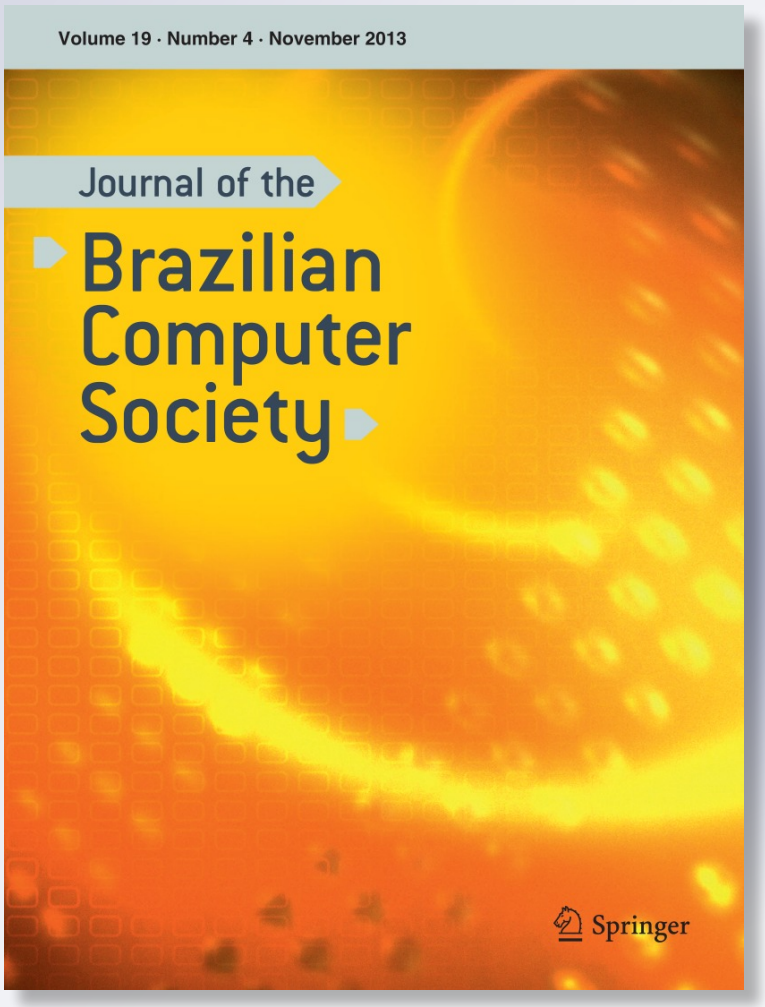
Mechanized metatheory for a $$\lambda $ $ λ -calculus with trust types

Rodrigo Ribeiro, Lucília Figueiredo & Carlos Camarão

Journal of the Brazilian Computer Society

ISSN 0104-6500
Volume 19
Number 4

J Braz Comput Soc (2013) 19:433-443
DOI 10.1007/s13173-013-0119-5



Springer

J Braz Comput Soc (2013) 19:433–443
DOI 10.1007/s13173-013-0119-5

ORIGINAL PAPER

Mechanized metatheory for a λ-calculus with trust types

Rodrigo Ribeiro · Lucília Figueiredo · Carlos Camarão

Received: 7 September 2012 / Accepted: 15 July 2013 / Published online: 4 August 2013

Abstract As computer programs become increasingly complex, techniques for ensuring trustworthiness of information manipulated by them become critical. In this work, we use the Coq proof assistant to formalise a λ-calculus with trust types, originally formulated by Ørbæk and Palsberg. We give formal proofs of type soundness, erasure and simulation theorems and also prove decidability of the typing problem. As a result of our formalisation a certified type checker is derived.

Keywords Trust · Type systems · Proof assistants · Soundness proofs

1 Introduction

Ensuring security of information manipulated by computer systems is a long-standing and increasingly important problem. There is little assurance that current computer systems keep data integrity and traditional (theoretical and practical) approaches to express and enforce security properties are, in general, unsatisfactory [46,53].

R. Ribeiro (✉) · C. Camarão
Departamento de Ciência da Computação, Instituto de Ciências Exatas, Universidade Federal de Minas Gerais,
Belo Horizonte, Brazil
e-mail: rodrigogribeiro@decea.ufop.br

C. Camarão
e-mail: camarao@dcc.ufmg.br

L. Figueiredo
Departamento de Computação, Instituto de Ciências Exatas e Biológicas, Universidade Federal de Ouro Preto,
Ouro Prêto, Brazil
e-mail: lucilia@iceb.ufop.br

One of such traditional approaches to protect data confidentiality is access control: privileges are required to access files or objects containing confidential data and information release is restricted according to some policy. Access control checks can restrict release but not propagation of information. Once information is released, a program can transmit it in some form and, since it is not feasible to suppose that all programs in a system are trustworthy, one cannot ensure that confidentiality is maintained. In order to guarantee that information is used only in accordance with relevant policies, it is necessary to analyse how information flows within the program. As modern computing systems are complex artefacts, automating such analysis is required [46].

An approach to ensure security property of computer software consists of the use of type systems in order to control information flow in software [46]. In programming languages with *security types*, variables and expressions types have annotations that indicate policies to be ensured by the compiler on uses of such data. This approach has the following benefits: (1) since these policies are checked at compile-time, there is no run-time overhead; (2) once security policies are expressed by a type system, standard techniques for guaranteeing type system soundness can be used to certify that security policies are enforced in an end-to-end way in the whole program.

However, proofs of programming language formalisms (e.g. type systems and semantics) are usually long and error prone. In order to give more reliability to these proofs, programming language researchers have been developing, in recent years, a large number of works devoted to machine assisted proofs [1,6,28,10].

In this work, we provide a formalisation of a variant of λ-calculus with trust types, as proposed by Ørbæk and Palsberg [38], using the Coq proof assistant [5]. Specifically, our contribution is to provide a machine checked proof of:

1. type soundness, using standard small-step call-by-value semantics. Intuitively, type soundness property ensures that if a program is well-typed it does not cause any run-time errors,
2. erasure and simulation theorems [38, Sections 3.3 and 3.4]. Erasure and simulation theorems ensure that the λ-calculus with trust types is a restriction of the simply typed λ-calculus. Together these theorems ensure that after type-checking a term, we can simply erase all trust constructs and annotations and evaluate it using the rules of the simply typed λ-calculus,
3. decidability of type checking. From this proof we extract a certified type checker for the language.

