J. Functional Programming, 3(2):75-85, 1997.

Trust in the λ-calculus

P. ØRBÆK AND J. PALSBERG

BRICS, Centre of the Danish National Research Foundation,
Dept. of Computer Science, University of Aarhus,
Ny Munkegade bld 540, 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 is presented as annotations of the underlying Curry types, and type inference is computable in $O(n^3)$ time.

1 Introduction

As computers are increasingly being used to handle important transactions (such as interchange of legal documents) in an open environment where also information that does not pertain directly to business is transferred between the same machines (for example e-mail), the issue of separating these two kinds of information flow has become more and more important. The issue is usually attacked using encryption and digital signatures for the important information flows between machines, but what good does this do if somewhere *inside* a program, there is a data path between reading a message and some dangerous operation such that the dangerous operation depends on the contents of a message whose signature is not checked?

An example where it is important to perform validity checks on all data paths from input to output, is in an HTTP (HyperText Transfer Protocol, the protocol used on the world-wide web) server that allows a web-browser to start a separate program (a so-called CGI script, for Common Gateway Interface) on the server machine by requesting a specific URL. The usual convention is that the browser requests a URL of the form `http://www.company.com/cgi-bin/`*program-name*. Since a CGI script can in principle be any program, it is important that the server checks that the requested program is one of the few programs that are allowed to be run in this fashion. In a complicated server, like most servers today, there are many data-paths from the initial reception of the URL request to the command to be executed, and it is imperative that the checks for allowed programs be performed on *all* these paths. The trust analysis presented in this paper is developed to help the programmer ensure that the requisite checks are always made.

A very similar situation existed in a part of the Gopher (an earlier and simpler

distributed information service than the world-wide web) server, where through one of the provided gateways to other services one could contrive a special request to the server and thereby get arbitrary commands executed on the server machine, including starting a remote terminal window on that machine and thus thwarting all security measures. This security bug was later fixed by the developers when the second author made them aware of the problem. We believe that the use of a trust-analysis could have helped prevent this problem in the first place.

A third example of where trust analysis could be useful is in a web browser that must forbid the retrieval of certain URLs, for example to prevent children from viewing on-line pornography. Since there are many ways in a typical browser program to enter a URL, (the command line, configuration files, dialog boxes, ...) it's important that the checks for forbidden URLs are made on all the paths from reading user input to getting the URL from a server.

Since we want our analysis to be generally applicable, we do not consider the very specific tests that have to be done on input data in various situations, such as checking digital signatures or verifying pathnames against a known pattern. Devising these tests is still up to the careful program designer. Instead our analysis offers a "trust" construct that is meant to be applied to data after they have passed the specific validity checks, the analysis will then propagate this knowledge around the program. At points of the program where something dangerous is about to happen, such as starting another program or starting a transaction against a database, the programmer can write a "check" construct to ensure that the arguments can indeed be trusted, which means that they only depend on data that have actually passed the validity checks.

As an example of a run-time version of trust-checking, the programming language Perl (Wall & Schwartz, 1991) has a switch that turns on so-called *taint*-checks at run-time that will abort the program with an error message if tainted input data are being used in "dangerous" functions such as `unlink`. There's also as construct to convert tainted data to un-tainted form so that dangerous functions can be used on it. Our trust analysis is inspired by this feature of Perl, but our analysis is entirely static, so we can avoid run-time errors and overhead due to taint-checks.

A static check inevitably loses some flexibility as compared to run-time checking, but in the case of trust checks, it's not very useful to get a run-time trust violation, as such an error would typically occur too late in the datapath for corrective action to be possible. So not much useful flexibility is lost.

In (Ørbæk, 1995) Ørbæk introduced the concept of *trust analysis* for a first order imperative language with pointers using abstract interpretation and constraints. This paper[†] investigates trust analysis as a type system for a pure functional language based on the λ-calculus.

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

[†] An extended abstract of an earlier version of this paper appears in the proceedings of the 1996 Static Analysis Symposium (Palsberg & Ørbæk, 1995).

operational reduction calculus. The calculus is shown to have the Church-Rosser property. We also give a denotational semantics for our language and relate it to the reduction rules. 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 obtain two simulation theorems relating reductions in our calculus to reductions in classical 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, modules and polymorphism, and we relate trust analysis to other program analyses.

1.1 Intuitions and Motivation

We distinguish two kinds of data: *trusted* data and *untrusted* data. Trusted data will typically arise from program constants, company databases, trustworthy persons, cryptographically verified input from a known partner etc. All other pieces of data, such as data obtained via an insecure network connection or from world writable files is regarded as untrusted.

Figure 1 is an abstract picture of the data-dependencies in a typical function. We see that the result of the function depends on both the functions arguments and its environment as symbolized by the paths entering the function through the sides of the box. Suppose the result of that function is stored in a company database or used in a transaction transferring money between accounts. We clearly want to be able to trust the output of that function, regardless of how the result of the function is derived from the arguments and the environment of the function. In order to make the result trustworthy the programmer has inserted certain checks in the function, symbolized by the small circles. However, there are still paths in the function such that untrustworthy information may leak through if the function is called with untrustworthy arguments.

The method we propose to help the programmer ensure that he inserts checks on all the required data-paths in his program is formulated in terms of an annotated type-system. This type-system must be able to type both trusted data and untrusted data as both kinds of data occur naturally in most programs, but still it must be able to distinguish between these two kinds of data.

As this paper is concerned with a higher-order language, functions are data as well so a function is itself either trusted or untrusted. Intuitively it should be clear that if we apply an untrustworthy function then the result of that application is untrustworthy as well, since the function itself may depend on untrustworthy information from the outside. This leads us to our type rule for function application as shown in Figure 8.

Our type system starts from the simply typed λ-calculus and annotates each type constructor occurring in the program with a trustworthiness. A trustworthy boolean has the annotated type $\mathsf{Bool}^{\mathsf{tr}}$, and an untrustworthy function sending trusted booleans to trusted booleans has the annotated type $(\mathsf{Bool}^{\mathsf{tr}} \to \mathsf{Bool}^{\mathsf{tr}})^{\mathsf{dis}}$.

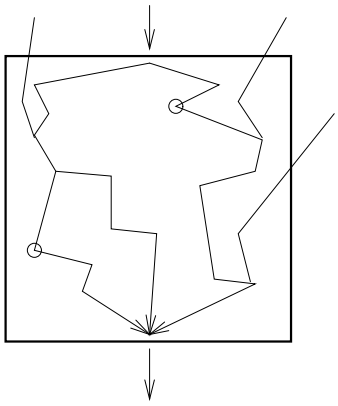

Fig. 1. Dataflow example

Suppose f is a function that can accept an untrusted argument. This must mean that f cannot use that argument in places where a trusted value is required unless it does some checking beforehand on the argument. If we give f a trusted argument then these checks should succeed which means that if a function can accept untrusted input then it can also accept trusted input. This leads us to a type system with subtyping, such that an expression of a trusted type can be typed as an untrusted type. The ordering between base-types as determined by their trustworthiness is extended to higher types using the usual contra/co-variant structural subtyping idea of Mitchell (Mitchell, 1984), Fuh and Mishra (Fuh & Mishra, 1990), Cardelli (Cardelli, 1984) and others.

