Trust in the λ-calculus

Jens Palsberg
Peter Ørbæk

BRICS Report Series RS-95-31

ISSN 0909-0878 June 1995

Trust in the λ-calculus

Jens Palsberg Peter Ørbæk

BRICS[*]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

This paper introduces trust analysis for higher-order languages. Trust analysis encourages the programmer to make explicit the trustworthiness of data, and in return it can guarantee that no mistakes with respect to trust will be made at run-time. We present a confluent λ-calculus with explicit trust operations, and we equip it with a trust-type system which has the subject reduction property. Trust information in presented as two annotations of each function type constructor, and type inference is computable in $O(n^3)$ time.

1 Introduction

With todays popular client-server software, server software needs to be very careful about the input they get from possibly untrustworthy clients. While writing server software one needs to be permanently aware of which data should be trusted (e.g., data coming from local databases or from authenticated clients) and which data cannot be trusted, such as input coming from unauthenticated clients.

Trust analysis aims to help the programmer write secure systems that cannot as easily be spoofed into trusting untrustworthy data. On one hand trust analysis enforces a certain discipline on the programmer, on the other hand it offers some guarantees that the software doesn't mistakenly trust untrustworthy data.

In [14] the second author introduced the concept of trust analysis for a simple first order imperative language using abstract interpretation. This paper extends trust analysis to a language with higher order functions.

[*]Basic Research in Computer Science, Centre of the Danish National Research Foundation. E-mail: {palsberg,poe}@daimi.aau.dk.

The remainder of the paper is structured as follows. First we give some intuitions about the intended program analysis, the semantics of our example language, and the type system. Then we present an extension of the λ-calculus together with an operational reduction semantics. The semantics is shown to have the Church-Rosser property. We define our static trust analysis in terms of a type system. The type system is shown to have the Subject Reduction property with respect to the reduction rules of the semantics. We then relate our type system to the classical Curry type system for λ-calculus and see that our system is a proper restriction of Currys system in that it accepts *fewer* programs. We also state some results relating reductions in our semantics with reductions in λ-calculus and finally we prove that well-typed terms are strongly normalizing. Then a type inference algorithm is presented and proved correct with respect to the type system. Finally we discuss how to extend the type system to handle recursion and polymorphism, and we relate trust analysis to other program analyses.

1.1 Motivation

Suppose we are writing a program using a patient database and assume the database contains sensitive data that should be accessible only to a few privileged people. In our library we are given two functions:

$$\begin{array}{ll} \mathsf{see\text{-}some}\colon & \mathsf{PatientId} \longrightarrow \mathsf{Info} \\ \mathsf{see\text{-}all}\colon & \mathsf{PatientId} \longrightarrow \mathsf{Info}. \end{array}$$

The see-some function returns non-sensitive information related to a patient, whereas see-all returns all recorded information about a patient.

With ordinary types these two functions seems to be interchangeable, and the lazy programmer may choose to use see-all at all times because it's just plain easier to remember one function than two. Our aim is to force the programmer to distinguish these two functions. In the type system presented in Section 3 the typings would be:

$$\begin{array}{ll} \mathsf{see\text{-}some}\colon & \mathsf{PatientId} \xrightarrow{\top\ \top} \mathsf{Info}\ \#\ \bot \\ \mathsf{see\text{-}all}\colon & \mathsf{PatientId} \xrightarrow{\bot\ \bot} \mathsf{Info}\ \#\ \bot\ . \end{array}$$

Each function (and in general each expression) is associated with a decorated type and an annotation about the trustworthiness of the value of that expression. An expression having annotation $\bot$ (bottom) is treated as trusted whereas the annotation $\top$ (top) means that the value cannot be trusted. Expressions of function type $s \xrightarrow{u\ v} t$ take arguments of decorated type s and these arguments must be at least as trustworthy as stated by the decoration u. The result of applying the function will have decorated type t, and the result will be trustworthy only if v is *and* the function applied is itself trusted.

With this in mind calls to see-all would not be allowed by the type system unless the actual argument had a *trustworthy* value, i.e., one derived solely from other trustworthy values or explicitly blessed into being trusted.

In order to allow programs to implement validity checks on data that subsequently should be regarded as trusted, our language includes a trust construct that behaves as the identity with respect to values, but informs the analysis that the result should henceforth be regarded as trustworthy. Symmetrically, there is a distrust construct that "curses" data into being untrusted.

The programmer is thus forced to make explicit when she needs trustworthy data and when untrusted data will do. Since raising of trustworthiness has to be made explicit in the program it can be better controlled, and we minimize the risk of using untrustworthy data (provided by some villain) unintentionally where trusted data are needed.

There is a third construct in our language: check, which intuitively aborts evaluation if the argument is not trustworthy. This would be what forces the type of see-all above. One might imagine the definition see-all looking like (ML'ish syntax):

```
fun see-all pid =
    let tpid = check pid
    in ...
```

We want a static analysis that can guarantee the programmer that no mistakes concerning trust exists in the program, i.e., that no untrustworthy data reach a check. Our decorated type system keeps explicit track of the trust "tags" together, and in parallel with, the ordinary type of values, and there are *no* trust tags around during run-time.

With all of this under our belts, we may now consider the following program fragment:

$$\text{credentialsOK: Cred} \xrightarrow{\perp\ \perp} \text{Bool} \# \perp$$

```
fun validateID credentials id g f =
    if credentialsOK credentials then // if we are credible
      g (trust id)                    // call the "good" continuation
    else
      f id                            // call "failure" continuation
    endif

... let info = validateID credentials id see-all see-some
    in ...
```

Assuming credentials is trusted and id untrusted, validateID would get the type:

$$\text{Cred} \xrightarrow{\perp\ \perp} \text{PatientId} \xrightarrow{\top\ \perp} (\text{PatientId} \xrightarrow{\perp\ \perp} \text{Info}) \xrightarrow{\perp\ \perp} (\text{PatientId} \xrightarrow{\top\ \top} \text{Info}) \xrightarrow{\perp\ \top} \text{Info}\ .$$

$$E, F, G, H \quad ::= \quad x \mid \lambda x.E \mid EE \mid \mathsf{trust}\ E \mid \mathsf{distrust}\ E \mid \mathsf{check}\ E$$

Figure 1: The syntax of expressions

If the program is written such that trust occurs only in such validate routines then the trust analysis will effectively guarantee us that see-all and similarly typed functions will only be called in situations where the process' credentials are in order.

Note the close coupling between checking credentials and the application of trust. This is a central point in the setup. It is here that the static analysis is coupled with the run-time properties of the program. Trust analysis deals with properties of data that cannot be decided from the values of that data. In some runs of the above program, a PatientId number might be trusted whereas in other runs of the *same* program that same number might be untrusted. This is a main point where trust analysis differs from many other program analyses. The differences between trust analysis and other program analyses is further discussed in Section 5.

In the remainder of the paper the above ideas are formalized in an extension of the λ-calculus.

2 Syntax and Semantics

This section presents the syntax and operational semantics of the trust language. Figure 1 defines the syntax of our language.

Variables, lambda abstraction and application behave as usual. Trust E is used to introduce trusted values in a program. One may view trust as a run-time tag on values that indicate the trustworthiness of the value. Symmetrically, distrust indicates untrusted values. The check construct will only reduce on trusted values, so evaluation may get *stuck* if an expression check(distrust E) occurs at some point during evaluation.

2.1 Reduction rules

The reduction (or evaluation) rules for the language are given in Figure 2. Stating $E \to E'$ means that there is a derivation of that reduction in the system.

There are three kinds of values around during reduction: trusted, distrusted and untagged. Untagged lambdas are treated as trusted program constants in the (Lambda Contraction) rules, since lambdas stem from the program text which the programmer is writing himself and they may therefore be trusted.

In order to facilitate the proof of the Church-Rosser property of the system, the reduction rules form a reflexive "one step" transition relation. This is inspired by the proof of Church-Rosser for the ordinary λ-calculus by Tait and Martin-Löf in [2, pp. 59–62].

$$E \to E \qquad \text{(Reflex)}$$

$$\frac{E \to E'}{\begin{array}{ccc} \lambda x.E & \to & \lambda x.E' \\ \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \\ \mathsf{check}\ E & \to & \mathsf{check}\ E' \end{array}} \qquad \text{(Sub)}$$

$$\frac{E \to \mathsf{trust}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \\ \mathsf{check}\ E & \to & \mathsf{trust}\ E' \end{array}} \qquad \text{(Trust Contraction)}$$

$$\frac{E \to \mathsf{distrust}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \end{array}} \qquad \text{(Distrust Contraction)}$$

$$\frac{E \to \mathsf{check}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{check}\ E' \\ \mathsf{check}\ E & \to & \mathsf{check}\ E' \end{array}} \qquad \text{(Check Contraction)}$$

$$\frac{E \to \lambda x.E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \lambda x.E' \\ \mathsf{check}\ E & \to & \lambda x.E' \end{array}} \qquad \text{(Lambda Contraction)}$$

$$\frac{E \to E' \qquad F \to F'}{\begin{array}{ccc} EF & \to & E'F' \\ (\lambda x.E)F & \to & E'[F'/x] \\ (\mathsf{distrust}\ (\lambda x.E))F & \to & \mathsf{distrust}\ E'[F'/x] \end{array}} \qquad \text{(Application)}$$

Figure 2: The reduction rules.

$$\frac{E \to const}{\begin{array}{c}\mathsf{trust}\ E \to const \\ \mathsf{check}\ E \to const\end{array}} \qquad \text{(Constant)}$$

$$\begin{array}{rl} \mathsf{T} \equiv \mathsf{K} \equiv \lambda xy.x & \text{(True)} \\ \mathsf{F} \equiv \lambda xy.y & \text{(False)} \\ \mathsf{if}\ E\ \mathsf{then}\ F\ \mathsf{else}\ G \equiv EFG & \text{(If)} \end{array}$$

$$\frac{E \to E' \quad F \to F'}{\begin{array}{c}\mathsf{if}\ \mathsf{T}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* E' \\ \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* F' \\ \mathsf{if}\ \mathsf{distrust}\ \mathsf{T}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* \mathsf{distrust}\ E' \\ \mathsf{if}\ \mathsf{distrust}\ \mathsf{F}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* \mathsf{distrust}\ F'\end{array}} \qquad \text{(If)}$$

Figure 3: Example rules.