The developed formalisation is axiom free, that is, all necessary results and properties were integrally proved in Coq. We choose Coq because it is an industrial strength proof assistant that has been used in several large scale projects such as a Certified C compiler [27] and Java Card platform [3]. The complete formalisation has approximately 1,400 lines of code. This makes it impossible to present here all details of the work. We only sketch the main proofs and some function definitions are omitted for brevity, when they are trivial. The Coq source code of this work is available on-line [43].

The rest of this paper is organised as follows. Section 2 presents a brief introduction to the Coq proof assistant and its features used in our formalisation. Section 3 briefly reviews the syntax and defines a small-step semantics for the λ-calculus with trust types. Section 4 presents the non-syntax directed type system for the λ-calculus with trust, as proposed in [38], and proves its type soundness property. We also define a syntax directed version of this original type system and prove soundness and completeness between these two versions. Finally, we prove that the typing problem for this calculus is decidable. Section 6 presents related work and Sect. 7 concludes.

2 A taste of Coq proof assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [5], a higher order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called BHK-correspondence,[1] where types represent logical formulas and λ-terms represent proofs [49]. Thus, the task of checking if a piece of text is a proof of a given formula corresponds to checking if the term that represents the proof has the type corresponding to the given formula.

[1] Abbreviation of Brower, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry–Howard "isomorphism".

```
Inductive nat : Set :=
   | O : nat
   | S : nat -> nat.

Fixpoint plus (n m : nat) : nat :=
   match n with
      | O => m
      | S n' => S (plus n' m)
   end.

Theorem plus_O_r : forall n, plus n O = n.
Proof.
   intros n.
   induction n as [| n'].
   (**Case n = O**)
      reflexivity.
   (**Case n = S n' **)
      simpl.
      rewrite -> IHn'.
      reflexivity.
Qed.
```

Fig. 1 Sample Coq code

However, writing a proof term whose type is that of a logical formula can be a hard task, even for very simple propositions. In order to make the writing of complex proofs easier, Coq provides *tactics*, which are commands that can be used to construct proof terms in a more user friendly way.

We briefly illustrate these notions by means of a small example, shown in Fig. 1.

The source code in Fig. 1 shows some basic features of the Coq proof assistant: types, functions and proof definitions. In this example, a new inductive type is defined to represent natural numbers in Peano notation. This type is formed by two data constructors: `O`, that represents the number 0; and `S`, the successor function. For instance, in this notation the number 2 is represented by the term `S (S O)` of type `nat`.

The command `Fixpoint` allows the definition of structural recursive functions. Function `plus` defines the sum of two unary natural numbers, in a straightforward way. It is noteworthy that, in order to maintain logical consistency, all functions in Coq must be total.

Besides the declaration of inductive types and functions, we can define and prove theorems in Coq. Figure 1 shows an example of a simple theorem about function `plus`, namely that, for an arbitrary value `n` of type `nat`, we have that `plus n O = n`. The command `Theorem` allows us to state some formula that we want to prove and it starts the *interactive proof mode*, in which tactics can be used to produce the wanted proof term. In an interactive section of Coq (after enunciation of theorem `plus_O_r`), we must prove the following goal:

```
============================
  forall n : nat, plus n O = n
```

After command `Proof.`, one can use tactics to build, step by step, a term of the given type. The first tactic, `intros`, is

used to move premisses and universally quantified variables from the goal to the hypothesis. Now, we need to prove:

```
n : nat
============================
plus n 0 = n
```

The quantified variable `n` has been moved from the `goal` to the hypothesis. Now, we can proceed by induction over the structure of `n`. This can be achieved by using tactic `induction`, that generates one goal for each constructor of type `nat`. This will leave us with the following two goals to be proved:

```
2 subgoals

============================
   plus 0 0 = 0

subgoal 2 is:
 S n' + 0 = S n'
```

The goal `plus 0 0 = 0` holds trivaly by the definition of `plus`. Tactic `reflexivity` proves trivial equalities, after reducing both sides of the equality to their normal forms. The next goal to be proved is:

```
n' : nat
IHn' : plus n' 0 = n'
============================
   plus (S n') 0 = S n'
```

The hypothesis `IHn'` is the automatically generated induction hypothesis for this theorem. In order to finish this proof, we need to transform the goal to use the inductive hypothesis. To do this, we use the tactic `simpl`, which performs reductions based on the definition of function `plus`. This changes the goal to:

```
n' : nat
IHn' : plus n' 0 = n'
============================
   S (plus n' 0) = S n'
```

Since the goal now has as a subterm the exact left hand side of the hypothesis `IHn'`, we can use the `rewrite` tactic, which replaces some term by another using some equality in the hypothesis. Now, we have the following goal:

```
n' : nat
IHn' : plus n' 0 = n'
============================
   S n' = S n'
```

This can be proved immediately using the `reflexivity` tactic. This tactic script builds the following term:

```
fun n : nat =>
  nat_ind
  (fun n0 : nat => n0 + 0 = n0) (eq_refl 0)
    (fun (n' : nat) (IHn' : n' + 0 = n') =>
     eq_ind_r (fun n0 : nat => S n0 = S n')
       (eq_refl (S n')) IHn') n
       : forall n : nat, n + 0 = n
```

Instead of using tactics, one could instead write CIC terms directly to prove theorems. This is, however, a complex task, even for simple theorems like `plus_O_r`, since the manual writing of proof terms requires knowledge of the CIC type system. Thus, tactics frees us from the details of constructing type correct CIC terms.

