Trust in the Pi-Calculus

Mark Hepburn
mark_h@postoffice.utas.edu.au

David Wright
David.Wright@utas.edu.au

School of Computing, University of Tasmania
GPO Box 252-100
Hobart, Tasmania
Australia, 7001
phone: +61 3 6226 2922
fax: +61 3 6226 1824

ABSTRACT

We introduce a new system of trust analysis for concurrent and distributed systems using the π-calculus[13, 14, 15] as a modelling tool. A Type system using boolean annotations guarantees that no run-time errors due to untrusted data being used in a trusted context are possible. We improve on other similar systems[18] by introducing a safe environment in which trust-coercion can be performed based on the results of run-time checks. An algorithm for deducing the most general types for the type system is presented.

Keywords

π-calculus, trust analysis, runtime coercion, type annotations

1. INTRODUCTION

Distributed computing is looming as one of the most important and influential computing paradigms of the near future. Among the benefits it offers are cheaper supercomputers through clustering (e.g. [1]) off-the-shelf personal computers, and perhaps more interestingly the ability to harness the spare CPU cycles and memory of other machines on the local intranet or even on the global internet itself (for example, [2, 3, 4]).

With this increase in the use of the internet, particularly the use of the internet for sharing computational resources, comes a corresponding increase in the number of threats to security using the very same pathways into your machine that your collaborators use to pursue more noble goals. To counteract these threats, there is a growing discipline of security analyses focusing on mobile agents and network oriented computation (e.g. [5, 23, 21, 26, 16, 11, 19]). Many of these[23, 8, 19] utilise a concept of *flow analysis*; the idea of somehow tracking the flow of data (or other properties or combination thereof) through a program in order to ascertain if any security violations are possible. Most such analyses are concerned with the possibility of security levels being compromised; for example data "leaking" from a high security level to a lower level.

We take another approach; we concern ourselves in this work with the possibility of your data being compromised by intruders rather than our data being viewed by those same intruders. Note that in some networks data integrity simply cannot be guaranteed; for instance a government keen to censor publicly available information may monitor its country's internet gateways, a malicious system administrator might alter peoples' sensitive data, or even a noisy channel could possibly render the integrity of information received along it suspect at best.

In *Trust in the lambda-calculus*[18], Ørbæk and Palsberg introduced a system of Trust Analysis involving type annotations and coercion operators that was able to demonstrate compile-time safety with respect to trust. Any possibility of data marked as untrusted being used in a computation that relied on data being trusted for its result would result in a type checking error. We feel this is very important work; however we have some reservations about the methodology through which the results are achieved. The cornerstones of their system are type annotations (tr, trusted; and dis, distrusted) and three additional operators to the language: *trust*, *distrust*, and *check*. The first two are explicit coercion operators; "trust x" means x is now trusted, no matter what its previous annotation, and similarly "distrust x" has the opposite effect. The third, *check*, is a safety operator: "check x" only type checks if x is trusted, and fails otherwise. The intention is the programmer places instances of "check" at judicious points in his program, then relevant data which is untrusted is coerced to trusted only after careful checking (of whatever nature) reveals that it can in fact be trusted. An advantage of this work is that it is completely decidable at compile time; if an annotated program successfully type checks then the bare program (with no *trust*s, *distrust*s, or *check*s) is also safe with respect to trust.

Ørbæk and Palsberg offer the following example of their system in action:

```
fun getRequest client =
  let (req, signature) = readFromNetwork(client) in
    if verifySignature(signature) then
      handleEvent(trust req)
```

```
    else
      handleWrongSignature(req, signature)
```

where the handler code resembles:

```
fun handleEvent req =
  let trustedReq = check req
  in ...
```

It is our belief however that this approach places too much responsibility on the programmer to correctly apply the appropriate casts and checks. Consider what happens if the "verifySignature" branches get mixed up; ie:

```
...
if verifySignature(signature) then
  handleWrongSignature(req, signature)
else
  handleEvent(trust req)
```

In this case the request has been established to be untrustworthy, however due to the explicit trust cast the program will still type-check and possibly use that untrusted data in a trusted context.

In our work presented here, we seek to follow a similar approach, with the following improvements:

- extend the approach into a distributed model (Ørbæk and Palsberg presented systems of a functional[18] and imperative[17] nature);
- provide a safe system of coercion by removing the onus of correct use of coercion operators from the programmer.

1.1 Motivation

Consider the simple case of a process wishing to send a message to a second process. Assume that the two processes are separated to some degree so that they must use an external communication medium, such as a network connection. Now suppose that that channel of communication has possibly been compromised, and an attacker may be altering all or some data passing through it. Further assume that our processes are called P and Q, that the channel they share for communication is called x, and that P wishes to transmit a message y which Q will bind to a variable z. In the π-calculus (see section 2, and also [13]) this scenario, represented pictorially in figure 1, is described by

$$\overline{x}.[y]\mathrm{P}|x.(\lambda z)Q$$

It is most likely that Q will not wish to trust the data (y) received along x without first performing some sort of integrity check, as it may well have been compromised in transit. In this case the danger is immediately apparent; however it is not difficult to imagine much more complicated examples in which the flow of data is a lot harder to follow without some kind of formal analysis. Our type system and syntactic additions to the π-calculus address this issue.

1.2 Layout

The remainder of this paper is presented as follows: in section 2 we briefly outline the π-calculus, the network calculus we will be using as our model. A typing system is also covered. In section 3 we present our system: first a set of boolean annotations (based on those presented in [24, 25]) to the base type system is described, which can be used to demonstrate type safety with the additional property of no instances of untrusted data being used in a trusted context being possible. Following this our safe model of coercion is introduced; a simple extension to the syntax of the π-calculus that removes the responsibility of placing checks in appropriate places from the programmer. The safety of the new system is demonstrated in section 4 as subject reduction is shown to hold. In section 5 an algorithm for determining the most general type for a program in our system is presented. The soundness and completeness of this algorithm is proven in section 6. Finally, in section 7 we conclude.

2. THE PI-CALCULUS

2.1 Syntax

The fundamental tenet of the π-Calculus is communication between processes, which makes it an ideal basis for our study. The main building blocks are channels, and all processes are constructed from channels. Channels carry data between processes, and in its purest form all data is also comprised of channels. There are two main forms of processes; those that can receive data and that can transmit data. As an example of a transmitting process, $\overline{x}.[y]P$ transmits, to a process capable of receiving along channel x, the data y then continues as P. The reciprocal case of a process receiving data is represented as $x.(\lambda z)Q$; the process receives a message from channel x and binds it to the variable z then continues as Q (with every instance of z being replaced with x). Note that a channel being used in an output context is presented with a vertical bar over it.

In this paper we consider a commonly used version of the π-Calculus as described in [13], with polyadic channels. We let P, Q, R range over processes, M, N range over normal processes (all processes can be expressed in this form ([13])), F represent abstractions and C concretions, A range over agents, x, y, z range over names and finally ω represent either x or $\overline{x}$ as appropriate for some name x. The complete syntax is shown in definition 1:

Definition 1.

$$\begin{array}{lcl} \mathrm{N} & ::= & \omega.\mathrm{A} \mid \mathbf{0} \mid \mathrm{M} + \mathrm{N} \\ \mathrm{P} & ::= & \mathrm{N} \mid \mathrm{P}|\mathrm{Q} \mid !\mathrm{P} \mid (\nu x)\mathrm{P} \\ \mathrm{F} & ::= & \mathrm{P} \mid (\lambda x)\mathrm{F} \mid (\nu x)\mathrm{F} \\ \mathrm{C} & ::= & \mathrm{P} \mid [x]\mathrm{C} \mid (\nu x)\mathrm{C} \\ \mathrm{A} & ::= & \mathrm{F}|\mathrm{C} \end{array}$$

An abstraction F is a process prepared to receive a name along an unspecified channel, and bind it to the variable *abstracted* (represented by λx where x is the bound variable) on F. A concretion is the equivalent output case; a process prepared to output a name along an unspecified channel. An agent is either an abstraction or a concretion. Restriction ($(\nu x)A$), which creates a unique name x in P is also a binding operator. In the π-calculus we consider, both input and output are blocking operations. The restriction operator $(\nu x)P$ creates a unique name x in P, while the replication operator $!P$ can be defined inductively as $!P = P|!P$.

