

Proof-Carrying Code in a Session-Typed Process Calculus

Frank Pfenning¹, Luis Caires², and Bernardo Toninho^{1,2}

¹ Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA, USA

² Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa,
Lisboa, Portugal

Abstract. Dependent session types allow us to describe not only properties of the I/O behavior of processes but also of the exchanged data. In this paper we show how to exploit dependent session types to express proof-carrying communication. We further introduce two modal operators into the type theory to provide detailed control about how much information is communicated: one based on traditional proof irrelevance and one integrating digital signatures.

Keywords: Process calculus, session types, proof irrelevance, proof-carrying code

1 Introduction

Session types [10] provide high-level specifications for the communication behavior of interacting processes along bidirectional channels. Recently, logical foundations for session types have been established via Curry-Howard correspondences with linear logic [5, 11]. Besides clarifying and unifying concepts in session types, such logical underpinnings provide simple means for generalization. One such extension to *dependent session types* [4, 18] allows us to express and enforce complex properties of data transmitted during sessions.

In this paper we build upon dependent session types to model various aspects of systems employing certified code. Already, just dependent session types can model basic proof-carrying code since a dependent type theory uniformly integrates proofs and programs. Briefly, a process implementing a session type $\forall x:\tau.A(x)$ will input a data value M of type τ and then behave as $A(M)$, while $\exists x:\tau.A(x)$ will output a data value M of type τ and then behave as $A(M)$. The data values are taken from an underlying functional layer which is dependently typed. The session type **1** indicates termination of the session. For example, the following is the specification of a session that accepts the code for a function on natural numbers, a proof that the function is decreasing, and emits a fixed point of that function.

$$\forall f:\text{nat} \rightarrow \text{nat}. \forall p:(\Pi x:\text{nat}. f(x) \leq x). \exists y:\text{nat}. \exists q:(y = f(y)). \mathbf{1}$$

In a session of the type above, two proof objects will be transmitted: one (labeled p) showing that the function f is decreasing will be passed to the server, and a second one (labeled q), that the returned value y is indeed a fixed point, will be passed back to the client. Note that the propositions $n \leq m$ and $n = m$ act as types of their proofs, according to the usual Curry-Howard correspondence.

The client may easily check that the returned value y is indeed a fixed point by computing $f(y)$ itself, so we would like to avoid transmitting a proof q of $y = f(y)$. But we do not want to erase this requirement entirely, of course, just avoid sending a proof term. We can do this by using the type-theoretic concept of *proof irrelevance* [15, 14, 2]. Generally, a type $[A]$ (pronounced “*bracket A*”) is the type inhabited by proofs of A , all of which are identified. This is only meaningful if such proofs play no computational role, so there is some subtlety to the type system presented in Section 3. The revised specification would be:

$$\forall f:\text{nat} \rightarrow \text{nat}. \forall p:(\Pi x:\text{nat}. f(x) \leq x). \exists y:\text{nat}. \exists q:[y = f(y)]. \mathbf{1}$$

Irrelevant proofs terms are eliminated in the operational semantics of our type theory, so just a unit element would be communicated instead of a proof. The residual communication overhead can also be optimized away using two different techniques (see Section 3).

The proof that a given function is decreasing may be complex, so the server may try to avoid checking this proof, delegating it instead to a trusted verifier. This verifier would sign a digital certificate to the effect that there is a proof that the function is decreasing. We integrate this into our type theory with a type constructor $\diamond_K A$ (read “ K says A ”), where K is a principal and A is a proposition. We want this certificate not to contain the proof, so the proof itself is marked as irrelevant. We obtain:

$$\forall f:\text{nat} \rightarrow \text{nat}. \forall p:\diamond_{\text{verif}}[\Pi x:\text{nat}. f(x) \leq x]. \exists y:\text{nat}. \exists q:[y = f(y)]. \mathbf{1}$$

In the implementation, we assume a public key infrastructure so that the verifier can sign a certificate containing the proposition $[\Pi x:\text{nat}. f(x) \leq x]$ and the server can reliably and efficiently check the signature. Our experience with a proof-carrying file system [8] shows that digitally signed certificates are much more compact and can be checked much more quickly than proofs themselves and are one of the cornerstones to make the architecture practical. In this paper we show that they can be accommodated elegantly within session types, based on logical grounds.

We begin in Section 2 with an overview of dependent session types in a term passing variant of the π -calculus, as formulated in previous work by the authors. In Section 3 we define proof irrelevance and how it is used in our operational model, followed by a discussion of affirmation as a way of integrating digitally signed certificates into sessions in Section 4. We sketch some standard meta-theoretic results regarding progress and preservation in Section 5 and conclude in Section 6.

2 Dependent session types

In this section we will briefly review dependent session types and the facilities they provide in terms of proof-carrying communication. Dependent session types [4, 18] are a conservative extension of session types [10, 5, 6, 9] that allow us to not only describe the behavior of processes in terms of their input and output behavior but also enable us to describe rich properties of the communicated data themselves.

In [18], the authors investigated a natural interpretation of linear type theory as a dependent session typed π -calculus.