An interesting feature of Coq is the possibility of defining inductive types that mix computational and logic parts. This allows us to define functions that compute values together with a proof that this value has some desired property. The type `sig`, also called "subset type", is defined in the Coq's standard library as:

```
Inductive sig (A : Set)
      (P : A -> Prop) : Set :=
  | exist : forall x : A, P x -> sig A P.
```

The `exist` constructor takes two arguments: the value `x` of type `A`—that represents the computational part—and an argument of type `P x`—the "certificate" that the value `x` has the property specified by the predicate `P`. As an example of a `sig` type, consider:

```
forall n : nat, n <> 0 -> {p | n = S p}.
```

This type represents a function that returns the predecessor of a natural number `n`, together with a proof that the returned value `p` really is the predecessor of `n`. Defining functions using the `sig` type requires writing the corresponding logical certificate. As with theorems, we can use tactics to define such functions.

```
Definition pred_certified :
   forall n : nat, n <> 0 -> {p | n = S p}.
   intros n H.
   destruct n as [| n'].
   (**Case n = 0**)
   elim H. reflexivity.
   (**Case n = S n'**)
   exists n'. reflexivity.
Defined.
```

Using the command `Extraction pred_certified` we can discard the logical part of this function definition and get a certified implementation of this function in OCaml [50], Haskell [23] or Scheme [16]. The OCaml code of this function, obtained through extraction, is the following:

```
(** val pred_cert : nat -> nat **)
let pred_cert = function
   | O -> assert false (* absurd case *)
   | S n0 -> n0
```

3 λ-Calculus with trust types

This section reviews some motivations for the use of trust types and gives definitions of the syntax and semantics of the trust λ-calculus, which differ from the original definitions in [38] as follows: (1) we use a small-step call-by-value semantics, and (2) without loss of generality, we consider only one base type: `bool`. Extensions to include other type constructors are straightforward.

3.1 Motivations

Data manipulated by computer programs can be classified as *trusted* or *untrusted*. Trusted data come from trusted sources, like company databases, program constants, cryptographically verified network data. All other data are considered untrusted [38].

Trust analysis is especially important in web applications, where user input data can be used to exploit security vulnerabilities, using attacks such as cross-site scripting (XSS). XSS attacks can occur when a user is able to "dump" HTML text in a dynamically generated page [44]. Through this vulnerability, it is possible to inject JavaScript code to steal cookies, in order to acquire session privileges. Such a threat occurs due to a lack of verification on input data since, ideally, HTML code cannot be considered as valid input.

In order to avoid such invalid inputs, one can insert checks that ensure data trustworthiness. But, how can we guarantee that all paths, in which probably untrusted information flows, pass all required checks? The solution proposed by Ørbæk and Palsberg [38] is to use a type system to track the flow of untrusted data in a program.

The language considered is a λ-calculus with additional constructs to check if some piece of data can be trusted and mark data as trusted or untrusted. If e is some program expression, then *trust* e indicates that the result of e can be trusted. Dually, *distrust* e indicates that the result of e cannot be trusted and *check* e indicates that e must be trustworthy. Well-typed programs do not have any sub-expression *check* e where e has an untrusted type.

3.2 Syntax of types and terms

Type syntax is given in Fig. 2, where metavariable usage is also given. It is exactly the type syntax of simply typed λ-calculus with boolean constants, except that each type t has a trust annotation to specify if t values can be trusted or not.

$$
\begin{aligned}
u &::= \texttt{tr} \mid \texttt{dis} \\
\tau &::= t^u \\
t &::= \texttt{bool} \mid \tau \to \tau
\end{aligned}
$$

```
Inductive trustty : Type :=
   | tr : trustty
   | dis : trustty.

Inductive ty : Type :=
   | ty_bool : trustty -> ty
   | ty_arrow : ty -> ty -> trustty -> ty.
```

Fig. 2 Syntax of trust types

$$
\begin{aligned}
e &::= x \\
&\mid \lambda x : \tau . e \\
&\mid e\, e \\
&\mid \texttt{true} \\
&\mid \texttt{false} \\
&\mid \texttt{trust}\ e \\
&\mid \texttt{distrust}\ e \\
&\mid \texttt{check}\ e
\end{aligned}
$$

```
Inductive term : Type :=
   | tm_var : id -> term
   | tm_lam : id -> ty -> term -> term
   | tm_app : term -> term -> term
   | tm_true : term
   | tm_false : term
   | tm_trust : term -> term
   | tm_distrust : term -> term
   | tm_check : term -> term.
```

Fig. 3 Syntax of terms

The translation of the type syntax to a Coq inductive type is straightforward, and is also presented in Fig. 2.

The syntax of terms consists of boolean constants, variables, abstractions and applications, and the three additional constructs to deal with trust types, explained previously. Figure 3 defines the syntax of terms and the corresponding Coq data type.

The syntax of types and terms used in our formalisation is identical to [38], except that we require type annotations in every λ-abstraction. We restrict ourselves to type annotated λ-terms, since our main interest is the development of a correct type checker for this language. Allowing non-annotated λ-abstractions characterises a type inference problem that would require a formalisation of a unification algorithm. The formalisation of a unification algorithm has been studied elsewhere [26,32]. We let a formalisation of the type inference problem for this trust-calculus for future work.

The `id` type, used in the definition of `term`, represents a generic identifier with a decidable function for testing equality and its simple definition is omitted to avoid unnecessary distraction.

3.3 Small-step operational semantics

In order to prove type soundness, we follow the standard approach of using a small-step operational semantics for

```
v ::= true
    | false
    | λx : τ.e
u ::= distrust v
```

```
Inductive prim_value : term -> Prop :=
   | v_true : prim_value tm_true
   | v_false : prim_value tm_false.
   | v_abs : forall x T e,
                prim_value (tm_abs x T e).

Inductive untrusted_value : term -> Prop :=
   | u_dist : forall v, prim_value v ->
           untrusted_value (tm_distrust v)
```

Fig. 4 Definition of Values

proving progress and preservation theorems [39]. This differs from the approach adopted in [38], where the semantics of the trust λ-calculus is formalised using a reduction semantics, with no predefined order of evaluation, and the Church–Rosser property and a subject reduction theorem are proved. The Church–Rosser property ensures that computation in the trust λ-calculus is deterministic and subject reduction guarantees that reduction preserves typing [2,20].

Let us first define the notion of *value*, i.e. a term that cannot be further reduced according to the intended semantics. We distinguish two kinds of values: primitive values and untrusted values. Primitive values (represented by metavariable v) are boolean constants and λ-abstractions. An untrusted value (represented by metavariable u) is a term of the form (*distrust v*), where v is a primitive value. Untrusted values arise as normal forms of terms that do not have any *check* construct.

The definition of values is given in Fig. 4. Corresponding Coq definitions for values are straightforward predicate definitions over `term`. Note that a term $\lambda\, x.e$ is always a value, no matter whether e is a value or not—in other words, we can say that reduction stops at abstractions.

The small-step semantics of the trust λ-calculus is an extension of the standard call-by-value semantics for the simply typed λ-calculus. The required extensions deal with trust specific constructs (terms `trust`, `distrust` and `check`). As usual, semantics for λ-calculi rely on substitution. For any e_1, e_2 and x, we define $[x \mapsto e_1]\, e_2$ to be the result of substituting every *free* occurrence of variable x in e_2, that follows the standard definition of capture free substitution [2,20].