The existence of binding operators enables us to (inductively) define the concept of free and bound names or variables (defn 2):

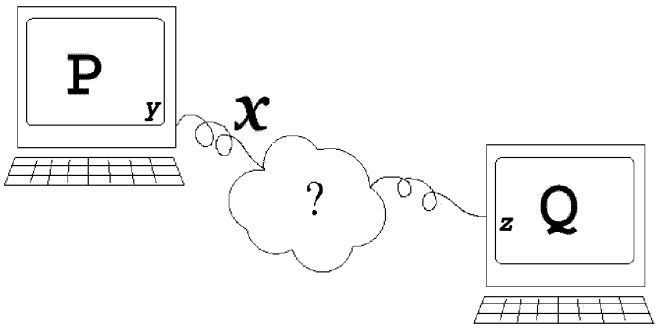

Figure 1: Basic Scenario

Definition 2.

$$\begin{aligned} FV(x.\mathrm{A}) = FV(\overline{x}.\mathrm{A}) &= \{x\} \cup FV(A) \\ FV([x]\mathrm{C}) &= \{x\} \cup FV(\mathrm{C}) \\ FV((\lambda x)\mathrm{F}) = FV((\nu x)\mathrm{F}) &= FV(\mathrm{F}) - \{x\} \end{aligned}$$

2.1.1 Reduction

There is only one reduction axiom in the basic π-Calculus; a communication step:

$$\frac{}{(\ldots + \overline{x}.[\vec{y}]\mathrm{P})|(\ldots + x.(\lambda\vec{z})\mathrm{Q}) \longrightarrow \mathrm{P}|\mathrm{Q}\{y/z\}}\ (comm.)$$

In the communication reduction shown above, process P sends a name y along channel x, then continues as P. Process Q receives the name y along x and binds it to the name z, then continues as Q. There are also three inference rules, summarised briefly as: reduction can occur in parallel (Par.); reduction can occur under a restriction (but *not* under abstraction or input/output) (Res.), and structural congruence is preserved under reduction (Struct.):

$$\frac{\mathrm{P} \longrightarrow \mathrm{P}' \quad \mathrm{Q} \longrightarrow \mathrm{Q}'}{\mathrm{P}|\mathrm{Q} \longrightarrow \mathrm{P}'|\mathrm{Q}'}\ (Par.) \qquad \frac{\mathrm{P} \longrightarrow \mathrm{P}'}{(\nu x)\mathrm{P} \longrightarrow (\nu x)\mathrm{P}'}\ (Res.)$$

$$\frac{\mathrm{Q} \equiv \mathrm{P} \quad \mathrm{P} \longrightarrow \mathrm{P}' \quad \mathrm{P}' \equiv \mathrm{Q}'}{\mathrm{Q} \longrightarrow \mathrm{Q}'}\ (Struct.)$$

2.2 Typing

The base typing system we use is also that of [13]. The usual goal of type systems is to prevent certain run-time errors from occurring; since the π-calculus is concerned with communication a likely cause of run-time errors is a process receiving a tuple of data from a channel that is of a different size to that which it is expecting. To this end our typing discipline will require each channel to be used in only a certain way; both in the lengths of data tuples it carries, and by extension since all data is itself comprised of channels the types of channels it can carry.

We begin by associating all names with a *sort*. Since all names are channels, then to each sort we associate a list of sorts that channels of that sort communicate: the *object* of that sort, denoted $\mathrm{ob}(\delta)$ for sort δ.

Definition 3. Judgements

$$\Gamma, \mathrm{ob} \vdash \mathrm{P} : \kappa$$

states that process P is well formed and has type κ, under the assumption sets Γ and ob (where Γ is a mapping of names to sorts, and ob is a mapping of sorts to objects of those sorts).

See definition 7 for a complete type syntax (with annotations; introduced in section 3.3); the reader is referred to [13] for a more detailed description of the base type system itself.

3. A SAFE SYSTEM OF COERCION

3.1 A safe coercion operator

We now start to consider the desirable properties of our system of coercion. We would like the programmer to be able to consider, and within a restricted environment to be able to use coercion, but as have stated previously we would like to do away with the need for explicit, programmer driven casts. As an alternative we envisage a system in which coercion is based not on programmer judgements (and hence potentially errors) but on run-time verification. We deliberately leave the method of verification unstated in keeping with the broad abstract framework provided by the π-calculus, however some examples could include signature verification, or in fact any form of security analysis/verification described by others (eg [16]) that would provide the necessary assurance. In this way we see our method as being an overall analysis framework that can utilise and encompass the work of others; we do not see security as being provided by any one method ([6]), but rather a constantly shifting approach.

Our addition to the existing π-calculus syntax (defn. 1) will coerce (to either trusted or untrusted) its argument based on the results of runtime certification (using an unspecified method). We would further like to alter the execution path of the program based on the certification results; to this end we provide two additional arguments (both π-calculus processes) to our operator: one to be executed if the data can be correctly verified, and the other if it cannot.

Our new operator then looks like this:

$$\text{certify } x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q}$$

where x is the data to be certified (note that in our current, first order system we do not allow processes themselves to be verified; this will be rectified in later work dealing with a higher-order system), P is the process to be executed if x can be guaranteed trusted, and Q to be executed in the event that x is untrusted. It will be seen later (Figure 3) that z is the variable x is bound to in the reduction.

We deal with typing and reduction details later; for the moment our complete syntax becomes (defn 4):

Definition 4.

$$\begin{array}{lcl} \mathrm{N} & ::= & \omega.\mathrm{A} \mid \mathbf{0} \mid \mathrm{M} + \mathrm{N} \mid \text{certify } x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q} \\ \mathrm{P} & ::= & \mathrm{N} \mid \mathrm{P}|\mathrm{Q} \mid !\mathrm{P} \mid (\nu x)\mathrm{P} \\ \mathrm{F} & ::= & \mathrm{P} \mid (\lambda x)\mathrm{F} \mid (\nu x)\mathrm{F} \\ \mathrm{C} & ::= & \mathrm{P} \mid [x]\mathrm{C} \mid (\nu x)\mathrm{C} \\ \mathrm{A} & ::= & \mathrm{F}|\mathrm{C} \end{array}$$

3.2 Example

As an example of our system in use, we present a contrast with the approach put forward in [18]. Consider again the example described in the introduction; in our system, the same scenario would be written as:

$\lambda x.$`certify` x (λ`z.handleEvent`) (λ`z.handleWrongSignature`)

(Note that in this example, the abstraction on x is taken to represent the `readFromNetwork` function, and the `verifySignature` functionality is incorporated in the certify operator). It can be observed that in our system the coercion is now implicit, and further it will be shown that the danger of the execution paths being placed in the wrong order is detected by the type system.

3.3 An Annotated Type System

To the existing type system, we add a system of boolean annotations, based on [25]. We consider two concrete values, T denoting a trusted channel sort (corresponding to truth in a boolean algebra) and U, denoting an untrusted channel sort (likewise corresponding to falsehood). In summary the following relations are admitted (defn 5):

Definition 5.

$$\begin{array}{rclrcl} \mathrm{T} \cdot b & = & b & \mathrm{U} \cdot b & = & \mathrm{U} \\ \mathrm{T} + b & = & \mathrm{T} & \mathrm{U} + b & = & b \\ \hat{\mathrm{T}} & = & \mathrm{U} & \hat{\mathrm{U}} & = & \mathrm{T} \end{array}$$

It should be noted that although in the interests of completeness we permit the operations of + and negation, they are not used in the system we present here. Preliminary work is underway however on an extended system for a higher order π-calculus which requires all the operations presented above.

Now we present the complete type syntax. We first define the symbols used in our type system:

Definition 6. For all elements of our type system we have three sets of symbols we use to range over those elements: concrete elements, variables, and ambiguous elements (table 1).