Definition 1 (Types). *Types in linear type theory are freely generated by the following grammar, given types τ from a standard dependent type theory:*

$$A, B ::= \mathbf{1} \quad | \quad A \multimap B \quad | \quad A \otimes B \quad | \quad A \& B \quad | \quad A \oplus B \quad | \quad !A \\ \forall x:\tau.A \quad | \quad \exists x:\tau.A$$

A process P offering a service A along a channel z is typed as $P :: z:A$ and we obtain an interpretation of the types as follows:

$$\begin{aligned} P :: x : \mathbf{1} & \quad \text{inaction} \\ P :: x : A \multimap B & \quad \text{input a channel of type } A \text{ along } x \text{ and continue as } B \\ P :: x : A \otimes B & \quad \text{output a fresh channel } y \text{ of type } A \text{ along } x \text{ and continue as } B \\ P :: x : A \& B & \quad \text{offer the choice between } A \text{ and } B \text{ along } x \\ P :: x : A \oplus B & \quad \text{provide either } A \text{ or } B \text{ along } x \\ P :: x : !A & \quad \text{provide a persistent (replicating) service } A \text{ along } x \\ P :: x : \forall y:\tau. A & \quad \text{input a value } M \text{ of type } \tau \text{ along } x \text{ and continue as } A\{M/y\} \\ P :: x : \exists y:\tau. A & \quad \text{output a value } M \text{ of type } \tau \text{ along } x \text{ and continue as } A\{M/y\} \end{aligned}$$

As an example consider the following type:

$$\mathbb{T} \triangleq \forall n:\text{nat}. \forall p:(n > 0). \exists y:\text{nat}. \exists q:(y > 0). \mathbf{1}$$

The type \mathbb{T} specifies a session that receives a positive number n and sends another positive number y . A process that implements this session (along channel x) is:

$$P :: x : \mathbb{T} \triangleq x(n).x(p).x\langle n+1 \rangle.x\langle \text{incp } n \, p \rangle. \mathbf{0}$$

where $\text{incp } n \, p$ denotes a proof term of type $n+1 > 0$, computed by a function:

$$\text{incp} : \Pi m : \text{int}. (m > 0) \rightarrow (m+1 > 0)$$

The properties of the communicated data (in this case, the positivity of both numbers) are made explicit by the exchange of terms that act as proof certificates for the properties, by inhabiting the appropriate types.

Our type system arises as a direct interpretation of the rules of linear logic as typing rules for processes, thus our typing judgment is essentially the same as that for a linear logic sequent calculus with a proof term assignment, but

singling out a specific channel in which the considered session is being offered. The typing judgment for our system is written as $\Psi; \Gamma; \Delta \Rightarrow P :: z : A$, where Ψ consists of assumptions of the form $x:\tau$, Γ consists of persistent assumptions of the form $u:A$, and Δ consists of linear assumptions of the form $x:A$. We assume all variables in these contexts to be distinct.

The typing judgment above denotes that process P implements session A along channel z , provided it is placed in a process environment that offers the sessions and values specified in contexts Ψ , Γ and Δ . The typing rules for our system are given below in Fig. 1, and are defined modulo structural congruence of processes. Following standard sequent calculus presentations of logic, our system is made up of so-called right and left rules that define the types, and structural rules that denote sound reasoning principles in logic. In our interpretation, right rules define how to implement a session of a particular type, while left rules define how to use such a session. The standard reasoning principles of cut and identity correspond to process composition and channel forwarding (i.e., communication along a channel being replaced by communication on another).

As previously mentioned, our process calculus is a π -calculus where processes can communicate not only channel names as usual, but also terms from a typed functional language, defined by the typing judgment $\Psi \vdash N:\tau$, whose proof rules we deliberately leave open.

Definition 2 (Processes). *Processes are defined by the following grammar, where P, Q range over processes, x, y over names and N over terms.*

$$P, Q ::= \mathbf{0} \mid P|Q \mid (\nu y)P \mid x\langle y \rangle.P \mid x(y).P \mid x\langle N \rangle.P \\ \mid !x(y).P \mid x.\text{inl}; P \mid x.\text{inr}; P \mid x.\text{case}(P, Q) \mid [y \leftrightarrow x]$$

Most constructs are standard. We highlight the term output construct $x\langle N \rangle.P$, the binary guarded choice constructs $x.\text{inl}; P$ and $x.\text{inr}; P$ with the corresponding case construct; the channel forwarding or renaming construct $[y \leftrightarrow x]$ that links the channels x and y .

Processes are equated up to a structural congruence \equiv , defined below.

Definition 3 (Structural Congruence). *Structural congruence is defined as the least congruence relation closed under the following rules:*

$$\begin{array}{ll} P \mid \mathbf{0} \equiv P & P \equiv_{\alpha} Q \Rightarrow P \equiv Q \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid Q \equiv Q \mid P \\ x \notin \text{fn}(P) \Rightarrow P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) & (\nu x)\mathbf{0} \equiv \mathbf{0} \\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & [y \leftrightarrow x] \equiv [x \leftrightarrow y] \end{array}$$

The operational semantics for the process calculus are standard. The semantics for the $[y \leftrightarrow x]$ construct, as informed by the proof theory, consist of channel renaming.

