Noname manuscript No.
(will be inserted by the editor)

Mechanized Metatheory for a λ-Calculus with Trust Types

Rodrigo Ribeiro · Carlos Camarão · Lucília Figueiredo

Received: date / Accepted: date

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 formalize 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 formalization 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 [?,?].

One of such traditional approaches to protect data confidentiality is access control: privileges are required to access files or objects containing confidential data. 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, we cannot ensure that confidentiality is maintained. In order to guarantee that information is used only in accordance with relevant policies, it is necessary to analyze how information flows within the program. Since modern computing systems are complex artefacts, a form of automating such analysis is required [?].

A promising approach has been recently developed, which consists on the use of type systems in order to control information flow in software [?]. 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 [?,?,?,?].

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

1. type soundness, using a standard small-step call-by-value semantics;

Rodrigo Ribeiro, Carlos Camarão
Instituto de Ciências Exatas, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. E-mail: {rribeiro,camarao}@dcc.ufmg.br.

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

2. erasure and simulation theorems [?, Sections 3.3 and 3.4];
3. decidability of type checking. From this proof we extract a certified type checker for the language.

The developed formalization is axiom free and has approximately 1400 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 omited for brevity, when they are trivial. The Coq source code of this work is available at trust-calculus github repository.

The rest of this paper is organized as follows. Section ?? presents a brief introduction to the Coq proof assistant and its features used in our formalization. Section ?? briefly reviews the syntax and defines a small-step semantics for the λ-calculus with trust types. Section ?? presents the non-syntax directed type system for the λ-calculus with trust, as proposed in [?], 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. We also prove that the typing problem for this calculus is decidable. Section ?? presents related work and Section ?? concludes.

2 A Taste of Coq Proof Assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC)[?], 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 [?]. 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.

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 Figure ??.

The source code in Figure ?? shows some basic features of the Coq proof assistant — types, functions and proof definitions. In this example, a new inductive type is firstly 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`.

```
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_0_r : forall n, plus n 0 = n.
Proof.
   intros n.
   induction n as [| n'].
   (**Case n = 0**)
      reflexivity.
   (**Case n = S n' **)
      simpl.
      rewrite -> IHn'.
      reflexivity.
Qed.
```

Fig. 1 Sample Coq code

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 allowing the declaration of inductive types and functions, we can define and prove theorems in Coq. Figure ?? 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 0 = 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 a interactive section of Coq (after enunciation of theorem `plus_0_r`), we must prove the following goal:

```
 ============================
   forall n : nat, plus n 0 = 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

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

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 trivally 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 sub-term 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 very 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 [?], Haskell [?] or Scheme [?]. 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 [?] 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 etc. All other data are considered untrusted [?].

Trust analysis is specially important in web applications, were 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 dynamicaly generated page [?]. Through this vulnerability, it is possible to inject JavaScript code to steal cookies, in order to acquire session privileges. Such 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 [?] 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 Figure **??**, where meta-variable 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. The translation of the type syntax to a Coq inductive type is straightforward, and is also presented in Figure **??**.

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 **??** defines the syntax of terms and the corresponding Coq data type.

The syntax of types and terms used in our formalization is identical to [?], 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.

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

```
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

Allowing non-annotated λ-abstractions characterizes a type inference problem that would require a formalization of a unification algorithm. The formalization of a unification algorithm has been studied elsewhere [?,?]. We let a formalization 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 proving progress and preservation theorems [?]. This differs from the approach addopted in [?], where the semantics of the trust λ-calculus is formalized using a reduction semantics, with no predefined order of evaluation, and the Church-Rosser property and a Subject Reduction Theorem are proved [?,?].

Let us firstly 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 meta-variable v) are boolean constants and λ-abstractions. An untrusted value (represented by meta-variable 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 Figure ??. Corresponding Coq definitions for values are straightforward predicate definitions over `term`.

$$\begin{array}{ll} v ::= & \mathit{true} \\ & \mid \mathit{false} \\ & \mid \lambda x : \tau . e \\ u ::= & \mathit{distrust}\ v \end{array}$$

```
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

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[2], that follows the standard definition of capture free substitution [?,?].

The Coq function presented in Figure ?? 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.

Figure ?? presents the small-step operational semantics. Most of its rules are standard, but some deserve attention. Rules $\texttt{Trust}_c$, $\texttt{Distrust}_c$, $\texttt{Distrust}_{ca1}$, $\texttt{Distrust}_{ca2}$, $\texttt{Trust}_v$ and $\texttt{Check}_v$ are rules for eliminating redundant uses of trust related constructs. For example, rule $\texttt{Distrust}_c$ 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

[2] The notion of free and bound variables is well-known. See e.g. [?], section 1B.