The trust type system differs from many other type systems in that given a bare value (eg. 7) it is not possible to see by just examining the value whether it is trustworthy or not. This is where our three extensions to the basic λ-calculus come in. They basically link the dynamic semantics of the calculus to our type system, and they can be seen as a kind of annotations to the program that the type inference algorithm will attempt to verify for consistency. The $\mathsf{trust}\ E$ construct indicates to the type system that the result of E may now be trusted. Dually, $\mathsf{distrust}\ E$ indicates that the result of E cannot be trusted, for example after a failing validity check, and last but not least: the $\mathsf{check}\ E$ construct indicates to the analysis that the result of E is *required* to be trustworthy. Well-written programs will only have a few syntactic places where the three new constructs are used. The type system propagates the trust information though the program, ensuring that for any instance of $\mathsf{check}\ E$ in a well-typed program the expression E is statically known to be trustworthy.

1.2 An Example

Consider the following piece of code written in an SML like syntax for a network server program:

$\mathsf{read\_from\_network} :: (\mathsf{Client}^{\mathsf{dis}} \rightarrow (\mathsf{Req}^{\mathsf{dis}}, \mathsf{Sig}^{\mathsf{tr}}))^{\mathsf{tr}}$

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

Fig. 2. The syntax of expressions

```
verify_signature :: (Sig^tr → Bool^tr)^tr
handle_event :: (Req^tr → Unit^tr)^tr
handle_wrong_signature :: ((Req^dis, Sig^tr)→ Unit^tr)^tr

fun get_request client =
    let (req, signature) = read_from_network(client) in
      if verify_signature(signature) then
        handle_event(trust req)
      else
        handle_wrong_signature(req, signature).
```

The server first reads a packet from the network client and if the signature of the packet can be verified, then we can trust the request data and the event handler is called with the request part of the packet. In case the signature cannot be verified an error handler is called which may eventually display some error message on the client's display.

The event handler could be called from many places in the program and to avoid by accident calling it with a request that has not yet been verified, we use the trust type system to require that the handler gets only trustworthy requests. The handler code may then look something like:

```
fun handle_event req =
    let trusted_req = check req
    in ...
```

and it will therefore get the argument type $\mathsf{Req}^{\mathsf{tr}}$. Notice the small number of program annotations of the form $\mathsf{trust}\ E$ and $\mathsf{check}\ E$ as opposed to the numerous trust annotations on the types that are inferred by our inference algorithm.

2 Syntax and Semantics

This section presents the syntax and operational semantics of the trust language. We have chosen to formalize our analysis in an extension of the traditional λ-calculus without any predetermined reduction order, that is, we study a pure calculus to gain results that will specialize equally well to call-by-value implementations as to call-by-need implementations. The drawback of this is of course that some results require more work, for example we need a lengthy proof of the Church-Rosser property of our calculus, something that would come essentially for free had we chosen to study just one specific reduction strategy such as left-most inner-most reduction for call-by-value. Figure 2 defines the syntax of our language.

Variables, λ-abstraction and application behave as usual. The $\mathsf{trust}\ E$ construct is used to introduce trusted values in a program. Symmetrically, distrust indicates

untrusted values. The check construct will reduce only on trusted values, so evaluation may get *stuck* if an expression check(distrust E) occurs at some point during evaluation. According to the previous section, the three new constructs should be regarded as the interface between the dynamic semantics of the program and the static analysis (the type system). But in order to prove the soundness of the analysis we have to give some dynamic meaning to the new constructs. This is done in terms of fairly obvious reduction rules for the constructs defining how they interact. One can also see the new constructs as operating on tags associated with all values at run-time, and this is the view taken in the denotational semantics given later.

2.1 Reduction rules

The reduction (or evaluation) rules for the language are given in Figure 3. 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. As discussed in Section 5 this may change in a larger scale situation with modules.

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 (Barendregt, 1981, pp. 59–62).

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 then explicitly trusting it is the same as just checking the expression (Check Contraction). 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 trustworthiness of E. The reason this particular choice of reduction rules is that we want check to act in a "call-by-value" fashion as discussed in the next section.

In the following we always consider equality of terms modulo α-renaming. Since we are working with a reflexive reduction relation we have to be careful in our definition of what is meant by a normal form. A *significant reduction* $E \to F$ is a reduction whose derivation uses at least one of the contraction rules or a β-rule. A term E is said to be in *normal form* when there are no significant reductions[‡] starting from E. There are *proper* and *improper* normal forms. A normal form containing a sub-term of the form check(distrust E) is said to be improper. All other normal forms are proper. We will write $\to^*$ for the reflexive transitive closure of the reduction relation $\to$.

[‡] A reduction $E \to F$ may be significant even when $E = F$: $\Omega = (\lambda x.xx)(\lambda x.xx) \to \Omega$ is a significant reduction.

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

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

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

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

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

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

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

Fig. 3. The reduction rules.

As an example of how to extend the language with usual programming constructs, we show in Figure 4 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, and how the trustworthiness of the condition in an if-then-else construct affects the trustworthiness of the result. It shows that our function application rule seamlessly handles what Denning in (Denning, 1976) called indirect data dependencies. In Section 5 we also show how to encode a rec construct in the language.

2.2 The nature of check

The contraction rules that we have in the case where check is the inner construction are given by the (Check Contraction) rules:

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

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

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

Fig. 4. Example rules.

$$\frac{E \to \mathsf{check}\ F}{\begin{array}{c}\mathsf{trust}\ E \to \mathsf{check}\ F \\ \mathsf{check}\ E \to \mathsf{check}\ F\end{array}}$$

and most notably, there is no rule for contracting $\mathsf{distrust}(\mathsf{check}\ E)$. There is at least one other set of rules for this case that may come to mind, namely this set of rules:

$$\frac{E \to \mathsf{check}\ F}{\begin{array}{c}\mathsf{trust}\ E \to \mathsf{trust}\ F \\ \mathsf{distrust}\ E \to \mathsf{distrust}\ F \\ \mathsf{check}\ E \to \mathsf{check}\ F\end{array}}$$

The intuition for the first of these alternative rules is that if we put trust around some expression there is really no need to perform the check inside the trust construct since the program is going to trust the resulting value anyway. The second rule is the symmetric case, and is needed to make the resulting calculus Church-Rosser. The last rule also occurs in our system. The calculus that results from this alternative set of rules makes fewer programs end up in a stuck configuration, because it is now possible to place a stuck expression in a context that will make it reducible, as in $\mathsf{trust}(\mathsf{check}(\mathsf{distrust}\ E))$ which is stuck in our calculus, but reduces to $\mathsf{trust}\ E$ under the alternative rules.

One way to think about this is to view our definition of check as "call-by-value" in that it really needs to see the (possibly implicit) tag on its subexpression before it can be reduced away, whereas with the alternative rules, check is "call-by-name" in that it can be reduced away, depending on its context, without considering the trustworthiness of its argument. The "call-by-value" nature of check is also reflected in the denotational semantics of check given later. Note, however, that for the core λ-calculus we have the full β-rule. It's only the new construct, check, that behaves in a "call-by-value" or "call-by-name" fashion.

