e_V : \tau_2}{\Gamma \vdash \mathsf{Lam}\ x\ e,\ \mathsf{Lam}\ x\ e_P,\ \mathsf{Lam}\ x\ e_V : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma[x] = \mathsf{Some}\ \tau}{\Gamma \vdash \mathsf{Var}\ x,\ \mathsf{Var}\ x,\ \mathsf{Var}\ x : \tau} \qquad \frac{\Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \mathsf{Fun}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \tau_1}{\Gamma \vdash \mathsf{App}\ e_1\ e_2,\ \mathsf{App}\ e_{P1}\ e_{P2},\ \mathsf{App}\ e_{V1}\ e_{V2} : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp (\Gamma, e_1, e_{P1}, e_{V1}) \qquad \Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2,\ e_{P2},\ e_{V2} : \tau_2}{\Gamma \vdash \mathsf{Let}\ e_1\ x\ e_2,\ \mathsf{Let}\ e_{P1}\ x\ e_{P2},\ \mathsf{Let}\ e_{V1}\ x\ e_{V2} : \tau_2}$$

$$\frac{\mathsf{atom}\ x \sharp \Gamma \qquad \mathsf{atom}\ y \sharp (\Gamma, x) \qquad \Gamma[x \mapsto \mathsf{Fun}\ \tau_1\ \tau_2] \vdash \mathsf{Lam}\ y\ e,\ \mathsf{Lam}\ y\ e_P,\ \mathsf{Lam}\ y\ e_V : \mathsf{Fun}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e),\ \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e_P),\ \mathsf{Rec}\ x\ (\mathsf{Lam}\ y\ e_V) : \mathsf{Fun}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \tau_1}{\Gamma \vdash \mathsf{Inj1}\ e,\ \mathsf{Inj1}\ e_P,\ \mathsf{Inj1}\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \tau_1}{\Gamma \vdash \mathsf{Inj2}\ e,\ \mathsf{Inj2}\ e_P,\ \mathsf{Inj2}\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Sum}\ \tau_1\ \tau_2 \qquad \Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \mathsf{Fun}\ \tau_1\ \tau \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \mathsf{Fun}\ \tau_2\ \tau}{\Gamma \vdash \mathsf{Case}\ e\ e_1\ e_2,\ \mathsf{Case}\ e_P\ e_{P1}\ e_{P2},\ \mathsf{Case}\ e_V\ e_{V1}\ e_{V2} : \tau}$$

$$\frac{\Gamma \vdash e_1,\ e_{P1},\ e_{V1} : \tau_1 \qquad \Gamma \vdash e_2,\ e_{P2},\ e_{V2} : \tau_2}{\Gamma \vdash \mathsf{Pair}\ e_1\ e_2,\ \mathsf{Pair}\ e_{P1}\ e_{P2},\ \mathsf{Pair}\ e_{V1}\ e_{V2} : \mathsf{Prod}\ \tau_1\ \tau_2}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj1}\ e,\ \mathsf{Prj1}\ e_P,\ \mathsf{Prj1}\ e_V : \tau_1} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{Prod}\ \tau_1\ \tau_2}{\Gamma \vdash \mathsf{Prj2}\ e,\ \mathsf{Prj2}\ e_P,\ \mathsf{Prj2}\ e_V : \tau_2}$$

$$\frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e,\ e_P,\ e_V : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}{\Gamma \vdash \mathsf{Roll}\ e,\ \mathsf{Roll}\ e_P,\ \mathsf{Roll}\ e_V : \mathsf{Mu}\ \alpha\ \tau} \qquad \frac{\mathsf{atom}\ \alpha \sharp \Gamma \qquad \Gamma \vdash e,\ e_P,\ e_V : \mathsf{Mu}\ \alpha\ \tau}{\Gamma \vdash \mathsf{Unroll}\ e,\ \mathsf{Unroll}\ e_P,\ \mathsf{Unroll}\ e_V : \tau[\mathsf{Mu}\ \alpha\ \tau/\alpha]}$$