The Coq function presented in Fig. 5 encodes term substitution. Function `subst` replaces every free occurrence of `x` in `t'` for `t`. It is straightforwardly defined by structural recursion over `t'`. In `tm_var` and `tm_abs` cases we have to check whether `x` is equal to the current variable. Substitution becomes trickier to define if we consider the case where `t`, the term being substituted for a variable in term `t'`, may itself contain free variables. However, since our interest is on extracting a mechanically verified type checker, our definition of the step relation may be restricted to closed terms (i.e. terms that don't have free variables) and thus we can avoid the extra complexity of dealing with the problem of free variable capture in the formalisation of substitution.[2]

```
Fixpoint subst(x : id)(t t' : term) : term:=
  match t' with
    | tm_var i =
        if beq_id x i then t else t'
    | tm_app l r =>
        tm_app (subst x t l) (subst x t r)
    | tm_abs i T t1 => tm_abs i T
        (if beq_id x i then t1
            else (subst x t t1))
    | tm_trust t1 =>
        tm_trust (subst x t t1)
    | tm_distrust t1 =>
        tm_distrust (subst x t t1)
    | tm_check t1 =>
        tm_check (subst x t t1)
    | tm_true        => tm_true
    | tm_false       => tm_false
  end.
```

Fig. 5 Coq function for term substitution.

Figure 6 presents the small-step operational semantics. Metavariable v denotes *values* and u denotes an *untrusted value*, as defined in Fig. 4. Most of the rules are standard, but some deserve attention. Rules $\texttt{Trust}_\texttt{c}$, $\texttt{Distrust}_\texttt{c}$, $\texttt{Distrust}_{\texttt{ca1}}$, $\texttt{Distrust}_{\texttt{ca2}}$, $\texttt{Trust}_\texttt{v}$ and $\texttt{Check}_\texttt{v}$ are rules for eliminating redundant uses of trust related constructs. For example, rule $\texttt{Distrust}_\texttt{c}$ specifies that distrusting a value twice is the same as distrusting it once. The other contraction rules have similar meanings.

We denote by $\rightarrow^*$ the reflexive, transitive closure of the small-step semantics. If a term e is not a value (primitive or untrusted), and e cannot be further reduced according to the rules of the small-step semantics, let's say that e is *stuck*. An example of a stuck term is *check*(*distrust true*); since *check* only reduces trusted values, this term does not reduce to any other term and it is not a primitive or untrusted value.

The main purpose of the type system is to rule out all programs that contain stuck expressions such as *check*(*distrust t*), for some term t.

The following lemma states the property that the proposed semantics is deterministic.

Lemma 1 (Determinism of small-step semantics) *For any e_1, e_2 and e_3, if $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$ then $e_2 = e_3$.*

Proof Induction over the derivation of $e_1 \rightarrow e_2$ and case analysis on the last rule used to conclude $e_1 \rightarrow e_3$.

[2] Several techniques can be used in order to deal with the problem of free variable capture on substitution, such as de Bruijn indexes [7], locally nameless representation [8] and high-order abstract syntax [14]. More about the formalisation of programming languages syntax with variable binding can be found at [1].

$$\frac{}{(\lambda\, x : \tau . e_1)\, v_2 \rightarrow [x \mapsto v_2]\, e_1}\ (\texttt{App})$$

$$\frac{}{(\lambda\, x : \tau . e_1)\, u_2 \rightarrow [x \mapsto u_2]\, e_1}\ (\texttt{App}_u)$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}\ (\texttt{App}_1)$$

$$\frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}\ (\texttt{App}_2)$$

$$\frac{e_2 \rightarrow e_2'}{u_1\ e_2 \rightarrow u_1\ e_2'}\ (\texttt{App}_{2u})$$

$$\frac{e \rightarrow e'}{\texttt{trust}\ e \rightarrow \texttt{trust}\ e'}\ (\texttt{Trust}_1)$$

$$\frac{e \rightarrow e'}{\texttt{distrust}\ e \rightarrow \texttt{distrust}\ e'}\ (\texttt{Distrust}_1)$$

$$\frac{e \rightarrow e'}{\texttt{check}\ e \rightarrow \texttt{check}\ e'}\ (\texttt{Check}_1)$$

$$\frac{}{\texttt{trust}(\texttt{distrust}\ v) \rightarrow \texttt{trust}\ v}\ (\texttt{Trust}_c)$$

$$\frac{}{\texttt{distrust}(\texttt{distrust}\ v) \rightarrow \texttt{distrust}\ v}\ (\texttt{Distrust}_c)$$

$$\frac{}{(\texttt{distrust}\ \ (\lambda x : \tau . e))\, v \rightarrow \texttt{distrust}\ \ ([x \mapsto v]\, e)}\ \texttt{Distrust}_{ca1}$$

$$\frac{}{(\texttt{distrust}\ \ (\lambda x : \tau . e))\, u \rightarrow \texttt{distrust}\ \ ([x \mapsto u]\, e)}\ \texttt{Distrust}_{ca2}$$

$$\frac{}{\texttt{trust}\ v \rightarrow v}\ (\texttt{Trust}_v)$$

$$\frac{}{\texttt{check}\ v \rightarrow v}\ (\texttt{Check}_v)$$

Fig. 6 Small-step Operational Semantics

$$\frac{s \preceq s' \quad \tau \leq \tau'}{\tau^s \leq \tau'^{s'}}\ (\texttt{S-}\tau)$$

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'}\ (\texttt{S-Arrow})$$

$$\frac{}{\tau \leq \tau}\ (\texttt{S-Refl})$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\texttt{S-Trans})$$

Fig. 7 Subtyping relation

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}^{tr}}\ (\texttt{T-True})$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}^{tr}}\ (\texttt{T-False})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\texttt{T-Var})$$

$$\frac{\Gamma \vdash e_1 : (\tau \rightarrow t^{s'})^s \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : t^{(s' \vee s)}}\ (\texttt{T-App})$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\, x : \tau . e : (\tau \rightarrow \tau')^{tr}}\ (\texttt{T-Abs})$$

$$\frac{\Gamma \vdash e : t^s}{\Gamma \vdash \texttt{trust}\ e : t^{tr}}\ (\texttt{T-Trust})$$

$$\frac{\Gamma \vdash e : t^s}{\Gamma \vdash \texttt{distrust}\ e : t^{dis}}\ (\texttt{T-Distrust})$$

$$\frac{\Gamma \vdash e : t^{tr}}{\Gamma \vdash \texttt{check}\ e : t^{tr}}\ (\texttt{T-Check})$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}\ (\texttt{T-Sub})$$

Fig. 8 Type System for λ-calculus with Trust Types

4 Type system

The type system proposed in [38] is based on the Curry version of simply typed λ-calculus. Since our main interest is the development of a certified type-checker and proofs about the type system, we use a variation of a Church like type system for the simply typed λ-calculus. The type system is defined in Fig. 8, as a set of rules for deriving judgements $\Gamma \vdash e : \tau$, meaning that term e has type τ, in typing context Γ (which contains type assumptions for the free variables in e). When e is a well-typed closed term, we omit Γ and simply write $\vdash e : \tau$.