```
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.

(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 an 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 $e_1 \rightarrow e_3$.

4 Type System

The type system proposed in [?] 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 Figure ??, 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).

Notation $\Gamma, x : \tau$ is the standard notation for extending typing context Γ with a new assumption, after deleting from Γ any type assumption for x; and we let $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$. Typing contexts are represented

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6 Small-step Operational Semantics

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 reflexive relation such that `trust` $\preceq$ `distrust` (the only non-reflexive element of relation $\preceq$ is `trust` $\preceq$ `distrust`).

Using the ordering relation over trust types, we can define a subtyping relation over types. For any types τ and τ', we say that $\tau \leq \tau'$ iff: 1) $\tau = \texttt{bool}^s$, $\tau' = \texttt{bool}^{s'}$ and $s \preceq s'$; 2) $\tau = \tau_1 \rightarrow \tau_2$, $\tau' = \tau_1' \rightarrow \tau_2'$, $\tau_1' \leq \tau_1$ and $\tau_2 \leq \tau_2'$. This subtyping relation is defined in Figure **??**.

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 [**?**]. The rules `T-Trust` and `T-Distrust` "cast" the trust type of an expression to `trust` and `untrust` respectively and rule

$$\frac{s \preceq s'}{\texttt{bool}^s \leq \texttt{bool}^{s'}}\ (\texttt{S-Bool})$$

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

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

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

Fig. 7 Subtyping Relation

`T-Check` checks whether an expression has a trusted type. In rule `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 least upper bound between the trust types s and s'.

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

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

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

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

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

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

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

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

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

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

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 Figure **??**. 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 $s \vee s'$.

```
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

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 [?]. We will not state each one of these "infrastructure" lemmas here, but only sketch the key ones.

Theorem 1 (Progress) *If $\Gamma \vdash e : \tau$, 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 $\Gamma \vdash e : \tau$ 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.

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

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

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

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

4.1 Syntax Directed Type System

The type system presented in Figure **??** 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. In this section, we develop a syntax-directed version of the type system for the trust λ-calculus and prove its soundness and completeness with respect to the original type system.

The syntax directed type system is presented in Figure **??**, 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 Figure **??**, 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 [?] 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.

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

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

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

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

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

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

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

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

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

Theorem 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 $\Gamma \vdash^{D} e : \tau$. Due to 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 Figure **??**. 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 [**?**], the type system for the λ-calculus with trust types is just a restriction of the classic (in our formalization) Church type system for λ-calculus. This notion is formalized by an erasure function that converts terms, types and contexts from the trust calculus to bare λ-calculus.

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

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

Lemma 3 (Lemma 12 of [?]) *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 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 Figure **??**. The proof of this theorem uses some lemmas relating erasure and operations over typing contexts and types. These lemmas are necessary just for "lifting" the erasure functions. Since these lemmas are simple consequences of these function definitions, they are omitted here.

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

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

```
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

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

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

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

Fig. 12 Church-Style Type System for λ-calculus

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 [**?**]. This fact is expressed by the following theorem.

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

Proof Induction over e.

6 Related Work

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

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 [?]. 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 [?]. 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. Volpano et. al. was the first to use type systems to enforce security policies by a compiler [?]. They defined the lattice based analysis proposed by Denning in [?] 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. Another proposal for a type system for ensuring security was described in [?], where a type system for a purely functional language was given. The JFlow type system [?] is used in a language that extends Java with security types. A production compiler for this language is available [?] and was used in the development of a secure voting system [?].

Barthe et. al. [?] 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. A mechanized proof of Barthe's work was given in [?], using the Coq proof assistant.

As pointed by Ørbæk and Palsberg in [?], security analysis focus on avoiding that classified information leaks out of a system to unprivileged users. The formalized 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. The use of proof assistants for mechanizing programming language metatheory has been the subject of extensive research in several directions [?,?,?,?,?,?,?]. Successful applications of proof assistants in programming languages are the Compcert verified C compiler [?] and formalizations of Java virtual machines [?].

7 Conclusion

We presented an axiom-free, fully constructive Coq formalization of λ-calculus with trust types. The use of a small-step semantics, instead of a reduction semantics, and a Church-style, instead of a Curry-style, type system are the major differences between the present work and its original formulation. 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 the syntax directed version and a correct type checker can be extracted from this proof.

The complete formalization has near 1.400 lines of Coq code and can be found at trust-calculus github repository.