$$\frac{\Gamma \vdash e,\ e_P,\ e_V : \tau}{\Gamma \vdash \mathsf{Auth}\ e,\ \mathsf{Auth}\ e_P,\ \mathsf{Auth}\ e_V : \mathsf{AuthT}\ \tau} \qquad \frac{\Gamma \vdash e,\ e_P,\ e_V : \mathsf{AuthT}\ \tau}{\Gamma \vdash \mathsf{Unauth}\ e,\ \mathsf{Unauth}\ e_P,\ \mathsf{Unauth}\ e_V : \tau}$$

$$\frac{\mathsf{value}\ v \qquad \mathsf{value}\ v_P \qquad \varnothing \vdash v,\ v_P,\ (\!|v_P|\!) : \tau \qquad \mathsf{hash}\ (\!|v_P|\!) = h}{\Gamma \vdash v,\ \mathsf{Hashed}\ h\ v_P,\ \mathsf{Hash}\ h : \mathsf{AuthT}\ \tau}$$

Figure 8 The agreement predicate.

modes differ in their semantics and their terms may differ at any point during evaluation, their evaluations are structurally identical. This observation is captured by the agreement relation, written as $\Gamma \vdash e,\ e_P,\ e_V : \tau$ and read as "in environment Γ, ideal expression e, prover expression e_P, and verifier expression e_V all agree at type τ" (quoted from *ADSG* [10]).

We formalize agreement as an inductive predicate, with the introduction rules presented in Figure 8. Most rules are straightforward extensions of the (strong) typing judgment to three terms. This immediately gives us the following result, which states that any well-typed expression can be used in the ideal, prover, and verifier positions to yield an agreeing triple.

▶ **Lemma 14** (Well-typedness implies agreement)**.**

$$\Gamma \vdash e : \tau \longrightarrow \Gamma \vdash e,\ e,\ e : \tau$$

The interesting exception to the agreement rules being extensions of the typing rules is the last rule. It is modeled after the Auth small-step rules for the three modes. This rule allows the three expressions to diverge during the evaluation of Auth and still be in agreement. Note that the agreeing triple in the rule's premises may not contain any free variables. This property is enforced by the empty type environment, using the agreement version of Lemma 7. Therefore, the use of the hash function in this rule is unproblematic.

Lemma 14 states that well-typedness implies agreement. Ideally, we would also like to show the other direction of this property: agreement implying well-typedness. Unfortunately this does not hold. This is due to the extra agreement rule, allowing the introduction of authenticated types for any ideal value. Consider for example that with $\varnothing \vdash \mathsf{Unit}, \mathsf{Unit}, \mathsf{Unit} : \mathsf{One}$, we can obtain $\varnothing \vdash \mathsf{Unit}, \mathsf{Hashed}\ h\ \mathsf{Unit}, \mathsf{Hash}\ h : \mathsf{AuthT}\ \mathsf{One}$. Clearly we cannot show $\varnothing \vdash \mathsf{Unit} : \mathsf{AuthT}\ \mathsf{One}$. However, we can show weak well-typedness:

▶ **Lemma 15** (Reformulated Lemma 2.3 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow \mathsf{fmmap\ erase}\ \Gamma \vdash_W\ e : \mathsf{erase}\ \tau$$

We now prove Lemma 16 and Lemma 17 that are used extensively in later proofs.

▶ **Lemma 16** (Lemma 2.1 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow (\!| e_P |\!) = e_V$$

▶ **Lemma 17** (Lemma 2.4 from *ADSG*).