$\Gamma, x : \tau$ is the standard notation for extending typing context Γ with a new assumption, after deleting from Γ any type assumption for x. We let $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$. Typing contexts are represented in Coq by lists of pairs of identifiers and types. Definitions of typing contexts, functions and properties over them (and their corresponding lemmas) are straightforward.

Trust annotations in types are subjected to a subtyping relation $s \preceq s'$, meaning that trust type s is a subtype of s', which is defined as the smallest reflexive relation (encoded as an inductive type) such that (the only non-reflexive element of relation $\preceq$ is) `tr` $\preceq$ `dis`.

Using the ordering relation over trust types, a subtyping relation over types is defined in Fig. 7.

The meaning of the typing rules for boolean constants, variables, subtyping and abstractions is standard. Constants and functions written by the programmer are considered as trusted, following [38]. The rules `T-Trust` and `T-Distrust` "cast" the trust type of an expression to `trust` and `untrust`, respectively, and rule `T-Check` checks whether an expression has a trusted type. In rule

```
Definition
   lubtrustty (x y : trustty):trustty :=
     match x with
      | Tr    => y
      | Dis   => Untrust
     end.

Definition
   updatetrustty
      (t : ty) (s : trustty) : ty :=
        match t with
          | ty_bool s' =>
            ty_bool (lub_trustty s s')
          | arrow l r s' =>
            arrow l r (lub_trustty s s')
        end.
```

Fig. 9 Functions for least upper bound over trust types

T-App, the annotated type of the actual argument is required to match the annotated type of the formal argument. This includes trustworthiness. The trust of the result of an application is the least upper bound of the trust of that function result type and the trust of the function type itself. We let $s \vee s'$ denote the maximum between the trust types s and s'.

In order to define the Coq inductive predicate for the typing relation, we need a function to compute the least upper bound of a pair of trust types. The definitions of the least upper bound and trust type update functions are given in Fig. 9. Function `lub_trustty` has a straightforward definition and `update_trusty` receives as parameters a type $\tau = t^s$ and a trust annotation s' and updates the trust annotation on type τ to $s \vee s'$.

We can now proceed to prove that the type system enjoys the type soundness property. In order to do this, we need to prove some lemmas about the typing relation, namely, inversion lemmas for the typing relation and canonical forms lemmas [39]. We will not state each one of these "infrastructure" lemmas here, but only sketch the key ones. Type soundness of the λ-calculus with trust types essentially guarantees that well typed terms do not stuck by checking trustworthiness of an untrusted term. This comes as a consequence of two properties: (1) progress, which guarantees that any well typed closed term reduces to a value; and (2) preservation, that ensures that term reduction preserves types. These results are formalised below.

Theorem 1 (Progress) *If* $\vdash e : \tau$, *then either* e *is a value, or it is an untrusted value, or there exists some term* e' *such that* $e \rightarrow e'$.

Proof Induction over the derivation of $\vdash e : \tau$ using canonical form lemmas.

Lemma 2 (Substitution lemma) *If* $\Gamma, x : \tau' \vdash e : \tau$ *and* e' *is such that* $\Gamma \vdash e' : \tau'$, *then* $\Gamma \vdash [x \mapsto e']\, e : \tau$.

Proof Induction over the structure of e using the corresponding inversion lemma for the typing relation in each case.

$$\frac{}{\Gamma \vdash^D \texttt{true} : \texttt{bool}^{tr}}\ (\texttt{D-True})$$

$$\frac{}{\Gamma \vdash^D \texttt{false} : \texttt{bool}^{tr}}\ (\texttt{D-False})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^D x : \tau}\ (\texttt{D-Var})$$

$$\frac{\Gamma \vdash^D e_1 : (\tau \rightarrow t^{s'})^s \quad \Gamma \vdash^D e_2 : \tau' \quad \tau' \leq_D \tau}{\Gamma \vdash^D e_1\ e_2 : t^{(s' \vee s)}}\ (\texttt{D-App})$$

$$\frac{\Gamma, x : \tau \vdash^D e : \tau'}{\Gamma \vdash^D \lambda x : \tau.\, e : (\tau \rightarrow \tau')^{tr}}\ (\texttt{D-Abs})$$

$$\frac{\Gamma \vdash^D e : t^u}{\Gamma \vdash^D \texttt{trust}\ e : t^{tr}}\ (\texttt{D-Trust})$$

$$\frac{\Gamma \vdash^D e : t^u}{\Gamma \vdash^D \texttt{distrust}\ e : t^{dis}}\ (\texttt{D-Distrust})$$

$$\frac{\Gamma \vdash^D e : t^{tr}}{\Gamma \vdash^D \texttt{check}\ e : t^{tr}}\ (\texttt{D-Check})$$

Fig. 10 Syntax Directed Type System for λ-calculus with Trust Types

Theorem 2 (Preservation) *If* $\vdash e : \tau$ *and* $e \rightarrow e'$, *then* $\vdash e' : \tau'$, *for some* τ' *s.t.* $\tau' \leq \tau$.

Proof Induction over the derivation of $\vdash e : \tau$ and case analysis over the last rule used to conclude $e \rightarrow e'$, using Lemma 2.

Corollary 1 (Type soundness) *If* $\vdash e : \tau$ *and* $e \rightarrow^* e'$, *then* e' *is not stuck (i.e.,* e' *is not of the form* `checke''`, *where* e'' *has an untrusted type*).

Proof Induction over $e \rightarrow^* e'$ using Theorems 1 and 2.