	Concrete	Variable	Either
Annotations	T, U	i, j, k	b,c,d
Sorts	S	α	δ
Object Sorts	r	ρ	κ

Table 1: Notation

Our complete type syntax then becomes:

Definition 7.

$$\begin{array}{rcl} b & ::= & \mathrm{T}|\mathrm{U}|i|b_1 \cdot b_2|b_1 + b_2|\hat{b} \\ \delta & ::= & \mathrm{S}|\alpha \\ \kappa & ::= & ()|\ \ \delta^b \left\{\delta^b\right\}^*\ \ |\rho \\ \Gamma & ::= & \left\{x : \delta^b\right\} \\ \mathrm{ob} & ::= & \left\{\delta^b \mapsto \kappa\right\} \end{array}$$

We also introduce here a few miscellaneous definitions used in the type rules:

Definition 8. Write $(\Gamma, \mathrm{ob}) \cup \{x : \delta^b \mapsto \kappa\}$ to mean

$$\Gamma \cup \{x : \delta^b\}, \mathrm{ob} \cup \{\delta^b \mapsto \kappa\}$$

Similarly, write $(\Gamma, \mathrm{ob})_x \cup \{x : \delta^b \mapsto \kappa\}$ as shorthand for

$$\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \cup \{\delta^b \mapsto \kappa\}$$

Definition 9. Define the '$^\wedge$' operation on object sorts as

$$(\mathrm{S})^\wedge(\mathrm{S}_1 \ldots \mathrm{S}_2) = (\mathrm{S}\mathrm{S}_1 \ldots \mathrm{S}_2)$$

We consider in this system two possible causes of untrustworthiness: one is data which for some reason or another is known to be untrusted, and the other is data that cannot be trusted because it has passed through an untrustworthy channel. These assumptions, in particular the latter, imply a relation between sorts and their objects: given a sort δ^{b_1} that we plan to receive data along then we would desire that for all $\delta_k^{b_2}$ in $\mathrm{ob}(\delta^{b_1})$, $b_2 \leq b_1$ where $\leq$ is the least reflexive, transitive relation inductively defined by the following (definition 10):

Definition 10.

$$\mathrm{U} \leq i \leq \mathrm{T}$$

Rather than have a separate set of constraints generated, we choose to express this relationship between sorts and their objects slightly differently in our type rules by integrating them into the types themselves. The same effect can be achieved by requiring that for any δ^b then for all δ_k^c in $\mathrm{ob}(\delta^b)$ that $c = b \cdot c'$ for some c'. Before introducing the complete type rules, first define multiplication as the operation inductively defined (definition 11) as

Definition 11.

$$\begin{array}{rcl} b \cdot () & = & () \\ b \cdot \rho & = & \rho \\ b \cdot\ (\delta^c)^\wedge\kappa & = & \delta^{bc\ \wedge}(b \cdot \kappa) \end{array}$$

We can now introduce the type rules: (see Figure 2)
Of these rules, three in particular are worth attention: *(inp.)* and *(out.)*, and *(cert.)*. The (*inp.*) rule requires that the types of the name(s) received along the channel x first of all form the same type as the object of the type of the channel x (which is required by the base type system), but also that that object sort κ' is equal to $b \cdot \kappa$ for some κ, thus enforcing the view that all data received along an untrusted channel is untrusted (by the definition of multiplication, defn. 11).

By contrast, we would expect that trusted data can by *sent* along an untrusted channel, but that it would be untrusted at the receiving end. To this end our (*out.*) rule is

$$\frac{}{\Gamma, \mathrm{ob} \vdash 0 : ()} \ (zero)$$

$$\frac{}{\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \vdash x : \delta^b} \ (var.)$$

$$\frac{\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \vdash \mathrm{A}{:}\kappa}{\Gamma, \mathrm{ob} \vdash (\nu x)\,\mathrm{A}{:}\kappa} \ (res.)$$

$$\frac{\Gamma, \mathrm{ob} \vdash \mathrm{P}{:}()}{\Gamma, \mathrm{ob} \vdash !\mathrm{P}{:}()} \ (repl.)$$

$$\frac{\Gamma, \mathrm{ob} \vdash \mathrm{M}{:}() \quad \Gamma, \mathrm{ob} \vdash \mathrm{N}{:}()}{\Gamma, \mathrm{ob} \vdash \mathrm{M} + \mathrm{N}{:}()} \ (sum.)$$

$$\frac{\Gamma, \mathrm{ob} \vdash \mathrm{P}{:}() \quad \Gamma, \mathrm{ob} \vdash \mathrm{Q}{:}()}{\Gamma, \mathrm{ob} \vdash \mathrm{P}|\mathrm{Q}{:}()} \ (comp.)$$

$$\frac{\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \vdash \mathrm{F}{:}\kappa}{\Gamma, \mathrm{ob} \vdash (\lambda x)\,\mathrm{F}{:}\,(\delta^b)^\wedge\,\kappa} \ (abs.)$$

$$\frac{\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \vdash \mathrm{C}{:}\kappa}{\Gamma_x \cup \{x : \delta^b\}, \mathrm{ob} \vdash [x]\,\mathrm{C}{:}\,(\delta^b)^\wedge\,\kappa} \ (conc.)$$

$$\frac{(\Gamma, \mathrm{ob})_x \cup \{x : \delta^b \mapsto b \cdot \kappa\} \vdash \mathrm{F}{:}b \cdot \kappa}{(\Gamma, \mathrm{ob})_x \cup \{x : \delta^b \mapsto b \cdot \kappa\} \vdash x.\mathrm{F}{:}()} \ (inp.)$$

$$\frac{(\Gamma, \mathrm{ob})_x \cup \{x : \delta^b \mapsto \kappa\} \vdash \mathrm{C}{:}\kappa}{(\Gamma, \mathrm{ob})_x \cup \{x : \delta^b \mapsto b \cdot \kappa\} \vdash \overline{x}.\mathrm{C}{:}()} \ (out.)$$

$$\frac{\begin{array}{l}\Gamma, \mathrm{ob} \vdash x : \delta^b \\ \Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{P} : (\delta^{\mathrm{T}}) \\ \Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{Q} : (\delta^{\mathrm{U}}) \\ x \notin FV(\mathrm{P}, \mathrm{Q})\end{array}}{\Gamma, \mathrm{ob} \vdash \mathrm{certify}\ x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q} : ()} \ (cert.)$$

Figure 2: Type Rules

slightly different: the base requirement that the sorts carried by x match those to be transmitted by C is still met, but now in the antecedent we are unconcerned about the annotation on the channel, thus a variable may be trusted in C and still be exported along an untrusted channel. The consequent of the rule matches that of the (*inp*) case; all sorts must be multiplied by the annotation of the channel. This allows trusted data to be sent along an untrusted channel and be considered untrusted by the receiving process.

The *(cert.)* rule forms the cornerstone of our system; it provides a safe environment in which coercion can be performed, removing the responsibility from the programmer. The first argument to $certify$ is the name to be certified; the second argument is a process P upon which a single name has been abstracted. This name must have the same base type as the name being certified (x), and be trusted. Similarly, the third argument also has a single name abstracted upon it with the same base type as x, and is untrusted. As we shall see when we examine the reduction rules for $certify$, in the event that x is certified (usually at runtime) as trusted then the type of x (δ^b) is coerced to trusted and every instance of z in P replaced with x. Conversely, if x cannot be certified then it is coerced to untrusted and every instance of z in Q replaced by x. Since the abstracted name z is trusted and untrusted in P and Q respectively, this ensures the safety of the system by subject reduction (see Section 4). Note that although subject reduction is present in [18] it is modulo the explicit trust casts: *any* program can be made "correct" by inserting enough casts. Finally, we also require that x not be in the set of free names of P or Q: this makes the system more elegant than if x were allowed free in P or Q as it removes the need to introduce explicit trust casting into the language.

3.4 Reduction Rules

Definition 12. Write $\Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}$ if x is certified as trusted; similarly if x is certified as untrusted.

The reduction rules remain the same as for the basic π-calculus, with the addition of extra rules to cater for $certify$; see Figure 3. These rules guarantee that if x is trusted then it will only be used within a trusted context by substituting it for a trusted name in P; conversely if it is untrusted then it will only be used within an untrusted context by substituting it for an untrusted name in Q. Note that all coercion is global in effect.

4. SAFETY OF THE SYSTEM

Now that we have a type system and syntax established, we can begin formulating a proof of subject reduction, which as we shall see is by necessity slightly different from the normal subject reduction statement.

First, some preliminary definitions (defn. 13):

Definition 13. Substitutions

- A substitution is a pair $(\mathbb{S}_T : Type \to Type, \mathbb{R}_B : boolexp \to boolexp)$.
- Usually written just as $\mathbb{S}$; sometimes write $\mathbb{R}$ to denote $(\mathrm{Id}, \mathbb{R}_B)$.
- Write Id for $(\mathrm{Id}, \mathrm{Id})$.
- Write $\mathbb{S}_1; \mathbb{S}_2$ for $\mathbb{S}_2 \circ \mathbb{S}_1$
- Application: if $\mathbb{S} = (\mathbb{S}_T, \mathbb{R}_B)$ then
  - $\mathbb{S}(\mathrm{S}^i) = \mathbb{S}_T; \mathbb{R}_B(\mathrm{S}^i)$
  - $\mathbb{S}((\mathrm{S}^i)^\wedge r) = (\mathbb{S}(\mathrm{S}^i))^\wedge \mathbb{S}(r)$
  - $\mathbb{S}(\{\mathrm{S}^i \mapsto r\} \cup \mathrm{ob}) = \{\mathbb{S}(\mathrm{S}^i) \mapsto \mathbb{S}(r)\} \cup \mathbb{S}(\mathrm{ob})$
- For $\mathbb{S} = (\mathbb{S}_T, \mathbb{R}_B)$, write $\mathbb{S}[\alpha^i := \mathrm{S}^j]$ for $(\mathbb{S}_T[\alpha^i := \mathrm{S}^j], \mathbb{R}_B)$ and $\mathbb{S}[i := j]$ for $(\mathbb{S}_T, \mathbb{R}_B[i := j])$. Similarly, write $\mathbb{S}; \mathbb{R}$ for $(\mathbb{S}_T, \mathbb{R}_B; \mathbb{R})$.

