Relating Cryptography and Polymorphism

Benjamin Pierce* Eijiro Sumii†
University of Pennsylvania

July 16, 2000

Abstract

Cryptography is information hiding. Polymorphism is also information hiding. So is cryptography polymorphic? Is polymorphism cryptographic?

To investigate these questions, we define the *cryptographic λ-calculus*, a simply typed λ-calculus with shared-key cryptographic primitives. Although this calculus is simply typed, it is powerful enough to encode recursive functions, recursive types, and dynamic typing. We then develop a theory of relational parametricity for our calculus as Reynolds did for the polymorphic λ-calculus. This theory is useful for proving equivalences in our calculus; for instance, it implies the non-interference property: values encrypted by a key cannot be distinguished from one another by any function ignorant of the key. We close with an encoding of the polymorphic λ-calculus into the cryptographic calculus that uses cryptography to protect type abstraction.

Our results shed a new light upon the relationship between cryptography and polymorphism, and offer a first step toward extending programming idioms based on type abstraction (such as modules and packages) from the civilized world of polymorphism, where only well-typed programs are allowed, to the unstructured world of cryptography, where friendly programs must cohabit with malicious attackers.

1 Introduction

Information hiding is a crucial concept in programming. It is essential not only for modularity but also for security of programs. *Type abstraction* and *encryption* are common approaches to information hiding.

Type abstraction forbids illegal access to secret data by concealing the *type* of the data. For example, consider the following package, which provides a list *ints* of secret integers and a function *isdiv* to test whether one of the secret integers is divisible by another.

$$\begin{array}{l}\textbf{pack int}, \{\mathit{ints} = [1,2,3],\ \mathit{isdiv} = \lambda i.\,\lambda j.\,(i \textbf{ mod } j = 0)\} \\ \textbf{as } \exists\alpha.\{\mathit{ints} : \alpha \textbf{ list},\ \mathit{isdiv} : \alpha \to \alpha \to \textbf{bool}\}\end{array}$$

This achieves the security by concealing the type **int** of the data 1, 2, and 3. The typing rules for existential types [14] guarantee that users of the package must treat the type abstractly.

On the other hand, *encryption* prevents illegal access to secret data by obfuscating the *value* of the data, for example as follows. (The terms $\lfloor - \rfloor_k$ and $\lceil - \rceil_k$ denote encryption and decryption, respectively, under the key k.)

$$\begin{array}{l}\textbf{let } k = \textbf{generate\_fresh\_key}() \textbf{ in} \\ \{\mathit{ints} = [\lfloor 1 \rfloor_k, \lfloor 2 \rfloor_k, \lfloor 3 \rfloor_k],\ \mathit{isdiv} = \lambda x.\,\lambda y.\,(\lceil x \rceil_k \textbf{ mod } \lceil y \rceil_k = 0)\}\end{array}$$

Type abstraction is static and high-level: a potential violation of secrecy causes a type error before execution, as long as the whole program (including the "attacker") abides by the type system. This leads to a powerful method of reasoning about type abstraction, known as *parametricity* [19, 21]. The principle of parametricity assures us that "related" programs are equivalent, in the sense that they are indistinguishable in any well-typed context, where the notion "related" is defined as follows:

* `bcpierce@cis.upenn.edu`

† `sumii@saul.cis.upenn.edu`. Visiting scholar from the University of Tokyo. Research fellow of the Japan Society for the Promotion of Science.

- Two values of a base type b are related if and only if they are equal.
- Two values of a function type $\tau_1 \rightarrow \tau_2$ are related if and only if they map any related arguments to related results.
- Two values of an abstract type α are related if and only if they satisfy the relation $\varphi(\alpha)$, where φ maps each abstract type to a relation between its concrete types.

The power of parametricity comes from the fact that we can take *any* relation as $\varphi(\alpha)$ for each α. For example, for the first program above, we can take $\varphi(\alpha) = \{(1,2),(2,4),(3,6)\}$. Then, by parametricity, this program is equivalent to the same program with $[1,2,3]$ substituted with $[2,4,6]$, because $isdiv(i)(j)$ is equal to $isdiv(i')(j')$ for any $(i,i') \in \varphi(\alpha)$ and $(j,j') \in \varphi(\alpha)$. Therefore, the packaging is secure in the sense that it never leaks the secret data, as long as everybody plays by the rules of the type system. However, type abstraction fails if the scope of a value of an abstract type exceeds the scope of the type checker, e.g. if the value is saved in a file or sent over a network. For instance, the security of the package above is broken if the secret data *ints* is written to a file and read from the file just as a list of integers.

By contrast, encryption is dynamic and low-level: illegal access to secret data causes a decryption failure either in the user code (like $\lceil ints \rceil_{bogus\_key}$) or in the provider code (like $isdiv(\lfloor 0 \rfloor_{bogus\_key})(\lfloor 0 \rfloor_{bogus\_key}))$ during execution. It keeps attackers from breaking the protection with no assumptions about what they do, except for the secrecy of the keys and the mathematical strength of the underlying cryptosystem.

These comparisons between type abstraction and encryption lead us to wonder whether we can establish some more formal relationships between them. In particular:

- Can we adapt the parametricity theory of type abstraction to reason about programs that use encryption for information hiding?
- Can we have our cake and eat it too, first writing programs in the high-level but restrictive setting of type abstraction, and then translating them to the lower-level but more flexible setting of encryption *without losing modularity and security*?

In this paper, we give a full answer to the first question and a partial answer to the second. Specifically, we (i) present the *cryptographic* λ*-calculus*, a simply typed λ-calculus with shared-key cryptographic primitives, and illustrate its expressive power through several examples (Section 2), (ii) establish a theory of relational parametricity à la Reynolds [19] for this calculus (Section 3), and (iii) sketch an encoding of System F [11,18] into the cryptographic λ-calculus (Section 4), using encryption to simulate type abstraction. Our primary contributions are (ii) and (iii). The expressiveness of the calculus is—though useful for modeling malicious attackers—a secondary issue.

2 The Cryptographic λ-Calculus

2.1 Syntax and Semantics

Our calculus is a standard simply typed call-by-value λ-calculus with records, booleans and integers, enriched with simple primitives for shared-key cryptography.

An encryption $\lfloor t_1 \rfloor_{t_2}$ encrypts the plaintext t_1 with the key t_2, and a decryption **let** $\lfloor x \rfloor_{t_1} = t_2$ **in** t_3 **else** t_4 tries to decrypt the ciphertext t_2 with the key t_1. If the decryption succeeds, it binds the plaintext to the variable x and evaluates the body t_3. If the decryption fails, it evaluates the handler t_4. A key generation **new**$\langle \tau, \alpha \rangle$ generates a key to encrypt and decrypt values of the type τ using the password α. A key $\langle \tau, \alpha, n \rangle$ consists of the type τ of the values to encrypt and decrypt, the password α, and the sequence number n. The type is necessary for type soundness: without it, two values of different types could be encrypted using the same key, and then confused when they are decrypted. The sequence number is taken from a global counter, incremented on each key generation. Passwords are used for (statically) tracking knowledge about sets of (dynamically generated) keys—i.e., who knows which keys—when stating security properties of the programs in our calculus, as we will do in Section 3 and 4. The primitive **fail** aborts evaluation.

We assume the following properties of these cryptographic primitives.

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda x.\, t \mid t_1 t_2 \mid \{\tilde{\ell} = \tilde{t}\} \mid \#_\ell(t) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \mid i \mid t_1 - t_2 \mid t > 0 \mid \\
 & & \langle \tau, \alpha, n\rangle \mid \mathbf{new}\langle \tau, \alpha\rangle \mid \lfloor t_1 \rfloor_{t_2} \mid \mathbf{let}\ \lfloor x \rfloor_{t_1} = t_2\ \mathbf{in}\ t_3\ \mathbf{else}\ t_4 \mid \mathbf{fail} \\
\tau & ::= & \tau_1 \to \tau_2 \mid \{\tilde{\ell} : \tilde{\tau}\} \mid \mathbf{bool} \mid \mathbf{int} \mid \langle \tau \rangle \mid \mathbf{bits} \\
v & ::= & \lambda x.\, t \mid \{\tilde{\ell} = \tilde{v}\} \mid \mathbf{true} \mid \mathbf{false} \mid i \mid \langle \tau, \alpha, n\rangle \mid \lfloor v \rfloor_{\langle \tau, \alpha, n\rangle} \\
V & ::= & [v]_n \mid \bot \mid \mathbf{fail} \mid \mathbf{error}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\lambda x.\, t]\!]_n & \stackrel{\text{def}}{=} & [\lambda x.\, t]_n \\
[\![t_1 t_2]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [\lambda x.\, t]_{n_1} \Rightarrow \\
 & & \quad (\texttt{case}\ [\![t_2]\!]_{n_1}\ \texttt{of}\ [v]_{n_2} \Rightarrow [\![t[x := v]]\!]_{n_2} \\
 & & \quad\ \mid V_2 \Rightarrow V_2) \\
 & & \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V_1 \Rightarrow V_1 \\
[\![\{\tilde{\ell} = \tilde{t}\}]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [v_1]_{n_1} \Rightarrow \\
 & & \quad \ldots \\
 & & \qquad (\texttt{case}\ [\![t_m]\!]_{n_{m-1}}\ \texttt{of}\ [v_m]_{n_m} \Rightarrow [\{\tilde{\ell} = \tilde{v}\}]_{n_m} \\
 & & \qquad\ \mid V_m \Rightarrow V_m) \\
 & & \quad \ldots \\
 & & \mid V_1 \Rightarrow V_1 \\
[\![\#_\ell(t)]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t]\!]_n\ \texttt{of}\ [\{\ell = v, \ldots\}]_{n'} \Rightarrow [v]_{n'} \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V \Rightarrow V \\
[\![\mathbf{true}]\!]_n & \stackrel{\text{def}}{=} & [\mathbf{true}]_n \\
[\![\mathbf{false}]\!]_n & \stackrel{\text{def}}{=} & [\mathbf{false}]_n \\
[\![\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [\mathbf{true}]_{n'} \Rightarrow [\![t_2]\!]_{n'} \mid [\mathbf{false}]_{n'} \Rightarrow [\![t_3]\!]_{n'} \\
 & & \hfill \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V \Rightarrow V \\
[\![i]\!]_n & \stackrel{\text{def}}{=} & [i]_n \\
[\![t_1 - t_2]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [i_1]_{n_1} \Rightarrow \\
 & & \quad (\texttt{case}\ [\![t_2]\!]_{n_1}\ \texttt{of}\ [i_2]_{n_2} \Rightarrow [i_1 - i_2]_{n_2} \\
 & & \quad\ \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V_2 \Rightarrow V_2) \\
 & & \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V_1 \Rightarrow V_1 \\
[\![t > 0]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t]\!]_n\ \texttt{of}\ [i]_{n'} \Rightarrow [i > 0]_{n'} \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V \Rightarrow V \\
[\![\langle \tau, \alpha, m\rangle]\!]_n & \stackrel{\text{def}}{=} & [\langle \tau, \alpha, m\rangle]_n \\
[\![\mathbf{new}\langle \tau, \alpha\rangle]\!]_n & \stackrel{\text{def}}{=} & [\langle \tau, \alpha, n+1\rangle]_{n+1} \\
[\![\lfloor t_1 \rfloor_{t_2}]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [v_1]_{n_1} \Rightarrow \\
 & & \quad (\texttt{case}\ [\![t_2]\!]_{n_1}\ \texttt{of}\ [v_2]_{n_2} \Rightarrow [\lfloor v_1 \rfloor_{v_2}]_{n_2} \\
 & & \quad\ \mid V_2 \Rightarrow V_2) \\
 & & \mid V_1 \Rightarrow V_1 \\
[\![\mathbf{let}\ \lfloor x \rfloor_{t_1} = t_2\ \mathbf{in}\ t_3\ \mathbf{else}\ t_4]\!]_n & \stackrel{\text{def}}{=} & \texttt{case}\ [\![t_1]\!]_n\ \texttt{of}\ [v_1]_{n_1} \Rightarrow \\
 & & \quad (\texttt{case}\ [\![t_2]\!]_{n_1}\ \texttt{of}\ [\lfloor v_2 \rfloor_k]_{n_2} \Rightarrow \\
 & & \qquad\qquad \texttt{if}\ v_1 = k\ \texttt{then}\ [\![t_3[x := v_2]]\!]_{n_2}\ \texttt{else}\ [\![t_4]\!]_{n_2} \\
 & & \quad\ \mid [\_]_\_ \Rightarrow \mathbf{error} \mid V_2 \Rightarrow V_2) \\
 & & \mid V_1 \Rightarrow V_1 \\
[\![\mathbf{fail}]\!]_n & \stackrel{\text{def}}{=} & \mathbf{fail}
\end{array}
$$