$$\begin{array}{c}
\frac{}{\Psi; \Gamma; \cdot \Rightarrow \mathbf{0} :: z : \mathbf{1}} \mathbf{1R} \quad \frac{\Psi; \Gamma; \Delta \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : \mathbf{1} \Rightarrow P :: z : C} \mathbf{1L} \\
\frac{\Psi; \Gamma; \cdot \Rightarrow P :: y : A}{\Psi; \Gamma; \cdot \Rightarrow !z(y).P :: z : !A} \mathbf{!R} \quad \frac{\Psi; \Gamma, u : A; \Delta \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : !A \Rightarrow P\{x/u\} :: z : C} \mathbf{!L} \\
\frac{\Psi; \Gamma; \Delta \Rightarrow P :: z : A \quad \Psi; \Gamma; \Delta \Rightarrow Q :: z : B}{\Psi; \Gamma; \Delta \Rightarrow z.\text{case}(P, Q) :: z : A \& B} \&R \\
\frac{\Psi; \Gamma; \Delta, x : A \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : A \& B \Rightarrow x.\text{inl}; P :: z : C} \&L_1 \\
\frac{\Psi; \Gamma; \Delta, x : B \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : A \& B \Rightarrow x.\text{inr}; P :: z : C} \&L_2 \\
\frac{\Psi; \Gamma; \Delta_1 \Rightarrow P :: y : A \quad \Psi; \Gamma; \Delta_2 \Rightarrow Q :: z : B}{\Psi; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu y)z\langle y \rangle.(P \mid Q) :: z : A \otimes B} \otimes R \\
\frac{\Psi; \Gamma; \Delta, y : A, x : B \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : A \otimes B \Rightarrow x(y).P :: z : C} \otimes L \\
\frac{\Psi; \Gamma; \Delta \Rightarrow P :: z : A}{\Psi; \Gamma; \Delta \Rightarrow z.\text{inl}; P :: z : A \oplus B} \oplus R_1 \quad \frac{\Psi; \Gamma; \Delta \Rightarrow P :: z : B}{\Psi; \Gamma; \Delta \Rightarrow z.\text{inr}; P :: z : A \oplus B} \oplus R_2 \\
\frac{\Psi; \Gamma; \Delta, x : A \Rightarrow P :: z : C \quad \Psi; \Gamma; \Delta, x : B \Rightarrow Q :: z : C}{\Psi; \Gamma; \Delta, x : A \oplus B \Rightarrow x.\text{case}(P, Q) :: z : C} \oplus L \\
\frac{}{\Psi; \Gamma; x : A \Rightarrow [x \leftrightarrow z] :: z : A} \text{id} \quad \frac{\Psi, x : \tau; \Gamma; \Delta \Rightarrow P :: z : A}{\Psi; \Gamma; \Delta \Rightarrow z(x).P :: z : \forall x : \tau. A} \forall R \\
\frac{\Psi \vdash N : \tau \quad \Psi; \Gamma; \Delta, x : A\{N/y\} \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : \forall y : \tau. A \Rightarrow x(N).P :: z : C} \forall L \\
\frac{\Psi \vdash N : \tau \quad \Psi; \Gamma; \Delta \Rightarrow P : A\{N/x\}}{\Psi; \Gamma; \Delta \Rightarrow z(N).P :: z : \exists x : \tau. A} \exists R \quad \frac{\Psi, y : \tau; \Gamma; \Delta, x : A \Rightarrow P :: z : C}{\Psi; \Gamma; \Delta, x : \exists y : \tau. A \Rightarrow x(y).P :: z : C} \exists L \\
\frac{\Psi; \Gamma, u : A; \Delta, y : A \Rightarrow P :: z : C}{\Psi; \Gamma, u : A; \Delta \Rightarrow (\nu y)u(y).P :: z : C} \text{copy} \\
\frac{\Psi; \Gamma; \Delta_1 \Rightarrow P :: x : A \quad \Psi; \Gamma; \Delta_2, x : A \Rightarrow Q :: z : C}{\Psi; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu x)(P \mid Q) :: z : C} \text{cut} \\
\frac{\Psi; \Gamma; \cdot \Rightarrow P :: x : A \quad \Psi; \Gamma, u : A; \Delta \Rightarrow Q :: z : C}{\Psi; \Gamma; \Delta \Rightarrow (\nu u)((!u(x).P) \mid Q) :: z : C} \text{cut}^!
\end{array}$$

Fig. 1. Dependent Session Types.

Definition 4 (Reduction). *The reduction relation on processes, $P \rightarrow Q$ is defined by the following rules:*

$$\begin{aligned}
& x\langle y \rangle.Q \mid x(z).P \rightarrow Q \mid P\{y/z\} \\
& x\langle y \rangle.Q \mid !x(z).P \rightarrow Q \mid P\{y/z\} \mid !x(z).P \\
& x\langle N \rangle.Q \mid x(z).P \rightarrow Q \mid P\{N/z\} \\
& (\nu x)([x \leftrightarrow y] \mid P) \rightarrow P\{y/x\} \\
& x.\text{inl}; P \mid x.\text{case}(Q, R) \rightarrow P \mid Q \\
& x.\text{inr}; P \mid x.\text{case}(Q, R) \rightarrow P \mid R \\
& Q \rightarrow Q' \Rightarrow P \mid Q \rightarrow P \mid Q' \\
& P \rightarrow Q \Rightarrow (\nu y)P \rightarrow (\nu y)Q \\
& P \equiv P', P' \rightarrow Q', Q' \equiv Q \Rightarrow P \rightarrow Q
\end{aligned}$$

A labeled transition system can be defined in a somewhat standard manner, where a label denotes a silent action, an output or input of a (bound) name or of a term (note that terms do not contain channel names, so no issues of scope extrusion arise).

The language of terms is intentionally left open-ended. We only suppose that they contain no (π -calculus) names, and that it satisfies substitution, progress, and preservation properties as we usually suppose for functional languages. In the next section we will postulate some particular constructs that allow us to specify different versions of proof-carrying code protocols.

3 Proof irrelevance

In a dependent type theory, proofs are represented as terms. Even with basic dependent function types we already have the means to model proof-carrying code, as explained in the introduction and the previous section. This assumes that data values transmitted along channels are type-checked when received before we continue to compute with them in a type-safe way.