The contraction rules exist to eliminate redundant uses of our new constructs in the calculus. For example, trusting an expression twice is the same as trusting it once (the first (Trust Contraction) rule) and checking the trustworthiness of an expression followed by explicitly trusting it is the same as just checking the expression. Checking an explicitly trusted expression succeeds and yields a trusted expression. Note that there is no rule contracting distrust(check E) since this would allow the removal of the check on the trustworthyness of E.

In the following we always consider equality of terms modulo α-renaming. Terms for which no further reductions are possible, i.e $\neg\exists F \neq E : E \to F$ are said to be in *normal form.* There are *proper* and *improper* normal forms. A normal form containing a subterm: check(distrust E') is said to be improper. All other normal forms are proper.

As an example of how to extend the language with usual programming constructs, we show in Figure 3 how a reduction rule for program constants would look and the derived rules we get for if-then-else with the usual coding of booleans in the λ-calculus. Notice how the (Constant) rules are patterned after the (Lambda Contraction) rules. In Section 5 we also show how to encode a rec construct in the language.

2.2 Church-Rosser

The Church-Rosser (or confluence) theorem for a reduction system states that for any term, if the term can reduce to two different terms there exists a successor term such that both of the two reduced terms can further reduce to that common successor. A corollary of this is that a normal form is unique if it exists.

We write $\to^*$ for the reflexive transitive closure of the reduction relation $\to$.

Theorem 1 (Church-Rosser) *For expressions E, F and G. If $E \to^* F$ and $E \to^* G$ then there is an expression H such that $F \to^* H$ and $G \to^* H$.*

Proof. By the Diamond lemma below (Lemma 7) and Lemma 3.2.2 of [2]. □

Lemma 2 *If $E \to F$ and E has a certain structure then some conditions on the structure of F hold, as made explicit below.*

- *If* $\mathsf{trust}\ E_1 \to F$ *then* $F = \alpha\ F_1$ *where* $\alpha \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$.
- *If* $\mathsf{check}\ E_1 \to F$ *then* $F = \alpha\ F_1$ *where* $\alpha \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$.
- *If* $\mathsf{distrust}\ E_1 \to F$ *then F is of the form* $\mathsf{distrust}\ F_1$.
- *If* $\lambda x.E_1 \to F$ *then F is of the form* $\lambda x.F_1$.

Proof. In each case by inspection of the reduction rules. □

Lemma 3 (Trust/Check Identity) *Let* $\alpha \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$.
If $E \to \alpha\ E_1$ *then* $\mathsf{check}\ E \to \alpha\ E_1$ *and* $\mathsf{trust}\ E \to \alpha\ E_1$

Proof. By inspection of the reduction rules, especially the contraction rules. □

Lemma 4 (Symmetry) *Let* $\alpha, \beta \in \{\mathsf{trust}, \mathsf{distrust}\}$. *If* $\alpha\ E \to \alpha\ E'$ *then* $\beta\ E \to \beta\ E'$

Proof. By induction on the structure of the derivation of $\alpha\ E \to \alpha\ E'$, verifying that in each case there is also a corresponding rule for the opposite combination. □

Lemma 5 (Pre-Substitution) *If $E \to F$ then $G[E/x] \to G[F/x]$.*

Proof. By induction on the structure of G. This is essentially a consequence of the (Sub) rules and the first (Application) rule. □

Lemma 6 (Substitution) *If $E \to F$ and $G \to H$ then $E[G/x] \to F[H/x]$.*

Proof. By induction on the structure of the derivation of $E \to F$. If $F = E$ by the (Reflex) axiom, we must show that $E[G/x] \to E[H/x]$ given that $G \to H$. This is the Pre-Substitution lemma (5).

For all the rules except the (Application) case: Suppose that α, β, and γ are in the set $\{\mathsf{check}, \mathsf{trust}, \mathsf{distrust}, \lambda x.\}$ as appropriate, and β and γ may be empty as well. Assume the rule

$$\frac{E_1 \to \beta\ F_1}{E = \alpha\ E_1 \to \gamma\ F_1 = F}$$

is the last rule in the derivation of $E \to F$. By the induction hypothesis we get

$$E_1[G/x] \to (\beta\ F_1)[H/x] = \beta(F_1[H/x]).$$

Now $E[G/x] = \alpha\ E_1[G/x]$ and $F[H/x] = \gamma\ F_1[H/x]$, and we may now deduce

$$\frac{E_1[G/x] \to \beta\ F_1[H/x]}{\alpha\ E_1[G/x] \to \gamma\ F_1[H/x]}$$

as required. Of course, in the case of a lambda, if the bound variable is the one substituted for, nothing happens during substitution, i.e.

$$E[G/x] = E \to F = F[H/x]$$

as the only rule applicable to the case of $\alpha = \lambda x.$ is the (Sub) rule.

The (Application) cases: If $E = E_1 E_2 \to F_1 F_2 = F$, where $E_1 \to F_1$ and $E_2 \to F_2$ then the result follows directly from the induction hypothesis. Suppose the last rule in the derivation of $E \to F$ was $(x \neq y)$:

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\lambda y.E_1)E_2 \to F_1[F_2/y] = F}$$

By the induction hypothesis $E_1[G/x] \to F_1[H/x]$ and similarly for E_2. Also $E[G/x] = (\lambda y.E_1[G/x])(E_2[G/x])$ and

$$F[H/x] = (F_1[F_2/y])[H/x] = (F_1[H/x])[F_2[H/x]/y]$$

where the last equality depends on y not being free in H. This can be assured by α-renaming H. We may now deduce:

$$\frac{E_1[G/x] \to F_1[H/x] \quad E_2[G/x] \to F_2[H/x]}{(\lambda y.E_1[G/x])(E_2[G/x]) \to (F_1[H/x])[F_2[H/x]/y]}$$

Suppose the last rule in the derivation of $E \to F$ was:

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\lambda x.E_1)E_2 \to F_1[F_2/x] = F}$$

By the induction hypothesis, $E_2[G/x] \to F_2[H/x]$. Also $E[G/x] = (\lambda x.E_1)(E_2[G/x])$ and

$$F[H/x] = (F_1[F_2/x])[H/x] = F_1[F_2[H/x]/x].$$

Now

$$\frac{E_1 \to F_1 \quad E_2[G/x] \to F_2[H/x]}{(\lambda x.E_1)(E_2[G/x]) \to F_1[F_2[H/x]/x]}$$

as required. Two similar cases apply to the distrust $\lambda x.E$ case. $\square$

Lemma 7 (Diamond) *For expressions E, F and G. If $E \to F$ and $E \to G$ then there is an expression H such that $F \to H$ and $G \to H$.*

Proof. By induction on the derivation of $E \to F$ and $E \to G$ and by cases on how F and G must look depending on E.

Depending on E there are a number of applicable rules. In all cases (Reflex) and (Sub) are applicable.

1. $E = \lambda x.E_1$: none other.
2. $E = \textsf{trust}\ E_1$:
   (a) $E \to \textsf{trust}\ E_1'$ since $E_1 \to \textsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \lambda x.E_1'$ since $E_1 \to \lambda x.E_1'$. (Lambda Contraction)
   (c) $E \to \textsf{check}\ E_1'$ since $E_1 \to \textsf{check}\ E_1'$. (Check Contraction)
   (d) $E \to \textsf{trust}\ E_1'$ since $E_1 \to \textsf{distrust}\ E_1'$. (Distrust Contraction)
3. $E = \textsf{distrust}\ E_1$:
   (a) $E \to \textsf{distrust}\ E_1'$ since $E_1 \to \textsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \textsf{distrust}\ E_1'$ since $E_1 \to \textsf{distrust}\ E_1'$. (Distrust Contraction)
4. $E = \textsf{check}\ E_1$:
   (a) $E \to \textsf{trust}\ E_1'$ since $E_1 \to \textsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \textsf{check}\ E_1'$ since $E_1 \to \textsf{check}\ E_1'$. (Check Contraction)
   (c) $E \to \lambda x.E_1'$ since $E_1 \to \lambda x.E_1'$. (Lambda Contraction)
5. $E = (\lambda x.E_1)E_2$: $E \to E_1'[E_2'/x]$ since $E_1 \to E_1'$ and $E_2 \to E_2'$.
6. $E = (\textsf{distrust}\ \lambda x.E_1)E_2$: $E \to \textsf{distrust}\ E_1'[E_2'/x]$ since $E_1 \to E_1'$ and $E_2 \to E_2'$.

If $E \to F$ or $E \to G$ by (Reflex) then there is no problem, one may just use the rule applied in the other branch to get to the common successor. Case 1 is easy as well: there is only one applicable rule except (Reflex) namely (Sub).

The tables below map pairs of "outgoing" reductions to proofs of the corresponding case.

Case 2	2a	2b	2c	2d	(Sub)
2a		B	B	C	A
2b			B	C	A
2c				C	A
2d					D

Case 3	3a	3b	(Sub)
3a		G	F
3b			E

Case 4	4a	4b	4c	(Sub)
4a		B	B	A
4b			B	A
4c				A

For case 6 the argument is as follows: Here the last rules in the derivation of $E \rightarrow F$ and $E \rightarrow G$ were:

$$\frac{E_1 \rightarrow F_1 \quad E_2 \rightarrow F_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \rightarrow \mathsf{distrust}\ F_1[F_2/x] = F}\text{(Application)}$$

and

$$\frac{E_1 \rightarrow G_1 \quad E_2 \rightarrow G_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \rightarrow (\mathsf{distrust}\ \lambda x.G_1)G_2 = G}\text{(Sub)}$$

respectively. By the induction hypothesis there are H_1 and H_2 such that $F_1 \rightarrow H_1$, $G_1 \rightarrow H_1$ and $F_2 \rightarrow H_2$, $G_2 \rightarrow H_2$. So by the Substitution lemma (Lemma 6) (and (Sub)):

$$F = \mathsf{distrust}\ F_1[F_2/x] \rightarrow \mathsf{distrust}\ H_1[H_2/x] = H$$

and by (Application)

$$G = (\mathsf{distrust}\ \lambda x.G_1)G_2 \rightarrow \mathsf{distrust}\ H_1[H_2/x] = H.$$

Case 5 without $\mathsf{distrust}$ is similar.