Figure 1: Syntax and Semantics

$$\Gamma, x:\tau \vdash x:\tau \qquad \Gamma \vdash \mathbf{true}:\mathbf{bool} \qquad \Gamma \vdash \mathbf{false}:\mathbf{bool} \qquad \Gamma \vdash i:\mathbf{int}$$

$$\frac{\Gamma, x:\tau_1 \vdash t:\tau_2}{\Gamma \vdash \lambda x.\, t:\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1:\tau_1 \to \tau_2 \quad \Gamma \vdash t_2:\tau_1}{\Gamma \vdash t_1 t_2:\tau_2} \qquad \frac{\Gamma \vdash \tilde{t}:\tilde{\tau}}{\Gamma \vdash \{\tilde{\ell}=\tilde{t}\}:\{\tilde{\ell}:\tilde{\tau}\}} \qquad \frac{\Gamma \vdash t:\{\ell:\tau,\ldots\}}{\Gamma \vdash \#_\ell(t):\tau}$$

$$\frac{\Gamma \vdash t_1:\mathbf{bool} \quad \Gamma \vdash t_2:\tau \quad \Gamma \vdash t_3:\tau}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3:\tau} \qquad \frac{\Gamma \vdash t_1:\mathbf{int} \quad \Gamma \vdash t_2:\mathbf{int}}{\Gamma \vdash t_1 - t_2:\mathbf{int}} \qquad \frac{\Gamma \vdash t:\mathbf{int}}{\Gamma \vdash t>0:\mathbf{bool}}$$

$$\Gamma \vdash \langle \tau, \alpha, n\rangle : \langle \tau \rangle \qquad \Gamma \vdash \mathbf{new}\langle \tau, \alpha\rangle : \langle \tau \rangle \qquad \Gamma \vdash \mathbf{fail}:\tau$$

$$\frac{\Gamma \vdash t_1:\tau \quad \Gamma \vdash t_2:\langle\tau\rangle}{\Gamma \vdash \lfloor t_1 \rfloor_{t_2}:\mathbf{bits}} \qquad \frac{\Gamma \vdash t_1:\langle\tau_1\rangle \quad \Gamma \vdash t_2:\mathbf{bits} \quad \Gamma, x:\tau_1 \vdash t_3:\tau_2 \quad \Gamma \vdash t_4:\tau_2}{\Gamma \vdash \mathbf{let}\ \lfloor x \rfloor_{t_1} = t_2\ \mathbf{in}\ t_3\ \mathbf{else}\ t_4:\tau_2}$$

Figure 2: Typing Rules

- No ciphertext can be decrypted without knowing the correct key.
- No password can be guessed by conjecture. In practice, this assumption can be fulfilled by choosing "good" passwords for keys.
- Ciphertexts cannot be compared for equality. In practice, this condition can be satisfied by including random nonces in ciphertexts.

Formally, the syntax and semantics of the cryptographic λ-calculus are given in Figure 1. (Throughout this paper, we abbreviate a sequence of the form $X_1, \ldots, X_n$ to a vector of the form $\tilde{X}$ when the number n is obvious.) In order to deal with the impurities such as key generation and failure in a rigorous manner, the evaluation function $[\![t]\!]_n$ takes an initial sequence number n and either diverges (written $\bot$) or returns (i) a value v and a final sequence number n' (written $[v]_{n'}$), (ii) **fail**, or (iii) **error**. Note that **fail** denotes a safe, intentional failure while **error** denotes an unsafe, unintentional error (a run-time type error).

We abbreviate $[\![t]\!]_n = [v]_{n'}$ to $[\![t]\!] = v$ when n and n' are unimportant. For the sake of readability, we also use the syntactic sugarings $(\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2) \stackrel{\text{def}}{=} (\lambda x.\, t_2)t_1$ and $\lceil t_1 \rceil_{t_2} \stackrel{\text{def}}{=} (\mathbf{let}\ \lfloor x \rfloor_{t_2} = t_1\ \mathbf{in}\ x\ \mathbf{else}\ \mathbf{fail})$.

2.2 Type System

The typing rules for the cryptographic λ-calculus are shown in Figure 2. They are standard except for the types given to keys and ciphertexts. The type $\langle\tau\rangle$ describes keys for encrypting and decrypting values of the type τ. The type **bits** ("bit string") includes *all* ciphertexts. The fact that the type of a plaintext is included in the type of the key used to encrypt it, rather than the type of the ciphertext, is the source of the surprising expressive power of our calculus (cf. Section 2.4).

The correctness of the type system with respect to the semantics guarantees that a well-typed program incurs no run-time type error.

Theorem 1 (Type Soundness). Suppose $\tilde{x}:\tilde{\tau} \vdash t:\tau$ and let $t' = t[\tilde{x} := \tilde{v}]$ for any $\emptyset \vdash \tilde{v}:\tilde{\tau}$. Then, for any n, we have $[\![t']\!]_n = \bot$, **fail**, or $[v]_{n'}$ for some $\emptyset \vdash v:\tau$ and $n' \geq n$.

Proof. By induction on the derivation of $\tilde{x}:\tilde{\tau} \vdash t:\tau$. □

2.3 Example

As a first example of programming in our calculus, consider the following situation. There are three entities S, A and B. The entities S and A share a password α, and the entities S and B share another password β. A wants to send an integer i to B in a secure manner, assuming that S is trustworthy.

To express A's transmission of i to S under the password α, we use encryption (under α) and application:

$$A \stackrel{\text{def}}{=} S(\lfloor i \rfloor_{\langle \mathbf{int},\alpha,0\rangle})$$

To capture S's actions—receiving the integer i under the password α, and forwarding it to B under the password β—we use abstraction, decryption (under α), encryption (under β), and application:

$$S \stackrel{\text{def}}{=} \lambda x'.\, \mathbf{let}\ x = \lceil x' \rceil_{\langle \mathbf{int},\alpha,0\rangle}\ \mathbf{in}\ B(\lfloor x \rfloor_{\langle \mathbf{int},\beta,0\rangle})$$

Similarly, B's receipt of i under the password β is expressed using abstraction and decryption:

$$B \stackrel{\text{def}}{=} \lambda y'.\, \mathbf{let}\ y = \lceil y' \rceil_{\langle \mathbf{int},\beta,0\rangle}\ \mathbf{in}\ t(y)$$

The whole program works as follows:

$$\begin{array}{rl}
& [\![\mathbf{let}\ B = \lambda y'.\,(\mathbf{let}\ y = \lceil y' \rceil_{\langle \mathbf{int},\beta,0\rangle}\ \mathbf{in}\ t(y))\ \mathbf{in} \\
& \ \mathbf{let}\ S = \lambda x'.\,(\mathbf{let}\ x = \lceil x' \rceil_{\langle \mathbf{int},\alpha,0\rangle}\ \mathbf{in}\ B(\lfloor x \rfloor_{\langle \mathbf{int},\beta,0\rangle})) \\
& \ S(\lfloor i \rfloor_{\langle \mathbf{int},\alpha,0\rangle})]\!] \\
= & [\![\mathbf{let}\ B = \lambda y'.\,(\mathbf{let}\ y = \lceil y' \rceil_{\langle \mathbf{int},\beta,0\rangle}\ \mathbf{in}\ t(y))\ \mathbf{in} \\
& \ B(\lfloor i \rfloor_{\langle \mathbf{int},\beta,0\rangle})]\!] \\
= & [\![t(i)]\!]
\end{array}$$

The security of each part of this program is guaranteed by the fact that, to any observer who does not know the passwords α and β, the functions S and B are indistinguishable from the function $\lambda\_.\,\mathbf{fail}$, and that the ciphertext $\lfloor i \rfloor_{\langle \mathbf{int},\alpha,0\rangle}$ is indistinguishable from the ciphertext $\lfloor 0 \rfloor_{\langle \mathbf{int},\alpha,0\rangle}$. Details of such reasoning will be presented in Section 3.