In our view the alternative rules are less intuitive to the programmer, in that if he writes check somewhere in the program he probably wants it to check the trustworthiness of its argument regardless of the surrounding context. But one can argue both ways: the "call-by-name" version of check might be the most natural choice in a lazy implementation of a functional language such as Haskell, whereas the "call-by-value" version might be most suitable for a call-by-value language such as SML. Either way it's easy to alter the proof of the Subject Reduction theorem (Theorem 11) for our type system to the alternative rules, so our type system is sound for both sets of rules.

2.3 Church-Rosser

The Church-Rosser (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.

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

Proof
By the Diamond lemma below (Lemma 7) and Lemma 3.2.2 of (Barendregt, 1981). □

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

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

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

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

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

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

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 $E \rightarrow F$ and $G \rightarrow H$ then $E[G/x] \rightarrow F[H/x]$.

Proof
By induction on the structure of the derivation of $E \rightarrow F$. If $F = E$ by the (Reflex) axiom, we must show that $E[G/x] \rightarrow E[H/x]$ given that $G \rightarrow H$. 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

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

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

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

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

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

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.

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

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

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

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

By the induction hypothesis $E_1[G/x] \rightarrow F_1[H/x]$ 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:

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

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 $\mathsf{distrust}\ \lambda x.E$ case. □

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. $E = \mathsf{trust}\ E_1$:
   (a) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \lambda x.E_1'$ when $E_1 \to \lambda x.E_1'$. (Lambda Contraction)
   (c) $E \to \mathsf{check}\ E_1'$ when $E_1 \to \mathsf{check}\ E_1'$. (Check Contraction)
   (d) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{distrust}\ E_1'$. (Distrust Contraction)
3. $E = \mathsf{distrust}\ E_1$:
   (a) $E \to \mathsf{distrust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \mathsf{distrust}\ E_1'$ when $E_1 \to \mathsf{distrust}\ E_1'$. (Distrust Contraction)
4. $E = \mathsf{check}\ E_1$:
   (a) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)
   (b) $E \to \mathsf{check}\ E_1'$ when $E_1 \to \mathsf{check}\ E_1'$. (Check Contraction)
   (c) $E \to \lambda x.E_1'$ when $E_1 \to \lambda x.E_1'$. (Lambda Contraction)
5. $E = (\lambda x.E_1)E_2$: $E \to E_1'[E_2'/x]$ when $E_1 \to E_1'$ and $E_2 \to E_2'$.
6. $E = (\mathsf{distrust}\ \lambda x.E_1)E_2$: $E \to \mathsf{distrust}\ E_1'[E_2'/x]$ when $E_1 \to E_1'$ 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 $E \to F$ and $E \to G$ were:

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

and

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

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

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

and by (Application)

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

Case 5 without 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 $E \to F$ and $E \to G$ were

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

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

By the induction hypothesis there is an H_1 such that $G_1 \to H_1$ and $\beta\ F_1 \to H_1$. 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), $G_1 \to \gamma\ H_2$ implies that $\alpha\ G_1 \to \gamma\ H_2 = H_1$. 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 $E \to F$ and $E \to G$ are:

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

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

By the induction hypothesis there is an H_1 such that $\beta\ F_1 \to H_1$ and $\gamma\ G_1 \to H_1$. 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 $E \to F$ and $E \to G$ are:

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

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

By the induction hypothesis we know there exists H_1 such that $\alpha\ F_1 \to H_1$. 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 $\mathsf{distrust}\ G_1 \to H_1$. And here Lemma 2 says that $H_1 = \mathsf{distrust}\ H_2$! This is a contradiction so it cannot be the case that both $E_1 \to \alpha\ F_1$ and $E_1 \to \mathsf{distrust}\ G_1$.

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

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

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

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

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

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

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

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

$$\frac{E_1 \to G_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ G_1 = G}(\text{Sub})$$

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

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

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

$$\frac{E_1 \to G_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ G_1 = G}(\text{Sub})$$

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

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

If $\alpha = \mathsf{check}$ or $\alpha = \lambda x.$ then by (Sub) we get $\mathsf{distrust}\ G_1 \to \mathsf{distrust}\ (\alpha\ H_2)$. Since $\mathsf{trust}\ F_1 \to \alpha\ H_2$ 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 $F_1 \to \alpha\ H_2$. Now by (Sub) we get $\mathsf{distrust}\ F_1 \to \mathsf{distrust}\ (\alpha\ H_2)$. So here we may use $H = \mathsf{distrust}\ (\alpha\ H_2)$.

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

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

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

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

This concludes the proof of the Diamond lemma. □

2.4 Denotational Semantics

In this section we give a denotational semantics for the calculus and prove it sound with respect to the calculus. Some readers may find the direct style denotational semantics easier to comprehend than the reduction rules. However, as we saw in the section on the nature of check, the reduction rules are easy to alter to get a different, but justifiable, semantics of check, whereas the denotational semantics for

$$
\begin{array}{rcl}
D & = & (((D \to D) \oplus \mathsf{base}) \otimes \{\mathsf{tr}, \mathsf{dis}\}_\perp) \oplus \{\mathsf{error}\}_\perp. \\
Env & = & Var \to D. \\
[\![\cdot]\!] & \in & Exp \to Env \to D \\
E, F & \in & Exp \\
x & \in & Var \\
\rho & \in & Env.
\end{array}
$$

Fig. 5. Domain equations.

$$
\begin{array}{lcl}
[\![x]\!]\rho & = & \rho(x) \\
[\![\lambda x.E]\!]\rho & = & \langle(\lambda d : D.\ \mathsf{case}\ [\![E]\!]\rho[x \mapsto d]\ \mathsf{of} \\
 & & \qquad \mid \mathsf{error} \to \mathsf{error} \\
 & & \qquad \mid \langle v, t\rangle \to \langle v, t\rangle), \mathsf{tr}\rangle \\
[\![EF]\!]\rho & = & \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \mid \mathsf{error} \to \mathsf{error} \\
 & & \mid \langle v, t\rangle \to \ \mathsf{case}\ v([\![F]\!]\rho)\ \mathsf{of} \\
 & & \qquad \mid \mathsf{error} \to \mathsf{error} \\
 & & \qquad \mid \langle v', t'\rangle \to \langle v', t \vee t'\rangle \\
[\![\mathsf{trust}\ E]\!]\rho & = & \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \mid \mathsf{error} \to \mathsf{error} \\
 & & \mid \langle v, t\rangle \to \langle v, \mathsf{tr}\rangle \\
[\![\mathsf{distrust}\ E]\!]\rho & = & \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \mid \mathsf{error} \to \mathsf{error} \\
 & & \mid \langle v, t\rangle \to \langle v, \mathsf{dis}\rangle \\
[\![\mathsf{check}\ E]\!]\rho & = & \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \mid \mathsf{error} \to \mathsf{error} \\
 & & \mid \langle v, \mathsf{tr}\rangle \to \langle v, \mathsf{tr}\rangle \\
 & & \mid \langle v, \mathsf{dis}\rangle \to \mathsf{error}.
\end{array}
$$

Fig. 6. Semantic equations.

the alternative set of rules would be substantially different, as we would have to abandon the direct style and instead use a continuation based semantics.