4.1 Syntax directed type system

The type system presented in Fig. 8 has the drawback of allowing applications of rule T-Sub at any place in the type derivation for some expression e. This makes this set of rules not immediately suitable for implementation. This section presents a syntax-directed version of the type system for the trust λ-calculus and proves its soundness and completeness with respect to the original type system.

The syntax directed type system is presented in Fig. 10 as a set of rules for deriving judgements of the form $\Gamma \vdash^D e : \tau$. The rules are almost the same as the ones in Fig. 8 except for the application rule that now includes, as a premise, a test of the subtyping relation $\tau' \leq_D \tau$, which represents a function that is true if and only if $\tau' \leq \tau$ holds. Termination, soundness and completeness of the subtyping test function follows the approach in [39] and their proofs are straightforward.

The next theorems state soundness and completeness of the syntax directed type system, and their proofs are in the companion Coq scripts.

Theorem 3 (Soundness) *If* $\Gamma \vdash^D e : \tau$, *then* $\Gamma \vdash e : \tau$

Proof Induction on the derivation of $\Gamma \vdash^D e : \tau$.

Theorem 4 (Completeness) *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash^D e : \tau'$ *for some* τ' *such that* $\tau' \leq \tau$.

Proof Induction on the derivation of $\Gamma \vdash e : \tau$.

Finally, we prove that the typing problem for the trust λ-calculus is decidable, that is, we prove that, given a typing context Γ and term e, it is decidable whether there exists a type τ such that $\Gamma \vdash^D e : \tau$. Because of the constructive nature of this proof, a certified algorithm for type checking an expression can be extracted from it. This theorem is stated as the following piece of Coq source code.

```
Theorem typecheck_dec :
   forall(e : term) (ctx : context),
     {t | has_type_alg ctx e t} +
      {forall t, ~ has_type_alg ctx e t}.
```

Predicate `has_type_alg` represents the syntax directed type system of Fig. 10. Intuitively, this theorem means that either there exists a type t such that `has_type_alg ctx e t` is provable or there is no such type `t`.

5 Erasure and simulation

As pointed out in [38], the type system for the λ-calculus with trust types is just a restriction of the classic (in our formalisation) Church type system for λ-calculus. This notion is formalised by an erasure function that converts terms, types and contexts from the trust calculus to simply typed λ-calculus.

Intuitively, the erasure function removes trust annotations from types, as well as trust constructs from terms. These functions are given in Fig. 11.

Following [38], we write these erasure functions using notation $|\phi|$, where ϕ is used as a term, type or context.

Lemma 3 (Lemma 12 of [38]) *For any trust types* τ *and* τ' *such that* $\tau \leq \tau'$ *we have that* $|\tau| = |\tau'|$.

Proof Induction over the derivation of $\tau \leq \tau'$.

The relationship between the trust calculus and original λ-calculus is stated by the next theorem, where the judgement $\Gamma \vdash^C e : t$ denotes the Church style type system for the λ-calculus presented in Fig. 12. The proof of this theorem uses some lemmas relating erasure and operations over typing contexts and types, that are necessary just for "lifting" the erasure functions. Since these lemmas are simple consequences of these function definitions, they are omitted here.

```
Fixpoint erase_ty (t : ty) : stlc_ty :=
  match t with
    | ty_bool _ => stlc_bool
    | arrow l r _ => stlc_arrow (erase_ty l)
                                (erase_ty r)
  end.

Fixpoint erase_term (t : term) : stlc_term :=
  match t  with
    | tm_false => stlc_false
    | tm_true => stlc_true
    | tm_var i => stlc_var i
    | tm_app l r => stlc_app (erase_term l)
                             (erase_term r)
    | tm_abs i T t
        => stlc_abs i (erase_ty T)
                      (erase_term t)
    | tm_trust t => erase_term t
    | tm_distrust t => erase_term t
    | tm_check t => erase_term t
  end.

Definition erase_context (ctx : context) :=
  map (fun p => match p with
                  | (i,t) => (i, erase_ty t)
                end) ctx.
```

Fig. 11 Erasure functions

$$\frac{}{\Gamma \vdash^C true : bool}\ \text{(TC-True)}$$

$$\frac{}{\Gamma \vdash^C false : bool}\ \text{(TC-False)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^C x : \tau}\ \text{(TC-Var)}$$

$$\frac{\Gamma, x : \tau \vdash^C e : \tau'}{\Gamma \vdash^C \lambda x : \tau . e : \tau \to \tau'}\ \text{(TC-Abs)}$$

$$\frac{\Gamma \vdash^C e_1 : \tau \to \tau' \quad \Gamma \vdash^C e_2 : \tau}{\Gamma \vdash^C e_1\, e_2 : \tau'}\ \text{TC-App}$$

Fig. 12 Church style type system for λ-calculus

Theorem 5 (Erasure) *If* $\Gamma \vdash e : \tau$, *then we have that* $|\Gamma| \vdash^C |e| : |\tau|$.

Proof Induction over $\Gamma \vdash e : \tau$.

For any well typed term, we can erase all `trust`, `distrust` and `check` constructs and evaluate the resulting term using a standard semantics of λ-calculus. In practice, this means that after type-checking a term, we can erase all trust related constructs and evaluate the term without any performance penalties [38]. This fact is expressed by the following theorem.

Theorem 6 (Simulation) *If* $\vdash e : \tau$ *and* $|e| \rightarrow^* e'$ *then there exists* e_1 *such that* $e \rightarrow^* e_1$ *and* $|e_1| = e'$.

Proof Induction over e.

6 Related work

Language support. The use of language based techniques for protecting information has as its most prominent example the security mechanism implemented by the Java run-time environment, which defines a set of security policies for applets [30,54].

Recently, an extension of Haskell was designed to deal with some language features that can be used to bypass the type system, referential transparency and module encapsulation [51]. The approach used by Safe Haskell is to classify modules and packages as safe, trust and unsafe based on its source code or in compiler pragmas that can be used to declare a possibly unsafe module as trustworthy. The Safe Haskell extension is available in the GHC compiler version 7.2 [45]. The authors used it to implement a web-based version of a Haskell interpreter, but no formal description of the safety inference process was given.