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \longrightarrow (\mathsf{value}\ e \wedge \mathsf{value}\ e_P \wedge \mathsf{value}\ e_V) \vee (\neg\mathsf{value}\ e \wedge \neg\mathsf{value}\ e_P \wedge \neg\mathsf{value}\ e_V)$$

In addition to Lemmas 15, 16, and 17, *ADSG* also states the following false property as Lemma 2.2. (Although *ADSG* states the property as a lemma, we did not encounter a situation where this statement was required to complete a proof.)

$$\Gamma \vdash e,\ e_P,\ e_V : \tau \wedge \Gamma \vdash e,\ e_P',\ e_V' : \tau \longrightarrow e_P = e_P' \wedge e_V = e_V'$$

To demonstrate why this property does not hold we construct a counterexample. We define $\mathsf{h} = \mathsf{Hash\ Unit}$ and we abbreviate Unit as u for better readability. Let us first consider the following two agreeing triples.

$$\varnothing \vdash \mathsf{u},\ \mathsf{u},\ \mathsf{u} : \mathsf{One}$$
$$\varnothing \vdash \mathsf{u},\ \mathsf{Hashed\ h\ u},\ \mathsf{Hash\ h} : \mathsf{AuthT\ One}$$

The second triple can be generated from the first one by applying the last agreement rule. Both triples share the environment and the first term but disagree in the second and third term as well as their type. Using the Pair rule we obtain the following two agreeing triples.

$$\varnothing \vdash \mathsf{Pair\ u\ u},\ \mathsf{Pair\ u\ u},\ \mathsf{Pair\ u\ u} : \mathsf{Prod\ One\ One}$$
$$\varnothing \vdash \mathsf{Pair\ u\ u},\ \mathsf{Pair\ u\ (Hashed\ h\ u)},\ \mathsf{Pair\ u\ (Hash\ h)} : \mathsf{Prod\ One\ (AuthT\ One)}$$

Applying Prj1 to these triples removes the difference in the types but preserves the differences in the second and third terms, completing our counterexample to *ADSG*'s Lemma 2.2.

$$\varnothing \vdash \mathsf{Prj1\ (Pair\ u\ u)},\ \mathsf{Prj1\ (Pair\ u\ u)},\ \mathsf{Prj1\ (Pair\ u\ u)} : \mathsf{One}$$
$$\varnothing \vdash \mathsf{Prj1\ (Pair\ u\ u)},\ \mathsf{Prj1\ (Pair\ u\ (Hashed\ h\ u))},\ \mathsf{Prj1\ (Pair\ u\ (Hash\ h))} : \mathsf{One}$$

In the following we prove that, given a well-typed $\lambda\bullet$ term, containing only free variables of authenticated types, substituting agreeing values of the same type produces an agreeing triple. This property is significant because it occurs in the following practical scenario. The verifier must represent the data structure in a query it sends to the prover. It does so by replacing it with a free variable, for which the prover substitutes its representation of the data structure. The prover then returns the generated proof stream to the verifier, who substitutes the free variable with its hash of the data structure and verifies the proof stream. We formalized this lemma as stated below, with *fmdom* returning a finite map's domain as a finite set and $|\in|$ denoting membership on finite sets.

▶ **Lemma 18** (Reformulated Lemma 3 from *ADSG*). *For* Δ, Δ_P, Δ_V :: $(var, term)$ *fmap:*

$$\left(\begin{array}{l}\Gamma \vdash e : \tau \wedge \\ \text{fmdom } \Delta = \text{fmdom } \Gamma \wedge \text{fmdom } \Delta_P = \text{fmdom } \Gamma \wedge \text{fmdom } \Delta_V = \text{fmdom } \Gamma \wedge \\ \left(\begin{array}{l}\forall x.\ x \mathrel{|\in|} \text{fmdom } \Gamma \longrightarrow (\exists \tau', v, v_P, h.\ \Gamma[x] = \text{Some } (\text{AuthT } \tau') \wedge \\ \quad \Delta[x] = \text{Some } v \wedge \Delta_P[x] = \text{Some } (\text{Hashed } h\ v_P) \wedge \Delta_V[x] = \text{Some } (\text{Hash } h) \wedge \\ \quad \varnothing \vdash v,\ \text{Hashed } h\ v_P,\ \text{Hash } h : \text{AuthT } \tau')\end{array}\right)\end{array}\right) \longrightarrow$$
$$\varnothing \vdash \text{psubst } e\ \Delta,\ \text{psubst } e\ \Delta_P,\ \text{psubst } e\ \Delta_V : \tau$$