In Figure 5 we define our semantic domains, using coalesced sums and smash products. A value is either error (standing for a semantic error) or a pair consisting of the "real" value (a function or a value of base-type from the base domain) and a trust tag, either tr or dis. The domain $\{\mathsf{tr}, \mathsf{dis}\}_\perp$ is the usual flat three-point domain where $\perp$ is the bottom element. For the definition of application we define another ordering between tr and dis, namely: $\mathsf{tr} \leq \mathsf{dis}$. We denote by $\vee$ the least upper bound according to this ordering.

$$\begin{array}{lcl} u, v, w & ::= & \mathsf{dis} \mid \mathsf{tr} \\ \tau, \sigma & ::= & t^u \\ s, t & ::= & \mathsf{base} \mid \tau \to \sigma. \end{array}$$

Fig. 7. Syntax of trust-types

The semantic equations are given in Figure 6. We employ a pattern matching case construct in the semantic description language.

Lemma 8 (Environment)
For expressions E and F and environment ρ we have:

$$[\![E[F/x]]\!]\rho = [\![E]\!]\rho[x \mapsto [\![F]\!]\rho].$$

Proof
By structural induction on E. □

The connection between the operational calculus and the denotational semantics is the following soundness theorem.

Theorem 9 (Semantic Soundness)
The denotation of an expression is invariant under reduction: If $E \to F$ then $[\![E]\!] = [\![F]\!]$.

Proof
By induction on the derivation of $E \to F$, applying the equational theory of the semantic description language and using Lemma 8 in the application case. □

3 The Type System

Our annotated type system is based on Currys monomorphic type system for the λ-calculus, also known as *simply typed* λ-calculus. Figure 7 shows the mutually recursive definition of the syntax of our types. Recall from Section 1.1 that tr means that the value is trusted and dis means that it is untrusted.

We write u, v or w (and primed and subscripted versions thereof) for trust annotations, s and t for bare types without their outermost annotation, and σ or τ for annotated types. The generic term "type" will be used both for bare types without their outermost annotation and for annotated types.

Just as was the case in the denotational semantics, trust annotations are subject to a partial ordering $\leq$, defined as the least partial ordering including the relation $\mathsf{tr} \leq \mathsf{dis}$. This ordering is extended to bare types and annotated types such that two bare base-types are ordered only if they identical; for annotated types we have:

$$t^u \leq s^v \text{ if and only if } u \leq v \text{ and } t \leq s,$$

and for bare arrow types we define:

$$\tau \to \sigma \leq \tau' \to \sigma' \text{ if and only if } \tau' \leq \tau \text{ and } \sigma \leq \sigma',$$

so argument types are ordered contravariantly. This is inspired by the work on

$$A \vdash x : \tau \text{ if } x \in \mathrm{dom}(A) \text{ and } A(x) = \tau \quad \text{(Var)}$$

$$\frac{A \vdash E : \tau \quad \tau \leq \tau'}{A \vdash E : \tau'} \quad \text{(Sub)}$$

$$\frac{A[x \mapsto \tau] \vdash E_1 : \sigma}{A \vdash \lambda x.E_1 : (\tau \to \sigma)^{\mathsf{tr}}} \quad \text{(Lambda)}$$

$$\frac{A \vdash E_1 : (\tau \to t^u)^w \quad A \vdash E_2 : \tau}{A \vdash E_1 E_2 : t^{u \vee w}} \quad \text{(App)}$$

$$\frac{A \vdash E_1 : t^u}{A \vdash \mathsf{trust}\ E_1 : t^{\mathsf{tr}}} \quad \text{(Trust)}$$

$$\frac{A \vdash E_1 : t^u}{A \vdash \mathsf{distrust}\ E_1 : t^{\mathsf{dis}}} \quad \text{(Distrust)}$$

$$\frac{A \vdash E_1 : t^{\mathsf{tr}}}{A \vdash \mathsf{check}\ E_1 : t^{\mathsf{tr}}} \quad \text{(Check)}$$

Fig. 8. The type system.

structural subtyping by Mitchell (Mitchell, 1984), Fuh and Mishra (Fuh & Mishra, 1990), Cardelli (Cardelli, 1984) and others.

As in the denotational semantics we denote by $\vee$ the least upper bound operation on the trust lattice according to the $\leq$ ordering. In Section 5 we discuss several extensions of the type system to cope with recursion, modules, polymorphism and more general lattices of annotations.

3.1 Rules

A type assumption A is a partial function which takes a program identifier to an annotated type τ. Figure 8 shows the inference rules for the type system. A type judgment $A \vdash E : \tau$ means that from the assumptions A we can deduce that the expression E has type τ.

The rule for variables and the subtyping rule should give no surprises. Since lambda abstractions in the program are written by the programmer, we treat them as trusted in the type system. This only means that the *function* is trusted, and does not indicate how its argument and result are treated. In Section 5 we discuss how this can be extended to a larger scale setting with multiple modules written by multiple programmers.

In the application rule, the annotated type of the actual argument is required to match the annotated type of the formal argument. This includes the trustworthiness. 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.

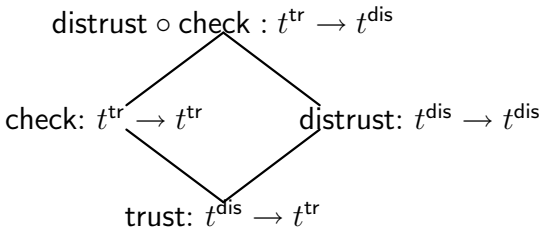

Fig. 9. The relationship between arrow types

The three rules for trust, distrust and check show that they behave as the identity on the underlying type. Trust makes any value trusted and distrust makes any value untrusted. Check E has a type only if E is trusted. This means that we cannot type improper normal forms and together with Subject Reduction (Theorem 11) this ensures the soundness of the type system.

Figure 9 shows the ordering of arrow types and how the constructs trust, distrust and check would fit into it.

3.2 Subject Reduction

An important part in proving the Subject Reduction theorem is that replacing an appropriately typed term for a variable in an expression does not change the type of the expression.

Lemma 10 (Substitution)
If $A[x \mapsto \sigma] \vdash E : \tau$ and $A \vdash F : \sigma$ then $A \vdash E[F/x] : \tau$.

Proof
By induction on the derivation of $A[x \mapsto \sigma] \vdash E : \tau$. □

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

Theorem 11 (Subject Reduction)
If $A \vdash E : \tau$ and $E \to F$ then $A \vdash F : \tau$.

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 τ then E *must* have type σ") 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 $A \vdash E_1 : t_1^{u_1}$

where $t_1 \leq t$. By the induction hypothesis we now get $A \vdash \mathsf{trust}\ F_1 : t_1^{u_1}$ and again we must have $A \vdash F_1 : t_2^{u_2}$ where $t_2 \leq t_1$. Now by the (Trust) rule of the type system we get $A \vdash \mathsf{trust}\ F_1 : t_2^{\mathsf{tr}}$ 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 $E \to F$ was

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

By assumption we know $A \vdash \mathsf{distrust}\ E_1 : t^u$ and therefore $u = \mathsf{dis}$, so by the rules we must have $A \vdash E_1 : t_1^{u_1}$ where $t_1 \leq t$. From the induction hypothesis we get $A \vdash \mathsf{distrust}\ F_1 : t_1^{u_1}$. By the (Distrust) rule we must have $A \vdash F_1 : t_2^{u_2}$ where $t_2 \leq t_1$ and $u_1 = \mathsf{dis}$. By the (Distrust) rule we now get $A \vdash \mathsf{distrust}\ F_1 : t_2^{\mathsf{dis}}$ which via (Sub) yields the required result.