2.4 Expressiveness

As we mentioned above, the cryptographic λ-calculus is rather expressive despite the fact that it is simply typed, thanks to the dynamic nature of the type **bits**. In this section, we illustrate its power by showing how to encode several constructs found in other λ-calculi. In these encodings, secrecy plays no role and a single, public password ε suffices for all encryptions. We write $\langle\!\langle \tau \rangle\!\rangle$ for the dummy key $\langle \tau, \varepsilon, 0\rangle$ using this non-secret password.

The correctness of each encoding $\mathcal{E}(\text{—})$ can be stated in the same way.

Theorem 2 (Soundness of Encodings). Suppose $\tilde{x} : \tilde{\tau} \vdash t : \tau$ and let $t' = t[\tilde{x} := \tilde{v}]$ for any $\emptyset \vdash \tilde{v} : \tilde{\tau}$, where t, $\tilde{v}$, τ and $\tilde{\tau}$ are terms, values, and types in the extended language. Then, $\emptyset \vdash \mathcal{E}(t') : \mathcal{E}(\tau)$ and $[\![\mathcal{E}(t')]\!]_n = \mathcal{E}([\![t']\!]_n)$ for any n, where $\mathcal{E}([v]_n) = [\mathcal{E}(v)]_n$, $\mathcal{E}(\bot) = \bot$ and $\mathcal{E}(\mathbf{fail}) = \mathbf{fail}$.

Proof. By induction on the derivation of $\tilde{x} : \tilde{\tau} \vdash t : \tau$ using the lemmas below, given at the end of each section for each extension introduced in that section. $\square$

2.4.1 Disjoint Sums

A disjoint sum $\tau_1 + \tau_2$ is the tagged union of the types τ_1 and τ_2. Its syntax, semantics, and typing rules are given below. (To simplify the encoding, we annotate each **case** expression with its result type τ.)

$$\begin{array}{rcl|l}
t & ::= & \ldots & \mathbf{inl}(t) \mid \mathbf{inr}(t) \mid \mathbf{case}\ t_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_2 \parallel \mathbf{inr}(y) \Rightarrow t_3 : \tau \\
\tau & ::= & \ldots & \tau_1 + \tau_2 \\
v & ::= & \ldots & \mathbf{inl}(v) \mid \mathbf{inr}(v)
\end{array}$$

$$\begin{array}{lcl}
[\![\mathbf{inl}(t)]\!]_n & \stackrel{\text{def}}{=} & \mathtt{case}\ [\![t]\!]_n\ \mathtt{of}\ [v]_{n'} \Rightarrow [\mathbf{inl}(v)]_{n'} \mid V \Rightarrow V \\
[\![\mathbf{inr}(t)]\!]_n & \stackrel{\text{def}}{=} & \mathtt{case}\ [\![t]\!]_n\ \mathtt{of}\ [v]_{n'} \Rightarrow [\mathbf{inr}(v)]_{n'} \mid V \Rightarrow V \\
\multicolumn{3}{l}{[\![\mathbf{case}\ t_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_2 \parallel \mathbf{inr}(y) \Rightarrow t_3 : \tau]\!]_n} \\
& \stackrel{\text{def}}{=} & \mathtt{case}\ [\![t_1]\!]_n\ \mathtt{of}\ [\mathbf{inl}(v)]_{n'} \Rightarrow [\![t_2[x := v]]\!]_{n'} \mid [\mathbf{inr}(v)]_{n'} \Rightarrow [\![t_3[y := v]]\!]_{n'} \\
& & \qquad \mid [_]_{_} \Rightarrow \mathbf{error} \mid V \Rightarrow V
\end{array}$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathbf{inl}(t) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathbf{inr}(t) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash t_2 : \tau \quad \Gamma, y : \tau_2 \vdash t_3 : \tau}{\Gamma \vdash (\mathbf{case}\ t_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_2 \mathbin{[\!]} \mathbf{inr}(y) \Rightarrow t_3 : \tau) : \tau}$$

Since disjoint sums are the dual of products, they can be encoded into the cryptographic calculus by a kind of CPS transformation as follows.

$$\begin{array}{rcl}
\mathcal{E}(\tau_1 + \tau_2) & \stackrel{\text{def}}{=} & (\mathcal{E}(\tau_1) \to \mathbf{bits}) \to (\mathcal{E}(\tau_2) \to \mathbf{bits}) \to \mathbf{bits} \\
\mathcal{E}(\mathbf{inl}(t)) & \stackrel{\text{def}}{=} & \mathbf{let}\ x = \mathcal{E}(t)\ \mathbf{in}\ \lambda f.\, \lambda g.\, f(x) \\
\mathcal{E}(\mathbf{inr}(t)) & \stackrel{\text{def}}{=} & \mathbf{let}\ x = \mathcal{E}(t)\ \mathbf{in}\ \lambda f.\, \lambda g.\, g(x) \\
\mathcal{E}(\mathbf{case}\ t_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_2 \mathbin{[\!]} \mathbf{inr}(y) \Rightarrow t_3 : \tau) & \stackrel{\text{def}}{=} & \lceil \mathcal{E}(t_1)(\lambda x.\, \lfloor \mathcal{E}(t_2) \rfloor_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle})(\lambda y.\, \lfloor \mathcal{E}(t_3) \rfloor_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle}) \rceil_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle}
\end{array}$$

For the other cases, the encoding works in a homomorphic manner, i.e., along the structure of the type or the term. For example, $\mathcal{E}(\tau_1 \to \tau_2) = \mathcal{E}(\tau_1) \to \mathcal{E}(\tau_2)$ and $\mathcal{E}(\lambda x.\, t) = \lambda x.\, \mathcal{E}(t)$.

Lemma 3. $[\![\mathcal{E}(\mathbf{case}\ \mathbf{inl}(v)\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_1 \mathbin{[\!]} \mathbf{inr}(y) \Rightarrow t_2 : \tau)]\!]_n = [\![\mathcal{E}(t_1)[x := \mathcal{E}(v)]]\!]_n$ and $[\![\mathcal{E}(\mathbf{case}\ \mathbf{inr}(v)\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow t_1 \mathbin{[\!]} \mathbf{inr}(y) \Rightarrow t_2 : \tau)]\!]_n = [\![\mathcal{E}(t_2)[y := \mathcal{E}(v)]]\!]_n$ for any v, t_1, t_2, and n.

We can now—if we like—define the type **bool** as the disjoint sum $\{\} + \{\}$ of the empty record $\{\}$. For the sake of presentation, however, we continue to take booleans as primitives.

2.4.2 Recursive Types

A recursive type is a type whose definition mentions itself, like $\mathit{intlist} \stackrel{\text{def}}{=} \{\} + \{\mathit{head} : \mathbf{int}, \mathit{tail} : \mathit{intlist}\}$. Such types are often written as $\mu\alpha.\, \{\} + \{\mathit{head} : \mathbf{int}, \mathit{tail} : \alpha\}$, using the least fixed-point operator μ. In general, a recursive type $\mu\alpha.\, \tau$ satisfies the equation $\mu\alpha.\, \tau \equiv \tau[\alpha := \mu\alpha.\, \tau]$. Its syntax, semantics, and typing rules are given below. Here, we use the simpler *iso-recursive* presentation (rather than the *equi-recursive* one), explicitly inserting the coercions **fold** and **unfold** to convert between values of the folded type $\mu\alpha.\, \tau$ and the unfolded type $\tau[\alpha := \mu\alpha.\, \tau]$. Again, we provide type annotations to assist the encoding.

$$\begin{array}{rcl}
t & ::= & \ldots \mid \mathbf{fold}_{\mu\alpha.\,\tau}(t) \mid \mathbf{unfold}_{\mu\alpha.\,\tau}(t) \\
\tau & ::= & \ldots \mid \alpha \mid \mu\alpha.\,\tau \\
v & ::= & \mathbf{fold}_{\mu\alpha.\,\tau}(v)
\end{array}$$

$$\begin{array}{rcl}
[\![\mathbf{fold}_{\mu\alpha.\,\tau}(t)]\!]_n & \stackrel{\text{def}}{=} & \mathtt{case}\ [\![t]\!]_n\ \mathtt{of}\ [v]_{n'} \Rightarrow [\mathbf{fold}_{\mu\alpha.\,\tau}(v)]_{n'} \mid V \Rightarrow V \\
[\![\mathbf{unfold}_{\mu\alpha.\,\tau}(t)]\!]_n & \stackrel{\text{def}}{=} & \mathtt{case}\ [\![t]\!]_n\ \mathtt{of}\ [\mathbf{fold}_{\mu\alpha.\,\tau}(v)]_{n'} \Rightarrow [v]_{n'} \mid [_]_{_} \Rightarrow \mathbf{error} \mid V \Rightarrow V
\end{array}$$

$$\frac{\Gamma \vdash t : \tau[\alpha := \mu\alpha.\,\tau]}{\Gamma \vdash (\mathbf{fold}_{\mu\alpha.\,\tau}(t)) : \mu\alpha.\,\tau} \qquad \frac{\Gamma \vdash t : \mu\alpha.\,\tau}{\Gamma \vdash \mathbf{unfold}_{\mu\alpha.\,\tau}(t) : \tau[\alpha := \mu\alpha.\,\tau]}$$

Recursive types can be encoded into the cryptographic λ-calculus by taking values of recursive types to be ciphertexts and the folding and unfolding operations to be encryption and decryption.

$$\begin{array}{lcl}
\mathcal{E}(\alpha) & \stackrel{\text{def}}{=} & \mathbf{bits} \\
\mathcal{E}(\mu\alpha.\,\tau) & \stackrel{\text{def}}{=} & \mathbf{bits} \\
\mathcal{E}(\mathbf{fold}_{\mu\alpha.\,\tau}(t)) & \stackrel{\text{def}}{=} & \lfloor t \rfloor_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle} \\
\mathcal{E}(\mathbf{unfold}_{\mu\alpha.\,\tau}(t)) & \stackrel{\text{def}}{=} & \lceil t \rceil_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle}
\end{array}$$

Lemma 4. $[\![\mathcal{E}(\mathbf{unfold}_{\mu\alpha.\,\tau}(\mathbf{fold}_{\mu\alpha.\,\tau}(v)))]\!]_n = [\![\mathcal{E}(v)]\!]_n$ for any v and n.