In each of the cases below, the quest is to find an appropriate common successor H to F and G.

Case A. Let $\alpha \in \{\mathsf{trust}, \mathsf{check}\}$ and $\beta \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. The last rules of the derivation of $E \rightarrow F$ and $E \rightarrow G$ were

$$\frac{E_1 \rightarrow \beta\ F_1}{E = \alpha\ E_1 \rightarrow \beta\ F_1 = F}(\beta \text{ Contraction})$$

$$\frac{E_1 \rightarrow G_1}{E = \alpha\ E_1 \rightarrow \alpha\ G_1 = G}\text{(Sub)}$$

By the induction hypothesis there is an H_1 such that $G_1 \rightarrow H_1$ and $\beta\ F_1 \rightarrow H_1$ By Lemma 2 and the restriction on β; $H_1 = \gamma\ H_2$ where $\gamma \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. By the Trust/Check Identity lemma (Lemma 3), $G_1 \rightarrow \gamma\ H_2$ implies that $\alpha\ G_1 \rightarrow \gamma\ H_2 = H_1$. So we can use $H = H_1$.

Case B. Let $\alpha, \beta, \gamma \in \{\mathsf{trust}, \mathsf{check}, \lambda x\}$ as appropriate. The last rules used in the derivation of $E \rightarrow F$ and $E \rightarrow G$ are:

$$\frac{E_1 \rightarrow \beta\ F_1}{E = \alpha\ E_1 \rightarrow \beta\ F_1 = F}(\beta \text{ Contraction})$$

$$\frac{E_1 \rightarrow \gamma\ G_1}{E = \alpha\ E_1 \rightarrow \gamma\ G_1 = G}(\gamma \text{ Contraction})$$

By the induction hypothesis there is an H_1 such that $\beta\ F_1 \rightarrow H_1$ and $\gamma\ G_1 \rightarrow H_1$. We may now use H_1 as H.

Case C. Let $\alpha \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. The two last rules used in the derivation of $E \rightarrow F$ and $E \rightarrow G$ are:

$$\frac{E_1 \rightarrow \alpha\ F_1}{E = \mathsf{trust}\ E_1 \rightarrow \alpha\ F_1 = F}(\alpha \text{ Contraction})$$

$$\frac{E_1 \rightarrow \mathsf{distrust}\ G_1}{E = \mathsf{trust}\ E_1 \rightarrow \mathsf{trust}\ G_1 = G}(\text{Distrust Contraction})$$

By the induction hypothesis we know there exists H_1 such that $\alpha\ F_1 \rightarrow H_1$. Here Lemma 2 says that $H_1 = \beta\ H_2$ where $\beta \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$. Also by the induction hypothesis we have $\mathsf{distrust}\ G_1 \rightarrow H_1$. And here Lemma 2 says that $H_1 = \mathsf{distrust}\ H_2$! This is a contradiction so it cannot be the case that both $E_1 \rightarrow \alpha\ F_1$ and $E_1 \rightarrow \mathsf{distrust}\ G_1$.

Case D. The two last rules used in the derivation of $E \rightarrow F$ and $E \rightarrow G$ are:

$$\frac{E_1 \rightarrow \mathsf{distrust}\ F_1}{E = \mathsf{trust}\ E_1 \rightarrow \mathsf{trust}\ F_1 = F}(\text{Distrust Contraction})$$

$$\frac{E_1 \rightarrow G_1}{E = \mathsf{trust}\ E_1 \rightarrow \mathsf{trust}\ G_1 = G}(\text{Sub})$$

By the induction hypothesis there is an H_1 such that $\mathsf{distrust}\ F_1 \rightarrow H_1$ and $G_1 \rightarrow H_1$. By Lemma 2 $H_1 = \mathsf{distrust}\ H_2$ so

$$\frac{G_1 \rightarrow \mathsf{distrust}\ H_2}{\mathsf{trust}\ G_1 \rightarrow \mathsf{trust}\ H_2}(\text{Distrust Contraction})$$

By Symmetry (Lemma 4) $\mathsf{distrust}\ F_1 \rightarrow \mathsf{distrust}\ H_2$ implies $\mathsf{trust}\ F_1 \rightarrow \mathsf{trust}\ H_2$. So we can use $\mathsf{trust}\ H_2$ as H.

Case E. The two last rules used in the derivation of $E \longrightarrow F$ and $E \longrightarrow G$ are:

$$\frac{E_1 \longrightarrow \textsf{distrust } F_1}{E = \textsf{distrust } E_1 \longrightarrow \textsf{distrust } F_1 = F}\text{(Distrust Contraction)}$$

$$\frac{E_1 \longrightarrow G_1}{E = \textsf{distrust } E_1 \longrightarrow \textsf{distrust } G_1 = G}\text{(Sub)}$$

By the induction hypothesis there is an H_1 such that $\textsf{distrust } F_1 \longrightarrow H_1$ and $G_1 \longrightarrow H_1$. By Lemma 2 $H_1 = \textsf{distrust } H_2$. By (Distrust Contraction) $G_1 \longrightarrow \textsf{distrust } H_2$ implies $\textsf{distrust } G_1 \longrightarrow \textsf{distrust } H_2 = H_1$. So we use $H = H_1$ in this case.

Case F. The two last rules used in the derivation of $E \longrightarrow F$ and $E \longrightarrow G$ are:

$$\frac{E_1 \longrightarrow \textsf{trust } F_1}{E = \textsf{distrust } F_1 \longrightarrow \textsf{distrust } F_1 = F}\text{(Trust Contraction)}$$

$$\frac{E_1 \longrightarrow G_1}{E = \textsf{distrust } E_1 \longrightarrow \textsf{distrust } G_1 = G}\text{(Sub)}$$

By the induction hypothesis there is an H_1 such that $\textsf{trust } F_1 \longrightarrow H_1$ and $G_1 \longrightarrow H_1$. By Lemma 2 $H_1 = \alpha\ H_2$ where $\alpha \in \{\textsf{trust}, \textsf{check}, \lambda x.\}$.

If $\alpha = \textsf{trust}$ then we have $G_1 \longrightarrow \textsf{trust } H_2$ and $\textsf{trust } F_1 \longrightarrow \textsf{trust } H_2$ and by Symmetry $\textsf{distrust } F_1 \longrightarrow \textsf{distrust } H_2$. By (Trust Contraction) we also get $\textsf{distrust } G_1 \longrightarrow \textsf{distrust } H_2$, so here we may use $H = \textsf{distrust } H_2$.

If $\alpha = \textsf{check}$ or $\alpha = \lambda x.$ then by (Sub) we get $\textsf{distrust } G_1 \longrightarrow \textsf{distrust } (\alpha\ H_2)$. Since $\textsf{trust } F_1 \longrightarrow \alpha\ H_2$ one sees by inspection of the rules that for each α there is just one possible last rule for this reduction so we must have $F_1 \longrightarrow \alpha\ H_2$. Now by (Sub) we get $\textsf{distrust } F_1 \longrightarrow \textsf{distrust } (\alpha\ H_2)$. So here we may use $H = \textsf{distrust } (\alpha\ H_2)$.

Case G. The two last rules used in the derivation of $E \longrightarrow F$ and $E \longrightarrow G$ are:

$$\frac{E_1 \longrightarrow \textsf{distrust } F_1}{E = \textsf{distrust } E_1 \longrightarrow \textsf{distrust } F_1 = F}\text{(Distrust Contraction)}$$

$$\frac{E_1 \longrightarrow \textsf{trust } G_1}{E = \textsf{distrust } E_1 \longrightarrow \textsf{distrust } G_1 = G}\text{(Trust Contraction)}$$

By the induction hypothesis there is an H_1 such that $\textsf{distrust } F_1 \longrightarrow H_1$ and $\textsf{trust } G_1 \longrightarrow H_1$, but by Lemma 2 this is a contradiction so this case cannot arise.

This concludes the proof of the Diamond lemma. □

$$\begin{array}{lll} u, v, w & ::= & \top \mid \bot \\ s, t & ::= & \mathsf{base} \mid s \xrightarrow{u\ \ v} t \end{array}$$

Figure 4: Syntax of trust-types

3 The Type System

Our decorated type system is based on Currys simple monomorphic type system for the λ-calculus, also known as *simply typed* λ-calculus. Figure 4 shows the syntax of the trust-types. Recall from Section 1.1 that $\bot$ means that the value is trusted and $\top$ means that it is untrusted.

Trust-types are subject to a partial ordering $\leq$, defined on trusts as: $\bot \leq \top$ and extended to types such that two base types are ordered only if they are identical, and for arrow types:

$$s \xrightarrow{u\ \ v} t \leq s' \xrightarrow{u'\ \ v'} t' \text{ if and only if } s' \leq s,\ u' \leq u,\ v \leq v' \text{ and } t \leq t'$$

so argument types are ordered contravariantly. This is inspired by the work on structural subtyping by Mitchell [12] and others.

We denote by $\sqcup$ the least upper bound operation on the trust lattice. In Section 5 we discuss several extensions of the type system to cope with recursion, polymorphism and more general lattices.

3.1 Rules

A type assumption A is a partial function which takes a program identifier to a pair of its type and trust. Figure 5 shows the inference rules for the type system. A type judgment $A \vdash E : t \,\#\, u$ means that from the assumptions A we can deduce that the expression E has type t and is as trustworthy as u.

The rule for variables and the subtyping rule should give no surprises. A lambda expression occurring in the program is always known to have a function type thus we can mark all lambdas in the program as trusted. This means that we trust that a lambda will evaluate to a function, but it says nothing about the arguments and results of the function. That information is kept in the arrow type.

In the application rule, the trust of the actual argument is required to match the argument trust coded in the arrow type of the function. The trust of the result of the application is the least upper bound of the result-trust from the arrow type and the trust of the function itself. The intuition is that if we cannot trust the function, we cannot trust the result of applying it.