Regarding application: If $E = E_1E_2 \to F_1F_2 = F$ then the result follows by two applications of the induction hypothesis. If the last rule used in the derivation of $E \to F$ was

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

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 $A \vdash \mathsf{distrust}(\lambda x.E_1) : (\sigma \to s^v)^w$ and $A \vdash E_2 : \sigma$ where $s \leq t$ and $v \vee w \leq u$. Again, we must also have $A \vdash \lambda x.E_1 : (\sigma_1 \to s_1^{v_1})^{w_1}$ and it must be case that $w = \mathsf{dis}$ and thus $u = \mathsf{dis}$. Also $s_1 \leq s$, $v_1 \leq v$, and $\sigma \leq \sigma_1$. Once again by the type rules we must have $A[x \mapsto \sigma_2] \vdash E_1 : s_2^{v_2}$ where $\sigma_1 \leq \sigma_2$, $s_2 \leq s_1$ and $v_2 \leq v_1$. By the (Sub) rule we get $A \vdash E_2 : \sigma_2$.

We can now apply the induction hypothesis to get $A[x \mapsto \sigma_2] \vdash F_1 : s_2^{v_2}$ and $A \vdash F_2 : \sigma_2$. By the Substitution lemma (Lemma 10) we then get $A \vdash F_1[F_2/x] : s_2^{v_2}$. By the (Distrust) rule we get $A \vdash \mathsf{distrust}\ F_1[F_2/x] : s_2^{\mathsf{dis}}$ and by (Sub) we get the desired result as $s_2 \leq s_1 \leq s \leq t$.

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:

$$\begin{array}{lcllcl} |x| & = & x & |\lambda x.E| & = & \lambda x.|E| \\ |E_1E_2| & = & |E_1||E_2| & |\mathsf{trust}\ E| & = & |E| \\ |\mathsf{distrust}\ E| & = & |E| & |\mathsf{check}\ E| & = & |E| \end{array}$$

and likewise the erasure of an annotated type as:

$$|\mathsf{base}^u| \ = \ \mathsf{base} \qquad |(\sigma \to \tau)^w| \ = \ |\sigma| \longrightarrow |\tau|.$$

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

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

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

Fig. 10. The Curry type system.

The Curry type rules for erased expressions are defined in Figure 10. Here type assumptions A map program identifiers to Curry types.

Lemma 12 (Erasure)
If σ and τ are annotated types and $\sigma \leq \tau$ then $|\sigma| = |\tau|$.

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

Theorem 13
If $A \vdash E : \tau$ then $|A| \vdash_C\ |E| : |\tau|$.

Proof
By induction on the derivation of $A \vdash E : \tau$.

- $E =$ x: By assumption we have $x \in \mathrm{dom}(A)$ and $\tau = A(x)$. Thus $x \in \mathrm{dom}(|A|)$ and $|A|(x) = |\tau|$.
- $E =$ $\alpha\ E_1$: Here $\alpha \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. By the definition of erasure, $|E| = |E_1|$ and the result follows from the induction hypothesis.
- $E =$ $\lambda x.E_1$: By assumption we must have $A[x \mapsto \sigma] \vdash E_1 : \sigma'$ where $(\sigma \to \sigma')^{\mathsf{tr}} \leq \tau$. By the induction hypothesis $|A|[x \mapsto |\sigma|] \vdash_C\ |E_1| : |\sigma'|$. By the lambda rule in the Curry system we get $|A| \vdash_C\ \lambda x.|E_1| : |\sigma| \longrightarrow |\sigma'|$. By definition of erasure and the Erasure lemma we get the desired result.
- $E =$ $E_1 E_2$: By assumption we must have $A \vdash E_1 : (\sigma \to t^u)^w$ and $A \vdash E_2 : \sigma$ where $t^{u \vee w} \leq \tau$. From the induction hypothesis we get $|A| \vdash_C\ |E_1| : |\sigma| \longrightarrow |t^u|$ and $|A| \vdash_C\ E_2 : |\sigma|$. By the application rule in the Curry system we get $|A| \vdash_C\ |E_1 E_2| : |t^u|$. Finally by the Erasure lemma we get the desired result.

□

The preceding theorem also implies that the erasure of a trust-typable program is trust-typeable, but the reverse does not hold in general. However, if there are no sub-terms 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 dis. The formalisation of this and its simple proof has been elided from the paper as it will not be used in the following.

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

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

Fig. 11. Reductions in the ordinary λ-calculus.

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 11 (this is taken from Definition 3.2.3 in (Barendregt, 1981).) 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 11 is a sub-relation of the reduction relation defined in Figure 3.

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 type-checking, an interpreter may erase all the constructs having to do with trust, and run the program without them, thus no run-time performance penalty is paid.

Lemma 14 (Step)
If $E \to^* \alpha \ F$ (α may be empty) and there is a reduction rule

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

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

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

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

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

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

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

Similarly, from

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

□

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 15 (CTD)
If $E = \mathsf{CTD}^*(\lambda x.E_1)$, $A \vdash E : \tau$ and $E_1 \to^* F_1$ then $E \to^* \mathsf{distrust}^?\ (\lambda x.F_1)$.

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 16 (Simulation 1)
If $A \vdash E : \tau$ and $|E| \to F$ then there is a term G such that $E \to^* G$ and $|G| = F$. Graphically:

$$\begin{array}{ccc} E & \overset{\to^*}{\dashrightarrow} & G \\ {\scriptstyle |\cdot|}\downarrow & & \downarrow{\scriptstyle |\cdot|} \\ |E| & \xrightarrow{\ \to\ } & F \end{array}$$

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 $|E| = |E_1|$, $A \vdash E_1 : \tau'$ 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 $A[x \mapsto \sigma] \vdash E_1 : \sigma'$. Also, $|E| = \lambda x.|E_1|$ and $F = \lambda x.F_1$. By the nature of the reduction rules, we must have $|E_1| \to F_1$. By the induction hypothesis we know there is a G_1 such that $E_1 \to^* G_1$ and $|G_1| = F_1$. By the (Sub) rule and the Step lemma we get

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

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 $|E_1| \to F_1$ and $|E_2| \to F_2$. By the induction hypothesis we get G_1 and G_2 such that $E_1 \to^* G_1$, $E_2 \to^* G_2$, $|G_1| = F_1$ 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 $|E_1| \to F_1$ and $|E_2| \to F_2$ 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 $H_1 \to F_1'$ and $H_2 \to F_2$ then also $|E_1| = \lambda x.H_1 \to \lambda x.F_1' = F_1$ by (Sub). By the above statements and the Step lemma we get $E \to^* G_1G_2$ and $|G_1G_2| = F$.