Under which circumstances can we avoid type-checking a proof object, or perhaps even avoid transmitting it entirely? One class of examples is provided by cases where the property of the objects we specified is (easily) decidable. Then we can check the property itself without the need to obtain an explicit proof object. However, this only works if the proof object is also of no actual *operational* significance, that is, it is computationally irrelevant. The previous section (e.g., $\forall p:(n > 0)$) and the introduction (e.g., $\exists q:(y = f(y))$) contain examples of this kind. But we do not want to presuppose or “bake in” any particular analysis or strategy, but formulate the type theory so that we can seamlessly move between different specifications. This is what a modality for *proof irrelevance* [15, 14, 2] in the type theory allows us to do.

Proof irrelevance is a technique that allows us to selectively hide portions of a proof (and by the proofs-as-programs principle, portions of a program). The idea is that these “irrelevant” proof objects are required to exist for the purpose

of type-checking, but they must have no bearing on the computational outcome of the program. This means that typing must ensure that these hidden proofs are never required to compute something that is not itself hidden.

We internalize proof irrelevance in our functional language by requiring a modal type constructor, $[\tau]$ (read *bracket* τ), meaning that there is a term of type τ , but the term is deemed irrelevant from a computational point of view. We give meaning to $[\tau]$ by adding an introduction form for irrelevant terms, written $[M]$, that states that M is not available computationally; and a new class of assumptions $x \div \tau$, meaning that x stands for a term of type τ that is not computationally available; we then define a promotion operation on contexts that transforms computationally irrelevant hypotheses into ordinary ones, to account for type-checking within the bracket operator.

Definition 5 (Promotion).

$$\begin{aligned} (\cdot)^\oplus &\triangleq \cdot \\ (\Psi, x : \tau)^\oplus &\triangleq \Psi^\oplus, x : \tau \\ (\Psi, x \div \tau)^\oplus &\triangleq \Psi^\oplus, x : \tau \end{aligned}$$

The introduction and elimination forms of proof irrelevant terms are defined by the following rules:

$$\frac{\Psi^\oplus \vdash M : \tau}{\Psi \vdash [M] : [\tau]} \quad \boxed{I} \qquad \frac{\Psi \vdash M : [\tau] \quad \Psi, x \div \tau \vdash N : \sigma}{\Psi \vdash \mathbf{let} [x] = M \mathbf{in} N : \sigma} \quad \boxed{E}$$

The introduction rule states that any term M (that may use irrelevant hypotheses) of type τ induces a proof irrelevant term $[M]$ of type $[\tau]$. The elimination rule states that we can unwrap the bracket operator only by binding its contents to a variable classified as proof irrelevant. This new class of variables is given meaning by an appropriate substitution principle.

Theorem 1 (Irrelevant substitution). *If $\Psi^\oplus \vdash M : \tau$ and $\Psi, x \div \tau, \Psi' \vdash N : \sigma$ then $\Psi, \Psi' \vdash N\{M/x\} : \sigma$*

Proof. By structural induction on the derivation of $\Psi, x \div \tau, \Psi' \vdash N : \sigma$

We generally prefer a call-by-value operational semantics for the type theory so that we can restrict communication to values without complications. We first extend this to a version that computes explicit evidence for inhabitation of type $[\tau]$, although the intent is to actually erase rather than compute irrelevant objects. The single-step reduction relation would then contain the following congruence and reduction rules (treating irrelevant terms lazily):

$$\frac{M \longrightarrow M'}{\mathbf{let} [x] = M \mathbf{in} N \longrightarrow \mathbf{let} [x] = M' \mathbf{in} N}$$

$$\frac{}{\mathbf{let} [x] = [M] \mathbf{in} N \longrightarrow N\{M/x\}}$$

As motivated above, the next step is to check that irrelevant terms do not need to be computed at the functional level or communicated at the process level. We formalize this through a notion of *erasure* that replaces computationally irrelevant types by a unit type `unit` and irrelevant terms by corresponding unit elements $\langle \rangle$.

Definition 6 (Erasure). *The erasure operation \dagger is defined on contexts, types, processes and terms. It is compositional everywhere, with the following special cases.*

$$\begin{aligned} (\Psi, x:\tau)^\dagger &\triangleq \Psi^\dagger, x:\tau^\dagger \\ (\Psi, x\dot{\div}\tau)^\dagger &\triangleq \Psi^\dagger \\ [\tau]^\dagger &\triangleq \text{unit} \\ [M]^\dagger &\triangleq \langle \rangle \\ (\text{let } [x] = M \text{ in } N)^\dagger &\triangleq N^\dagger \end{aligned}$$

The erasure from the definition above does not affect the process structure. It simply traverses processes down to the functional terms they manipulate and replaces bracketed terms by the unit element as specified above.

Theorem 2 (Erasure correctness).

If $\Psi; \Gamma; \Delta \Rightarrow P :: z : A$ then $\Psi^\dagger; \Gamma^\dagger; \Delta^\dagger \Rightarrow P^\dagger :: z : A^\dagger$.

Proof. Straightforward, by induction on the typing derivation. Note that in the case for the let-binding for bracket types we rely on the fact that the variable $[x]$ can only occur in a bracketed term (which is itself replaced by $\langle \rangle$ in \dagger).

Note that the lack of computational significance of proof-irrelevant terms ensures that the meanings of programs are preserved. Since erasure does not affect the structure of processes, we need only focus on the functional language itself (which we fix to be well-behaved in terms of the standard properties of progress and preservation). We can establish that erasure and evaluation commute, in the following sense (where \equiv is a standard notion of equality).

Theorem 3 (Erasure Preservation). *If $\Psi \vdash M : \tau$ and $M \longrightarrow N$, then there exists N' such that $M^\dagger \longrightarrow^* N'$ and $N^\dagger \equiv N'$.*

Proof. By induction on the operational semantics.