ADSG's Lemma 3 includes an additional premise:

$\Gamma \vdash e : \tau$ where e contains no values of type $\text{AuthT } \tau$

Since variables are values, this premise implies that e contains neither bound nor free variables of type $\text{AuthT } \tau$ (only for this particular τ, it can contain other variables with other authenticated types). The premise does not impose any further restrictions, since variables are the only expressions that are values and can have type $\text{AuthT } \sigma$ for some σ. We are unclear as to what this premise's purpose is. Fortunately, the lemma holds without it.

Finally, we prove a straightforward but crucial lemma, which states that substituting agreeing values of the correct type for a free variable in an agreeing triple preserves agreement.

▶ **Lemma 19** (Lemma 4 from *ADSG*).

$$\left(\begin{array}{l}\Gamma[x \mapsto \tau'] \vdash e,\ e_P,\ e_V : \tau \wedge \varnothing \vdash v,\ v_P,\ v_V : \tau' \wedge \\ \text{value } v \wedge \text{value } v_P \wedge \text{value } v_V\end{array}\right) \longrightarrow \Gamma \vdash e[v/x],\ e_P[v_P/x],\ e_V[v_V/x] : \tau$$

5 Correctness

Having formalized $\lambda\bullet$ and proved a number of lemmas about it, we now take a look at the main claims formulated in *ADSG*, concerning the correctness and security of $\lambda\bullet$. We start with some agreeing terms e, e_P, e_V. The properties we would then like to obtain can be informally stated as follows:

1. *Correctness*: If e takes i steps in mode I, then e_P and e_V can also take i steps in their respective modes, with the verifier consuming the prover's output proof stream. The resulting terms agree.
2. *Security*: If e_V takes i steps in mode V, consuming the proof stream π (which may be legit or created by an adversary trying to trick the verifier) then either e and e_P can also take i steps in their respective modes, with the prover generating π and the resulting terms agreeing, or otherwise there exists a term in the proof stream π, such that we can show the presence of a hash collision.

Besides these primary claims *ADSG* formulates a third claim (named *Remark 1*) that starts with the prover's computation and lets the other two modes follow:

3. *Remark 1*: If e_P takes i steps in mode P generating the proof stream π, then e and e_V can also take i steps in their respective modes, with the verifier consuming π. The resulting terms agree.

In a first step we formulate and prove these three properties on the single-step relation. Afterwards we will lift these lemmas to obtain the main results on the multi-step relation.

▶ **Lemma 20** (Single step version of Correctness, Lemma 5 from *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle [],\ e\rangle \ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \longrightarrow \left(\begin{array}{l}\exists e'_P, e'_V, \pi.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \\ \quad (\forall \pi_P.\ \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle) \wedge (\forall \pi'.\ \langle \pi \mathbin{@} \pi',\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle \pi',\ e'_V\rangle)\end{array}\right)$$

Proof. The proof is by induction on the agreement relation. Most cases are straightforward, using the lemmas about agreement and various freshness lemmas. The most interesting cases are those for Let, Auth and Unauth. Let is the only construct with a binder that allows recursive evaluation, requiring an additional freshness lemma to show that the recursive step preserves freshness. The Auth and Unauth cases require us to show that the expressions being hashed are closed. In both cases we have an agreeing triple with an empty typing context, so we can apply the counterpart of Lemma 7 for agreement to show that property. ◀