The three rules for trust, $\mathsf{distrust}$ and check show that they behave as the identity on the type. Trust makes any value trusted and $\mathsf{distrust}$ makes any value untrusted. Check E has a type only if E is trusted. This means that we cannot type improper

$$A \vdash x : t \,\#\, u \text{ if } x \in \mathrm{dom}(A) \text{ and } A(x) = \langle t, u \rangle \qquad \text{(Var)}$$

$$\frac{A \vdash E : t \,\#\, u \quad t \leq t' \quad u \leq u'}{A \vdash E : t' \,\#\, u'} \qquad \text{(Sub)}$$

$$\frac{A[x \mapsto \langle t, u \rangle] \vdash E_1 : s \,\#\, v}{A \vdash \lambda x.E_1 : t \xrightarrow{u \;\; v} s \,\#\, \bot} \qquad \text{(Lambda)}$$

$$\frac{A \vdash E_1 : s \xrightarrow{u \;\; v} t \,\#\, w \quad A \vdash E_2 : s \,\#\, u}{A \vdash E_1 E_2 : t \,\#\, v \sqcup w} \qquad \text{(App)}$$

$$\frac{A \vdash E_1 : t \,\#\, u}{A \vdash \mathsf{trust}\ E_1 : t \,\#\, \bot} \qquad \text{(Trust)}$$

$$\frac{A \vdash E_1 : t \,\#\, u}{A \vdash \mathsf{distrust}\ E_1 : t \,\#\, \top} \qquad \text{(Distrust)}$$

$$\frac{A \vdash E_1 : t \,\#\, \bot}{A \vdash \mathsf{check}\ E_1 : t \,\#\, \bot} \qquad \text{(Check)}$$

Figure 5: The type system.

$$\begin{tikzcd}
& \mathsf{distrust} \circ \mathsf{check} : t \xrightarrow{\perp\ \top} t \arrow[dl, no head] \arrow[dr, no head] & \\
\mathsf{check}\text{: } t \xrightarrow{\perp\ \perp} t \arrow[dr, no head] & & \mathsf{distrust}\text{: } t \xrightarrow{\top\ \top} t \arrow[dl, no head] \\
& \mathsf{trust}\text{: } t \xrightarrow{\top\ \perp} t &
\end{tikzcd}$$

Figure 6: The relationship between arrow types

normal forms and together with Subject Reduction (Theorem 9) this ensures the soundness of the type system.

Figure 6 shows the ordering of arrow types and how the constructs trust, $\mathsf{distrust}$ and check would fit into it.

3.2 Subject Reduction

The main result of this section is the Subject Reduction theorem. The theorem states that types are invariant under reduction.

Lemma 8 (Substitution) *If $A[x \mapsto \langle s, u\rangle] \vdash E : t \,\#\, v$ and $A \vdash F : s \,\#\, u$ then $A \vdash E[F/x] : t \,\#\, v$.*

Proof. By induction on the derivation of $A[x \mapsto \langle s, u\rangle] \vdash E : t \,\#\, v$. □

Theorem 9 (Subject Reduction) *If $A \vdash E : t \,\#\, u$ and $E \to F$ then $A \vdash F : t \,\#\, u$.*

Proof. By induction on the structure of the derivation of $E \to F$ and by cases on the structure of E. The (Reflex) case is trivial. In the (Sub) cases the result follows directly from the induction hypothesis. The type rule (Sub) is applicable in all cases, so when reasoning "backwards" (as in "when α E has type t then E *must* have type t'") we must take care to handle the case where the (Sub) type rule was used in between.

For the contraction rules we show just two illustrative cases, the remaining cases are extremely similar. Suppose the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to \mathsf{trust}\ F_1}{E = \mathsf{trust}\ E_1 \to \mathsf{trust}\ F_1 = F}\text{(Trust Contraction)}$$

By assumption we have $A \vdash \mathsf{trust}\ E_1 : t \,\#\, u$ so by the rules we must have $A \vdash E_1 : t' \,\#\, u'$ where $t' \leq t$. By the induction hypothesis we now get $A \vdash \mathsf{trust}\ F_1 : t' \,\#\, u'$ and again we must have $A \vdash F_1 : t'' \,\#\, u''$ where $t'' \leq t'$. Now by the (Trust) rule of the type system we get $A \vdash \mathsf{trust}\ F_1 : t'' \,\#\, \perp$ and finally by (Sub) we get $A \vdash \mathsf{trust}\ F_1 : t \,\#\, u$ as required.

Another case: Suppose the last rule used in the derivation of $E \longrightarrow F$ was

$$\frac{E_1 \longrightarrow \mathsf{distrust}\ F_1}{E = \mathsf{distrust}\ E_1 \longrightarrow \mathsf{distrust}\ F_1 = F}\text{(Distrust Contraction)}$$

By assumption we know $A \vdash \mathsf{distrust}\ E_1 : t \,\#\, u$ and therefore $u = \top$, so by the rules we must have $A \vdash E_1 : t' \,\#\, u'$ where $t' \leq t$. From the induction hypothesis we get $A \vdash \mathsf{distrust}\ F_1 : t' \,\#\, u'$. By the (Distrust) rule we must have $A \vdash F_1 : t'' \,\#\, u''$ where $t'' \leq t'$ and $u' = \top$. By the (Distrust) rule we now get $A \vdash \mathsf{distrust}\ F_1 : t'' \,\#\, \top$ which via (Sub) yields the required result.

Regarding application: If $E = E_1 E_2 \longrightarrow F_1 F_2 = F$ then the result follows by two applications of the induction hypothesis.

If the last rule used in the derivation of $E \longrightarrow F$ was

$$\frac{E_1 \longrightarrow F_1 \quad E_2 \longrightarrow F_2}{E = (\mathsf{distrust}(\lambda x.E_1))E_2 \longrightarrow \mathsf{distrust}\ F_1[F_2/x] = F}\text{(Application)}$$

then by assumption $A \vdash (\mathsf{distrust}(\lambda x.E_1))E_2 : t \,\#\, u$. By definition of the type rules this must mean that $A \vdash \mathsf{distrust}(\lambda x.E_1) : s_1 \xrightarrow{v_1\ v_2} t_1 \,\#\, w_1$ and $A \vdash E_2 : s_1 \,\#\, v_1$ where $t_1 \leq t$ and $v_2 \sqcup w_1 \leq u$. Again, we must also have $A \vdash \lambda x.E_1 : s'_1 \xrightarrow{v'_1\ v'_2} t'_1 \,\#\, w'_1$ and it must be case that $w_1 = \top$ and thus $u = \top$. Also $t'_1 \leq t_1$, $v'_2 \leq v_2$, $v_1 \leq v'_1$ and $s_1 \leq s'_1$. Once again by the type rules we must have $A[x \mapsto \langle s''_1, v''_1 \rangle] \vdash E_1 : t''_1 \,\#\, v''_2$ where $s'_1 \leq s''_1$, $v'_1 \leq v''_1$, $t''_1 \leq t'_1$ and $v''_2 \leq v'_2$. By the (Sub) rule we get $A \vdash E_2 : s''_1 \,\#\, v''_1$.

We can now apply the induction hypothesis to get $A[x \mapsto \langle s''_1, v''_1 \rangle] \vdash F_1 : t''_1 \,\#\, v''_2$ and $A \vdash F_2 : s''_1 \,\#\, v''_1$. By the Substitution lemma (Lemma 8) we now get $A \vdash F_1[F_2/x] : t''_1 \,\#\, v''_2$. By the (Distrust) rule we get $A \vdash \mathsf{distrust}\ F_1[F_2/x] : t''_1 \,\#\, \top$ and by (Sub) we get the desired result as $t''_1 \leq t'_1 \leq t_1 \leq t$.

The case without distrust is similar. □

3.3 Comparison with the Curry System

Our type system may be viewed as a restriction of the classic Curry type system for λ-calculus. This notion is formalized in the following. Define the *erasure* $|\cdot|$ of a term as:

$$\begin{array}{lcllcl} |x| & = & x & |\lambda x.E| & = & \lambda x.|E| \\ |E_1 E_2| & = & |E_1||E_2| & |\mathsf{trust}\ E| & = & |E| \\ |\mathsf{distrust}\ E| & = & |E| & |\mathsf{check}\ E| & = & |E| \end{array}$$

and likewise the erasure of a trust-type as:

$$|\mathsf{base}| = \mathsf{base} \qquad |s \xrightarrow{u\ v} t| = |s| \longrightarrow |t|.$$

The notion of erasure is extended pointwise to environments: $|A|(x) = |t|$ if and only if $A(x) = \langle t, u \rangle$.

The Curry type rules for erased expressions are defined in Figure 7. Here type assumptions A map program identifiers to Curry types.

$$A \vdash_C x : t \text{ if } x \in \mathrm{dom}(A) \text{ and } A(x) = t$$

$$\frac{A[x \mapsto s] \vdash_C E_1 : t}{A \vdash_C \lambda x.E_1 : s \longrightarrow t} \qquad \frac{A \vdash_C E_1 : s \longrightarrow t \quad A \vdash_C E_2 : s}{A \vdash_C E_1 E_2 : t}$$

Figure 7: The Curry type system.

Lemma 10 (Erasure) *If s and t are trust types and $s \leq t$ then $|s| = |t|$.*

Proof. For base-types $s \leq t$ implies $s = t$. For arrow types note that by definition of $\leq$, s and t must have the same arrow structure. So the result follows by an induction on the common structure of s and t. □

Theorem 11 *If $A \vdash E : t \# u$ then $|A| \vdash_C |E| : |t|$.*

Proof. By induction on the derivation of $A \vdash E : t \# u$.

$E =$ x: By assumption we have $x \in \mathrm{dom}(A)$ and $\langle t, u\rangle = A(x)$. Thus $x \in \mathrm{dom}(|A|)$ and $|A|(x) = |t|$.

$E =$ $\alpha\ E_1$: Here $\alpha \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. By the definition of erasure, $|E| = |E_1|$ and the result follows by induction.

$E =$ $\lambda x.E_1$: By assumption we must have $A[x \mapsto \langle s_1, v_1\rangle] \vdash E_1 : t_1 \# u_1$ where $s_1 \xrightarrow{v_1\ u_1} t_1 \leq t$. By the induction hypothesis $|A|[x \mapsto |s_1|] \vdash_C |E_1| : |t_1|$. By the lambda rule in the Curry system we get $|A| \vdash_C \lambda x.|E_1| : |s_1| \longrightarrow |t_1|$. By definition of erasure and the Erasure lemma we get the desired result.