2.4.3 Recursive Functions

Since recursive types can be encoded, it is straightforward to implement recursive *functions*. Using the encoding of recursive types above, the standard call-by-value fixed-point operator

$$\begin{array}{lcl}\mathbf{fix}_{\tau_1 \to \tau_2} & \stackrel{\text{def}}{=} & \lambda f.\, (\lambda x.\, f(\lambda a.\, \mathbf{unfold}_\tau(x)xa)) \\ & & \quad (\mathbf{fold}_\tau(\lambda x.\, f(\lambda a.\, \mathbf{unfold}_\tau(x)xa)))\end{array}$$

with $\tau = \mu\alpha.\, \alpha \to \tau_1 \to \tau_2$ can be implemented as follows, with $k = \langle\!\langle \mathbf{bits} \to \tau_1 \to \tau_2 \rangle\!\rangle$.

$$\mathcal{E}(\mathbf{fix}_{\tau_1 \to \tau_2}) \;=\; \lambda f.\, (\lambda x.\, f(\lambda a.\, \lceil x \rceil_k xa))\lfloor \lambda x.\, f(\lambda a.\, \lceil x \rceil_k xa)\rfloor_k$$

2.4.4 Dynamic Typing

Dynamic typing [2] is a mechanism for injecting values of different types into a single dynamic type **dyn** in a type-safe manner.

$$\begin{array}{rcl}t & ::= & \ldots \mid \mathbf{indyn}_\tau(t) \mid \mathbf{let}\ \mathbf{indyn}_\tau(x) = t_1\ \mathbf{in}\ t_2\ \mathbf{else}\ t_3 \\ \tau & ::= & \ldots \mid \mathbf{dyn} \\ v & ::= & \mathbf{indyn}_\tau(v)\end{array}$$

$$\begin{array}{l}[\![\mathbf{indyn}_\tau(t)]\!]_n \stackrel{\text{def}}{=} \mathtt{case}\ [\![t]\!]_n\ \mathtt{of}\ [v]_{n'} \Rightarrow [\mathbf{indyn}_\tau(v)]_{n'} \mid V \Rightarrow V \\ [\![\mathbf{let}\ \mathbf{indyn}_\tau(x) = t_1\ \mathbf{in}\ t_2\ \mathbf{else}\ t_3]\!]_n \\ \quad \stackrel{\text{def}}{=} \mathtt{case}\ [\![t_1]\!]_n\ \mathtt{of}\ [\mathbf{indyn}_\tau(v)]_{n'} \Rightarrow [\![t_2[x := v]]\!]_{n'} \mid [\mathbf{indyn}\_(\_)]_{n'} \Rightarrow [\![t_3]\!]_{n'} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid [\_]\_ \Rightarrow \mathbf{error} \mid V \Rightarrow V\end{array}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{indyn}_\tau(t) : \mathbf{dyn}} \qquad \frac{\Gamma \vdash t_1 : \mathbf{dyn} \quad \Gamma, x : \tau \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{let}\ \mathbf{indyn}_\tau(x) = t_1\ \mathbf{in}\ t_2\ \mathbf{else}\ t_3 : \tau}$$

Thanks to the "dynamic" type **bits**, it is also straightforward to implement dynamic typing in the cryptographic calculus.

$$\begin{array}{rcl}\mathcal{E}(\mathbf{dyn}) & \stackrel{\text{def}}{=} & \mathbf{bits} \\ \mathcal{E}(\mathbf{indyn}_\tau(t)) & \stackrel{\text{def}}{=} & \lfloor t \rfloor_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle} \\ \mathcal{E}(\mathbf{let}\ \mathbf{indyn}_\tau(x) = t_1\ \mathbf{in}\ t_2\ \mathbf{else}\ t_3) & \stackrel{\text{def}}{=} & \mathbf{let}\ \lfloor x \rfloor_{\langle\!\langle \mathcal{E}(\tau) \rangle\!\rangle} = \mathcal{E}(t_1)\ \mathbf{in}\ \mathcal{E}(t_2)\ \mathbf{else}\ \mathcal{E}(t_3)\end{array}$$

Lemma 5. $[\![\mathcal{E}(\mathbf{let}\ \mathbf{indyn}_\tau(x) = \mathbf{indyn}_\tau(v)\ \mathbf{in}\ t_1\ \mathbf{else}\ t_2)]\!]_n = [\![\mathcal{E}(t_1)[x := \mathcal{E}(v)]]\!]_n$ and $[\![\mathcal{E}(\mathbf{let}\ \mathbf{indyn}_\tau(x) = v'\ \mathbf{in}\ t_1\ \mathbf{else}\ t_2)]\!]_n = [\![\mathcal{E}(t_2)]\!]_n$ for any n, v and $v' \neq \mathbf{indyn}_\tau(v)$.

3 Parametricity

In this section, we develop a tool for reasoning about information hiding in the cryptographic λ-calculus, adapting the concept of *parametricity* [19, 21] from the polymorphic λ-calculus [11, 18].

Consider the packages $p_1 \stackrel{\text{def}}{=} \{c = \lfloor 1 \rfloor_k,\ f = \lambda x.\, \lceil x \rceil_k + 2\}$ and $p_2 \stackrel{\text{def}}{=} \{c = \lfloor 3 \rfloor_k,\ f = \lambda x.\, \lceil x \rceil_k\}$, where $k = \langle \mathbf{int}, \alpha, n \rangle$ for some α and n, and $t_1 + t_2$ stands for $t_1 - (0 - t_2)$. A function that does not know the password α cannot distinguish p_1 and p_2, because it cannot decrypt the ciphertexts $\#_c(p_1)$ and $\#_c(p_2)$, and the function applications $\#_f(p_1)\#_c(p_1)$ and $\#_f(p_2)\#_c(p_2)$ (which are the only ways to do anything on the ciphertexts $\#_c(p_1)$ and $\#_c(p_2)$) return the same integer 3.

To formalize such reasoning, we are going to define a relation $\mathcal{R}_S(\tau)\varphi$ between values of the type τ, where S is a set of "secret" passwords $\{\tilde{\alpha}\}$. Intuitively, two values v_1 and v_2 will be related by $\mathcal{R}_S(\tau)\varphi$ if and only if

- v_1 and v_2 do not reveal any information about the secret passwords S, and

$$
\begin{array}{rcl}
\mathcal{R}_S(\tau \to \tau')\varphi & \stackrel{\text{def}}{=} & \{(\lambda x.\, t_1, \lambda x.\, t_2) \mid \forall (v_1, v_2) \in \mathcal{R}_S(\tau)\varphi.\, (t_1[x := v_1], t_2[x := v_2]) \in \mathcal{R}'_S(\tau')\varphi \\
\mathcal{R}_S(\{\tilde{\ell} : \tilde{\tau}\})\varphi & \stackrel{\text{def}}{=} & \{(\{\tilde{\ell} = \tilde{v}_1\}, \{\tilde{\ell} = \tilde{v}_2\}) \mid (\tilde{v}_1, \tilde{v}_2) \in \mathcal{R}_S(\tilde{\tau})\varphi\} \\
\mathcal{R}_S(\mathbf{bool})\varphi & \stackrel{\text{def}}{=} & \{(b, b) \mid b = \mathbf{true}, \mathbf{false}\} \\
\mathcal{R}_S(\mathbf{int})\varphi & \stackrel{\text{def}}{=} & \{(i, i) \mid i = \ldots, -2, -1, 0, 1, 2, \ldots\} \\
\mathcal{R}_S(\langle\tau\rangle)\varphi & \stackrel{\text{def}}{=} & \{(\langle\tau, \alpha, n\rangle, \langle\tau, \alpha, n\rangle) \mid \alpha \notin S\} \\
\mathcal{R}_S(\mathbf{bits})\varphi & \stackrel{\text{def}}{=} & \{(\lfloor v_1 \rfloor_{\langle\tau,\alpha,n\rangle}, \lfloor v_2 \rfloor_{\langle\tau,\alpha,n\rangle}) \mid \emptyset \vdash v_1 : \tau \wedge \emptyset \vdash v_2 : \tau \wedge \alpha \notin S \wedge (v_1, v_2) \in \mathcal{R}_S(\tau)\varphi \\
 & \cup & \{(\lfloor v_1 \rfloor_{\langle\tau_1,\alpha_1,n_1\rangle}, \lfloor v_2 \rfloor_{\langle\tau_2,\alpha_2,n_2\rangle}) \mid \emptyset \vdash v_1 : \tau_1 \wedge \emptyset \vdash v_2 : \tau_2 \wedge \alpha_1, \alpha_2 \in S \wedge \\
 & & \qquad (v_1, v_2) \in \varphi(\langle\tau_1, \alpha_1, n_1\rangle, \langle\tau_2, \alpha_2, n_2\rangle)\} \\
\mathcal{R}'_S(\tau)\varphi & \stackrel{\text{def}}{=} & \{(t_1, t_2) \mid \emptyset \vdash t_1 : \tau \wedge \emptyset \vdash t_2 : \tau \wedge \\
 & & \quad (\forall n.\, [\![t_1]\!]_n = [\![t_2]\!]_n = \bot \vee \\
 & & \quad \forall n.\, [\![t_1]\!]_n = [\![t_2]\!]_n = \mathbf{fail} \vee \\
 & & \quad \forall n.\, \exists v_1.\, \exists v_2.\, \exists n' \geq n.\, [\![t_1]\!]_n = [v_1]_{n'} \wedge [\![t_2]\!]_n = [v_2]_{n'} \wedge (v_1, v_2) \in \mathcal{R}_S(\tau)\varphi)\}
\end{array}
$$

Figure 3: Logical Relation

- they are indistinguishable by any function that does not know the secret passwords.