4.1 Substitution Lemma

First we must establish that the substitution lemma holds:

LEMMA 4.1. *Substitution Lemma: Let ξ be either δ or κ, then:*

$$\frac{\Gamma_x \cup \quad x : \delta^b \quad , \mathrm{ob} \vdash \mathrm{A} : \xi \quad \Gamma, \mathrm{ob} \vdash y : \delta^b}{\Gamma, \mathrm{ob} \vdash \mathrm{A}\,\{y/x\} : \xi}$$

Proof:

By induction on the structure of A and the type of x & y.

4.2 Subject Reduction

Having established that substitution is sound, we can now consider the case of subject reduction. The fact that we must also consider run-time coercion complicates matters however: the basic subject reduction property states that the type of a given expression remains immutable as the expression is reduced. This is obviously not the case in the presence of run-time coercion; if the reduction involves any coercion then the type and its enclosing environment may be altered. Given this, we must find some way of including this in the statement without relaxing the subject reduction property itself.

To this end we present a boolean substitution inductively defined by the reduction path: (Figure 4)

Now we can formulate our new subject reduction statement; similar to the classical case, but the environment after the reduction step must have the substitution defined by the reduction and the relation given in figure 4 applied to it.

THEOREM 4.1. *Subject Reduction:*

$$\frac{\Gamma, \mathrm{ob} \vdash \mathrm{P} : () \quad \mathrm{P} \to \mathrm{P}' \quad \mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}}{\mathbb{R}(\Gamma), \mathbb{R}(\mathrm{ob}) \vdash \mathrm{P}' : ()}$$

Proof:

By induction on the structure of the derivation of $\mathrm{P} \to \mathrm{P}'$, and by cases on the structure of P. In the interests of space, we present only two illustrative cases:

- Suppose the last rule used in the derivation of $\mathrm{P} \to \mathrm{P}'$ was

  $$\frac{\mathrm{Q} \to \mathrm{Q}' \quad \mathrm{R} \to \mathrm{R}'}{\mathrm{P} = \mathrm{Q}|\mathrm{R} \to \mathrm{Q}'|\mathrm{R}' = \mathrm{P}'}$$

  By the statement $\Gamma, \mathrm{ob} \vdash \mathrm{Q}|\mathrm{R} : ()$, so from the (comp.) type rule we must have $\Gamma, \mathrm{ob} \vdash \mathrm{Q} : ()$ and $\Gamma, \mathrm{ob} \vdash \mathrm{R} : ()$. By the induction hypothesis, we have $\mathbb{R}_1(\Gamma), \mathbb{R}_1(\mathrm{ob}) \vdash \mathrm{Q}' : ()$ and $\mathbb{R}_2(\Gamma), \mathbb{R}_2(\mathrm{ob}) \vdash \mathrm{R}' : ()$ so $\mathbb{R}(\Gamma), \mathbb{R}(\mathrm{ob}) \vdash \mathrm{Q}'|\mathrm{R}' : ()$ as required, where $\mathbb{R} = \mathbb{R}_1; \mathbb{R}_2$ by the definition of $\Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}}$.

$$\frac{\Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}}{\mathrm{certify}\ x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q} \to \mathrm{P}\{x/z\}}$$

$$\frac{\Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{U}}}{\mathrm{certify}\ x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q} \to \mathrm{Q}\{x/z\}}$$

Figure 3: certify Reduction Rules

$$\frac{}{\mathrm{P} \to \mathrm{P} \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathrm{Id}}\ (reflex)$$

$$\frac{}{\overline{x}.[\vec{y}]\mathrm{C} \xrightarrow{\overline{x}.[\vec{y}]} \mathrm{C} \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathrm{Id}}\ (out.)$$

$$\frac{}{x.(\lambda\vec{y})\mathrm{F} \xrightarrow{x.(\lambda\vec{z})} \mathrm{F} \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathrm{Id}}\ (inp.)$$

$$\frac{\mathrm{M} \to \mathrm{M}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}}{\mathrm{M} + \mathrm{N} \to \mathrm{M}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}}\ (sum.)$$

$$\frac{\mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_1 \quad \mathrm{Q} \to \mathrm{Q}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_2}{\mathrm{P}|\mathrm{Q} \to \mathrm{P}'|\mathrm{Q}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_1; \mathbb{R}_1}\ (comp.)$$

$$\frac{\mathrm{P} \to \mathrm{P}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_1 \quad \mathrm{P}' \to \mathrm{P}'' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_2}{\mathrm{P} \to \mathrm{P}'' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}_1; \mathbb{R}_2}\ (trans.)$$

$$\frac{\Gamma, \mathrm{ob} \vdash x : \delta^i \quad \Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}}{\mathrm{certify}\ x\ \mathrm{P}\ \mathrm{Q} \to \mathrm{P} \cdot x \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathrm{Id}[i := \mathrm{T}]}\ (certify - \mathrm{T})$$

$$\frac{\Gamma, \mathrm{ob} \vdash x : \delta^i \quad \Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{U}}}{\mathrm{certify}\ x\ \mathrm{P}\ \mathrm{Q} \to \mathrm{Q} \cdot x \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathrm{Id}[i := \mathrm{U}]}\ (certify - \mathrm{U})$$

Figure 4: Reduction Substitution Relation

- Suppose the last rule used in the derivation of $\mathrm{P}\rightarrow\mathrm{P}'$ was

$$\frac{\Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}}{\mathrm{P} = \mathrm{certify}\ x\ (\lambda z)\mathrm{R}\ (\lambda z)\mathrm{Q} \rightarrow \mathrm{R}\{x/z\} = \mathrm{P}'}$$

From the statement, $\Gamma, \mathrm{ob} \vdash \mathrm{certify}\ x\ (\lambda z)\mathrm{R}\ (\lambda z)\mathrm{Q}{:}()$, where $\Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{R} : (\delta^{\mathrm{T}})$, $\Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{Q} : (\delta^{\mathrm{U}})$, and $\Gamma, \mathrm{ob} \vdash x : \delta^{b}$ by the (cert.) type rule. Given $\Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}$, then
$(\mathrm{certify}\ x\ (\lambda z)\mathrm{R}\ (\lambda z)\mathrm{Q}) \rightarrow \mathrm{R}\{x/z\}$ by the semantics of certify, and by subject reduction

$$\frac{\Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{R} : (\delta^{\mathrm{T}}) \quad \Gamma, \mathrm{ob} \vdash_{\mathrm{certify}} x : \delta^{\mathrm{T}}}{\mathbb{R}(\Gamma), \mathbb{R}(\mathrm{ob}) \vdash \mathrm{R}\{x/z\} : ()}$$

where the original annotation b on x has been coerced to T by certify; so $\mathbb{R} = \mathrm{Id}[b := \mathrm{T}]$ and $\mathrm{P} \rightarrow \mathrm{P}' \Vdash^{\Gamma,\mathrm{ob}}_{\mathrm{certify}} \mathbb{R}$ as required.

5. IMPLEMENTATION

In figure 5 we present an algorithm for determining the most general type of a given expression. Most of the algorithm is recursive and fairly trivial, with the base case of the rule for a name which searches in its existing environment for an entry for that name; if it finds one it returns the type entry, otherwise it creates a new type variable and returns that, updating the entry in the environments.

Those that deserve closer inspection include the input and output cases which must use $\mathbb{M}$ (definition 17) to perform the multiplication required by the type rules, and the rule for certify which ensures that the certify variable (x) does not appear free in either continuation (see section 3.3). The input rule ensures the object of the sort of x is multiplied by the annotation on the sort before deducing a type for F; in case the object of x is a variable (in which case multiplication has no effect; defn 17) it again multiplies the object after it has been unified with the type of F. The output rule performs the multiplication after unification, as is consistent with the type rules. Note that the rule for certify uses unification to ensure the abstracted variable in the continuations are trusted and untrusted respectively, as required by the type rules.