However, the erasure operation is just a step in the optimization mentioned above, since the processes in the image of the erasure still perform some communication (of unit elements) in the same places where proof objects were previously exchanged. To fully remove the potentially unnecessary communication, we consistently appeal to type isomorphisms regarding the interaction of `unit` with the universal and existential quantifiers:

$$\begin{aligned} \forall x:\text{unit}.A &\cong A \\ \exists x:\text{unit}.A &\cong A \end{aligned}$$

Since we only allow for types of the functional language in the universal and existential quantifiers (and terms in the appropriate process constructs), the isomorphisms above allow us to remove a communication step. For example, if we revisit our initial example of Section 2, we can reformulate the type and process as:

$$\begin{aligned} \mathsf{T}_1 &\triangleq \forall n:\mathsf{nat}. \forall p:[n > 0]. \exists y:\mathsf{nat}. \exists q:[y > 0]. \mathbf{1} \\ \mathsf{P}_1 &:: x : \mathsf{T}_1 \triangleq x(n).x(p).x\langle n+1 \rangle.x\langle [\mathsf{incp} \ n \ p] \rangle.\mathbf{0} \end{aligned}$$

By bracketing the types for the universally and existentially quantified variables p and q , we are effectively stating that we only require some proof that p and y are positive, but the content of the proof itself does not matter. Of course, since determining the positivity of an integer is easily decidable, and the form of the proof is irrelevant, we can erase the proofs using \dagger , obtaining the following process (and type):

$$\begin{aligned} \mathsf{T}_1^\dagger &\triangleq \forall n:\mathsf{nat}. \forall p:\mathsf{unit}. \exists y:\mathsf{nat}. \exists q:\mathsf{unit}. \mathbf{1} \\ \mathsf{P}_1^\dagger &:: x : \mathsf{T}_1 \triangleq x(n).x(p).x\langle n+1 \rangle.x\langle \langle \rangle \rangle.\mathbf{0} \end{aligned}$$

By consistently appealing to the type isomorphisms mentioned above, we obtain the process below that simply inputs a number n and outputs its increment:

$$\mathsf{P}_{1\cong}^\dagger \triangleq x(n).x\langle n+1 \rangle.\mathbf{0}$$

An alternative technique familiar from type theories is to replace sequences of data communications by a single communication of pairs. When proof objects are involved, these become Σ -types which are inhabited by pairs. For example, we can rewrite the example above as

$$\begin{aligned} \mathsf{T}_2 &\triangleq \forall p:(\Sigma n:\mathsf{nat}. [n > 0]). \exists q:(\Sigma y:\mathsf{nat}. [y > 0]). \mathbf{1} \\ \mathsf{P}_2 &:: x : \mathsf{T}_2 \triangleq x(\langle n, p \rangle).x\langle n+1, [\mathsf{incp} \ n \ p] \rangle.\mathbf{0} \end{aligned}$$

where we have take the liberty of using pattern matching against $\langle n, p \rangle$ instead of writing first and second projections. Applying erasure here only simplifies the communicated terms without requiring us to change the structure of the communication.

$$\begin{aligned} \mathsf{T}_2^\dagger &\triangleq \forall p:(\Sigma n:\mathsf{nat}. \mathsf{unit}). \exists q:(\Sigma y:\mathsf{nat}. \mathsf{unit}). \mathbf{1} \\ \mathsf{P}_2^\dagger &:: x : \mathsf{T}_2 \triangleq x(\langle n, _ \rangle).x\langle n+1, \langle \rangle \rangle.\mathbf{0} \end{aligned}$$

This solution is popular in type theory, where $\Sigma x:\tau. [\sigma]$ is a formulation of a *subset type* [15], $\{x:\tau \mid \sigma\}$. Conversely, bracket types $[\sigma]$ can be written as $\{x:\mathsf{unit} \mid \sigma\}$, except that the proof object is *always* erased. Under some restrictions on σ , subset types can be seen as predicate-based type refinement as available, for example, in Fine [17] where it used for secure communication in distributed computation.

4 Affirmation

In many distributed communicating systems there are trade-offs between trust and explicit proofs. For example, when we download a large application we may be willing to trust its safety if it is digitally signed by a reputable vendor. On the other hand, if we are downloading and running a piece of Javascript code embedded in a web page, we may insist on some explicit proof that it is safe and adheres to our security policy. The key to making such trade-offs explicit in session types is a notion of *affirmation* (in the sense of [7]) of propositions and proofs by principals. Such affirmations can be realized through explicit digital signatures on proofs by principals, based on some underlying public key infrastructure.

An affirmation judgment, written $\Psi \vdash M :_K \tau$, means that principal K attests a proof M for τ . As in prior work [7], this may be realized by a digitally signed certificate, although in our case it will be both the proof and the propositions that are signed by a principal K , written as $\langle M:\tau \rangle_K$.

We add the affirmation judgment to the type system of our functional language through the following rule:

$$\frac{\Psi \vdash M : \tau}{\Psi \vdash \langle M:\tau \rangle_K :_K \tau} \text{ (affirms)}$$

The rule states that any principal can affirm the property τ by virtue of a proof M . In the implementation, a process wishing to create such an affirmation must have access to K 's private key so it can sign the pair consisting of the term M and its type τ .

Such an affirmation may seem redundant: after all, the certificate contains the term M which can be type-checked. However, checking a digitally signed certificate may be faster than checking the validity of a proof, so we may speed up the system if we trust K 's signature. More importantly, if we have proof irrelevance, and some parts of M have been erased, then we have in general no way to reconstruct the proofs. In this case we must trust the signing principal K to accept the τ as true, because we cannot be sure if K played by the rules and did indeed have a proof. Therefore, in general, the affirmation of τ by K is *weaker* than the truth of τ , for which we demand explicit evidence. Conversely, when τ is true K can always sign it and be considered as “playing by the rules”, as the inference rule above shows.