The first condition is essential because if the values themselves leak the secret passwords, then the observer can break the abstraction without knowing any secret password in advance. For example, suppose $k = \langle \mathbf{int}, \alpha, n\rangle$ for some $\alpha \in S$ and n. Although the ciphertexts $\lfloor 4 \rfloor_k$ and $\lfloor 5 \rfloor_k$ are indistinguishable to any observer who does not know the secret password α, the records $\{c = \lfloor 4 \rfloor_k,\ k = k\}$ and $\{c = \lfloor 5 \rfloor_k,\ k = k\}$ are distinguishable to the observer $\lambda r.\, \lceil \#_c(r) \rceil_{\#_k(r)}$.

Just as we can take any relation for values of an abstract type in the parametricity theory of type abstraction (cf. Section 1), we can take any relation for ciphertexts encrypted with secret passwords, because they are anyway indistinguishable to the observer. The map φ in the relation $\mathcal{R}_S(\tau)\varphi$ provides such a relation between values of the type τ_1 and values of the type τ_2, for each pair of keys $\langle\tau_1, \alpha_1, n_1\rangle$ and $\langle\tau_2, \alpha_2, n_2\rangle$ with $\alpha_1, \alpha_2 \in S$.

Formally, the relation $\mathcal{R}_S(\tau)\varphi$ is defined in Figure 3 by induction on the type τ. It is a standard logical relation except for the following points.

- Since the cryptographic calculus is impure (because of key generation, failure and divergence), we actually define two relations $\mathcal{R}_S(\tau)\varphi$ and $\mathcal{R}'_S(\tau)\varphi$, the former over *values* (i.e., results of evaluation) and the latter over *computations* (i.e., closed terms with potential effects).

- Two keys are *not* related if their passwords are secret, because they would leak the secret and break the first condition as we saw above.

- Two ciphertexts encrypted with a non-secret key are related if and only if the plaintexts are related.

- Two ciphertexts encrypted with secret keys k_1 and k_2 are related if and only if they satisfy the relation $\varphi(k_1, k_2)$.

For instance, in the first example above, let us take $S = \{\alpha\}$, $\varphi(k, k) = \{(1, 3)\}$ and $\varphi(k_1, k_2) = \emptyset$ when $k_1 \neq k$ or $k_2 \neq k$. Then, $(\lfloor 1 \rfloor_k, \lfloor 3 \rfloor_k) \in \mathcal{R}_S(\mathbf{bits})\varphi$ because $(1, 3) \in \varphi(k, k)$ and $\alpha \in S$, and $(\lambda x.\, \lceil x \rceil_k + 2, \lambda x.\, \lceil x \rceil_k) \in \mathcal{R}_S(\mathbf{bits} \to \mathbf{int})\varphi$ because $[\![(\lambda x.\, \lceil x \rceil_k + 2)(\lfloor 1 \rfloor_k)]\!] = [\![(\lambda x.\, \lceil x \rceil_k)(\lfloor 3 \rfloor_k)]\!] = 3$. Therefore, $(p_1, p_2) \in \mathcal{R}_S(\{c : \mathbf{bits},\ f : \mathbf{bits} \to \mathbf{int}\})\varphi$.

Thus, we expect that the packages p_1 and p_2 are observationally equivalent, in the sense that a function that does not know the password α cannot distinguish them. That is, $g(p_1) = g(p_2)$ for any $g : \{c : \mathbf{bits},\ f : \mathbf{bits} \to \mathbf{int}\} \to \mathbf{bool}$ where α does not appear in g. In order to prove this fact, it suffices to show $(g, g) \in \mathcal{R}_S(\{c : \mathbf{bits},\ f : \mathbf{bits} \to \mathbf{int}\} \to \mathbf{bool})\varphi$, which is a special case of the general theorem below.

Definition 6. *knows*$(t) = \{\alpha \mid \alpha \text{ appears in } t\}$.

Theorem 7 (Parametricity). Suppose $\tilde{x} : \tilde{\tau} \vdash t : \tau$. Then, $(t[\tilde{x} := \tilde{v}_1], t[\tilde{x} := \tilde{v}_2]) \in \mathcal{R}'_S(\tau)\varphi$ for any $(\tilde{v}_1, \tilde{v}_2) \in \mathcal{R}_S(\tilde{\tau})\varphi$, provided that $S \cap \mathit{knows}(t) = \emptyset$ and $\mathit{dom}(\varphi) = \{(\langle \tau'_1, \alpha_1, n_1\rangle, \langle \tau'_2, \alpha_2, n_2\rangle) \mid \alpha_1, \alpha_2 \in S\}$.

Proof. By induction on the derivation of $\tilde{x} : \tilde{\tau} \vdash t : \tau$. The least trivial case is when $t = (\mathbf{let}\ \lfloor x \rfloor_{t_a} = t_b\ \mathbf{in}\ t_c\ \mathbf{else}\ t_d)$ for some t_a, t_b, t_c and t_d. In that case, $(t_a[\tilde{x} := \tilde{v}_1], t_a[\tilde{x} := \tilde{v}_2]) \in \mathcal{R}'_S(\langle \tau' \rangle)\varphi$ for some τ', by the induction hypothesis. Then, the definition of the logical relation for key types tells us that, for any n_1, we have $[\![t_a[\tilde{x} := \tilde{v}_1]]\!]_{n_1} = [\![t_a[\tilde{x} := \tilde{v}_2]]\!]_{n_1} = [\langle \tau', \alpha, n\rangle]_{n'_1}$ for some $\alpha \notin S$, n, and $n'_1 \geq n_1$, unless the evaluations fail or diverge. On the other hand, $(t_b[\tilde{x} := \tilde{v}_1], t_b[\tilde{x} := \tilde{v}_2]) \in \mathcal{R}'_S(\mathbf{bits})\varphi$ by the induction hypothesis. Therefore, for any n_2, we have either $[\![t_b[\tilde{x} := \tilde{v}_1]]\!]_{n_2} = [\lfloor v'_1 \rfloor_{\langle \tau', \alpha, n\rangle}]_{n'_2}$ and $[\![t_b[\tilde{x} := \tilde{v}_2]]\!]_{n_2} = [\lfloor v'_2 \rfloor_{\langle \tau', \alpha, n\rangle}]_{n'_2}$ for some $(v'_1, v'_2) \in \mathcal{R}_S(\tau')\varphi$ and $n'_2 \geq n_2$, or $[\![t_b[\tilde{x} := \tilde{v}_1]]\!]_{n_2} = [\lfloor v'_1 \rfloor_k]_{n'_2}$ and $[\![t_b[\tilde{x} := \tilde{v}_2]]\!]_{n_2} = [\lfloor v'_2 \rfloor_k]_{n'_2}$ for some v'_1, v'_2, $k \neq \langle \tau', \alpha, n\rangle$, and $n'_2 \geq n_2$, unless the evaluations fail or diverge. In the former case, $(t_c[\tilde{x} := \tilde{v}_1][x := v'_1], t_c[\tilde{x} := \tilde{v}_1][x := v'_2]) \in \mathcal{R}'_S(\tau)\varphi$ by the induction hypothesis. In the latter case, $(t_d[\tilde{x} := \tilde{v}_1], t_d[\tilde{x} := \tilde{v}_2]) \in \mathcal{R}'_S(\tau)\varphi$ by the induction hypothesis, which completes the proof. $\square$

From the parametricity theorem above, the non-interference property below follows immediately. It states that values encrypted with secret passwords cannot be distinguished from one another by any function ignorant of the secret passwords.

Corollary 8 (Non-Interference). Suppose $\emptyset \vdash t : \mathbf{bits} \to \mathbf{bool}$ and $\alpha_1, \alpha_2 \notin \mathit{knows}(t)$. Then, $[\![t(\lfloor v_1 \rfloor_{k_1})]\!]_n = [\![t(\lfloor v_2 \rfloor_{k_2})]\!]_n$ for any n, $k_1 = \langle \tau_1, \alpha_1, n_1\rangle$, $k_2 = \langle \tau_2, \alpha_2, n_2\rangle$, $\emptyset \vdash v_1 : \tau_1$, and $\emptyset \vdash v_2 : \tau_2$.

Proof. Let $S = \{\alpha_1, \alpha_2\}$, $\varphi(k_1, k_2) = \{(v_1, v_2)\}$, and $\varphi(k'_1, k'_2) = \emptyset$ whenever $(k'_1, k'_2) \neq (k_1, k_2)$. Then, $(t, t) \in \mathcal{R}_S(\mathbf{bits} \to \mathbf{bool})\varphi$ by Theorem 7 and $(\lfloor v_1 \rfloor_{k_1}, \lfloor v_2 \rfloor_{k_2}) \in \mathcal{R}_S(\mathbf{bits})\varphi$ by the definition of $\mathcal{R}_S(\mathbf{bits})\varphi$. Therefore, by the definition of $\mathcal{R}_S(\mathbf{bits} \to \mathbf{bool})\varphi$, we have $(t(\lfloor v_1 \rfloor_{k_1}), t(\lfloor v_2 \rfloor_{k_2})) \in \mathcal{R}'_S(\mathbf{bool})\varphi$. Thus, by the definition of $\mathcal{R}'_S(\mathbf{bool})\varphi$, $[\![t(\lfloor v_1 \rfloor_{k_1})]\!]_n = [\![t(\lfloor v_2 \rfloor_{k_2})]\!]_n$. $\square$

More generally, it can be proved as follows that related terms are indistinguishable to any observer who does not know the secret passwords.

Definition 9. Two closed terms t_1 and t_2 of type τ are said to be *observationally equivalent* with respect to a set S of secret passwords, written $t_1 \overset{\text{obs}}{=}_S t_2 : \tau$, if and only if $[\![t(t_1)]\!]_n = [\![t(t_2)]\!]_n$ for any n and t with $\emptyset \vdash t : \tau \to \mathbf{bool}$ and $S \cap \mathit{knows}(t) = \emptyset$.