5.1 Auxiliary Algorithms

The following macro (definition 14) is used for conciseness in the algorithm:

Definition 14.

$$\mathrm{SORT}\,(\Gamma, x) \triangleq \{x : \Gamma(x)\}$$

Unification algorithms for both single sorts and lists of sorts (object sorts) are also required by the algorithm, and these are presented here. Note that the algorithms for unifying types are mutually recursive; U_π (which unifies single pairs of sorts) attempts to equate the two sorts, then calls $\mathrm{U}_{\pi l}$ to unify their object sorts. $\mathrm{U}_{\pi l}$ simply walks down the two lists, calling U_π to unify matching pairs of sorts. A third algorithm, $\mathbb{M}$, is used to perform the multiplication. BUNIFY is the boolean unification algorithm returning a substitution which is the most general unifier of its two arguments ([12]).

Definition 15.

$$\begin{aligned}
\mathrm{U}_\pi\ \ \mathrm{ob}, \alpha^b, \mathrm{S}^c &= \text{let } \mathbb{S} = \mathrm{Id}[\alpha^b := \mathrm{S}^c]; \mathrm{BUNIFY}\,(b, c) \text{ in} \\
&\quad \mathbb{S}; \mathrm{U}_{\pi l}\ \ \mathbb{S}(\mathrm{ob}), \mathbb{S}(\mathrm{ob}(\alpha^b)), \mathbb{S}(\mathrm{ob}(\mathrm{S}^c)) \\
\mathrm{U}_\pi\ \ \mathrm{ob}, \mathrm{S}^c, \alpha^b &= \mathrm{U}_\pi\ \ \mathrm{ob}, \alpha^b, \mathrm{S}^c \\
\mathrm{U}_\pi\ \ \mathrm{ob}, \mathrm{S}_1^b, \mathrm{S}_2^c &= \text{if } \mathrm{S}_1 \neq \mathrm{S}_2 \text{ then } \bot \\
&\quad \text{else if } \mathrm{S}_1^b =_\pi \mathrm{S}_2^c \text{ then} \\
&\quad\ \ \mathrm{Id} \\
&\quad \text{else let } \mathbb{R} = \mathrm{BUNIFY}\,(b, c) \text{ in} \\
&\quad\ \ \mathbb{R}; \mathrm{U}_{\pi l}\ \ \mathbb{R}(\mathrm{ob}), \mathbb{R}(\mathrm{ob}(\mathrm{S}_1^b)), \mathbb{R}(\mathrm{ob}(\mathrm{S}_2^c)) \\
\mathrm{U}_\pi\ \ \mathrm{ob}, \alpha_1^b, \alpha_2^c &= \text{if } \alpha_1^b =_\pi \alpha_2^c \text{ then} \\
&\quad\ \ \mathrm{Id} \\
&\quad \text{else let } \mathbb{S} = \mathrm{Id}[\alpha_1^b := \alpha_2^c]; \mathrm{BUNIFY}\,(b, c) \\
&\quad\ \ \text{in} \\
&\quad\ \ \mathbb{S}; \mathrm{U}_{\pi l}\ \ \mathbb{S}(\mathrm{ob}), \mathbb{S}(\mathrm{ob}(\alpha_1^b)), \mathbb{S}(\mathrm{ob}(\alpha_2^c))
\end{aligned}$$

Definition 16.

$$\begin{aligned}
\mathrm{U}_{\pi l}\,(\mathrm{ob}, \rho, t) &= \text{if } \rho \in t \wedge \rho \neq t \text{ then } \bot \\
&\quad \text{else } \mathrm{Id}[\rho := t] \\
\mathrm{U}_{\pi l}\,(\mathrm{ob}, t, \rho) &= \mathrm{U}_{\pi l}\,(\mathrm{ob}, \rho, t) \\
\mathrm{U}_{\pi l}\ \ \mathrm{ob}, (\mathrm{S}_1^i)^\wedge r, (\mathrm{S}_2^c)^\wedge t &= \text{let } \mathbb{S} = \mathrm{U}_\pi\ \ \mathrm{ob}, \mathrm{S}_1^b, \mathrm{S}_2^c \text{ in} \\
&\quad\ \ \mathbb{S}; \mathrm{U}_{\pi l}\,(\mathbb{S}(\mathrm{ob}), \mathbb{S}(r), \mathbb{S}(t)) \\
\mathrm{U}_{\pi l}\,(\mathrm{ob}, \rho_1, \rho_2) &= \mathrm{Id}[\rho_1 := \rho_2] \\
\mathrm{U}_{\pi l}\,(\mathrm{ob}, (), ()) &= \mathrm{Id} \\
\mathrm{U}_{\pi l}\ \ \mathrm{ob}, (\mathrm{S}_1^b)^\wedge r, () &= \mathrm{U}_{\pi l}\ \ \mathrm{ob}, (), (\mathrm{S}_1^b)^\wedge r\ \ = \bot
\end{aligned}$$

Definition 17.

$$\begin{aligned}
\mathbb{M}\ \ b, (\mathrm{S}^c)^\wedge r &= \text{let } \mathbb{S} = \mathbb{M}\,(b, \mathrm{S}^c) \text{ in } \mathbb{S}; \mathbb{M}\,(\mathbb{S}(b), \mathbb{S}(r)) \\
\mathbb{M}\,(b, \mathrm{S}^c) &= \mathrm{BUNIFY}\,(c, b \cdot d) \quad d \text{ new} \\
\mathbb{M}\ \ b, \mathrm{S}^{\mathrm{T}} &= \mathrm{BUNIFY}\,(b, \mathrm{T}) \\
\mathbb{M}\ \ b, \mathrm{S}^{\mathrm{U}} &= \mathrm{Id} \\
\mathbb{M}\,(b, \beta) &= \mathrm{Id}
\end{aligned}$$

6. SOUNDNESS AND COMPLETENESS OF THE ALGORITHMS

In this section we prove that the type inference algorithm presented in section 5 is sound and complete. In order to preserve space (and the reader's attention!) we restrict the proofs - all by induction - to a few suitable cases.

6.1 Soundness of Type_π

THEOREM 6.1. *If* $\langle \Gamma, \mathrm{ob}, \kappa \rangle = \mathrm{Type}_\pi\,(\Gamma', \mathrm{ob}', \mathrm{P})$ *is defined then* $\Gamma, \mathrm{ob} \vdash \mathrm{P} : \kappa$.

Proof:

By induction on the structure of P. For example:

$$
\begin{array}{rcl}
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, 0\right) & = & \langle \Gamma, \mathrm{ob}, () \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x\right) & = & \text{If } x \in \mathrm{dom}\left(\Gamma\right) \\
 & & \text{then } \langle \Gamma, \mathrm{ob}, \Gamma(x) \rangle \\
 & & \text{else } \left\langle \Gamma \cup \{x : \alpha^i\}, \mathrm{ob} \cup \{\alpha^i \mapsto \beta\}, \alpha^i \right\rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, !\mathrm{P}\right) & = & \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \mathrm{P}\right) \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, (\nu x)\mathrm{A}\right) & = & \text{let } \langle \Gamma_1, \mathrm{ob}_1, \kappa \rangle = \mathrm{Type}_\pi\left(\Gamma_x, \mathrm{ob}, \mathrm{A}\right) \text{ in} \\
 & & \quad \langle (\Gamma_1)_x \cup \mathrm{SORT}\left(\Gamma, x\right), \mathrm{ob}_1, \kappa \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \mathrm{P}|\mathrm{Q}\right) & = & \text{Let } \langle \Gamma_1, \mathrm{ob}_1, () \rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \mathrm{P}\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, () \rangle = \mathrm{Type}_\pi\left(\Gamma_1, \mathrm{ob}_1, \mathrm{Q}\right) \text{ in} \\
 & & \quad\quad \langle \Gamma_2, \mathrm{ob}_2, () \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \mathrm{M} + \mathrm{N}\right) & = & \text{Let } \langle \Gamma_1, \mathrm{ob}_1, () \rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \mathrm{M}\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, () \rangle = \mathrm{Type}_\pi\left(\Gamma_1, \mathrm{ob}_1, \mathrm{N}\right) \text{ in} \\
 & & \quad\quad \langle \Gamma_2, \mathrm{ob}_2, () \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, (\lambda x)\mathrm{F}\right) & = & \text{let } \left\langle \Gamma_1, \mathrm{ob}_1, \delta^b \right\rangle = \mathrm{Type}_\pi\left(\Gamma_x, \mathrm{ob}, x\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, \kappa \rangle = \mathrm{Type}_\pi\left(\Gamma_1, \mathrm{ob}_1, \mathrm{F}\right) \text{ in} \\
 & & \quad\quad \left\langle (\Gamma_2)_x \cup \mathrm{SORT}\left(\Gamma, x\right), \mathrm{ob}_2, \quad \delta^b \quad \widehat{} \quad \kappa \right\rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, [x]\mathrm{C}\right) & = & \text{let } \left\langle \Gamma_1, \mathrm{ob}_1, \delta^b \right\rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, \kappa \rangle = \mathrm{Type}_\pi\left(\Gamma_1, \mathrm{ob}_1, \mathrm{C}\right) \text{ in} \\
 & & \quad\quad \left\langle \Gamma_2, \mathrm{ob}_2, \quad \delta^b \quad \widehat{} \quad \kappa \right\rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x.\mathrm{F}\right) & = & \text{let } \left\langle \Gamma_1, \mathrm{ob}_1, \delta^b \right\rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x\right) \text{ in} \\
 & & \quad \text{let } \mathbb{S}_1 = \mathbb{M} \quad b, \mathrm{ob}(\delta^b) \quad \text{in} \\
 & & \quad\quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, \kappa \rangle = \mathrm{Type}_\pi\left(\mathbb{S}_1(\Gamma_1), \mathbb{S}_1(\mathrm{ob}_1), \mathrm{F}\right) \text{ in} \\
 & & \quad\quad\quad \text{let } \mathbb{S}_2 = \mathbb{S}_1; \mathrm{U}_{\pi l} \quad \mathrm{ob}_2, \kappa, \mathrm{ob}_2(\delta^b) \quad \text{in} \\
 & & \quad\quad\quad\quad \text{let } \mathbb{S}_3 = \mathbb{S}_2; \mathbb{M}\left(\mathbb{S}_2(b), \mathbb{S}_2(\kappa)\right) \text{ in} \\
 & & \quad\quad\quad\quad\quad \langle \mathbb{S}_3(\Gamma_2), \mathbb{S}_3(\mathrm{ob}_2), () \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \overline{x}.\mathrm{C}\right) & = & \text{let } \left\langle \Gamma_1, \mathrm{ob}_1, \delta^b \right\rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, \kappa \rangle = \mathrm{Type}_\pi\left(\Gamma_1, \mathrm{ob}_1, \mathrm{C}\right) \text{ in} \\
 & & \quad\quad \text{let } \mathbb{S}_1 = \mathrm{U}_{\pi l} \quad \mathrm{ob}_2, \kappa, \mathrm{ob}(\delta^b) \quad \text{in} \\
 & & \quad\quad\quad \text{let } \mathbb{S}_2 = S_1; \mathbb{M}\left(\mathbb{S}_1(b), \mathbb{S}_1(\kappa)\right) \text{ in} \\
 & & \quad\quad\quad\quad \langle \mathbb{S}_2(\Gamma_2), \mathbb{S}_2(\mathrm{ob}_2), () \rangle \\
\mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, \text{certify } x\ (\lambda z)\mathrm{P}\ (\lambda z)\mathrm{Q}\right) & = & \text{let } \left\langle \Gamma_1, \mathrm{ob}_1, \delta_1^b \right\rangle = \mathrm{Type}_\pi\left(\Gamma, \mathrm{ob}, x\right) \text{ in} \\
 & & \quad \text{let } \langle \Gamma_2, \mathrm{ob}_2, (\delta_2^c) \rangle = \mathrm{Type}_\pi\left((\Gamma_x)_1, \mathrm{ob}_1, (\lambda z)\mathrm{P}\right) \text{ in} \\
 & & \quad\quad \text{if } x \in \mathrm{dom}(\Gamma_2) \text{ then } \bot \text{ else} \\
 & & \quad\quad \text{let } \left\langle \Gamma_3, \mathrm{ob}_3, (\delta_3^d) \right\rangle = \mathrm{Type}_\pi\left(\Gamma_2, \mathrm{ob}_2, (\lambda z)\mathrm{Q}\right) \text{ in} \\
 & & \quad\quad\quad \text{if } x \in \mathrm{dom}(\Gamma_3) \text{ then } \bot \text{ else} \\
 & & \quad\quad\quad \text{let } \mathbb{S}_1 = \mathrm{U}_\pi \quad \mathrm{ob}_3, \delta_1^{\mathrm{T}}, \delta_2^c \quad \text{in} \\
 & & \quad\quad\quad\quad \text{let } \mathbb{S}_2 = \mathbb{S}_1; \mathrm{U}_\pi \quad \mathbb{S}_1(\mathrm{ob}_3), \mathbb{S}_1(\delta_1^{\mathrm{U}}), \mathbb{S}_1(\delta_3^d) \quad \text{in} \\
 & & \quad\quad\quad\quad\quad \langle \mathbb{S}_2(\Gamma_3 \cup \mathrm{SORT}\left(\Gamma_1, x\right)), \mathbb{S}_2(\mathrm{ob}_3), () \rangle
\end{array}
$$

Figure 5: Sort Inference Algorithm

- P≡ x.F: By the induction hypothesis $\Gamma_1, \mathrm{ob}_1, \delta^b = \mathrm{Type}_\pi(\Gamma', \mathrm{ob}', x)$, so $\Gamma_1, \mathrm{ob}_1 \vdash x : \delta^b$. From the statement, $\mathbb{S}_1 = \mathbb{M}\ b, \mathrm{ob}_1(\delta^b)$ must be defined and hence by the soundness of $\mathbb{M}$,
$\forall \delta_k^c \in \mathrm{ob}_1(\delta^b).\exists d.\mathbb{S}_1(\delta_k^c) = \mathbb{S}_1(b \cdot \delta_k^d)$; i.e.
$\mathbb{S}_1(\mathrm{ob}_1(\delta^b)) = b \cdot \kappa'$ for some κ'. Again by the induction hypothesis, $\langle \Gamma_2, \mathrm{ob}_2, \kappa\rangle = \mathrm{Type}_\pi(\mathbb{S}_1(\Gamma_1), \mathbb{S}_1(\mathrm{ob}_1), \mathrm{F})$, and therefore $\Rightarrow \Gamma_2, \mathrm{ob}_2 \vdash \mathrm{F} : \kappa$ By the soundness of $\mathrm{U}_{\pi l}$, $\mathbb{S}_2 = \mathbb{S}_1; \mathrm{U}_{\pi l}\ \mathrm{ob}_2, \kappa, \mathrm{ob}_2(\delta^b)$ is sound and hence $\mathbb{S}_2(\Gamma_2, \mathrm{ob}_2) \vdash x.\mathrm{F}; ()$ as required, where
$\langle \mathbb{S}_2(\Gamma_2), \mathbb{S}_2(\mathrm{ob}_2), ()\rangle = \mathrm{Type}_\pi(\Gamma', \mathrm{ob}', x.\mathrm{F})$

- P≡certify x $(\lambda z)\mathrm{P}'$ $(\lambda z)\mathrm{Q}$: By the induction hypothesis, given $\Gamma_1, \mathrm{ob}_1, \delta_1^b = \mathrm{Type}_\pi(\Gamma, \mathrm{ob}, x)$ then $\Gamma_1, \mathrm{ob}_1 \vdash x : \delta_1^b$. Similarly
$\langle \Gamma_2, \mathrm{ob}_2, (\delta_2^c)\rangle = \mathrm{Type}_\pi((\Gamma_1)_x, \mathrm{ob}_1, (\lambda z)\mathrm{Q})$
$\Rightarrow \Gamma_2, \mathrm{ob}_2 \vdash (\lambda z)\mathrm{Q} : (\delta_2^c)$
and
$\Gamma_3, \mathrm{ob}_3, (\delta_3^d) = \mathrm{Type}_\pi((\Gamma_2, \mathrm{ob}_2, (\lambda z)\mathrm{R})$
$\Rightarrow \Gamma_3, \mathrm{ob}_3 \vdash (\lambda z)\mathrm{R} : (\delta_3^d)$
Note that if $x \in fn(\mathrm{Q}, \mathrm{R})$ then Type_π fails; however according to the statement it is defined hence both these steps are.
Now by the soundness of U_π we have $\mathbb{S}_1(\delta_1\mathrm{T}) = \mathbb{S}_1(\delta_2^c)$ and $\mathbb{S}_2(\delta_1^{\mathrm{U}}) = \mathbb{S}_2(\delta_3^d)$ where $\mathbb{S}_1 = \mathrm{U}_\pi(\mathrm{Id}, \delta_1\mathrm{T}, \delta_2^c)$ and $\mathbb{S}_2 = \mathrm{U}_\pi\ \mathbb{S}_1, \delta_1^{\mathrm{U}}, \delta_3^d$

  Then by lemma Sub1 $\mathbb{S}_2(\Gamma_3, \mathrm{ob}_3) \vdash (\lambda z)\mathrm{Q} : (\delta^{\mathrm{T}})$ where $\mathbb{S}_2(\delta_2^c) = \mathbb{S}_2(\delta_1^{\mathrm{T}}) = \delta^{\mathrm{T}}$
  Similarly $\mathbb{S}_2(\Gamma_3, \mathrm{ob}_3) \vdash (\lambda z)\mathrm{R} : (\delta^{\mathrm{U}})$ Then by the (cert.) type rule
  $\mathbb{S}_2(\Gamma_3 \cup \mathrm{SORT}(\Gamma, x), \mathrm{ob}_3) \vdash \mathrm{certify}\ x\ (\lambda z)\mathrm{Q}\ (\lambda z)\mathrm{R} : ()$ as required.