Now, to actually be able to use affirmation with the other types in our system, we internalize the judgment as a modal operator. We write $\diamond_K \tau$ for the type that internalizes the judgment $:_K \tau$ (e.g. in the same way that implication internalizes entailment), and $\mathbf{let} \langle x:\tau \rangle_K = M \mathbf{in} N$ for the corresponding destructor.

$$\frac{\Psi \vdash M :_K \tau}{\Psi \vdash M : \diamond_K \tau} \diamond I \quad \frac{\Psi \vdash M : \diamond_K \tau \quad \Psi, x:\tau \vdash N :_K \sigma}{\Psi \vdash \mathbf{let} \langle x:\tau \rangle_K = M \mathbf{in} N :_K \sigma} \diamond E$$

The introduction rule simply internalizes the affirmation judgment. The elimination rule requires the type we are determining to be an affirmation of the

same principal K , adding an assumption of τ – we can assume the property τ from an affirmation made by K only if we are reasoning about affirmations of K . Affirmation in this sense works as a principal-indexed monad. The reduction rules for affirmation are straightforward:

$$\frac{M \longrightarrow M'}{\text{let } \langle x:\tau \rangle_K = M \text{ in } N \longrightarrow \text{let } \langle x:\tau \rangle_K = M' \text{ in } N}$$

$$\frac{}{\text{let } \langle x:\tau \rangle_K = \langle M:\tau \rangle_K \text{ in } N \longrightarrow N\{M/x\}}$$

Returning now to the example in the introduction, the type

$$\text{fpt} : \forall f:\text{nat} \rightarrow \text{nat}. \forall p:\diamond_{\text{verif}}[\Pi x:\text{nat}. f(x) \leq x]. \exists y:\text{nat}. \exists q:[y = f(y)]. \mathbf{1}$$

expresses the type of a server that inputs a function f , accepts a verifier's word that it is decreasing, and returns a fixed point of f to the client. A client that passes the identity function to fpt may be written as follows:

$$\text{fpt}(\lambda x. x). \text{fpt}(\langle [\lambda x. \text{refl } x]:[\Pi x:\text{nat}. f(x) \leq x] \rangle_{\text{verif}}). \text{fpt}(y). \text{fpt}(q). \mathbf{0}.$$

If we want to explicate that the digital signature is supplied by another process associated with access to the private key with the principal verif , we could write a polymorphic process with type

$$\mathbf{v} : \forall \alpha:\text{type}. \forall x:\alpha. \exists y:\diamond_{\text{verif}}[\alpha]. \mathbf{1}$$

which could be

$$\mathbf{v}(\alpha). \mathbf{v}(x). \mathbf{v}(\langle [x]:[\alpha] \rangle_{\text{verif}}) :: \mathbf{v} : \forall \alpha:\text{type}. \forall x:\alpha. \exists y:\diamond_{\text{verif}}[\alpha]. \mathbf{1}$$

The client would then call upon this service and pass the signed certificate (without the proof term) on to fpt .

$$\text{fpt}(\lambda x. x). \mathbf{v}(\text{nat} \rightarrow \text{nat}). \mathbf{v}(\lambda x. x). \mathbf{v}(c). \text{fpt}(c). \text{fpt}(y). \text{fpt}(q). \mathbf{0}.$$

In fact, the implementation of the proof-carrying file system [8] (PCFS) provides such a generic trusted service. In PCFS, the access control policy is presented as a logical theory in the access control logic. Access to a file is granted if a proof of a corresponding access theorem can be constructed with the theory in access control logic and is presented to the file system. Such proofs are generally small when compared to proof-carrying code in the sense of Necula and Lee [13, 12] in which the type safety and memory safety of binary code is certified, but they are still too big to be transmitted and checked every time a file is accessed. Instead, we call upon the trusted verification service to obtain a digitally signed certificate of type $\langle \text{verif}:[\alpha] \rangle$ called a *procap* (for *proven capability*). Procaps are generally very small and fast to verify, leading to an acceptably small overhead when compared to checking access control lists.

As another example, we consider a toy scenario where the customer of a store uses a paying machine to make a purchase. The machine receives the account balance from the bank in order to ensure that the client has enough money for the purchase (realistically the bank would decide if the client has enough money, not the machine, but this suits our illustrative purposes best), if that is not the case it must abort the transaction, otherwise the purchase goes through. We can model this system in our setting by specifying a type for the bank and a type for the machine. We abbreviate $\forall x:\tau. A$ as $\tau \supset A$ and $\exists x:\tau. A$ as $\tau \wedge A$ when x is not free in A :

$$\text{TBank} \triangleq \forall s:\text{string}. \diamond_{\text{M}}[\text{uid}(s)] \supset (\Sigma n:\text{int}. \diamond_{\text{B}}[\text{bal}(s, n)]) \wedge ((\forall m:\text{nat}. \diamond_{\text{M}}[\text{charge}(s, m)] \supset \mathbf{1}) \& \mathbf{1})$$

The type for the bank describes part of the protocol we wish this system to observe: the bank will receive a string and a signed certificate from the paying machine (we use M and B as the principal identifiers for the machine and for the bank, respectively), that asserts the client's identification data. It then sends back the account balance to the machine, attaching a signed certificate that it is indeed the appropriate balance information. It will wait for the decision of the machine to charge the account or not. This is embodied in the use of the additive conjunction ($\&$), that allows the bank to branch on doing nothing ($\mathbf{1}$) or inputting the appropriate charge information. The type for the interface of the machine with the client is as follows:

$$\text{TMClient} \triangleq \forall s:\text{string}. ((\diamond_{\text{M}}[\text{ok}] \wedge \mathbf{1}) \oplus (\diamond_{\text{M}}[\text{nok}] \wedge \mathbf{1}))$$