▶ **Lemma 21** (Single step version of Security, Lemma 6 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_A,\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle \pi',\ e'_V\rangle \longrightarrow \left(\begin{array}{l}\exists e', e'_P, \pi.\ \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \wedge (\forall \pi_P.\ \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle) \wedge \\ \quad ((\varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \pi_A = \pi \mathbin{@} \pi') \vee \\ \quad\ (\exists s, s'.\ \pi = [s] \wedge \pi_A = [s'] \mathbin{@} \pi' \wedge s \neq s' \wedge \mathsf{hash}\ s = \mathsf{hash}\ s' \wedge \mathsf{closed}\ s \wedge \mathsf{closed}\ s'))\end{array}\right)$$

Proof. The proof is similar to that of Lemma 20, though the Unauth case here does not involve hashes and therefore does not need special treatment. ◀

▶ **Lemma 22** (Single step version of Remark 1).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle \longrightarrow (\exists e', e'_V.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}\ \langle [],\ e'\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow}\ \langle [],\ e'_V\rangle)$$

Proof. The proof is by straightforward induction on the agreement relation, without any of the special cases of Lemmas 20 and 21. ◀

Having proven Lemmas 20, 21 and 22 we can now lift the results to the small-step semantics' transitive closure to obtain the main results, described informally above.

▶ **Theorem 23** (Correctness, Theorem 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}_i\ \langle [],\ e'\rangle \longrightarrow (\exists e'_P, e'_V, \pi.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow}_i\ \langle \pi,\ e'_P\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow}_i\ \langle [],\ e'_V\rangle)$$

▶ **Theorem 24** (Security, Theorem 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_A,\ e_V\rangle\ \mathsf{V}{\rightarrow}_i\ \langle \pi',\ e'_V\rangle \longrightarrow \left(\begin{array}{l}(\exists e', e'_P, \pi.\ \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}_i\ \langle [],\ e'\rangle \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow}_i\ \langle \pi,\ e'_P\rangle \wedge \\ \quad \pi_A = \pi \mathbin{@} \pi' \wedge \varnothing \vdash e',\ e'_P,\ e'_V : \tau) \vee \\ (\exists e'_P, j, \pi_0, \pi'_0, s, s'.\ j \leq i \wedge \langle [],\ e_P\rangle\ \mathsf{P}{\rightarrow}_j\ \langle \pi_0 \mathbin{@} [s],\ e'_P\rangle \wedge \\ \quad \pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'_0 \mathbin{@} \pi' \wedge s \neq s' \wedge \mathsf{hash}\ s = \mathsf{hash}\ s' \wedge \mathsf{closed}\ s \wedge \mathsf{closed}\ s')\end{array}\right)$$

▶ **Theorem 25** (Remark 1 in *ADSG*).

$$\varnothing \vdash e,\ e_P,\ e_V : \tau \wedge \langle \pi_P,\ e_P\rangle\ \mathsf{P}{\rightarrow}_i\ \langle \pi_P \mathbin{@} \pi,\ e'_P\rangle \longrightarrow (\exists e', e'_V.\ \varnothing \vdash e',\ e'_P,\ e'_V : \tau \wedge \langle [],\ e\rangle\ \mathsf{I}{\rightarrow}_i\ \langle [],\ e'\rangle \wedge \langle \pi,\ e_V\rangle\ \mathsf{V}{\rightarrow}_i\ \langle [],\ e'_V\rangle)$$

The statement of Theorem 24 differs from the one in *ADSG*. In the case where colliding hashes cause the verifier to falsely accept a computation as correct, the theorem ensures that the offending proof stream π_A has a specific shape. *ADSG* claims this shape to be $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'$, i.e., the evaluation must stop after a hash collision is encountered. For Lemma 21, the single-step version, this holds, since we only evaluate a single step. However, this fact is no longer true when taking multiple steps, since the verifier may continue to evaluate and consume valid (or invalid) elements of the proof stream after encountering the hash collision. In fact, the verifier cannot recognize that a hash collision has occurred. Formally, this means that $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'_0 \mathbin{@} \pi'$ for some π'_0, as our corrected theorem states. We illustrate the problem with *ADSG*'s formulation with a concrete counterexample:

$$\mathsf{Let}\ (\mathsf{Unauth}\ (\mathsf{Auth}\ (\mathsf{Inj1}\ \mathsf{Unit})))\ x\ (\mathsf{Let}\ (\mathsf{Unauth}\ (\mathsf{Auth}\ \mathsf{Unit}))\ y\ (\mathsf{Var}\ x))$$