6.2 Completeness of Type$_\pi$

THEOREM 6.2. *If for some $\Gamma', \mathrm{ob}', \kappa'$ there is a valid deduction $\Gamma', \mathrm{ob}' \vdash \mathrm{P} : \kappa'$ for P, then $\mathrm{Type}_\pi(\emptyset, \emptyset, \mathrm{P})$ is defined, and if $\langle \Gamma, \mathrm{ob}, \kappa\rangle = \mathrm{Type}_\pi(\emptyset, \emptyset, \mathrm{P})$ then $\langle \Gamma, \mathrm{ob}, \kappa\rangle \leq \langle \Gamma', \mathrm{ob}', \kappa'\rangle$.*

Proof:

By induction on the structure of P. For example:

- P≡ (λx)F: The deduction
  $\Gamma', \mathrm{ob}' \vdash (\lambda x)\mathrm{F} : (\delta'^{b'})^\wedge\kappa'$ must end in a use of the (abs.) rule with antecedent $\Gamma' \cup \{x : \delta'^{b'}\}, \mathrm{ob}' \vdash \mathrm{F} : \kappa'$ Given $\Gamma_1 \cup \{x : \delta^b\}, \mathrm{ob}_1, \delta^b = \mathrm{Type}_\pi(\emptyset, \emptyset, x)$ we have $\Gamma_1 \cup \{x : \delta^b\}, \mathrm{ob}_1, \delta^b \leq \left\langle \Gamma' \cup \{x : \delta'^{b'}\}, \mathrm{ob}', \delta'^{b'}\right\rangle$
  Then by the induction hypothesis $\Gamma \cup \{x : \delta^b\}, \mathrm{ob}, \kappa = \mathrm{Type}_\pi\ \Gamma_1 \cup \{x : \delta^b\}, \mathrm{ob}_1, \mathrm{F}$ is defined, and
  $\Gamma \cup \{x : \delta^b\}, \mathrm{ob}, \kappa \leq \left\langle \Gamma' \cup \{x : \delta'^{b'}\}, \mathrm{ob}', \kappa'\right\rangle$ Then by the type rules, $\mathrm{Type}_\pi(\emptyset, \emptyset, (\lambda x)\mathrm{F})$ is defined and
  $\Gamma, \mathrm{ob}, (\delta^b)^\wedge\kappa \leq \left\langle \Gamma', \mathrm{ob}', (\delta'^{b'})^\wedge\kappa'\right\rangle$

- P≡ certify x $(\lambda z)\mathrm{Q}$ $(\lambda z)\mathrm{R}$: The deduction
  $\Gamma', \mathrm{ob}' \vdash \mathrm{certify}\ x\ (\lambda z)\mathrm{Q}\ (\lambda z)\mathrm{R}$ must end in a use of the (cert.) rule, with antecedents $\Gamma', \mathrm{ob}' \vdash x : \delta'^{b'}$, $\Gamma', \mathrm{ob}' \vdash (\lambda z)\mathrm{Q} : (\delta'^{\mathrm{T}})$, and $\Gamma', \mathrm{ob}' \vdash (\lambda z)\mathrm{R} : (\delta'^{\mathrm{U}})$
  Given $\Gamma_1, \mathrm{ob}_1, \delta^b = \mathrm{Type}_\pi(\emptyset, \emptyset, x)$, then
  $\Gamma_1, \mathrm{ob}_1, \delta^b \leq \left\langle \Gamma', \mathrm{ob}', \delta'^{b'}\right\rangle$ By the induction hypothesis, the following are defined:
  $\langle \Gamma_2, \mathrm{ob}_2, (\delta_2^c)\rangle = \mathrm{Type}_\pi((\Gamma_1)_x, \mathrm{ob}_1, (\lambda z)\mathrm{Q})$ and
  $\Gamma_3, \mathrm{ob}_3, (\delta_3^d) = \mathrm{Type}_\pi(\Gamma_2, \mathrm{ob}_2, (\lambda z)\mathrm{R})$. Note that if $x \in fn(\mathrm{Q}, \mathrm{R})$ then Type_π fails; however according to the statement a valid deduction exists so this is not the case. Then by completeness and soundness of U_π, the following statements hold: $\mathbb{S}_1 = \mathrm{U}_\pi\ \mathrm{Id}, \delta^{\mathrm{T}}, \delta_2^c \Rightarrow \mathbb{S}_1(\delta^{\mathrm{T}}) =_\pi \mathbb{S}_1(\delta_2^c)$ and $\mathbb{S}_2 = \mathrm{U}_\pi\ \mathbb{S}_1, \delta^{\mathrm{U}}, \delta_3^d \Rightarrow \mathbb{S}_2(\delta^{\mathrm{U}}) =_\pi \mathbb{S}_1(\delta_3^d)$. Hence
  $\langle \mathbb{S}_2(\Gamma_3 \cup \mathrm{SORT}(\Gamma_1, x), \mathrm{ob}_3), ()\rangle \leq \langle \Gamma', \mathrm{ob}', ()\rangle$ as required.

The proofs of soundness and completeness of Type_π depend on the corresponding proofs of the auxiliary algorithms, primarily the unification algorithms used. These results are stated here, but again to preserve space the proofs themselves are not presented, as they are fairly standard.

6.3 Soundness and Completeness of Unification

THEOREM 6.3. *If $\mathbb{S} = \mathrm{U}_\pi\ \mathrm{ob}, \mathrm{S}_1^i, \mathrm{S}_2^j$ is defined, then $\mathbb{S}(\mathrm{S}_1^i) = \mathbb{S}(\mathrm{S}_2^j)$.*

Proof:

By induction on S_1^i, S_2^j and the definition of U_π.

One must be careful proving completeness of U_π; because the sorts in ob in effect form a graph, we must consider not only the sorts being unified but the entire network. To this end, we introduce the following definition (figure 6):

THEOREM 6.4. *If $\mathbb{S}(\mathrm{S}_1^i) = \mathbb{S}(\mathrm{S}_2^j)$ and consistent $\emptyset, \mathrm{ob}, \mathrm{ob}(\mathrm{S}_1^i), \mathrm{ob}(\mathrm{S}_2^j)$ then $\mathrm{U}_\pi\ \mathrm{ob}, \mathrm{S}_1^i, \mathrm{S}_2^j$ is defined.*

Proof:

By induction on S_1^i, S_2^j and the definition of U_π.

THEOREM 6.5. *If $\mathbb{S} = \mathrm{U}_{\pi l}(\mathrm{ob}, b, \kappa_1)\,\kappa_2$ is defined, then $\mathbb{S}(\kappa_1) = \mathbb{S}(\kappa_2)$.*

Proof:

By cases on κ_1 and κ_2, and by induction on the definition of $\mathrm{U}_{\pi l}$.

THEOREM 6.6. *If $\mathbb{S}(r) = \mathbb{S}(t)$ and consistent $(\emptyset, \mathrm{ob}, r, t)$ is true, then $\mathrm{U}_{\pi l}(\mathrm{ob}, b, r)\,t$ is defined.*

Proof:

By induction on r, and t, and on the definition of $\mathrm{U}_{\pi l}$.

THEOREM 6.7.
- *If $\mathbb{S} = \mathbb{M}(b, \delta^c)$ is defined then $\exists d.\mathbb{S}(\delta^c) = \mathbb{S}(b \cdot \delta^d)$*
- *If $\mathbb{S} = \mathbb{M}(b, \kappa)$ is defined then $\forall \delta^c \in \kappa.\exists d.\mathbb{S}(\delta^c) = \mathbb{S}(b \cdot \delta^d)$*

Proof:

By induction on the structure of δ^c or κ as appropriate.