Otherwise a β-reduction happens. Here $H_1 \to F_1'$, $H_2 \to F_2$ 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 $Q_1 \to^* Q_1'$ and $|Q_1'| = F_1'$.

By the CTD lemma $E_1 \to^* \mathsf{distrust}^?\ (\lambda x.Q_1')$ and by the Subject Reduction theorem (Theorem 11) we get that $\mathsf{distrust}^?\ (\lambda x.Q_1')$ is well-typed.

We may now reason as follows:

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

and

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

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 17 (Simulation 2)
If $E \to F$ then $|E| \to |F|$.

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.

□

Well-typed terms in our extended calculus are strongly normalizing, that is: If $A \vdash E : \tau$ then there is a normal form G such that $E \to G$. This is proved by an argument using Theorem 13, Strong Normalization for Curry typed λ-calculus, Theorem 1 and the two simulation theorems.

4 Type Inference

The type inference problem is:

Given an untyped program E possibly with trust, distrust, and check expressions in it, is E typable? If so, annotate it.

From Theorem 13 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 $V_1^{W_1} \to V_2^{W_2}$, and where $V_1, V_2 \in \mathcal{V}_y$ and $W_1, W_2 \in \mathcal{V}_r$.*
- *D is a finite set of constraint of the forms $W \leq W'$, $W = \mathsf{tr}$, or $W = \mathsf{dis}$, 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 without their outermost annotation, 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.* □

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	$x_y \leq [\![x]\!]_y$	$x_r \leq [\![x]\!]_r$
$\lambda x.F$	$x_y^{x_r} \to [\![F]\!]_y^{[\![F]\!]_r} \leq [\![\lambda x.F]\!]_y$	
GH	$[\![G]\!]_y \leq [\![H]\!]_y^{[\![H]\!]_r} \to [\![GH]\!]_y^{\langle GH\rangle_r}$	$[\![G]\!]_r \leq [\![GH]\!]_r$ $\langle GH\rangle_r \leq [\![GH]\!]_r$
trust F	$[\![F]\!]_y \leq [\![\mathsf{trust}\ F]\!]_y$	
distrust F	$[\![F]\!]_y \leq [\![\mathsf{distrust}\ F]\!]_y$	$[\![\mathsf{distrust}\ F]\!]_r = \mathsf{dis}$
check F	$[\![F]\!]_y \leq [\![\mathsf{check}\ F]\!]_y$	$[\![F]\!]_r = \mathsf{tr}$

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 $A(x) = \delta(x_y)^{\varphi(x_r)}$.

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 agrees with the two functions on their respective domains.

Lemma 19 (Soundness)
If $(\delta, \varphi) \models T(E)$, and δ, φ extend A then $A \vdash E : \delta([\![E]\!]_y)^{\varphi([\![E]\!]_r)}$.

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 $\delta([\![E]\!]_y) = t$ and $\varphi([\![E]\!]_r) = u$.

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

- $E =$ x: As $A \vdash x : t^u$ we must have $x \in \mathrm{dom}(A)$, $A(x) = s^v$ and $s \leq t, v \leq u$. In this case $T(E) = \{x_y \leq [\![x]\!]_y, x_r \leq [\![x]\!]_r\}$. Put $\delta(x_y) = s$ and $\varphi(x_r) = v$ so that (δ, φ) extends A. Finally assign $\delta([\![x]\!]_y) = t$ and $\varphi([\![x]\!]_r) = u$ to satisfy the constraints.
- $E =$ $\lambda x.F$: By the type rules we must have $A[x \mapsto \sigma] \vdash F : s^v$ where $\sigma \to s^v \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta([\![F]\!]_y) = s$, $\varphi([\![F]\!]_r) = v$, and (δ, φ) extends $A[x \mapsto \sigma]$. Now assign $\delta' = \delta[[\![\lambda x.F]\!]_y \mapsto t]$ and $\varphi' = \varphi[[\![\lambda x.F]\!]_r \mapsto u]$. Now check that $\sigma \to s^v \leq t$ implies

$$\delta'(x_y)^{\varphi'(x_r)} \to \delta'([\![F]\!]_y)^{\varphi'([\![F]\!]_r)} \leq \delta'([\![\lambda x.F]\!]_y)$$

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

$E =$ GH: By the type rules we must have $A \vdash G : (\sigma \to s^v)^w$, $A \vdash H : \sigma$ where $s \leq t$ and $v \vee w \leq u$. 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

$$T(E) = T(G) \cup T(H) \cup (\{[\![G]\!]_y \leq [\![H]\!]_y^{[\![H]\!]_r} \to [\![GH]\!]_y^{\langle GH\rangle_r}\}, \\ \{[\![G]\!]_r \leq [\![GH]\!]_r,\ \langle GH\rangle_r \leq [\![GH]\!]_r\}) \ .$$

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

$$\delta([\![G]\!]_y) \leq \sigma \to t^v = \delta'([\![H]\!]_y)^{\varphi'([\![H]\!]_r)} \to \delta''([\![GH]\!]_y)^{\varphi''(\langle GH\rangle_r)}$$

$$\begin{aligned} \varphi([\![G]\!]_r) = w &\leq u = \varphi''([\![GH]\!]_r) \\ \varphi''(\langle GH\rangle_r) = v &\leq u = \varphi''([\![GH]\!]_r) \ . \end{aligned}$$

and clearly (δ'', φ'') extend A.

$E =$ $\mathsf{check}\ F$: From the type rules we must have $A \vdash F : s^{\mathsf{tr}}$ where $s \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta([\![F]\!]_y) = s$, and $\varphi([\![F]\!]_r) = \mathsf{tr}$. Now, $T(E) = T(F) \cup (\{[\![F]\!]_y \leq [\![\mathsf{check}\ F]\!]_y\}, \{[\![F]\!]_r = \mathsf{tr}\})$. Put $\delta' = \delta[[\![\mathsf{check}\ F]\!]_y \mapsto t]$ and $\varphi' = \varphi[[\![\mathsf{check}\ F]\!]_r \mapsto u]$.
Clearly, δ', φ' extend A, $\delta'([\![F]\!]_y) \leq \delta'([\![\mathsf{check}\ F]\!]_y)$, and $\varphi'([\![F]\!]_r) = \mathsf{tr}$ as required. The cases for trust and $\mathsf{distrust}$ are very similar.

□

Theorem 21
The judgment $A \vdash E : t^u$ is derivable if and only if there exists a solution (δ, φ) of $T(E)$ with (δ, φ) extending A such that $\delta([\![E]\!]_y) = t$ and $\varphi([\![E]\!]_r) = u$. 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 $\delta([\![E]\!]_y) = t$ and $\varphi([\![E]\!]_r) = u$.