$E =$ $E_1 E_2$: By assumption we must have $A \vdash E_1 : s_1 \xrightarrow{v_1\ u_1} t_1 \# w_1$ and $A \vdash E_2 : s_1 \# v_1$ where $t_1 \leq t$ and $u_1 \sqcup w_1 \leq u$. From the induction hypothesis we get $|A| \vdash_C |E_1| : |s_1| \longrightarrow |t_1|$ and $|A| \vdash_C E_2 : |s_1|$. By the application rule in the Curry system we get $|A| \vdash_C |E_1 E_2| : |t_1|$. Finally by the Erasure lemma we get the desired result.

□

If there are no subterms of the form $\mathsf{check}\ E$ in a program and the erasure of the program is Curry typable then the program is trust-typable and all the trusts may be chosen as $\top$. This idea is formalized below. Define the *decoration* of a Curry type as

$$\begin{array}{lcl} \Delta(\mathsf{base}) & = & \mathsf{base} \\ \Delta(s \longrightarrow t) & = & \Delta s \xrightarrow{\top\ \top} \Delta t \end{array}$$

Extend decorations to Curry type assumptions: $(\Delta A)(x) = \langle \Delta t, \top\rangle$ whenever $A(x) = t$. Clearly $|\Delta\ t| = t$.

$$E \to E \ \text{(Reflex)} \qquad \frac{E \to E'}{\lambda x.E \ \to \ \lambda x.E'} \ \text{(Sub)}$$

$$\frac{E \to E' \qquad F \to F'}{\begin{array}{lcl} EF & \to & E'F' \\ (\lambda x.E)F & \to & E'[F'/x] \end{array}} \ \text{(Application)}$$

Figure 8: Reductions in the ordinary λ-calculus.

Theorem 12 *If $A \vdash_C |E| : t$ and E contains no subterm of the form* check E' *then* $\Delta A \vdash E : \Delta t \,\#\, \top$.

Proof. By induction on the structure of E.

$E =$ x: By assumption we have $x \in \text{dom}(A)$ and $A(x) = t$, so $x \in \text{dom}(\Delta A)$ and $(\Delta A)(x) = \langle \Delta t, \top \rangle$. By the (Var) rule we get $\Delta A \vdash x : \Delta t \,\#\, \top$.

$E =$ $\lambda x.E_1$: By the assumptions and the Curry rules we must have $A[x \mapsto s_1] \vdash_C |E_1| : t_1$ where $s_1 \longrightarrow t_1 = t$. By induction $\Delta(A[x \mapsto s_1]) \vdash E_1 : \Delta t_1 \,\#\, \top$. By the (Lambda) and (Sub) rules of the trust system, we get $\Delta A \vdash \lambda x.E_1 : \Delta s_1 \overset{\top\ \top}{\longrightarrow} \Delta t_1 \,\#\, \top$ as required.

$E =$ $E_1 E_2$: By the assumptions and the Curry rules we must have $A \vdash_C |E_1| : s \longrightarrow t$ and $A \vdash_C |E_2| : s$. By the induction hypothesis we get $\Delta A \vdash E_1 : \Delta(s \longrightarrow t) \,\#\, \top$ and $\Delta A \vdash E_2 : \Delta s \,\#\, \top$. By the (App) rule we then get $\Delta A \vdash E_1 E_2 : \Delta t \,\#\, \top$ as required.

$E =$ trust E_1: Since $|$trust $E_1| = |E_1|$ and $A \vdash_C |E_1| : t$ by the induction hypothesis one gets $\Delta A \vdash E_1 : \Delta t \,\#\, \top$. Now we may apply the (Trust) and (Sub) rules to get $\Delta A \vdash$ trust $E_1 : \Delta t \,\#\, \top$ as wanted. The case for distrust is similar.

This exhausts the possible cases since E was assumed check-free. □

3.4 Simulation

The aim of this section is to show that for well-typed terms one may erase all the trust, distrust and check constructs and reduce expressions according to the ordinary λ-calculus as displayed in Figure 8 (this is taken from Definition 3.2.3 in [2].) We use the same symbol for this reduction relation as for our own and it will be clear from the context which reduction relation is meant. Note that the relation defined in Figure 8 is a subrelation of the reduction relation defined in Figure 2.

More formally the two following simulation theorems show that for well-typed terms, reduction and erasure commute: $|\cdot| \circ \to^* \ = \ \to^* \circ |\cdot|$.

In terms of implementation this means that after typechecking an interpreter may erase all trust tags and run the program without tags, thus no run-time performance penalty is paid.

Lemma 13 (Step) *If $E \rightarrow^* \alpha\ F$ (α may be empty) and there is a reduction rule*

$$\frac{E_1 \rightarrow \alpha\ F_1}{\beta\ E_1 \rightarrow \gamma\ F_1}$$

then $\beta\ E \rightarrow^* \gamma\ F$.

Proof. By induction on the length of the sequence $E \rightarrow^* \alpha\ F$. If $E = \alpha\ F$ then by (Reflex) we have $E \rightarrow \alpha\ F$ and we may apply the rule to get $\beta\ E \rightarrow \gamma\ F$ and since $\rightarrow \subseteq \rightarrow^*$ this is the required result.

Otherwise the last step in the reduction sequence $E \rightarrow^* \alpha\ F$ must look like $E' \rightarrow \alpha\ F$, where $E \rightarrow^* E'$ and $E' \neq \alpha\ F$. Now we apply the rule mentioned in the statement of the lemma:

$$\frac{E' \rightarrow \alpha\ F}{\beta\ E' \rightarrow \gamma\ F}$$

By the induction hypothesis one gets (via the (Sub) rule and using β for γ): $E \rightarrow^* E'$ implies $\beta\ E \rightarrow^* \beta\ E'$. By appending the two reductions we get $\beta\ E \rightarrow^* \gamma\ F$ as we wanted. In effect we get this derived rule:

$$\frac{E_1 \rightarrow^* \alpha\ F_1}{\beta\ E_1 \rightarrow^* \gamma\ F_1}$$

Similarly, from

$$\frac{E \rightarrow G \quad F \rightarrow H}{EF \rightarrow GH} \qquad \text{we get} \qquad \frac{E \rightarrow^* G \quad F \rightarrow^* H}{EF \rightarrow^* GH}$$

□

Some notation: We write $E_0 = \mathsf{CTD}^* F$ to mean that E_0 is produced by the following grammar, where F is an ordinary term.

$$E_0 ::= \mathsf{check}\ E_0 \mid \mathsf{trust}\ E_0 \mid \mathsf{distrust}\ E_0 \mid F$$

We also write $\mathsf{distrust}^?\ E$ to mean either E or $\mathsf{distrust}\ E$.

Lemma 14 (CTD) *If $E = \mathsf{CTD}^*(\lambda x.E_1)$, $A \vdash E : t \# u$ and $E_1 \rightarrow^* F_1$ then $E \rightarrow^* \mathsf{distrust}^?\ (\lambda x.F_1)$.*

Proof. By induction on the length of the CTD sequence. Suppose that

$$\mathsf{CTD}^*\lambda x.E_1 = (\alpha\ (\beta\ \ldots(\lambda x.E_1)\ldots)).$$

In the base case (the empty sequence) $E_1 \to^* F_1$ implies (via Sub and Step) that $\lambda x.E_1 \to^* \lambda x.F_1$.

Otherwise, there are two cases depending on whether $(\beta\ \ldots)$ reduces to a lambda or a distrusted lambda.

Suppose that $(\beta\ \ldots) \to^* \lambda x.F_1$ by the induction hypothesis then via the Step lemma and (Lambda Contraction):

$\alpha =$ trust: $(\mathsf{trust}\ (\beta\ \ldots)) \to^* \lambda x.F_1$.

$\alpha =$ $\mathsf{distrust}$: $(\mathsf{distrust}\ (\beta\ \ldots)) \to^* \mathsf{distrust}\ \lambda x.F_1$.

$\alpha =$ check: $(\mathsf{check}\ (\beta\ \ldots)) \to^* \lambda x.F_1$.

Finally, suppose that $(\beta\ \ldots) \to^* \mathsf{distrust}\ \lambda x.F_1$ by the induction hypothesis then via the Step lemma and (Distrust Contraction):

$\alpha =$ trust: $(\mathsf{trust}\ (\beta\ \ldots)) \to^* \mathsf{trust}\ \lambda x.F_1$ and via a (Lambda Contraction) step: $\mathsf{trust}\ \lambda x.F_1 \to \lambda x.F_1$.

$\alpha =$ $\mathsf{distrust}$: $(\mathsf{distrust}\ (\beta\ \ldots)) \to^* \mathsf{distrust}\ \lambda x.F_1$.

$\alpha =$ check: As E is well-typed this case cannot occur since $\mathsf{check}(\mathsf{distrust}\ E_1)$ is untypable.

□

Theorem 15 (Simulation 1) *If $A \vdash E : t \# u$ and $|E| \to F$ then there is a term G such that $E \to^* G$ and $|G| = F$.*

Proof. By structural induction on E.

$E = x$: Here $|E| = E$ and the only applicable rule is (Reflex), thus we get $E = F = G$.

$E = \alpha\ E_1$, where $\alpha \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. Here we have $|E| = |E_1|$, $A \vdash E_1 : t_1 \# u_1$ and $|E| = |E_1| \to F$. So by the induction hypothesis there is a G_1 such that $E_1 \to^* G_1$ and $|G_1| = F$. By the Step lemma we can deduce:

$$\frac{E_1 \to^* G_1}{E = \alpha\ E_1 \to^* \alpha\ G_1 = G}\,(\text{Sub})$$

and the erasure of G is F as required.

$E = \lambda x.E_1$: By the assumptions we must have $A[x \mapsto \langle t_1, u_1\rangle] \vdash E_1 : t_2 \# u_2$. Also, $|E| = \lambda x.|E_1|$ and $F = \lambda x.F_1$. By the nature of the reduction rules, we must have $|E_1| \rightarrow F_1$. By the induction hypothesis we know there is a G_1 such that $E_1 \rightarrow^* G_1$ and $|G_1| = F_1$. By the (Sub) rule and the Step lemma we get

$$\frac{E_1 \rightarrow^* G_1}{E = \lambda x.E_1 \rightarrow^* \lambda x.G_1 = G}\text{(Sub)}$$

and $|G| = \lambda x.|G_1| = \lambda x.F_1 = F$ as required.