Corollary 10 (Soundness of Parametricity). $(t_1, t_2) \in \mathcal{R}'_S(\tau)\varphi$ implies $t_1 \overset{\text{obs}}{=}_S t_2 : \tau$ for any t_1, t_2, τ, S, and φ with $\mathit{dom}(\varphi) = \{(\langle \tau_1, \alpha_1, n_1\rangle, \langle \tau_2, \alpha_2, n_2\rangle) \mid \alpha_1, \alpha_2 \in S\}$.

Proof. Immediate from Theorem 7 and the definition of the logical relation for arrow types. $\square$

Example 11. Suppose $p_1 \overset{\text{def}}{=} \{c = \lfloor 1 \rfloor_k,\ f = \lambda x.\, \lceil x \rceil_k + 2\}$ and $p_2 \overset{\text{def}}{=} \{c = \lfloor 3 \rfloor_k,\ f = \lambda x.\, \lceil x \rceil_k\}$ where $k = \langle \mathbf{int}, \alpha, n\rangle$ for some α and n. Then, as we saw above, $(p_1, p_2) \in \mathcal{R}_S(\tau)\varphi$ where $\tau = \{c : \mathbf{bits},\ f : \mathbf{bits} \to \mathbf{int}\}$, $S = \{\alpha\}$, $\varphi(k, k) = \{(1, 3)\}$, and $\varphi(k_1, k_2) = \emptyset$ for $(k_1, k_2) \neq (k, k)$. Therefore, $p_1 \overset{\text{obs}}{=}_S p_2 : \tau$ by Corollary 10.

Example 12. Suppose $k = \langle \mathbf{int}, \alpha, n\rangle$, $\alpha \in S$, and $(4, 5) \in \varphi(k, k)$. Then, $(\lfloor 4 \rfloor_k, \lfloor 5 \rfloor_k) \in \mathcal{R}_S(\mathbf{bits})\varphi$ but $(k, k) \notin \mathcal{R}_S(\langle \mathbf{int} \rangle)\varphi$, so $(\{c = \lfloor 4 \rfloor_k,\ k = k\}, \{c = \lfloor 5 \rfloor_k,\ k = k\}) \notin \mathcal{R}_S(\{c : \mathbf{bits},\ k : \langle \mathbf{int} \rangle\})\varphi$ as we saw above. Indeed, $\lfloor 4 \rfloor_k \overset{\text{obs}}{=}_S \lfloor 5 \rfloor_k$ but $\{c = \lfloor 4 \rfloor_k,\ k = k\} \overset{\text{obs}}{\neq}_S \{c = \lfloor 5 \rfloor_k,\ k = k\}$.

The next example suggests how we can enforce the abstractness of polymorphic types (e.g., a value of the type $\forall \alpha.\, \alpha \to \alpha$ is the universal identity function) by wrapping a term with encryption and decryption. Section 4 develops this intuition in greater detail.

Example 13. Suppose $k = \langle \tau, \alpha, n\rangle$ and $\emptyset \vdash f : \mathbf{bits} \to \mathbf{bits}$, with $\alpha \notin \mathit{knows}(f)$. Then, $[\![\lceil f \lfloor v \rfloor_k \rceil_k]\!] = v$ for any $\emptyset \vdash v : \tau$, provided that the evaluation does not fail or diverge.

Proof. Let $S = \{\alpha\}$, $\varphi(k, k) = \{(v, v)\}$, and $\varphi(k_1, k_2) = \emptyset$ when $(k_1, k_2) \neq (k, k)$. Then, by the definition of the logical relation for ciphertexts, $(\lfloor v \rfloor_k, \lfloor v \rfloor_k) \in \mathcal{R}_S(\mathbf{bits})\varphi$. At the same time, by Theorem 7, $(f, f) \in \mathcal{R}_S(\mathbf{bits} \to \mathbf{bits})\varphi$. Therefore, by the definition of the logical relation for arrow types, $([\![f \lfloor v \rfloor_k]\!], [\![f \lfloor v \rfloor_k]\!]) \in \mathcal{R}_S(\mathbf{bits})\varphi$. On the other hand, since the decryption $\lceil f \lfloor v \rfloor_k \rceil_k$ does not fail, $[\![f \lfloor v \rfloor_k]\!] = \lfloor v' \rfloor_k$ for some v'. Thus, we have $(\lfloor v' \rfloor_k, \lfloor v' \rfloor_k) \in \mathcal{R}_S(\mathbf{bits})\varphi$, which implies $v = v'$ by the definition of the logical relation for ciphertexts. Therefore, $[\![\lceil f \lfloor v \rfloor_k \rceil_k]\!] = [\![\lceil \lfloor v' \rfloor_k \rceil_k]\!] = v' = v$. $\square$

4 Encoding Type Abstraction

In the previous section, we adapted the parametricity theory of type abstraction for the cryptographic λ-calculus, providing a high-level tool for reasoning about "cryptographic abstraction." An even more attractive scenario would be one where we could simply write our programs in terms of polymorphism (for simplicity), and then translate uses of polymorphism into encryption to allow our programs to operate in a more flexible setting (including networking, etc.) with a broader range of attackers *while preserving the abstraction guarantees of the original program*. In this section, we propose such an encoding of the polymorphic λ-calculus into the cryptographic calculus. We begin in Section 4.1 with a naive version that illustrates most of the ideas but does not protect itself from malicious functions that may be passed in from the environment. In Section 4.2 we show how to add such protection to achieve what we conjecture to be a fully abstract encoding.

The source language of the encoding is the standard polymorphic λ-calculus (System F) [11, 18] with existential types, with two minor changes:

- Each type application $(e_1 : \forall\alpha.\, T)[e_2]$ is annotated with the type $\forall\alpha.\, T$ of the term e_1, so that the encoding can be defined on typed terms instead of their typing derivation trees. (This type annotation is elided when it is unimportant.)
- The primitive **fail** is included for the sake of convenience, since it is also present in the target language. Alternatively, we could exclude it from the source language and weaken the conjecture of full abstraction to "full abstraction modulo failure," i.e., the encodings of equivalent terms are equivalent as long as both evaluations succeed.

In the encoding, we assume that all bound type variables are pairwise distinct by α-conversion.

4.1 Naive Encoding

Before presenting an encoding that preserves type abstraction, let us consider an encoding that just preserves well-typedness of terms. This is already non-trivial, since we are translating a polymorphic language into a monomorphic one. Consider, for example, the term

$$\begin{array}{l}\mathbf{let}\ id : \forall\alpha.\, \alpha \to \alpha = \Lambda\alpha.\, \lambda x : \alpha.\, x\\ \mathbf{in}\ \{i = id[\mathbf{int}]123,\ b = id[\mathbf{bool}]\mathbf{true}\}\end{array}$$

in the source language. If we only erase the types, we get an ill-typed term:

$$\mathbf{let}\ id = \lambda\_.\, \lambda x.\, x\ \mathbf{in}\ \{i = id\{\}123,\ b = id\{\}\mathbf{true}\}$$

Instead, as we did in the encoding of recursive types, we encode each type variable α as the ciphertext type **bits** and insert encryption and decryption when we use values of universal types (and, dually, when we make values of existential types). In the present example, the universal type $\forall\alpha.\, \alpha \to \alpha$ is encoded as the type $\{\} \to \mathbf{bits} \to \mathbf{bits}$, the type abstraction $\Lambda\alpha.\, \lambda x : \alpha.\, x$ is encoded as the function $\lambda\_.\, \lambda x.\, x$ of type $\{\} \to \mathbf{bits} \to \mathbf{bits}$, and the instantiations $id[\mathbf{int}]$ and $id[\mathbf{bool}]$ are encoded as functions $\lambda z.\, \lceil id\{\}\lfloor z\rfloor_{\langle\!\langle \mathbf{int}\rangle\!\rangle}\rceil_{\langle\!\langle \mathbf{int}\rangle\!\rangle}$ and $\lambda z.\, \lceil id\{\}\lfloor z\rfloor_{\langle\!\langle \mathbf{bool}\rangle\!\rangle}\rceil_{\langle\!\langle \mathbf{bool}\rangle\!\rangle}$ of types $\mathbf{int} \to \mathbf{int}$ and $\mathbf{bool} \to \mathbf{bool}$, respectively. These translations together yield the following well-typed term:

$$\begin{array}{l}\mathbf{let}\ id = \lambda\_.\, \lambda x.\, x\\ \mathbf{in}\ \{i = (\lambda z.\, \lceil id\{\}\lfloor z\rfloor_{\langle\!\langle \mathbf{int}\rangle\!\rangle}\rceil_{\langle\!\langle \mathbf{int}\rangle\!\rangle})123,\\ \qquad b = (\lambda z.\, \lceil id\{\}\lfloor z\rfloor_{\langle\!\langle \mathbf{bool}\rangle\!\rangle}\rceil_{\langle\!\langle \mathbf{bool}\rangle\!\rangle})\mathbf{true}\}\end{array}$$

Formally, the encoding $\mathcal{E}(e)$ is given in Figure 4. The coercions $\mathcal{C}^{\pm}_{\alpha}(x, k, T)$ insert the encryption and decryption using the key k for the type variable α by η-expanding the variable x along the type T. The coercion $\mathcal{C}^{-}$ instantiates an abstract value to a concrete one, while the other coercion $\mathcal{C}^{+}$ generalizes a concrete value to an abstract one. In general, provided that $k = \langle \mathcal{E}(T'), \beta, n\rangle$ for some β and n, they convert between values of the types $\mathcal{E}(T)$ and $\mathcal{E}(T[\alpha := T'])$. For example, provided that $T = \alpha \to \alpha$ and $T' = \mathbf{int}$, they convert functions of the type $\mathbf{bits} \to \mathbf{bits}$ to functions of the type $\mathbf{int} \to \mathbf{int}$ and vice versa.