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 $V_1^{W_1} \to V_2^{W_2} \leq V_3^{W_3} \to V_4^{W_4}$ is in $\bar{C}$, then $V_3 \leq V_1$ and $V_2 \leq V_4$ are in $\bar{C}$, and $W_3 \leq W_1$ and $W_2 \leq W_4$ 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 8 and from the constraints defined earlier in this section, we obtain two equivalent formulations of Curry typability (Palsberg & Schwartzbach, 1995). 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	$[\![G]\!]_y = [\![H]\!]_y \to [\![GH]\!]_y$
$\mathsf{trust}\ F$	$[\![\mathsf{trust}\ F]\!]_y = [\![F]\!]_y$
$\mathsf{distrust}\ F$	$[\![\mathsf{distrust}\ F]\!]_y = [\![F]\!]_y$
$\mathsf{check}\ F$	$[\![\mathsf{check}\ F]\!]_y = [\![F]\!]_y$

To aid the definition of our type inference algorithm we define the following operations: If s, t are bare trust types such that $|s| = |t|$, then define the operators $\sqcup^{|s|}$ and $\sqcap^{|t|}$ as follows.

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

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

If $t_1, \ldots, t_n$ are bare 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

$$\begin{array}{rclrcl} \mathsf{small}(\mathsf{base}) & = & \mathsf{base} & \mathsf{big}(\mathsf{base}) & = & \mathsf{base} \\ \mathsf{small}(s \to t) & = & \mathsf{big}(s)^{\mathsf{dis}} \to \mathsf{small}(t)^{\mathsf{tr}} & \mathsf{big}(s \to t) & = & \mathsf{small}(s)^{\mathsf{tr}} \to \mathsf{big}(t)^{\mathsf{dis}} \end{array}$$

If s is a bare 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* bare type with erasure t.

For each constraint expression X define

$$L(C, X) = \{V_1^{W_1} \to V_2^{W_2} \mid V_1^{W_1} \to V_2^{W_2} \leq X \text{ is in } \bar{C}\} \ .$$

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:

$$\begin{aligned} |V| &= V \\ |V_1^{W_1} \to V_2^{W_2}| &= V_1 \to V_2 \end{aligned}$$

where $V_1, V_2 \in \mathcal{V}_y$ and $W_1, W_2 \in \mathcal{V}_r$.

Lemma 24

If $T(E) = (C, D)$, and ψ is a solution to $\mathsf{Curry}(E)$, and $X_1 \leq X_2$ 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 13 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.

$$\begin{aligned} \delta(V) = \mathsf{if}\ & \psi(|V|) = \mathsf{base}\ \mathsf{then}\ \mathsf{base} \\ & \mathsf{else\ let}\ \{V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}\} = L(C, V) \cup \{\mathsf{small}(\psi(|V|))\} \\ & \quad \mathsf{in}\ \bigsqcup_i^{\psi(|V|)} (\delta(V_{1i})^{\varphi(W_{1i})} \to \delta(V_{2i})^{\varphi(W_{2i})}) \end{aligned}$$

To see that δ is well-defined, we need that the Curry types of the variables V_{1i} and V_{2i} are of strictly less size than the Curry type of V. For $(V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}) \in L(C, V)$, we get by Lemma 24 that $\psi(|V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}|) = \psi(|V|) = s \to t$ for some s, t. This means that $\psi(|V_{1i}|) = s$ and $\psi(|V_{2i}|) = t$ 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 $X_1 \leq X_2$ 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 $\delta(X_1) \leq \delta(X_2)$ 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 $\delta(X_1) \leq \delta(X_2)$ as required. □

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

Input: A λ-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 (Palsberg, 1995)).
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

In this section we discuss several possible extensions of the type system.

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:

$$\frac{E \rightarrow^* F}{\mathsf{Y}E \rightarrow^* F(\mathsf{Y}E)} \qquad \frac{A[x \mapsto \tau] \vdash E_1 : \tau}{A \vdash \mathsf{rec}\ x.E_1 : \tau}$$

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.

Finding a good type inference algorithm for a type-system with both structural subtyping and polymorphism is a nontrivial task although the work by Aiken and Wimmers (Aiken & Wimmers, 1993) and by Eifrig, Smith and Trifonov (Eifrig *et al.*, 1995) is promising.

A Trust-case Construction. One could imagine the usefulness of a trust-case construction that would allow dynamic dispatch on the trustworthiness of a value. The reduction rules added for such a construction could be:

$$\frac{E \rightarrow \mathsf{trust}\ E'}{\mathsf{trust\_case}\ E\ F\ G \rightarrow F(\mathsf{trust}\ E')} \qquad \frac{E \rightarrow \lambda x.E'}{\mathsf{trust\_case}\ E\ F\ G \rightarrow F(\lambda x.E')}$$

$$\frac{E \rightarrow \mathsf{distrust}\ E'}{\mathsf{trust\_case}\ E\ F\ G \rightarrow G(\mathsf{distrust}\ E')}$$

and the corresponding type-rule:

$$\frac{A \vdash E : t^u \quad A \vdash F : t^{\mathsf{tr}} \to \tau \quad A \vdash G : t^{\mathsf{dis}} \to \tau}{A \vdash \mathsf{trust\_case}\ E\ F\ G : \tau}$$

Church-Rosser and the Subject Reduction theorem still holds with these extensions and generating constraints for this construct is not hard either. However, this would only make sense in the presence of a polymorphic trust type system. With monomorphic trust-types all the trust-case choices would be statically determinable from the type system, so such a construction would be of very limited use. And since we have not developed an inference algorithm for a polymorphic trust type system, this has not been an issue.

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 to be modeled at once: Take the four point lattice $(\mathcal{P}(\{\mathsf{path\_ok}, \mathsf{signature\_ok}\}), \subseteq)$ with the empty set denoting completely untrusted. This could be used in a web server that can both verify digital signatures and do consistency checking on URL paths. In such a situation one would extend check to a construct checking for reverse subset inclusion.

Modules. In a larger scale system with many program modules and many programmers, it is useful to differentiate between functions located in different modules such that there would be trusted and untrusted modules, where functions defined in untrusted modules would not be trusted in any other module. This can be realized in our simple system by having a preprocessor that wraps all lambdas in an untrusted module in the distrust construct. Some external programming environment might also be used to ensure that only trustworthy programmers get to write trusted modules.

One might also make another distinction among modules, akin to the difference between safe and unsafe modules in Modula-3 (Cardelli *et al.*, 1989), where only unsafe modules are allowed to use arbitrary type casts and unlimited address arithmetic. In a trust analysis system, unsafe modules would then correspond to modules where the trust construct is used, and just as in Modula-3 one has to take extra care in the unsafe modules.

6 Related Work

The original notion of trust analysis was presented in (Ørbæk, 1995) where an abstract interpretation analysis and a constraint based analysis for an imperative, first order language with pointers were given. This work extends trust analysis to the higher order functional case and formalizes it in terms of an annotated type system.

In (Mitchell, 1984) 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.

In an earlier version of the paper we used a different syntax for trust-types inspired by the work on effect systems by for example Gifford and Jouvelot, writing for example $\mathsf{Bool} \xrightarrow{\mathsf{tr}\ \mathsf{dis}} \mathsf{Bool}\ \#\ \mathsf{tr}$ for what is now written $(\mathsf{Bool}^{\mathsf{tr}} \to \mathsf{Bool}^{\mathsf{dis}})^{\mathsf{tr}}$. This turned out to be misleading in that our type system does not involve accumulating representations of side effects and input/output. We thank our referees for pointing this out and making us change the syntax of types.