$E = E_1E_2$: By the assumptions E is well-typed thus E_1 and E_2 are well-typed. By definition of the reduction rules we must have $|E_1| \rightarrow F_1$ and $|E_2| \rightarrow F_2$. By the induction hypothesis we get G_1 and G_2 such that $E_1 \rightarrow^* G_1$, $E_2 \rightarrow^* G_2$, $|G_1| = F_1$ and $|G_2| = F_2$.

There are two cases depending on the form of $|E|$:

$|E| =$ *not a β-redex*: Here $F = F_1F_2$ where $|E_1| \rightarrow F_1$ and $|E_2| \rightarrow F_2$ so by the reasoning above $|G_1G_2| = F$ and we are done.

$|E| = (\lambda x.H_1)H_2$: If $F = (\lambda x.F_1')F_2$ where $H_1 \rightarrow F_1'$ and $H_2 \rightarrow F_2$ then also $|E_1| = \lambda x.H_1 \rightarrow \lambda x.F_1' = F_1$ by (Sub). By the above statements and the Step lemma we get $E \rightarrow^* G_1G_2$ and $|G_1G_2| = F$.

Otherwise a β-reduction happens. Here $H_1 \rightarrow F_1'$, $H_2 \rightarrow F_2$ and $F = F_1'[F_2/x]$.

Clearly, E_1 must have form $\mathsf{CTD}^*(\lambda x.Q_1)$ where $|Q_1| = H_1$. By the induction hypothesis there is a Q_1' such that $Q_1 \rightarrow^* Q_1'$ and $|Q_1'| = F_1'$.

By the CTD lemma $E_1 \rightarrow^* \mathsf{distrust}^? \ (\lambda x.Q_1')$ and by the Subject Reduction theorem (Theorem 9) we get that $\mathsf{distrust}^? \ (\lambda x.Q_1')$ is well-typed.

We may now reason as follows:

$$\frac{E_1 \rightarrow^* \mathsf{distrust}^? \ (\lambda x.Q_1') \quad E_2 \rightarrow^* G_2}{E_1E_2 \rightarrow^* (\mathsf{distrust}^? \ (\lambda x.Q_1'))G_2}\text{(Sub + Step)}$$

and

$$\frac{Q_1' \rightarrow Q_1' \quad G_2 \rightarrow G_2}{(\mathsf{distrust}^? \ (\lambda x.Q_1'))G_2 \rightarrow \mathsf{distrust}^? \ Q_1'[G_2/x]}\text{(Application)}$$

since $|Q_1'| = F_1'$ and $|G_2| = F_2$:

$$|\mathsf{distrust}^? \ Q_1'[G_2/x]| = |Q_1'[G_2/x]| = F_1'[F_2/x] = F$$

as required.

This concludes the proof of the Simulation theorem. □

Theorem 16 (Simulation 2) *If $E \rightarrow F$ then $|E| \rightarrow |F|$.*

Proof. By induction on the derivation of $E \to F$.

- If $E \to F$ by (Reflex) then $|E| = |F|$ and the result holds trivially.
- If $E \to F$ by the (Sub) rule. The subterm(s) E_i of E then must reduce $E_i \to F_i$ and by the induction hypothesis $|E_i| \to |F_i|$. We may now apply the (Sub) rule to these erased terms and get $|E| \to |F|$.
- Let $\alpha, \beta, \gamma \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. If the last rule in the derivation of $E \to F$ was

  $$\frac{E_1 \to \alpha\ F_1}{E = \beta\ E_1 \to \gamma\ F_1 = F}(\alpha\text{-Contraction})$$

  then $|E| = |E_1|$ and $|F| = |F_1|$. By the induction hypothesis we know $|E_1| \to |F_1|$ which is the desired result.
- If the last rule used in the derivation of $E \to F$ was

  $$\frac{E_1 \to \lambda x.F_1}{E = \alpha\ E_1 \to \lambda x.F_1 = F}(\text{Lambda Contraction})$$

  where $\alpha \in \{\mathsf{trust}, \mathsf{check}\}$ then by the induction hypothesis we get $|E_1| \to \lambda x.|F_1|$ and since $|E| = |E_1|$ and $|F| = \lambda x.|F_1|$ this is the desired result.
- If the last rule used in the derivation of $E \to F$ was

  $$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \to \mathsf{distrust}\ F_1[F_2/x] = F}(\text{Application})$$

  We have $|E| = (\lambda x.|E_1|)|E_2|$ and $|F| = |F_1|[|F_2|/x]$. By the induction hypothesis $|E_1| \to |F_1|$ and $|E_2| \to |F_2|$. We may now apply the (Application) rule to get the desired result. The case for the trusted lambda is similar.

□

3.5 Strong Normalization

In Curry typed λ-calculus all typable terms are strongly normalizing, i.e. they reduce to a normal form. This result carries over to our language, essentially since our type system admits fewer terms than the Curry system. On the other hand we have terms not occurring in the standard λ-calculus and a lot more reduction rules so this requires a proof.

Theorem 17 (Strong Normalization) *If $A \vdash E : t \# u$ then there is a normal form G' such that $E \to G'$.*

Proof. By Theorem 11, $A \vdash E : t \# u$ implies $|A| \vdash_C |E| : |t|$. By the Strong Normalization theorem for Curry typed λ-calculus [7] there is a Curry normal form F such that $|E| \rightarrow^* F$. We can now apply the first Simulation theorem (Theorem 15) to obtain a term G such that $E \rightarrow^* G$ and $|G| = F$.

As $|G| = F$ is a normal form it has no β-redexes, but in G some other reductions may be applicable. Because of the Church-Rosser theorem, only using contraction and Sub rules will not block any possible other reduction that might be applicable, so let $G \rightarrow^* G'$ be such a reduction sequence. In all the contraction rules the number of $\{\mathsf{check}, \mathsf{trust}, \mathsf{distrust}\}$ constructs decrease with one. Writing $\rightarrow^!$ for the irreflexive part of the $\rightarrow$ relation we have that in the sequence $G \rightarrow^! G_1 \rightarrow^! \cdots \rightarrow^! G'$ there are a finite number of steps (less than the size of G). Thus we may take G' such that in G' no more of these non-β reductions are applicable.

After reaching G' can it be the case that the contractions revealed a β-redex in G'? Suppose for a contradiction that this is the case. This means that G' has a subterm of the form $(\mathsf{distrust}^? \ \lambda x.G_1)G_2$. So we have $G' \rightarrow G''$ by reducing this redex. By Theorem 16 this means that $F = |G'| \rightarrow^! |G''|$ and clearly $|G''| \neq F$. But this contradicts the fact the F could be reduced no further! So it cannot be the case that the final contractions reveal a β-redex, and thus G' is a normal form. □

4 Type Inference

The type inference problem is:

Given an untyped program E, is E typable? If so, annotate it.

From Theorem 11 we have that trust typing implies Curry typing. Our type inference algorithm works by first checking if the program has a Curry type and then checking a condition that only involves trust values.

4.1 Constraints

The type inference problem can be rephrased in terms of solving a system of constraints.

Definition 18 Given two disjoint denumerable sets of variables $\mathcal{V}_y$ and $\mathcal{V}_r$, a *T-system* is a pair (C, D) where:

- C is a finite set of inequalities $X \leq X'$ between constraint expressions, where X and X' are of the forms V or $V \overset{W\ W'}{\longrightarrow} V'$, and where $V, V' \in \mathcal{V}_y$ and $W, W' \in \mathcal{V}_r$.
- D is a finite set of constraint of the forms $W \leq W'$, $W = \bot$, or $W = \top$, where $W, W' \in \mathcal{V}_r$.

A *solution* for a T-system is a pair of maps (δ, φ), where δ maps variables in $\mathcal{V}_y$ to types, and where φ maps variables in $\mathcal{V}_r$ to trusts, such that all constraints are satisfied.

If φ satisfies all constraints in D, we say that D has solution φ. $\square$

Given a λ-term E, assume that E has been α-converted so that all bound variables are distinct. Let $\mathcal{V}_y$ be the set consisting of:

- A variable $[\![F]\!]_y$ for each occurrence of a subterm F of E; and
- A variable x_y for each λ-variable x occurring in E.

(The notation $[\![F]\!]_y$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant.) Intuitively, $[\![F]\!]_y$ denotes the type of F after the use of subsumption. Moreover, x_y denotes the type assigned to the bound variable x.

Let $\mathcal{V}_r$ be the set consisting of:

- A variable $[\![F]\!]_r$ for each occurrence of a subterm F of E; and
- A variable x_r for each λ-variable x occurring in E.
- A variable $\langle GH\rangle_r$ for each occurrence of an application GH in E.

(As before, the notation $[\![F]\!]_r$ is ambiguous.) Intuitively, $[\![F]\!]_r$ denotes the trust value of F after the use of subsumption. Moreover, x_r denotes the trust value assigned to the bound variable x. Finally, $\langle GH\rangle_r$ denotes the trust value of GH before the use of subsumption.

From the λ-term E, we generate the T-system (C, D) where:

For each occurrence in E	We have in C	We have in D
x	$x_y \le [\![x]\!]_y$	$x_r \le [\![x]\!]_r$
$\lambda x.F$	$x_y \xrightarrow{x_r\ [\![F]\!]_r} [\![F]\!]_y \le [\![\lambda x.F]\!]_y$	
GH	$[\![G]\!]_y \le [\![H]\!]_y \xrightarrow{[\![H]\!]_r\ \langle GH\rangle_r} [\![GH]\!]_y$	$[\![G]\!]_r \le [\![GH]\!]_r$ $\langle GH\rangle_r \le [\![GH]\!]_r$
$\mathsf{trust}\ F$	$[\![F]\!]_y \le [\![\mathsf{trust}\ F]\!]_y$	
$\mathsf{distrust}\ F$	$[\![F]\!]_y \le [\![\mathsf{distrust}\ F]\!]_y$	$[\![\mathsf{distrust}\ F]\!]_r = \top$
$\mathsf{check}\ F$	$[\![F]\!]_y \le [\![\mathsf{check}\ F]\!]_y$	$[\![F]\!]_r = \bot$

Denote by $T(E)$ the T-system of constraints generated from E in this fashion. The solutions of $T(E)$ correspond to the possible type annotations of E in a sense made precise by Theorem 21.