The client inputs his pin number in the machine and then simply waits for the machine to inform him if the transaction went through or not. A process implementing the bank session (along channel x) is given below:

$$\text{Bank} \triangleq x(s).x(u).x(\text{sign}_2(\text{db_getbal}(s))). \\ x.\text{case}(x(m).x(c).\mathbf{0}; \mathbf{0}) :: x : \text{TBank}$$

We assume a function `db_getbal` that interacts with the bank database to fetch the appropriate balance information and a generic function `sign2` (making type arguments implicit) which is like the earlier generic verifier and uses the bank's private key.

$$\text{db_getbal} : \Pi s:\text{string}. \Sigma n:\text{int}. \text{bal}(s, n) \\ \text{sign}_2 : (\Sigma n:\alpha. \beta) \rightarrow (\Sigma n:\alpha. \diamond_{\text{B}}[\beta])$$

The machine process is typed in an environment containing the bank session along channel x and implementing the interface with the client along channel z , as follows:

$$\text{Machine} \triangleq z(s).x\langle s \rangle. x\langle \langle [\text{gen_uid}]:[\text{uid}(s)] \rangle_{\text{M}} \rangle. x(n).x(b).P_{\text{decide}}$$

We assume a function `gen_uid` of type $\Pi s:\text{string}. \text{uid}(s)$ that takes the clients input and generates the appropriate uid object. We abstract away the details

of deciding if the client has enough money for the purchase in process P_{decide} . This process will simply perform the check and then either terminate and send to the client the `nok` signal, if the client has insufficient balance, or send the charge information to the bank and inform the client that the transaction went through.

As another example, illustrating an application to distributed certified access control, consider the following types

$$\begin{aligned} \text{Server} &\triangleq \forall \text{uid}:\text{string}. (\mathbf{1} \oplus (\diamond_S[\text{perm}(\text{uid})] \wedge \text{Session}(\text{uid}))) \\ \text{Session}(\text{uid}) &\triangleq (\text{productid} \multimap \diamond_S[\text{may}(\text{uid}, \text{buy})] \supset \text{rcp} \otimes \mathbf{1}) \\ &\quad \& \\ &\quad (\text{productid} \multimap \diamond_S[\text{may}(\text{uid}, \text{quote})] \supset \text{ans} \otimes \mathbf{1}) \end{aligned}$$

The type `Server` specifies a server that receives an user id (of type `string`), and then either refuses the session (`1`), or sends back a proof of access permissions granted to the given user, before proceeding. Here, we might have

$$\text{perm}(\text{uid}) \triangleq \text{may}(\text{uid}, \text{quote}) \vee \text{may}(\text{uid}, \text{buy}) \vee \text{may}(\text{uid}, \text{all})$$

In order to access an operation (say `buy`), the client must exhibit a proof of authorization, necessarily computed from the permission proof sent by the server (assuming that only the server can provide such proofs).

The examples above illustrates how proof certificates might be used in our process setting. Recall that, since the proof certificates are always marked as proof irrelevant, we can use the erasure of Section 3 and remove them from the protocol if we so desire.

5 Progress and Preservation

In [18] we established the type safety results of progress and preservation for our dependent session type theory for an unspecified functional language. In fact, we made no mention of when reduction of the functional terms happens. Here, we work under the assumption that processes always evaluate a term to a value *before* communication takes place, and therefore progress and preservation are contingent on the functional layer also being type safe in this sense (which can easily be seen to be the case for the connectives we have presented in this development).

The proof of type preservation then follows the same lines of [18], using a series of reduction lemmas that relate process reductions with parallel composition through an instance of the cut rule and appealing to the type preservation of the functional layer when necessary.

Theorem 4 (Type Preservation). *If $\Psi; \Gamma; \Delta \Rightarrow P :: z : A$ and $P \rightarrow Q$ then $\Psi; \Gamma; \Delta \Rightarrow Q :: z : A$*

Proof. By induction on the typing derivation. When the last rule is an instance of `cut`, we appeal to the reduction lemmas mentioned above (and to type preservation of the functional language when the premises of `cut` are of existential or universal type), which are presented in more detail in [18].

The case for the proof of progress is identical. The result in [18] combined with progress of the functional language establishes progress for the system of this paper. For the purpose of having a self-contained document, we will sketch the proof here as well.

Definition 7 (Live Process).

$$\text{live}(P) \triangleq P \equiv (\nu \bar{n})(Q \mid R) \quad \text{for some } Q, R, \bar{n}$$

where $Q \equiv \pi.Q'$ (π is a non-replicated prefix) or $Q \equiv [x \leftrightarrow y]$

We begin by defining the form of processes that are live. We then establish a contextual progress theorem from which progress follows (Theorem 5 relies on several inversion lemmas that relate types to action labels). Given an action label α , we denote by $s(\alpha)$ the subject of the action α (i.e., the name through which the action takes place).

Theorem 5 (Contextual Progress). *Let $\Psi; \Gamma; \Delta \Rightarrow P :: z : C$. If $\text{live}(P)$ then there is Q such that one of the following holds:*

- (a) $P \rightarrow Q$,
- (b) $P \xrightarrow{\alpha} Q$ for some α where $s(\alpha) \in z, \Gamma, \Delta$ and $s(\alpha) \in \Gamma, \Delta$ if $C = !A$,
- (c) $P \equiv_S [x \leftrightarrow z]$, for some $x \in \Delta$.