6.1 Why trust analysis isn't...

- **Binding-time analysis**: If trust analysis was equivalent to binding-time analysis then one would equate tr with static and dis with dynamic, and without using any of our special constructs this analogy goes a long way. However, the trust construct would correspond to an unrestricted "down-lift" operation able to convert arbitrary dynamic data to static data, something that is clearly unsound in a binding-time analysis. Our distrust construct would correspond nicely to the lift operation, but again the check construct has no counterpart in binding-time analysis.
- **Security Analysis**: Since the seminal papers by Denning and Denning (Denning, 1976; Denning & Denning, 1977; Denning, 1982), there has been a great deal of work on using static analysis to ensure that classified information would not leak out of information systems, cf. (Banâtre *et al.*, 1994). In security analysis, the basic distinction is between unclassified information and classified (secret) information. The task is to prevent classified information from being shown to unprivileged users. Usually this is done by assigning security classes to users as well as to data, and making sure that information is never transferred from higher (more secret) security classes to lower (less secret) classes.

  Whereas security analysis focuses on preventing classified information leaking *out* of the system to unprivileged users, trust analysis focuses on preventing untrustworthy information flowing *into* the system. In this sense, trust analysis can be viewed as the dual of security analysis.

  A simple example of a security analysis setup would have just two classes: secret and unclassified. The key relation between these classes is that wherever secret information is needed, unclassified information will do just as well, so in analogy with the subtyping system for trust, one can automatically coerce unclassified information into secret information. Continuing this analogy one would equate unclassified with tr and secret with dis, and furthermore the trust construct would correspond to a declassify construct, something rarely, if ever, found in security analysis systems. We think, however, that for trust analysis, the trust construct is a very natural construction, it is in fact the cornerstone of the analysis.
- **Dynamic typing**: Dynamic typing also known as tagging/untagging analysis (Aiken & Murphy, 1991; Henglein, 1992; Wright & Cartwright, 1994) aims to

remove type tags as much as possible in a dynamically typed language. One might be tempted to view, say, distrust as a tagging operation and trust as the corresponding untagging 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 tagged function to an argument does not necessarily result in a tagged result.

One idea is that trust analysis might be used as a kind of soft typing extension to languages like C or C++ which are almost strongly typed, but contain loopholes such as unrestricted type-casts. The idea is to essentially have two copies of every C type, a trusted variant and an untrusted variant, such that the compiler could guarantee no type errors for variables having a trusted type, whereas the compiler could insert run-time checks for values of untrusted types. However, it turns out that this kind of analysis is not equivalent to trust analysis, as illustrated by the following C example:

```
if ((int) p)
  x = 5;
else
  x = 7;
```

Since type-casting is supposed to correspond to distrust and the two assignments to x are dependent on the condition; following (Ørbæk, 1995) x would have to be treated as untrusted after the if-statement. This is not what we want for this kind of analysis, because in both cases x would clearly contain an integer. This illustrates that trust analysis does not serve the purpose of this analysis.

7 Summary

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 as well as a sound denotational semantics. The reduction calculus is proved Church-Rosser. We then 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.

We have related our extension of the λ-calculus to the classical λ-calculus and obtained two simulation theorems, as well as shown that well-typed terms in our calculus are strongly normalizing.

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

Finally we have discussed certain possible extensions to our analysis and given several examples of why it is different from already known analyses.

7.1 Acknowledgements

Part of this work was completed while both authors were visiting the Laboratory for Computer Science at the Massachusetts Institue of Technology. We want to thank the anonymous referees for many valuable comments and Philip Wadler for encouragement.

References

Aiken, Alexander, & Murphy, Brian R. (1991). Static Type Inference in a Dynamically Typed Language. *Pages 279–290 of: Proc. of the 18'th annual ACM symp. on principles of programming languages (POPL 18)*. ACM.

Aiken, Alexander, & Wimmers, Edward. (1993). Type inclusion constraints and type inference. *Pages 31–41 of: Proc. conference on functional programming languages and computer architecture*.

Banâtre, Jean-Pierre, Bryce, Ciarán, & Metayer, Daniel Le. (1994). Compile-time detection of information flow in sequential programs. *Pages 55–73 of:* Gollmann, Dieter (ed), *Computer security – ESORICS 94, 3rd european symp. on research in comp. security*. Lecture Notes in Computer Science, vol. 875. Brighton, UK: Springer-Verlag.

Barendregt, Henk P. (1981). *The lambda calculus: Its syntax and semantics*. North-Holland.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., & Nelson, G. 1989 (Nov.). *Modula-3 report (revised)*. Tech. rept. TR-52. DEC-SRC.

Cardelli, Luca. (1984). A semantics of multiple inheritance. *Pages 51–68 of:* Kahn, Gilles, MacQueen, David, & Plotkin, Gordon (eds), *Semantics of data types*. Springer-Verlag (*LNCS* 173).

Denning, Dorothy E. (1976). A Lattice Model of Secure Information Flow. *Communications of the ACM*, **19**(5), 236–242.

Denning, Dorothy E. (1982). *Cryptography and data security*. Addison-Wesley.

Denning, Dorothy E., & Denning, Peter J. (1977). Certifications of Programs for Secure Information Flow. *Communications of the ACM*, **20**(7), 504–512.

Eifrig, Jonathan, Smith, Scott, & Trifonov, Valery. (1995). Type inference for recursively constrained types and it application to OOP. *Proc. mathematical foundations of programming semantics*. To appear.

Fuh, You-Chin, & Mishra, Prateek. (1990). Type Inference With Subtypes. *Theoretical computer science*, **73**(1), 155–175.

Henglein, Fritz. (1992). Dynamic typing. *Pages 233–253 of: Proc. esop'92, european symposium on programming*. Springer-Verlag (*LNCS* 582).

Mitchell, John. (1984). Coercion and type inference. *Pages 175–185 of: Eleventh symposium on principles of programming languages*.

Ørbæk, Peter. (1995). Can you Trust your Data? *Pages 575–590 of:* Mosses, P. D. (ed), *Proceedings of the TAPSOFT/FASE'95 conference*. Lecture Notes in Computer Science, vol. 915. Aarhus, Denmark: Springer-Verlag. URL: ftp://ftp.daimi.aau.dk/pub/empl/-poe/index.html.

Palsberg, Jens. (1995). Efficient inference of object types. *Information and computation*, **123**(2), 198–209. Preliminary version in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

Palsberg, Jens, & Ørbæk, Peter. (1995). Trust in the λ-calculus. *Pages 314–330 of:* Mycroft, Alan (ed), *SAS'95: Static Analysis*. Lecture Notes in Computer Science, vol. 983. Glasgow: Springer-Verlag.

Palsberg, Jens, & Schwartzbach, Michael I. (1995). Safety analysis versus type inference. *Information and computation*, **118(1)**, 128–141.

Wall, Larry, & Schwartz, Randal L. (1991). *Programming Perl*. O'Reilly and Associates.

Wright, Andrew K., & Cartwright, Robert. (1994). A Practical Soft Type System for Scheme. *Proceedings of the 1994 ACM conference on lisp and functional programming (lfp'94)*. ACM. URL: `ftp://cs.rice.edu/public/wright/HomePage.html`.