Let A be a trust-type environment. If δ is a function assigning types to variables in $\mathcal{V}_y$ and φ a function assigning trusts to variables in $\mathcal{V}_r$, we say that (δ, φ) *extend* A if for every x in the domain of A, we have $A(x) = \langle \delta(x_y), \varphi(x_r)\rangle$.

As a shorthand in the following, we write $(\delta, \varphi) \models (C, D)$ to mean that (δ, φ) is a solution to the constraints (C, D). Define also, for two functions δ and δ' agreeing on $\mathrm{dom}(\delta) \cap \mathrm{dom}(\delta')$, $\delta + \delta'$ as the unique function on $\mathrm{dom}(\delta) \cup \mathrm{dom}(\delta')$ that agree with the two functions on their respective domains.

Lemma 19 (Soundness) *If $(\delta, \varphi) \models T(E)$, and δ, φ extend A then $A \vdash E : \delta(\llbracket E \rrbracket_y) \# \varphi(\llbracket E \rrbracket_r)$.*

Proof. By induction on the structure of E. □

Lemma 20 (Completeness) *If $A \vdash E : t \# u$ then there is a solution $(\delta, \varphi) \models T(E)$ with δ and φ extending A, and $\delta(\llbracket E \rrbracket_y) = t$ and $\varphi(\llbracket E \rrbracket_r) = u$.*

Proof. By induction on the derivation of $A \vdash E : t \# u$.

$E = x$: As $A \vdash x : t \# u$ we must have $x \in \mathrm{dom}(A)$, $A(x) = \langle t', u' \rangle$ and $t' \leq t, u' \leq u$. In this case $T(E) = \{x_y \leq \llbracket x \rrbracket_y,\ x_r \leq \llbracket x \rrbracket_r\}$. Put $\delta(x_y) = t'$ and $\varphi(x_r) = u'$ so that (δ, φ) extends A. Finally assign $\delta(\llbracket x \rrbracket_y) = t$ and $\varphi(\llbracket x \rrbracket_r) = u$ to satisfy the constraints.

$E = \lambda x.F$: By the type rules we must have $A[x \mapsto \langle s, v \rangle] \vdash F : s' \# u'$ where $s \xrightarrow{v\ u'} s' \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta(\llbracket F \rrbracket_y) = s'$, $\varphi(\llbracket F \rrbracket_r) = u'$, and (δ, φ) extends $A[x \mapsto \langle s, v \rangle]$. Now assign $\delta' = \delta[\llbracket \lambda x.F \rrbracket \mapsto t]$ and $\varphi' = \varphi[\llbracket \lambda x.F \rrbracket \mapsto u]$. Now check that $s \xrightarrow{v\ u'} s' \leq t$ implies

$$\delta'(x_y) \xrightarrow{\varphi'(x_r)\ \varphi'(\llbracket F \rrbracket_r)} \delta'(\llbracket F \rrbracket_y) \leq \delta'(\llbracket \lambda x.F \rrbracket_y)$$

as required. So we get $(\delta', \varphi') \models T(E)$.

$E = GH$: By the type rules we must have $A \vdash G : s' \xrightarrow{u'\ v'} t' \# w'$, $A \vdash H : s' \# u'$ where $t' \leq t$ and $v' \sqcup w' \leq u$. By the induction hypothesis we get $(\delta, \varphi) \models T(G)$ and $(\delta', \varphi') \models T(H)$ and both solutions extending A which means that they agree on their common domain (the x_y's and the x_r's in $\mathrm{dom}(A)$). The definition of $T(E)$ says

$$\begin{aligned} T(E) = T(G) \cup T(H) \cup (&\{\llbracket G \rrbracket_y \leq \llbracket H \rrbracket_y \xrightarrow{\llbracket H \rrbracket_r\ \langle GH \rangle_r} \llbracket GH \rrbracket_y\}, \\ &\{\llbracket G \rrbracket_r \leq \llbracket GH \rrbracket_r,\ \langle GH \rangle_r \leq \llbracket GH \rrbracket_r\})\ . \end{aligned}$$

Define $\delta'' = \delta + \delta'[\llbracket GH \rrbracket_y \mapsto t]$ and $\varphi'' = \varphi + \varphi'[\llbracket GH \rrbracket_r \mapsto u, \langle GH \rangle_r \mapsto v']$. Now, $(\delta'', \varphi'') \models T(E)$ because

$$\delta(\llbracket G \rrbracket_y) = s' \xrightarrow{u'\ v'} t' \leq s' \xrightarrow{u'\ v'} t = \delta'(\llbracket H \rrbracket_y) \xrightarrow{\varphi'(\llbracket H \rrbracket_r)\ \varphi''(\langle GH \rangle_r)} \delta''(\llbracket GH \rrbracket_y)$$

$$\begin{aligned}\varphi(\llbracket G \rrbracket_r) = w' &\leq u = \varphi''(\llbracket GH \rrbracket_r) \\ \varphi''(\langle GH \rangle_r) = v' &\leq u = \varphi''(\llbracket GH \rrbracket_r) \ .\end{aligned}$$

and clearly (δ'', φ'') extend A.

$E =$ $\mathsf{check}\ F$: From the type rules we must have $A \vdash F : t' \# \bot$ where $t' \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta(\llbracket F \rrbracket_y) = t'$, and $\varphi(\llbracket F \rrbracket_r) = \bot$. Now, $T(E) = T(F) \cup \{\llbracket F \rrbracket_y \leq \llbracket \mathsf{check}\ F \rrbracket_y, \llbracket F \rrbracket = \bot\}$. Put $\delta' = \delta[\llbracket \mathsf{check}\ F \rrbracket_y \mapsto t]$ and $\varphi' = \varphi[\llbracket \mathsf{check}\ F \rrbracket_y \mapsto u]$.

Clearly, δ', φ' extend A, $\delta'(\llbracket F \rrbracket_y) \leq \delta'(\llbracket \mathsf{check}\ F \rrbracket_y)$, and $\varphi'(\llbracket F \rrbracket_r) = \bot$ as required. The cases for trust and $\mathsf{distrust}$ are very similar.

□

Theorem 21 *The judgement $A \vdash E : t \# u$ is derivable if and only if there exists a solution (δ, φ) of $T(E)$ with (δ, φ) extending A such that $\delta(\llbracket E \rrbracket_y) = t$ and $\varphi(\llbracket E \rrbracket_r) = u$. In particular, if E is closed, then E is typable with type t and trust u if and only if there exists a solution (δ, φ) of $T(E)$ such that $\delta(\llbracket E \rrbracket_y) = t$ and $\varphi(\llbracket E \rrbracket_r) = u$.*

Proof. Combine Lemma 19 and 20. □

4.2 Algorithm

Definition 22 Given a T-system (C, D), define the *deductive closure* $(\bar{C}, \bar{D})$ to be the smallest T-system such that:

- $C \subseteq \bar{C}$.
- $D \subseteq \bar{D}$.
- If $V_1 \xrightarrow{W_1\ W_2} V_2 \leq V_1' \xrightarrow{W_1'\ W_2'} V_2'$ is in $\bar{C}$, then $V_1' \leq V_1$ and $V_2 \leq V_2'$ are in $\bar{C}$, and $W_1' \leq W_1$ and $W_2 \leq W_2'$ are in $\bar{D}$.
- If $X_1 \leq X_2$ and $X_2 \leq X_3$ are in $\bar{C}$, then $X_1 \leq X_3$ is in $\bar{C}$.

□

Lemma 23 *(C, D) and $(\bar{C}, \bar{D})$ have the same solutions.*

Proof. Since $C \subseteq \bar{C}$ and $D \subseteq \bar{D}$, any solution of $(\bar{C}, \bar{D})$ is also a solution of (C, D). The converse can be proved by induction on the construction of $(\bar{C}, \bar{D})$. □

If we remove all mentioning of trust and subtyping from the type rules in figure 5 and from the constraints defined earlier in this section, we obtain two equivalent formulations of Curry typability [16]. Clearly, E is Curry typable if and only if $|E|$ is Curry typable. The constraint system (written out below) that expresses Curry typability will be denoted $\mathsf{Curry}(E)$.

For each occurrence in E	We have in $\mathsf{Curry}(E)$
x	$x_y = [\![x]\!]_y$
$\lambda x.F$	$[\![\lambda x.F]\!]_y = x_y \to [\![F]\!]_y$
GH	$[\![G]\!]_y = [\![H]\!]_y \to [\![GH]\!]_y$
$\mathsf{trust}\ F$	$[\![\mathsf{trust}\ F]\!]_y = [\![F]\!]_y$
$\mathsf{distrust}\ F$	$[\![\mathsf{distrust}\ F]\!]_y = [\![F]\!]_y$
$\mathsf{check}\ F$	$[\![\mathsf{check}\ F]\!]_y = [\![F]\!]_y$

If s, t are trust types such that $|s| = |t|$, then define the operators $\sqcup^{|s|}$ and $\sqcap^{|t|}$ as follows.

$$s \sqcup^{|s|} t \quad = \quad \begin{cases} \mathsf{base} & \text{if } s = t = \mathsf{base} \\ (s_1 \sqcap^{|s_1|} t_1) \xrightarrow{u_1 \sqcap v_1 \ \ u_2 \sqcup v_2} (s_2 \sqcup^{|s_2|} t_2) & \text{if } s = s_1 \xrightarrow{u_1 \ \ u_2} s_2 \\ & \text{and } t = t_1 \xrightarrow{v_1 \ \ v_2} t_2 \end{cases}$$

$$s \sqcap^{|s|} t \quad = \quad \begin{cases} \mathsf{base} & \text{if } s = t = \mathsf{base} \\ (s_1 \sqcup^{|s_1|} t_1) \xrightarrow{u_1 \sqcup v_1 \ \ u_2 \sqcap v_2} (s_2 \sqcap^{|s_2|} t_2) & \text{if } s = s_1 \xrightarrow{u_1 \ \ u_2} s_2 \\ & \text{and } t = t_1 \xrightarrow{v_1 \ \ v_2} t_2\ . \end{cases}$$

If $t_1, \ldots, t_n$ are trust types, and s is a Curry type such that $|t_i| = s$ for all $i \in 1..n$, then define $\bigsqcup_i^s t_i = t_1 \sqcup^s \ldots \sqcup^s t_n$. If t is a Curry type, define

$$\begin{aligned} \mathsf{small}(\mathsf{base}) &= \mathsf{base} & \qquad \mathsf{big}(\mathsf{base}) &= \mathsf{base} \\ \mathsf{small}(s \to t) &= \mathsf{big}(s) \xrightarrow{\top \ \ \bot} \mathsf{small}(t) & \qquad \mathsf{big}(s \to t) &= \mathsf{small}(s) \xrightarrow{\bot \ \ \top} \mathsf{big}(t) \end{aligned}$$

If s is a trust type and t is a Curry type such that $|s| = t$, then $s \sqcup^t \mathsf{small}(t) = s$ and $s \sqcap^t \mathsf{big}(t) = s$. In other words, $\mathsf{small}(t)$ is the *least* trust type with erasure t.