Type systems for security. The work of Volpano et al. was the first to use type systems to enforce security policies by a compiler [52]. They defined the lattice based analysis proposed by Denning in [15] as a type system for a prototypical imperative language with first order procedures. Their type system relies on polymorphism, thus allowing that commands and expression types depend on the context in which they occur. The calculus proposed by Ørbæk and Palsberg in [38] does not support polymorphism as well as our formalisation. In order to provide polymorphism in the λ-calculus with trust types, we need to deal with the combination of structural subtyping and parametric polymorphism that was developed in [22,47]. We leave this extension for future work.

Another proposal for a type system for ensuring security was described in [19], where a type system for a purely functional language was given and extensions like concurrency support were also discussed. The core calculus presented in [19] supports products and sums types and their related term constructs (projections and case expressions, respectively) and they prove type soundness results for closed expressions like the present work. Extending the formalisation of the λ-calculus with trust types with sum and product types is straightforward (e.g. [39, Chapter 15]).

The JFlow type system [35] is used in a language that extends Java with security types. Unlike other works in security type systems [52,19], JFlow applies the idea of a type system to ensure security policies over information flow on a full programming language, but no formal soundness proof is presented. A production compiler for this language is available [34] and was used in the development of a secure voting system [13]. Later, Simonet et al. uncovered a couple of flaws in JFlow type system [41], during his development of an extension of ML with types for information flow control, called Flow-Caml [42]. Following ML tradition of providing full type inference, Simonet developed a type inference algorithm for polymorphism and subtyping that was used as basis for Flow-Caml type system [47].

Barthe et al. [4] describe a security type system for a low level language with jumps and calls and prove that information flow types are preserved by the compilation from a high level imperative language. A mechanised proof that the type system proposed in Barthe's work is sound was given in [24], using the Coq proof assistant. Compared to the present work, Kammüler's formalisation deals with the problems of the lack of structure present in low level languages which make control flow more intricate than in a pure functional language.

As pointed out by Ørbæk and Palsberg in [38], security analysis focus on avoiding that classified information leaks out of a system to unprivileged users. The formalised type system ensures that untrustworthy information does not flow *into* the system. So, a trust type system can be seen as the "dual" of security type systems.

Use of Proof Assistants. Proof assistants have been used with success in several verification tasks. The Compcert project aims to develop a certified C compiler for embedded systems programming, using Coq proof assistant [27]. Several intermediate results were reported such as a mechanisation of a subset of C semantics [6], verification of the Compcert back-end [28] and a formalisation of C memory model [29]. The works of Chlipala describe compilers for small functional languages also using Coq proof assistant [10,11] and proofs about low level programs using separation logic [12]. The main differential of Chlipala works was the use of dependent types and proof automation to enable maintainability of proof scripts. Coq was also used with success in the formalisation of well known mathematical theorems such as the four colour theorem and the Feit–Thompson theorem by Georges Gonthier team at Microsoft-INRIA joint research center [17,18].

Agda is a dependently typed language [37], based on Martin Löf's type theory [31], that can be used as a proof assistant. Licata et al. [33] developed a library, called Aglet, for embedding secure-typed programming in Agda. Security policies are, in Aglet, proof terms that ensure some security policy. In order to ease the task of writing proof terms, this library provides an implementation of proof search procedure.

Isabelle/HOL [36] is a proof assistant that has been used in several projects like the formalisation of a general purpose operating system kernel, in which C code can be extracted from the produced Isabelle theories [25]. The Archive of

Formal Proofs [21] is a online repository for Isabelle developments that contains several formalisations of programming languages and mathematical theorems, such as the formal proofs of Volpano et al. type system for security [48,52].

A common issue in the formalisation of programming languages metatheory using proof assistants is how to deal with languages whose syntax supports variable binding. The main issue with variable binding is the possibility of variable capture in substitutions. Several techniques were developed to avoid variable capture such as de Bruijn indexes [7], high-order abstract syntax [14], locally nameless [8] and nominal logic [40]. For a detailed survey on binding techniques we refer the interested reader to [1]. In our work, we decided to avoid the extra complications of dealing with binding techniques (such as de Bruijn indexes and high-order abstract syntax) considering the formalisation of typing properties for closed terms only. There is a library for using the locally nameless approach for Coq proof assistant [9], but it relies on nonconstructive features that would not allow us to extract a verified type-checker from the developed formalisation.

7 Conclusion

We presented an axiom-free, fully constructive Coq formalisation of λ-calculus with trust types. The major differences between the original formulation of the trust λ-calculus and its presentation in this work is the use of a small-step semantics, instead of a reduction semantics, and a Church style, instead of a Curry style, type system. This allowed us to give concise proofs of type soundness, erasure and simulation theorems.

We also presented a syntax directed formulation of the original type system, that is sound and complete with respect to the former. Decidability of type checking is proved using this syntax directed version and a correct type checker can be extracted from this proof.

Future directions for extending this work include: (1) a reviewed formalisation which deals with free variables capture on substitutions, that could be used as a basis for the formalisation of other properties of the calculus; (2) modifying the type system in order to account for polymorphism; (3) formalising the type inference problem for the calculus and extracting a certified type inferencer; and (4) extending the calculus to a high level functional language.