Proof. By induction on typing, following [18].

The theorem above states that live processes are either able to reduce outright, are able to take an action α or are equivalent to a channel forwarding (modulo structural congruence extended with a “garbage collection rule” for replicated processes that are no longer usable).

Theorem 6 (Progress). *If $\cdot; \cdot; \cdot \Rightarrow P :: x : \mathbf{1}$, and $\text{live}(P)$, then there exists a process Q such that $P \rightarrow Q$.*

Finally, Theorem 6 follows straightforwardly from Theorem 5 since P can never offer an action α along x , due to its type. Note that $\mathbf{1}$ types not just the inactive process but also all closed processes (i.e. processes that consume all ambient sessions).

6 Concluding remarks

In this paper, we have built upon previous work on dependent session types to account for a flexible notion of proof-carrying code, including digitally signed certificates in lieu of proof objects. To this end, we integrated proof irrelevance and affirmations to the underlying functional language, giving the session type language fine control over which code and data are accompanied by explicit proof, which are supported by digital signature only, and which are trusted outright. We had previously considered proof irrelevance only as a means of optimizing communication in trusted or decidable settings. In a concrete implementation,

the operational semantics must be supported by cryptographic infrastructure to digitally sign propositions and proofs and check such signatures as authentic.

Ours is one amongst several Curry-Howard interpretations connecting linear logic to concurrency. Perhaps closest to session types is work by Mazurak and Zdancewic [11] who develop a Curry-Howard interpretation of classical linear logic as a functional programming language with explicit constructs for concurrency. Their system is based on natural deduction and is substantially different from ours, and they consider neither dependent types nor unrestricted sessions.

The work on Fine [17], F7 [3], and more recently F* [16] has explored the integration of dependent and refinement types in a suite of functional programming languages, with the aim of statically checking assertions about data and state, and enforcing security policies. In our line of research, investigating how closely related mechanisms may be essentially extracted from a Curry-Howard interpretation of fragments of linear and affirmation logics, building on proof irrelevance to express a counterpart of the so-called ghost refinements in F*.

The work on PCML5 [1] has some connection to our own in the sense that they also use affirmation in their framework. PCML5, however, is mostly concerned with authorization and access control, while we employ affirmation as a way of obtaining signatures. Furthermore, PCML5 has no concurrency primitives, while our language consists of a process calculus and thus is inherently concurrent. Nevertheless, it would be quite interesting to explore the possibilities of combining PCML5's notion of authorization with our concurrent setting.

For future work, we wish to explore the applications of proof irrelevance and affirmation in the process layer. Proof irrelevance at the process level is not well understood since it interacts with linearity (if a channel is linear, it must be used, but because it is irrelevant it may not) and communication, considered as an effect. The monadic flavor of affirmation seems to enforce a very strong notion of information flow restrictions on processes, where a process that provides a session of type $\diamond_K A$ is only able to do so using public sessions, or other sessions of type $\diamond_K T$. It would nevertheless be very interesting to investigate how more flexible information flow disciplines might be expressed in our framework, based on modal logic interpretations.

References

1. K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *Proceedings of the 5th Workshop on Types in Language Design and Implementation*, TLDI'10, pages 27–38, New York, NY, USA, 2010. ACM.
2. S. Awodey and A. Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.
3. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffei. Refinement types for secure implementations. In *21st Computer Security Foundations Symposium (CSF'08)*, pages 17–32, Pittsburgh, Pennsylvania, June 2008. IEEE Computer Society.
4. E. Bonelli, A. Compagnoni, and E. L. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. of Func. Prog.*, 15(2):219–247, 2005.

5. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory, CONCUR'10*, pages 222–236. Springer LNCS 6269, 2010.
6. M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: an overview. In *6th International Workshop on Web Services and Formal Methods (WS-FM'09)*, pages 1–28. Springer LNCS 6194, 2010.
7. D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. A linear logic of affirmation and knowledge. In *Proceedings of the 11th European Symposium on Research in Computer Security, ESORICS'06*, pages 297–312. Springer LNCS 4189, Sept. 2006.
8. D. Garg and F. Pfenning. A proof-carrying file system. In D. Evans and G. Vigna, editors, *Proceedings of the 31st Symposium on Security and Privacy (Oakland 2010)*, Berkeley, California, May 2010. IEEE. Extended version available as Technical Report CMU-CS-09-123, June 2009.
9. K. Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory, CONCUR'93*, pages 509–523. Springer LNCS 715, 1993.
10. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems, ESOP'98*, pages 122–138. Springer LNCS 1381, 1998.
11. K. Mazurak and S. Zdancewic. Lollipop: To concurrency from classical linear logic via Curry-Howard and control. In P. Hudak and S. Weirich, editors, *Proceedings of the 15th International Conference on Functional Programming (ICFP'10)*, pages 39–50, Baltimore, Maryland, Sept. 2010. ACM.
12. G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, Jan. 1997. ACM Press.
13. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, Oct. 1996.
14. F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE.
15. A. Salvesen and J. M. Smith. The strength of the subset type in Martin-Löf's type theory. In *3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 384–391, Edinburgh, Scotland, July 1988. IEEE.
16. N. Swamy, J. Checn, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In O. Danvy, editor, *International Conference on Functional Programming (ICFP'11)*, Tokyo, Japan, Sept. 2011. ACM. To appear.
17. N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In A. D. Gordon, editor, *19th European Symposium on Programming (ESOP'10)*, pages 529–549, Paphos, Cyprus, Mar. 2010. Springer LNCS 6012.
18. B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International Symposium on Principles and Practice of Declarative Programming (PPDP'11)*, pages 161–172. ACM, July 2011.