This term can be evaluated in the prover mode to generate the proof stream [Inj1 Unit, Unit]. We assume a hash function, which satisfies hash (Inj1 Unit) = hash (Inj2 Unit) and hash Unit $\neq$ hash t for all $t \neq$ Unit. Note that, since all theorems are formulated to be agnostic to the choice of the hash function, this is an entirely reasonable hash function to use in a counterexample. A verifier using the adversarial proof stream $\pi_A =$ [Inj2 Unit, Unit] evaluates the given term to Inj2 Unit. The original statement of the theorem would require the proof stream to be of the shape $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'$ with $\pi' = []$. However, our adversarial proof stream does not fit this pattern since the term with a colliding hash is not the last term from the proof stream that is evaluated. With our amended, formally verified version, the shape $\pi_A = \pi_0 \mathbin{@} [s'] \mathbin{@} \pi'_0 \mathbin{@} \pi'$ can be matched as $\pi_A = [] \mathbin{@}$ [Inj1 Unit] @ [Unit] @ [].

Since *ADSG* requires terms to be in administrative normal form, the above counterexample cannot be expressed in *ADSG*'s definition of λ•. However, in our formalization we include a (more verbose) counterexample in administrative normal form.

6 Discussion

We have formalized λ• and proved its correctness and security in Isabelle/HOL. Our work can be seen as the mechanized supplement to Miller et al.'s *ADSG* [10]. Ultimately, *ADSG* passed the test of formalization. However, achieving this result turned out to be harder than we first had expected, given the mistakes and imprecisions we had to overcome. We discovered major problems in the paper's Lemmas 1 and 2.2. We repaired Lemma 1 in a rather unsatisfactory fashion. However, in our view type soundness, and more specifically type preservation, is not very relevant for λ•; what is more important is the preservation of agreement, which correctness and security establish. Lemma 2.2 could not be salvaged. Moreover, we removed a redundant (and nonsensical) assumption from *ADSG*'s Lemma 3 and corrected a slip in the formal statement of *ADSG*'s main security theorem. We have not reported here the minor typos we found in *ADSG*'s informal definitions and refer to the first author's Bachelor's thesis [4] for such an overview. Taken together, our findings confirm the value of formal proofs. The formalization could (and arguably should) have been undertaken as part of the research on *ADSG*.

The last point is typically countered by the disproportional effort needed to obtain the formalization. However, in this case the effort was modest: The main difficulties stemmed from the fact that on several occasions we first tried to prove false statements from *ADSG*.

At 3500 lines of proof, our formalization is concise. In our view, Nominal was the main asset behind this conciseness, because it allowed us to closely follow the informal proofs, while discharging straightforward freshness obligations along the way. Nominal's seamless integration with the type of finite maps provided the right level of abstraction to reason about type environments and term substitutions.

However, we also noticed a few points where Nominal could provide a better user experience. First, the introduction of binding-aware recursive functions and inductive predicates requires some boilerplate proofs, which in many cases seem automatable. This impression is confirmed by the fact that we could literally copy these proofs from unrelated formalizations that were also using Nominal and perform minor adjustments to make them work in our case. Second, *ADSG* uses terms of the form rec x $\lambda y.\ t$ for defining recursive functions, which we model with the term Rec x (Lam y t). The more faithful way to model this form would be a single Nominal datatype constructor that simultaneously binds two variables:

Rec $(x :: \mathit{var})$ $(y :: \mathit{var})$ $(t :: \mathit{term})$ **binds** x **and** y **in** t