For each a constraint expression X define

$$L(C, X) = \{V_1 \xrightarrow{W_1 \ \ W_1'} V_1' \mid V_1 \xrightarrow{W_1 \ \ W_1'} V_1' \leq X \text{ is in } \bar{C}\}\ .$$

Intuitively, $L(C, X)$ is the set of syntactic lower bounds for X.

We also define the erasure of a constraint expression used in C, mapping trust-type constraint expressions to Curry constraint expressions:

$$\begin{aligned} |V| &= V \\ |V \xrightarrow{W \ \ W'} V'| &= |V| \to |V'| \end{aligned}$$

where $V, V' \in \mathcal{V}_y$ and $W, W' \in \mathcal{V}_r$.

Lemma 24 *If $T(E) = (C, D)$, and ψ is a solution to* $\mathsf{Curry}(E)$*, and $X_1 \le X_2$ is a constraint in $\bar{C}$, then $\psi(|X_1|) = \psi(|X_2|)$.*

Proof. By induction on the construction of $\bar{C}$. □

Theorem 25 *Suppose $T(E) = (C, D)$. Then $T(E)$ is solvable if and only if E is Curry typable and $\bar{D}$ is solvable.*

Proof. Suppose first that $T(E)$ is solvable. By Theorem 21, E is trust typable. It follows from Theorem 11 and the remark above that E is Curry typable, and from Lemma 23 that $\bar{D}$ is solvable.

For the reverse implication, suppose that $\mathsf{Curry}(E)$ has solution ψ and that $\bar{D}$ has solution φ. We define δ inductively in the Curry types of the constraint variables.

$$\begin{aligned}\delta(V) = {} & \mathsf{if}\ \psi(|V|) = \mathsf{base\ then\ base} \\ & \quad \mathsf{else\ let}\ \{V_i \overset{W_i\ W_i'}{\longrightarrow} V_i'\} = L(C, V) \cup \{\mathsf{small}(\psi(|V|))\} \\ & \quad \mathsf{in}\ \textstyle\bigsqcup_i^{\psi(|V|)} (\delta(V_i) \overset{\varphi(W_i)\ \varphi(W_i')}{\longrightarrow} \delta(V_i'))\end{aligned}$$

To see that δ is well-defined, we need that the Curry types of the variables V_i and V_i' are of strictly less size than the Curry type of V. For $(V_i \overset{W_i\ W_i'}{\longrightarrow} V_i') \in L(C, V)$, we get by Lemma 24 that $\psi(|V_i \overset{W_i\ W_i'}{\longrightarrow} V_i'|) = \psi(|V|) = s \to t$ for some s, t. This means that $\psi(|V_i|) = s$ and $\psi(|V_i'|) = t$ which are both of smaller size than $s \to t$, so δ is well-defined.

To see that (δ, φ) is a solution of $T(E)$, consider an inequality $X_1 \le X_2$ in C. If $\psi(|X_1|) = \mathsf{base}$, then by Lemma 24, $\psi(|X_2|) = \mathsf{base}$, $\delta(X_1) = \delta(X_2) = \mathsf{base}$, thus $\delta(X_1) \le \delta(X_2)$ as required.

In case $\psi(|X_1|) = s \to t$, we have by Lemma 24 that $\psi(|X_2|) = s \to t$ and since $\bar{C}$ is transitively closed we get $L(C, X_1) \subseteq L(C, X_2)$ so $\delta(X_1) \le \delta(X_2)$ as required. □

Using the characterization of Theorem 25, we get a type inference algorithm:

```
Input: A $\lambda$-term $E$ of size $n$.
   1: Construct $T(E) = (C, D)$ (in log space).
   2: Close $(C, D)$, yielding $(\bar{C}, \bar{D})$ (in $O(n^3)$ time, see for example [15]).
   3: Check if $E$ is Curry typable (in $O(n)$ time).
   4: Check if $\bar{D}$ is solvable (in $O(n^2)$ time).
   5: If $E$ is Curry typable and $\bar{D}$ is solvable,
      then output "typable"
      else output "not typable".
```

The entire algorithm requires $O(n^3)$ time. To construct an annotation of a typable program, we can use the construction of the second half of the proof of Theorem 25.

5 Extensions and Related Work

In this section we discuss several extensions of the type system and related work.

Recursion. The type system can be extended to handle recursion by adding a rec rule. In the (untyped) reduction system, the rec combinator can be coded with the classical Y combinator: $\mathsf{rec}\ x.E \equiv \mathsf{Y}(\lambda x.E)$. The following reduction rule is a derived rule in the λ-calculus and in our system, and correspondingly we would have a rec rule in the type system:

$$\frac{E \to^* F}{\mathsf{Y}E \to^* F(\mathsf{Y}E)} \qquad\qquad \frac{A[x \mapsto \langle t, u\rangle] \vdash E_1 : t \# u}{A \vdash \mathsf{rec}\ x.E_1 : t \# u}$$

Subject Reduction still holds, but Strong Normalization of course fails in this case. The type inference algorithm can also be extended in a straightforward way to deal with the rec construct.

Polymorphism. ML style let polymorphism can be achieved in the usual way by replacing let bound variables with their definition. This is of course inefficient as each definition might then be type-checked many times. The type system can be extended along the same ideas that extend Curry types to Hindley-Milner types. An extension of our type inference algorithm remains to be found.

Other Lattices. The values of trust-tags may be extended from the two point lattice used in this paper to any finite lattice. Extending the lattice to a longer linear lattice accommodates multiple levels of trust. Extensions to non-linear orderings may allow different properties of values.

Related Work. The original notion of trust analysis was presented in [14] where an abstract interpretation [4] analysis and a constraint based analysis for a simple imperative, first order, language were given. This work extends trust analysis to the higher order functional case and formalizes it in terms of a decorated type system.

Security flow analysis [5, 13, 1] aims at restricting the flow of confidential information *out* of a trusted computer system, dually to our analysis that aims at preventing untrustworthy information from entering *into* a secure facility. Also, the above analyses all deal with procedural, imperative languages without higher-order functions.

In [12] Mitchell developed the structural subtyping idea and our type system borrows some of these ideas to handle automatic coercion from trusted data to untrusted data.

Our type system extends the basic Curry types with annotations in the same spirit as Talpin and Jouvelot's effect systems [17], Wright's annotated types [18] and others.

The idea of accepting *fewer* programs than a well-known type system is in the spirit of the refinement types of [6].

Other Analyses. Without support for trust in the type system one might approximate the same effect by coding the relevant types as sums with a trusted and a distrusted variant. This has two drawbacks: the values have to be explicitly packed and unpacked each time they are used and the tags will be present at run-time which means that hackers might be able to fiddle with these tags and thereby circumvent the security measures.

As mentioned in Section 1 many other program analyses approximate information inherent in the *values* that exists at run-time. Strictness analysis [3, 11] aims to find out whether functions in a higher order lazy language are strict or not. This may not be a decidable property but it is nevertheless a property of a value (function). In binding time analysis [8] one aims to approximate whether a value can be computed at compile-time or not. At compile-time this is a property of the values. Also, this is something that does not change from one compilation of the same program to the next.

Another kind of analysis that looks related to trust analysis is dynamic typing or boxing/unboxing analysis [9, 10] which aims to remove type tags as much as possible in a dynamically typed language. One might be tempted to view, say, distrust as a boxing operation and trust as the corresponding unboxing operation. However, this does not explain how check should be interpreted and it doesn't match with our application type rule, in that applying a boxed function to an argument does not necessarily result in a boxed result.

6 Conclusion

We have argued for the usefulness of so-called trust analysis to help programmers produce safer and more trustworthy software. We have presented an extension of the λ-calculus together with a reduction semantics. The reduction relation is proved Church-Rosser. Then we gave a type system that enables the static inference of the trustworthiness of values and the type system was proved to have the Subject Reduction property with respect to the semantics of our language.

Last a constraint based type inference algorithm was presented and proved correct with respect to the type system.

References

[1] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Metayer. Compile-time detection of information flow in sequential programs. In Dieter Gollmann, editor, *Computer Security – ESORICS 94, 3rd European Symp. on Research in Comp. Security*, pages 55–73. Springer-Verlag (*LNCS* 875), 1994.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[3] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[5] Dorothy E. Denning and Peter J. Denning. Certifications of programs for secure information flow. *Communications of the ACM*, 20(7):504–512, July 1977.

[6] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[7] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[8] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[9] Fritz Henglein. Dynamic typing. In *Proc. ESOP'92, European Symposium on Programming*, pages 233–253. Springer-Verlag (*LNCS* 582), 1992.

[10] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Proc. POPL'94, 21st Annual Symposium on Principles of Programming Languages*, pages 213–226, 1994.

[11] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, 1989.

[12] John Mitchell. Coercion and type inference. In *Eleventh Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[13] Masaaki Mizuno and David Schmidt. A security flow control algorithm and its denotational semantics correctness proof. Technical Report TR-CS-90-21, Kansas State University, 1990.

[14] Peter Ørbæk. Can you trust your data? In P. D. Mosses, editor, *Proc. TAPSOFT'95, Theory and Practice of Software Development*, pages 575–590. Springer-Verlag (*LNCS* 915), 1995.

[15] Jens Palsberg. Efficient inference of object types. *Information and Computation.* To appear. Also in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[16] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[17] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[18] David A. Wright. A new technique for strictness analysis. In *Proc. TAPSOFT'91*, pages 235–258. Springer-Verlag (*LNCS* 494), 1991.