$$
\begin{aligned}
\mathcal{E}(\Lambda\alpha.\, e) &\stackrel{\text{def}}{=} \lambda\_.\, \mathcal{E}(e) \\
\mathcal{E}((e : \forall\alpha.\, T_1)[T_2]) &\stackrel{\text{def}}{=} \textbf{let } x = \mathcal{E}(e)\{\} \textbf{ in } \mathcal{C}^-_\alpha(x, \langle\!\langle \mathcal{E}(T_2) \rangle\!\rangle, T_1) \\
\mathcal{E}(\textbf{pack } T_1, e \textbf{ as } \exists\alpha.\, T_2) &\stackrel{\text{def}}{=} \textbf{let } x = \mathcal{E}(e) \textbf{ in } \mathcal{C}^+_\alpha(x, \langle\!\langle \mathcal{E}(T_1) \rangle\!\rangle, T_2) \\
\mathcal{E}(\textbf{open } e_1 \textbf{ as } \alpha, x \textbf{ in } e_2) &\stackrel{\text{def}}{=} \textbf{let } x = \mathcal{E}(e_1) \textbf{ in } \mathcal{E}(e_2) \\
\mathcal{E}(x) &\stackrel{\text{def}}{=} x \\
\mathcal{E}(\lambda x : T.\, e) &\stackrel{\text{def}}{=} \lambda x.\, \mathcal{E}(e) \\
\mathcal{E}(e_1 e_2) &\stackrel{\text{def}}{=} \mathcal{E}(e_1)\mathcal{E}(e_2) \\
\mathcal{E}(\{\tilde{\ell} = \tilde{e}\}) &\stackrel{\text{def}}{=} \{\tilde{\ell} = \mathcal{E}(\tilde{e})\} \\
\mathcal{E}(\#_\ell(e)) &\stackrel{\text{def}}{=} \#_\ell(\mathcal{E}(e)) \\
\mathcal{E}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) &\stackrel{\text{def}}{=} \textbf{if } \mathcal{E}(e_1) \textbf{ then } \mathcal{E}(e_2) \textbf{ else } \mathcal{E}(e_3) \\
\mathcal{E}(\textbf{true}) &\stackrel{\text{def}}{=} \textbf{true} \\
\mathcal{E}(\textbf{false}) &\stackrel{\text{def}}{=} \textbf{false} \\
\mathcal{E}(i) &\stackrel{\text{def}}{=} i \\
\mathcal{E}(e_1 - e_2) &\stackrel{\text{def}}{=} \mathcal{E}(e_1) - \mathcal{E}(e_2) \\
\mathcal{E}(e > 0) &\stackrel{\text{def}}{=} \mathcal{E}(e) > 0 \\
\mathcal{E}(\textbf{fail}) &\stackrel{\text{def}}{=} \textbf{fail}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}(\alpha) &\stackrel{\text{def}}{=} \textbf{bits} \\
\mathcal{E}(\forall\alpha.\, T) &\stackrel{\text{def}}{=} \{\} \to \mathcal{E}(T) \\
\mathcal{E}(\exists\alpha.\, T) &\stackrel{\text{def}}{=} \mathcal{E}(T) \\
\mathcal{E}(T_1 \to T_2) &\stackrel{\text{def}}{=} \mathcal{E}(T_1) \to \mathcal{E}(T_2) \\
\mathcal{E}(\{\tilde{\ell} : \tilde{T}\}) &\stackrel{\text{def}}{=} \{\tilde{\ell} : \mathcal{E}(\tilde{T})\} \\
\mathcal{E}(\textbf{bool}) &\stackrel{\text{def}}{=} \textbf{bool} \\
\mathcal{E}(\textbf{int}) &\stackrel{\text{def}}{=} \textbf{int}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}^+_\alpha(x, k, \alpha) &\stackrel{\text{def}}{=} \lfloor x \rfloor_k \\
\mathcal{C}^-_\alpha(x, k, \alpha) &\stackrel{\text{def}}{=} \lceil x \rceil_k \\
\mathcal{C}^\pm_\alpha(x, k, \beta) &\stackrel{\text{def}}{=} x \qquad\qquad (\text{if } \alpha \neq \beta) \\
\mathcal{C}^\pm_\alpha(x, k, \forall\beta.\, T) &\stackrel{\text{def}}{=} \lambda\_.\, \textbf{let } y = x\{\} \textbf{ in } \mathcal{C}^\pm_\alpha(y, k, T) \\
\mathcal{C}^\pm_\alpha(x, k, \exists\beta.\, T) &\stackrel{\text{def}}{=} \mathcal{C}^\pm_\alpha(x, k, T) \\
\mathcal{C}^\pm_\alpha(x, k, T_1 \to T_2) &\stackrel{\text{def}}{=} \lambda y.\, \textbf{let } a = \mathcal{C}^\mp_\alpha(y, k, T_1) \textbf{ in} \\
&\qquad\quad \textbf{let } r = xa \textbf{ in } \mathcal{C}^\pm_\alpha(r, k, T_2) \\
\mathcal{C}^\pm_\alpha(x, k, \{\tilde{\ell} : \tilde{T}\}) &\stackrel{\text{def}}{=} \textbf{let } \tilde{y} = \#_{\tilde{\ell}}(x) \textbf{ in } \{\tilde{\ell} = \mathcal{C}^\pm_\alpha(\tilde{y}, \}, \tilde{T})k \\
\mathcal{C}^\pm_\alpha(x, k, \textbf{bool}) &\stackrel{\text{def}}{=} x \\
\mathcal{C}^\pm_\alpha(x, k, \textbf{int}) &\stackrel{\text{def}}{=} x
\end{aligned}
$$

Figure 4: Naive Encoding

$$
\begin{array}{rcl}
\mathcal{T}(e:T) & \stackrel{\text{def}}{=} & \textbf{let } x = \mathcal{E}(e) \textbf{ in } \mathcal{G}^{+}(x,T) \\
\\
\mathcal{G}^{\pm}(x,\alpha) & \stackrel{\text{def}}{=} & x \\
\mathcal{G}^{+}(x,\forall\alpha.\,T) & \stackrel{\text{def}}{=} & \lambda\_.\,\textbf{let } y = x\{\} \textbf{ in } \mathcal{G}^{+}(x,T) \\
\mathcal{G}^{-}(x,\forall\alpha.\,T) & \stackrel{\text{def}}{=} & \lambda\_.\,\textbf{let } k = \textbf{new}\langle\textbf{bits},\gamma\rangle \textbf{ in} \\
 & & \quad \textbf{let } y = x\{\} \textbf{ in} \\
 & & \quad \textbf{let } z = \mathcal{C}^{-}_{\alpha}(y,k,T) \textbf{ in } \mathcal{G}^{-}(z,T) \\
\mathcal{G}^{+}(x,\exists\alpha.\,T) & \stackrel{\text{def}}{=} & \textbf{let } k = \textbf{new}\langle\textbf{bits},\gamma\rangle \textbf{ in} \\
 & & \textbf{let } z = \mathcal{C}^{+}_{\alpha}(x,k,T) \textbf{ in } \mathcal{G}^{+}(z,T) \\
\mathcal{G}^{-}(x,\exists\alpha.\,T) & \stackrel{\text{def}}{=} & \mathcal{G}^{-}(x,T) \\
\mathcal{G}^{\pm}(x,T_1 \to T_2) & \stackrel{\text{def}}{=} & \lambda y.\,\textbf{let } a = \mathcal{G}^{\mp}(y,T_1) \textbf{ in} \\
 & & \quad \textbf{let } r = xa \textbf{ in } \mathcal{G}^{\pm}(r,T_2) \\
\mathcal{G}^{\pm}(x,\{\tilde{\ell}:\tilde{T}\}) & \stackrel{\text{def}}{=} & \textbf{let } \tilde{y} = \#_{\tilde{\ell}}(x) \textbf{ in } \{\tilde{\ell} = \mathcal{G}^{\pm}(\tilde{y},\tilde{T})\} \\
\mathcal{G}^{\pm}(x,\textbf{bool}) & \stackrel{\text{def}}{=} & x \\
\mathcal{G}^{\pm}(x,\textbf{int}) & \stackrel{\text{def}}{=} & x
\end{array}
$$

Figure 5: Better Encoding

4.2 Better Encoding

Although the encoding above suffices as far as only "good citizens" translated from the source language are concerned, it does not protect type abstractions from malicious attackers hand-coded in the target language, because not all terms of type $\mathcal{E}(T)$ (such as $\{\} \to \textbf{bits} \to \textbf{bits}$) in the target language behave like encodings of terms of the type T (such as $\forall\alpha.\,\alpha \to \alpha$) in the source language. For example, consider the terms $e_1 \stackrel{\text{def}}{=} \lambda f : (\forall\alpha.\,\alpha \to \alpha).\,f$ and

$$
\begin{array}{rcl}
e_2 & \stackrel{\text{def}}{=} & \lambda f : (\forall\alpha.\,\alpha \to \alpha).\,\Lambda\beta.\,\textbf{let } f' : \beta \to \beta = f[\beta] \textbf{ in} \\
 & & \qquad\qquad\qquad\qquad \lambda x : \beta.\,\textbf{let } y : \beta = f'x \textbf{ in } x
\end{array}
$$

of type $T \stackrel{\text{def}}{=} (\forall\alpha.\,\alpha \to \alpha) \to (\forall\alpha.\,\alpha \to \alpha)$. In the source language, they are indistinguishable because $f[\tau]v = v$ for any $f : \forall\alpha.\,\alpha \to \alpha$ and $v : \tau$ unless the evaluation fails. On the other hand, their naive encodings $t_1 = \lambda f.\,f$ and

$$
t_2 \quad = \quad \lambda f.\,\lambda\_.\,\textbf{let } f' = f\{\} \textbf{ in } \lambda x.\,\textbf{let } \_ = f'x \textbf{ in } x
$$

of type $\tau = (\{\} \to \textbf{bits} \to \textbf{bits}) \to (\{\} \to \textbf{bits} \to \textbf{bits})$ are distinguishable, say, to the observer

$$
\lambda F.\,\lceil F(\lambda\_.\,\lambda x.\,\lfloor \textbf{true} \rfloor_{\langle\!\langle \textbf{bool} \rangle\!\rangle})\{\}\lfloor \textbf{false} \rfloor_{\langle\!\langle \textbf{bool} \rangle\!\rangle} \rceil_{\langle\!\langle \textbf{bool} \rangle\!\rangle}
$$

of type $\tau \to \textbf{bool}$.