THEOREM 6.8.
- *If $\exists b, c, \mathbb{S}.\mathbb{S}(\delta^c) = \mathbb{S}(b \cdot \delta^d)$ then $\mathbb{M}(b, \delta^c)$ is defined.*
- *If $\exists b, c, \mathbb{S}.\mathbb{S}(\delta^c) = \mathbb{S}(b \cdot \delta^d)$ then $\mathbb{M}(b, \kappa)$ is defined $\forall \delta^c \in \kappa$.*

$$
\begin{array}{rcl}
\text{consistent}\quad \Sigma, \text{ob}, \delta_1^b, \delta_2^c & = & \text{if } \{\delta_1^b, \delta_2^c\} \in \Sigma \text{ then true else} \\
 & & \text{isVar}(\delta_1^b) \vee \text{isVar}(\delta_2^c) \vee \\
 & & \text{isCon}(\delta_1^b) \wedge \text{isCon}(\delta_2^c) \wedge (\delta_1^b =_\pi \delta_2^c) \\
 & & \wedge \text{isDefined}(\text{BUNIFY}\ (b, c)) \\
 & & \wedge \text{consistent}\quad \Sigma \cup \{\{\delta_1^b, \delta_2^c\}\}, \text{ob}, \text{ob}(\delta_1^b), \text{ob}(\delta_2^c) \\
\text{consistent}\quad \Sigma, \text{ob}, \delta_1^b, \delta_1^b & = & true \\
\text{consistent}\ (\Sigma, \text{ob}, \beta, \kappa) & = & \text{consistent}\ (\Sigma, \text{ob}, \kappa, \beta) = true \\
\text{consistent}\quad \Sigma, \text{ob}, (\delta_1^b)^\wedge \kappa_1, (\delta_2^c)^\wedge \kappa_2 & = & \text{consistent}\quad \Sigma, \text{ob}, \delta_1^b, \delta_2^c \quad \wedge \\
 & & \text{consistent}\ (\Sigma, \text{ob}, \kappa_1, \kappa_2) \\
\text{consistent}\ (\Sigma, \text{ob}, (), ()) & = & true \\
\text{consistent}\quad \Sigma, \text{ob}, (\delta_1^b)^\wedge \kappa, () & = & \text{consistent}\quad \Sigma, \text{ob}, (), (\delta_1^b)^\wedge \kappa \quad = false
\end{array}
$$

Figure 6: Consistency Definition

Proof:

By induction on δ^c or κ as appropriate.

6.4 Termination of Unification Algorithms

Given that the type unification algorithms are mutually recursive, it is desirable to prove that they will eventually terminate. A proof of this exists; however due to space reasons we have ommited it. As a summary, it works by proving that each recursive call operates on a smaller ob than its parent, and thus must eventually terminate.

7. CONCLUSIONS

We have presented a system of trust analysis for networks using the π-calculus. Boolean expressions as annotations to the base type system are used for this purpose. A safe environment for performing coercion of trustedness information based on the results of runtime verification was introduced. An algorithm for determining the most general type of an expression in our system was also presented.

7.1 Related Work

The work presented here bares many similarities to the discipline of *flow analysis*; for example that presented in [7, 23, 9]. Such work usually deals with a concept of security levels; typically the idea that data should not be able to flow (eg by assignment) from a high security level to a low one. More generally, this is extended to use multi-point lattices which may include notions of trust. We believe that our approach differs from this area in several key points: our system naturally handles non-determinism, whereas traditional methods have difficulty in this regard ([23]); our system requires no extra run-time type tag information; and we provide a safe and natural method for performing runtime coercion of these trust levels. We also believe our approach can be extended to higher order systems fairly simply, and importantly without much more complexity being introduced into the analysis; current work appears to verify this.

Sewell and Vitek[21] present a system in which "off the shelf" software can run in a constrained environment provided by software wrappers. Their analysis uses the *box-*π calculus, and they investigate the flow of information across the boundaries of these wrappers. No avenue for allocating trustedness is provided however.

Igarishi and Kobayashi[10] study linear type systems for the π-calculus, using a similar system of type annotations (derived indepedently of [25, 24]).

7.2 Future Directions

Future work primarily involves study of a higher-order π-calculus (and corresponding type system). It has been demonstrated[20] that in the regular π-calculus system higher-order terms can be encoded as first-order with no loss of expressiveness. We do not believe this to be the case for our system (whether this view proves correct remains to be seen!); we view a higher-order calculus as encoding mobile code and thus potentially a situation involving not just untrusted data but also arbitrary code executing on your system. One consequence of a higher-order system is it seems unreasonable to trust data coming from an untrusted process, even along a trusted channel. This implies that processes themselves must be annotated as well; for example

$$\frac{\Gamma, \text{ob} \vdash \text{P} : ()^b \quad \Gamma, \text{ob} \vdash \text{Q} : ()^c}{\Gamma, \text{ob} \vdash \text{P}|\text{Q} : ()^{b \cdot c}}$$

and perhaps more interestingly,

$$\frac{\begin{array}{l}\Gamma, \mathrm{ob} \vdash x : \delta^b \\ \Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{P} : (\delta^{\mathrm{T}})^c \\ \Gamma, \mathrm{ob} \vdash (\lambda z)\mathrm{Q} : (\delta^{\mathrm{U}})^c \\ x \notin FV(\mathrm{P}, \mathrm{Q})\end{array}}{\Gamma, \mathrm{ob} \vdash \text{certify } x \text{ P Q} : ()^{b \cdot c + \hat{b} \cdot d}}$$

Note that this last example provides a primitive kind of dependent type; if x can be trusted then the statement reduces as P and the expression $b \cdot c + \hat{b} \cdot d$ reduces to c; the annotation on P above the line. Similarly if x proves to untrusted.

The authors also intend to investigate a *type* system (as opposed to the current system of sorts) as is currently in vogue; with this sort of arrangement instead of a mapping between sorts and their objects, all information about what is carried by the channels is contained in the types. (Note that to get the same power as sorts it is then necessary to admit recursive types). While this scheme does appear to lose name-equivalence among types, it is hoped that the corresponding inference algorithms can be simplified (both in time and readability); in [22] a type inference algorithm that is linear on the size of the input process is presented (for an un-annotated type system).

8. ACKNOWLEDGEMENTS

Hepburn was supported by a University of Tasmania Postgraduate Research Scholarship. The authors are grateful to the anonymous referees for their insightful and helpful comments.

9. REFERENCES

[1] http://www.beowulf.org/.

[2] http://www.seti.org/.

[3] http://www.distributed.net/.

[4] http://www.popularpower.com/.

[5] Abadi. Secrecy by typing in security protocols. In *TACS: 3rd International Conference on Theoretical Aspects of Computer Software*, 1997.

[6] D. E. Denning. The limits of formal security models. National Computer Systems Security Award Acceptance Speech.

[7] D. E. Denning. A lattice model of information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[8] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *27th International Colloquium on Automata, Languages and Programming (ICALP '2000)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, July 2000. A longer version appeared as Computer Science Technical Report 2000:03, School of Cognitive and Computing Sciences, University of Sussex.

[9] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming (ESOP 2000), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2000), (Berlin, Germany, March/April 2000)*, volume 1782 of *lncs*, pages 180–199. sv, 2000.

[10] A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Journal of Information and Computation*, 161(1):1–44, 2000. An extended abstract appeared in the *Proceedings of SAS '97*, LNCS 1302.

[11] F. Levi and C. Bodei. Security analysis for mobile ambients.

[12] U. Martin and T. Nipkow. Boolean unification — the story so far. In C. Kirchner, editor, *Unification*, pages 437–456. Academic Press, 1990.

[13] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, Sept. 1992.

[16] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[17] P. Ørbæk. Can you Trust your Data? In M. I. S. P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, volume 915 of LNCS of *Springer Lecture Notes in Computer Science*, pages 575–590, Aarhus, Denmark, may 1995. Springer-Verlag.

[18] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In A. Mycroft, editor, *SAS'95: Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 314–330. Springer-Verlag, 1995.

[19] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Lecture Notes in Computer Science*, 1576:40–58, 1999.

[20] D. Sangiorgi. From pi-calculus to higher-order pi-calculus—and back. In *TAPSOFT*. Springer Verlag, LNCS 668, 1993.

[21] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. Technical Report 478, Computer Laboratory, University of Cambridge, 1999.

[22] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic pi–calculus. *Lecture Notes in Computer Science*, 715:524–??, 1993.

[23] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, Dec. 1996.

[24] D. A. Wright. *Reduction Types and Intensionality in the Lambda-Calculus*. PhD thesis, University of Tasmania, 1992.

[25] D. A. Wright. Linear, strictness and usage logics. In M. E. Houle and P. Eades, editors, *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 73–80, Townsville, Jan. 29–30 1996. Australian Computer Science Communications.

[26] R. Yahalom, B. Klein, and T. Beth. Trust-based navigation in distributed systems, 1994.