Nominal supports this declaration. However, the reasoning infrastructure it provides for such constructors is significantly more difficult to use than the one for the special case of constructors binding a single variable. We had started our formalization with the above formulation, but soon switched to the presented Rec constructor that only binds the recursive variable x. Note that both typing and agreement require Rec's second argument to be of a function type, which is what the above form used in *ADSG* aims to hardwire into the syntax. Third, unlike *ADSG* we do not consider actually running $\lambda\bullet$ programs. Here, in our opinion, Nominal does not score very well by not being integrated with Isabelle's code generator. And moreover, it is not clear in general how to execute recursive functions that carry freshness assumptions. Executability can be regained by translating the Nominal types to a nameless representation (e.g., de Bruijn indices) and lifting all definitions to this representation. Developing a more principled approach to executing Nominal programs is interesting future work.

References

1 Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 40 of *Studies in Logic*. Elsevier, 1984.

2 Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *PACMPL*, 3(POPL):22:1–22:34, 2019. doi:10.1145/3290335.

3 Joachim Breitner. The adequacy of Launchbury's natural semantics for lazy evaluation. *J. Funct. Program.*, 28:e1, 2018. doi:10.1017/S0956796817000144.

4 Matthias Brun. *Authenticated Data Structures in Isabelle/HOL*. B.Sc. thesis, ETH Zürich, 2019.

5 Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In Bhavani M. Thuraisingham, Reind P. van de Riet, Klaus R. Dittrich, and Zahir Tari, editors, *DBSec 2000*, volume 201 of *IFIP Conference Proceedings*, pages 101–112. Kluwer, 2000. doi:10.1007/0-306-47008-X_9.

6 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *PLDI 1993*, pages 237–247. ACM, 1993. doi:10.1145/155090.155113.

7 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.

8 Brian Huffman and Christian Urban. A new foundation for Nominal Isabelle. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 35–50. Springer, 2010. doi:10.1007/978-3-642-14052-5_5.

9 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO 1987*, volume 293 of *LNCS*, pages 369–378. Springer, 1987. doi:10.1007/3-540-48184-2_32.

10 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In Suresh Jagannathan and Peter Sewell, editors, *POPL 2014*, pages 411–424. ACM, 2014. doi:10.1145/2535838.2535851.

11 Julian Nagele, Vincent van Oostrom, and Christian Sternagel. A short mechanized proof of the Church-Rosser theorem by the Z-property for the $\lambda\beta$-calculus in Nominal Isabelle. *CoRR*, abs/1609.03139, 2016.

12 Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In Robert Nieuwenhuis, editor, *CADE 2005*, volume 3632 of *LNCS*, pages 424–440. Springer, 2005. doi:10.1007/11532231_31.

13 Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *J. Autom. Reasoning*, 55(1):1–37, 2015. doi:10.1007/s10817-015-9322-8.

14 Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002.

15 George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In June Andronick and Amy P. Felty, editors, *CPP 2018*, pages 78–90. ACM, 2018. doi:10.1145/3167086.

16 Roberto Tamassia. Authenticated data structures. In Giuseppe Di Battista and Uri Zwick, editors, *ESA 2003*, volume 2832 of *LNCS*, pages 2–5. Springer, 2003. doi:10.1007/978-3-540-39658-1_2.

17 Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. *Logical Methods in Computer Science*, 8(2), 2012. doi:10.2168/LMCS-8(2:14)2012.

18 Christian Urban and Julien Narboux. Formal SOS-proofs for the lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 247:139–155, 2009. doi:10.1016/j.entcs.2009.07.053.

19 Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *CADE 2005*, volume 3632 of *LNCS*, pages 38–53. Springer, 2005. doi:10.1007/11532231_4.

20 Bill White. A theory for lightweight cryptocurrency ledgers. Accessed on 30.03.2019, 2015. URL: https://github.com/input-output-hk/qeditas-ledgertheory.