References

1. Aydemir BE, Charguéraud A, Pierce BC, Pollack R, Weirich S (2008) Engineering formal metatheory. In: Necula GC, Wadler P (eds) POPL. ACM, New York, pp 3–15
2. Barendrecht HP: The Lambda calculus: its syntax and semantics, studies in logic and the foundations of mathematics, vol 103. Elsevier, New York (1984)
3. Barthe G, Dufay G, Jakubiec L, de Sousa SM (2002) A formal correspondence between offensive and defensive javacard virtual machines. In: Cortesi A (ed) VMCAI, Lecture Notes in Computer Science, vol 2294. Springer, Berlin, pp 32–45
4. Barthe G, Rezk T, Basu A (2007) Security types preserving compilation. Computer Lang Syst Struct 33(2):35–59
5. Bertot Y, Castéran P (2004) Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions. In: Texts in theoretical computer science. Springer, New York
6. Blazy S, Leroy X (2009) Mechanized semantics for the clight subset of the C language. J Autom Reason 43(3):263–288
7. de Bruijn N (1972) Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church–rosser theorem. Indag Math (Proceedings) 75(5): 381–392. doi:10.1016/1385-7258(72)90034-0
8. Charguéraud A (2012) The locally nameless representation. J Autom Reason 49(3):363–408
9. Arthur C (2013) Locally nameless Coq library. http://www.chargueraud.org/softs/ln/
10. Chlipala A (2007) A certified type-preserving compiler from lambda calculus to assembly language. In: Ferrante J, McKinley KS (eds) PLDI. ACM, New Yok, pp 54–65
11. Chlipala A (2010) A verified compiler for an impure functional language. In: Hermenegildo MV, Palsberg J (eds) POPL. ACM, New York, pp 93–106
12. Chlipala A (2011) Mostly-automated verification of low-level programs in computational separation logic. In: Hall MW, Padua DA (eds) PLDI. ACM, New York, pp 234–245
13. Clarkson MR, Chong S, Myers AC (2008) Civitas: toward a secure voting system. In: IEEE symposium on security and privacy. IEEE Computer Society, Los Alamitos, pp 354–368
14. Crary K, Harper R (2006) Higher-order abstract syntax: setting the record straight. SIGACT News 37(3):93–96
15. Denning DE (1976) A lattice model of secure information flow. Commun ACM 19(5):236–243
16. Dybvig RK (2009) The Scheme Programming Language, 4th edn. MIT Press, Cambridge
17. Gonthier G (2007) The four colour theorem: engineering of a formal proof. In: Kapur D (ed) ASCM, Lecture notes in computer science, vol 5081. Springer, Heidelberg, p 333
18. Gonthier G (2013) Engineering mathematics: the odd order theorem proof. In: Giacobazzi R, Cousot R (eds) POPL. ACM, New York, pp 1–2
19. Heintze N, Riecke JG (1998) The slam calculus: programming with secrecy and integrity. In: MacQueen DB, Cardelli L (eds) POPL. ACM, New York, pp 365–377
20. Hindley JR, Seldin JP (2008) Lambda-calculus and combinators: an introduction, 2nd edn. Cambridge University Press, New York
21. Isabelle Team (2013) The archieve of formal proofs. http://afp.sourceforge.net/
22. Jones M (1994) Qualified types: theory and practice. PhD thesis. Distinguished Dissertations in Computer Science. Cambridge University Press
23. Jones SP (2003) Haskell 98 language and libraries: the revised report
24. Kammüller F (2008) Formalizing non-interference for a simple bytecode language in Coq. Formal Asp. Comput. 20(3):259–275
25. Klein G, Andronick J, Elphinstone K, Heiser G, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2010) seL4: formal verification of an operating-system kernel. Commun ACM 53(6):107–115
26. Kothari S, Caldwell J (2010) A machine checked model of idempotent mgu axioms for lists of equational constraints. In: Fernandez M (ed) Proceedings 24th international workshop on unification, EPTCS, vol 42. pp 24–38

27. Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 52(7):107–115
28. Leroy X (2009) A formally verified compiler back-end. J Autom Reason 43(4):363–446
29. Leroy X, Blazy S (2008) Formal verification of a C-like memory model and its uses for verifying program transformations. J Autom Reason 41(1):1–31
30. Lindholm T, Yellin F (1999) Java virtual machine specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
31. Martin-Löf P (1984) Intuitionistic type theory. Bibliopolis, Naples
32. McBride C (2003) First-order unification by structural recursion. J Funct Program 13(6):1061–1075
33. Morgenstern J, Licata DR (2010) Security-typed programming within dependently-typed programming. In: Proceedings of the International Conference on Functional Programming.
34. Myers A et al (1998) Jif Compiler. http://www.cs.cornell.edu/jif/
35. Myers AC (1999) Jflow: practical mostly-static information flow control. In: Appel AW, Aiken A (eds) POPL. ACM, New York, pp 228–241
36. Nipkow T, Paulson, LC, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic. In: LNCS, vol 2283. Springer, Heidelberg
37. Norell U (2009) Dependently typed programming in Agda. In: Kennedy A, Ahmed A (eds) TLDI. ACM, New York, pp 1–2
38. Ørbæk P, Palsberg J (1997) Trust in the lambda-calculus. J Funct Program 7(6):557–591
39. Pierce BC (2002) Types and programming languages. MIT Press, Cambridge
40. Pitts AM (2003) Nominal logic, a first order theory of names and binding. Inf Comput 186(2):165–193
41. Pottier F, Simonet V (2002) Information flow inference for ML. In: Proceedings of the 29th ACM symposium on principles of programming languages. In: (POPL'02). Portland, Oregon, pp 319–330
42. Pottier F, Simonet V (2003) Information flow inference for ML. ACM transactions on programming languages and systems 25(1):117–158
43. Ribeiro R, et al (2013) A formalization of a lambda-calculus with trust types—on-line repository. https://github.com/rodrigogribeiro/trust-calculus
44. Rimsa A, d'Amorim M, Pereira FMQ (2011) Tainted flow analysis on e-ssa-form programs. In: Knoop J (ed) CC, Lecture notes in computer science, vol 6601. Springer, Berlin, pp 124–143
45. Jones SP et al (1998) GHC—the Glasgow Haskell Compiler. http://www.haskell.org/ghc/
46. Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas n Commun 21(1):5–19
47. Simonet, V (2003) Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In: Ohori A (ed) APLAS, Lecture notes in computer science, vol 2895. Springer, Heidelberg, pp 283–302
48. Snelting G, Wasserrab D (2008) A correctness proof for the Volpano–Smith security typing system. Isabelle Archive of Formal Proofs. http://afp.sourceforge.net/entries/VolpanoSmith.shtml
49. Sørensen M, Urzyczyn P (2006) Lectures on the Curry–Howard isomorphism. No. v. 10 in studies in logic and the foundations of mathematics. Elsevier, Amsterdam.
50. Team IO. Objective Caml (OCaml) programming language website. http://caml.inria.fr/
51. Terei D, Mazires D, Marlow S, Jones SP (2012) Safe Haskell. In: Haskell '12: proceedings of the fifth ACM SIGPLAN symposium on Haskell. ACM, New York
52. Volpano D, Irvine C, Smith G (1996) A sound type system for secure flow analysis. J Comput Secur 4(2–3):167–187
53. Volpano D, Smith G (1997) A type-based approach to program security. In: Lecture notes in computer science, vol 1214, (chap. 48). Springer, Berlin / Heidelberg, Berlin/Heidelberg, pp. 607–621. doi:10.1007/BFb0030629
54. Wallach DS, Appel AW, Felten EW (2000) Safkasi: a security mechanism for language-based systems. ACM Trans Softw Eng Methodol 9(4):341–378. doi:10.1145/363516.363520