We can improve this naive encoding by wrapping all values coming from the outside of the encoding with encryption and decryption, and thereby forcing them to behave as they should, so that the decryption fails if they do not. For instance, in the example above, we can wrap the incoming function f like

$$
\begin{array}{rcl}
t_2' & = & \lambda f.\,\textbf{let } f = \lambda\_.\,(\textbf{let } k = \textbf{new}\langle\textbf{bits},\gamma\rangle \textbf{ in} \\
 & & \qquad\qquad\qquad\quad \textbf{let } f' = f\{\} \textbf{ in } \lambda x.\,\lceil f' \lfloor x \rfloor_k \rceil_k) \\
 & & \quad \textbf{in } \ldots
\end{array}
$$

so that it does nothing but behaving as a function of type $\forall\alpha.\,\alpha \to \alpha$.

Formally, the better encoding $\mathcal{T}(e : T)$ is given in Figure 5. To protect type abstractions, the guard $\mathcal{G}^{\pm}(x, T)$ wraps incoming values of universal types (and, dually, outgoing values of existential types) by η-expanding the variable x along the type T and inserting encryption and decryption using a single secret password γ. Note that it never changes the types of the values that it wraps—for instance, in the example above, both t_2 and t'_2 have type τ—since it encrypts values of the type **bits**. This "double encryption" is not a waste of work: the purpose of this second encryption is protection, while the first encryption (in the naive encoding) was just for well-typedness.

The correctness of the encoding can be formalized as follows.

Definition 14. $(e_1 \overset{\text{obs}}{=} e_2 : T) \overset{\text{def}}{\iff} (\emptyset \vdash e_1 : T) \wedge (\emptyset \vdash e_2 : T) \wedge [\![e(e_1)]\!] = [\![e(e_2)]\!]$ for any $\emptyset \vdash e : T \to \mathbf{bool}$.

Conjecture 15 (Full Abstraction). For any e_1 and e_2, $e_1 \overset{\text{obs}}{=} e_2 : T$ if and only if $\mathcal{T}(e_1 : T) \overset{\text{obs}}{=}_{\{\gamma\}} \mathcal{T}(e_2 : T)$.

Establishing the truth (or falsity) of this conjecture is left for future work. It is not straightforward for at least the following reasons:

- As far as we are aware, it remains an open problem to find *any* fully-abstract model of the polymorphic λ-calculus. Indeed, this was a hard problem even for the simply typed λ-calculus (with recursion) [12, pp. 212–215].

- The target language (which is Turing complete) is much more expressive than the source language (which is strongly normalizing). Consider, for instance, two functions F_1 and F_2 of type $(\mathbf{int} \to \mathbf{int}) \to \mathbf{bool}$. Even if $[\![\mathcal{E}(F_1)f]\!] = \mathbf{true}$ and $[\![\mathcal{E}(F_2)f]\!] = \mathbf{false}$ for some function $f : \mathbf{int} \to \mathbf{int}$ in the target language, we cannot conclude that F_1 and F_2 are inequivalent in the source language, because the function f might be undefinable. Neither does this imply that the encoding is *not* fully-abstract, because the "test" for the function f might also be undefinable.

5 Related Work

The history of abstraction is as long as the history of programming languages. More than 27 years ago, Morris [15] informally presented the idea of protecting type abstractions by "sealing" values of the abstract types. Our encoding of polymorphism into cryptography in Section 4 can be regarded as a formalization of his idea.

Parametricity was first formalized by Reynolds [19] and further popularized by Wadler [21]. Although it was developed primarily in the setting of universal types, it can be applied easily to the setting of abstract types, because they can be interpreted as existential types [14], which can be encoded by universal types. Alternatively, parametricity can be extended directly to existential types [17].

Abadi, Fournet, and Gonthier [4–6] have studied process calculi with security features such as authentication and private communication, and presented fully-abstract encodings of these systems into process calculi with cryptographic primitives (adaptations of the spi calculus [1, 7] and the join calculus [9, 10]). As their work itself shows, however, direct reasoning about such calculi is somewhat difficult.

Zdancewic, Grossman, and Morrisett [22] developed a variant of the simply typed λ-calculus that preserves abstractions between multiple principals by distinguishing them through the syntax and the type system. They also gave a syntactic proof of (an adaptation of) parametricity in their calculus, using the small-step reduction semantics of their calculus. It would be interesting to see whether and how their syntactic approach applies to our cryptographic framework and vice versa.

Heintze and Riecke [13] proposed a typed λ-calculus with information flow control, and proved the non-interference property (i.e., a value of high security does not leak to any context of low security) by using a logical relation like parametricity as we did in Section 3. While their calculus is high-level and uses static typing to keep secrecy, our calculus is low-level and uses dynamic encryption to hide information. As a result, it is forbidden in their scheme to, say, communicate a secret value over a public network, because the potential illegal access leads to a static type error.

Lillibridge and Harper [personal communication, July 2000] have independently developed a *typed seal calculus* quite similar to our cryptographic λ-calculus, and studied encodings of various constructs such as recursive types and polymorphism. The main thrust of their work is finding various alternative implementations of the seal primitives in terms of lower-level mechanisms such as exceptions and references, rather than establishing techniques for reasoning about properties of programs in their calculus or protecting translations of high-level programs from low-level attackers.

6 Conclusion

We have (i) developed a λ-calculus with cryptography, (ii) adapted the parametricity theory of polymorphism to this calculus, and (iii) proposed an encoding—conjectured to be fully abstract—of type abstraction into encryption. Besides proving full abstraction, the following issues are interesting for further research.

Dynamic Typing and Polymorphism It is known that dynamic typing does not to coexist well with polymorphism, because it breaks the abstraction of polymorphism [3]. By implementing polymorphism with cryptography, however, it would be possible to allow the coexistence without losing the abstraction.

Aliasing and Information Hiding It is also known that language features such as references cells and communication channels break type abstractions in surprising ways via aliasing [16]. It would be interesting to see how such constructs affect the information hiding that cryptography achieves.

Public-Key Cryptography Although we have focused on shared-key cryptography so far, it is natural to wonder what *public-key* cryptography corresponds to in term of the type theory. A possible answer to this question might involve *bounded quantification* [8]. Conventional upper-bounded quantification corresponds roughly to digital signatures or *trademarks* [15]. If a package exports an abstract type $\alpha \leq \tau$, then clients of the package can read α-values without help from the package, but only authorized code inside the package can create elements of α. Conversely, a lower-bounded abstract type $\alpha \geq \tau$ roughly corresponds to a public key for encrypting values of type τ, because those who do not know the concrete type (i.e., the private key) can make a value of the type α (i.e., a ciphertext) but cannot use it. For example, using a "public key" $\alpha \geq \mathbf{int}$, we can program transmission of a secret datum $i : \mathbf{int}$ to a "remote host" $f : \alpha \to \tau'$ over a "public network" $g : \alpha \to \alpha$, as $f(g(\mathbf{coerce}_\alpha(i)))$. Because the type α is abstract, the function g *must* return the argument as it is, or just diverge and discard it (which a real network might also do).

Encoding Encryption into Type Abstraction As the reverse of the encoding in Section 4, it also seems possible to implement encryption using type abstraction with some extensions such as references and dynamic typing. The following program in Standard ML illustrates this intuition, implementing public-key cryptography in terms of existential polymorphism. The **exn** type is used for dynamic typing and functor application is used for fresh key generation.

```
type ciphertext = exn
exception DecryptionFailure

signature CRYPT =
sig type plaintext
    val encrypt : plaintext → ciphertext
    val decrypt : ciphertext → plaintext
end

functor Crypt (type plaintext) : CRYPT =
struct type plaintext = plaintext
       exception C of plaintext
       fun encrypt x = C x
       fun decrypt (C x) = x
```

```
            | decrypt _ = raise DecryptionFailure
    end
```

In this program, the functor *Crypt* takes the place of the fresh key generator, and the functions *encrypt* and *decrypt* play the role of the encryption and decryption keys, respectively. For example, it works as follows.

```
- structure IntCrypt1 = Crypt (type plaintext = int);
structure IntCrypt1 : CRYPT
- structure IntCrypt2 = Crypt (type plaintext = int);
structure IntCrypt2 : CRYPT
- val c = IntCrypt1.encrypt 123;
val c = C(-) : ciphertext
- IntCrypt1.decrypt c;
val it = 123 : IntCrypt1.plaintext
- IntCrypt2.decrypt c;
uncaught exception DecryptionFailure
```

Since it is possible to interpret the module system of Standard ML in terms of universal and existential polymorphism [20], it might be possible to encode cryptography into polymorphism (with some extensions) by generalizing this example.

Acknowledgements

Some of the ideas presented here originated in a coffee-break conversation with Greg Morrisett. Martín Abadi offered useful pointers to related work. Mark Lillibridge helped us understand the relation with his typed seal calculus and gave us valuable comments on a late draft of the manuscript.

This work was supported by the National Science Foundation under NSF Career grant CCR-9701826 and by the Japan Society for the Promotion of Science.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[3] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.

[4] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. Available at `http://pauillac.inria.fr/~fournet/papers/secure-implementation.ps.gz`, 1999. Preliminary papers on this work appeared in *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 74–88 and *1999 IEEE Symposium on Security and Privacy*, pp. 105–116.

[5] Martín Abadi, Cédric Fournet, and Georges Gonthier. A top-down look at a secure message. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1999.

[6] Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, 2000.

[7] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[9] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[10] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR '96*, volume 1119, pages 406–421, 1996.

[11] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[12] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[13] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[14] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[15] James H. Morris Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[16] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000.

[17] Andrew M. Pitts. Existential types: Logical relations and operational equivalence. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 1998.

[18] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[19] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, pages 513–523, 1983.

[20] Claudio V. Russo. *Types For Modules.* PhD thesis, University of Edinburgh, 1998. Available at `http://www.dcs.ed.ac.uk/home/cvr/ECS-LFCS-98-389.html`.

[21] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[22] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 197–207, 1999.