Theories of Information Hiding in Lambda-Calculus: Logical Relations and Bisimulations for Encryption and Type Abstraction

Eijiro Sumii

Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo on September 10, 2004
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Abstract

Two methods are studied for proving equivalence of programs involving two forms of information hiding. The proof methods are *logical relations* and *bisimulations*; the forms of information hiding are *type abstraction* and *perfect encryption* (also known as *dynamic sealing*). Our thesis is that these theories are useful for reasoning about programs involving information hiding. We prove it through soundness and completeness theorems as well as examples including abstract data structures and cryptographic protocols.

Type abstraction is the most foundational form of information hiding in programming languages. Logical relations are the primary method for reasoning about type abstraction, which is often called *relational parametricity* or *representation independence*. Encryption is another foundational form of information hiding that is predominant in communication systems. In fact, an encryption-like primitive is useful for abstraction in programming languages as well, where it is called dynamic sealing. Given this intuitive connection between two forms of information hiding in computer software, it is natural to wonder whether we can establish more formal connections between them and transfer reasoning techniques from one to the other. We give affirmative answers to these questions.

First, we adapt the theory of relational parametricity from type abstraction to dynamic sealing by defining a simply typed λ-calculus extended with primitives for dynamic sealing, named $\lambda^{\rightarrow}_{\texttt{seal}}$, and its logical relations. As an illustrative application of this theory, we prove security properties of cryptographic protocols by means of careful encodings into the calculus.

Second, we develop a theory of bisimulations for dynamic sealing. Unlike logical relations, it extends with no difficulty to richer languages with recursive functions, recursive types, or even no types at all. We illustrate its power by defining untyped λ-calculus with dynamic sealing, which is named $\lambda_{\texttt{seal}}$, and proving the equivalence of different implementations of abstract data structures as well as the correctness of a complex cryptographic protocol.

Third, we "feed back" this theory of bisimulations from dynamic sealing to type abstraction to obtain the first sound, complete, and yet elementary theory of type abstraction in λ-calculus with full universal, existential, and recursive types (called $\lambda^{\forall\exists}_{\mu}$). Our examples include abstract data types, generative functors, and an object encoding.

We conclude with a conjecture of full abstraction for type-directed translation from type abstraction into dynamic sealing and with a possible direction for application to language environments supporting statically checked, dynamically checked, and unchecked programs at the same time without sacrificing abstraction.

Acknowledgments

I would like to thank many people who helped my study and life in general. First of all, I appreciate all the support from my family—in particular, my wife. She came with me all the way from Tokyo to Philadelphia almost as soon as we married. My parents, with my brother, brought me up teaching how to think in Western (or even American) ways, perhaps because they all have once lived in New York and have also studied (and worked) in international circumstances.

Most of the research in this thesis was carried out while I was working with Benjamin Pierce, once as a visiting student and again as a research associate, in the University of Pennsylvania. He first mentioned the intuitive connection between encryption and type abstraction—which he himself noticed during an informal conversation with Greg Morrisett—and has been an excellent adviser throughout the whole work. Especially, he spent significant energy to improve various aspects of my technical writings.

Prof. Akinori Yonezawa and Prof. Naoki Kobayashi have also been great mentors since I was a student, both undergraduate and graduate, and a research associate in the University of Tokyo. Colleagues in their groups have stimulated me as well in positive and important ways.

Members of the Logic and Computation Seminar and the Security Seminar in the University of Pennsylvania—Andre Scedrov, in particular—gave me useful suggestions concerning the application of this work to cryptographic protocols. People in the Programming Language Club (also in the University of Pennsylvania), including Steve Zdancewic and Stephanie Weirich, offered various opportunities to learn and think about relevant topics in programming languages and their theories, such as security typing and intensional polymorphism.

Communications with Martín Abadi, Karl Crary, Andy Gordon, Bob Harper, Mark Lillibridge, and Andrew Pitts helped me understand relationships of my work to theirs. Anonymous referees for the Theoretical Computer Science journal, Journal of Computer Security, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, IEEE Computer Security Foundations Workshop, and JSSST Workshop on Programming and Programming Languages (in Japan) gave insightful comments to all the technical details.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Information hiding or *abstraction* is of central importance when building large systems—not only computer software, but other complex human artifacts as well. By dividing a system into separate modules with restrictive interfaces and concealing their internal implementations, architects avoid considering all of the details at once and can often reuse similar (or even identical) modules for constructing another system.

Type abstraction is the primary method of information hiding in programming languages. It conceals the concrete type of data structures and thereby prevents direct access to their internal representations. Consider, for example, the following two implementations of complex numbers in ML-like pseudo-code

```
module Cartesian implements Complex
  abstype t = real * real
  make_complex(x, y) = (x, y)
  multiply((x1, y1), (x2, y2)) =
    (x1 * x2 - y1 * y2, x1 * y2 + y1 * x2)
  get_re_and_im(x, y) = (x, y)
end
```

and

```
module Polar implements Complex
  abstype t = real * real
  make_complex(x, y) = (sqrt(x * x + y * y), atan2(y, x))
  multiply((r1, a1), (r2, a2)) = (r1 * r2, a1 + a2)
  get_re_and_im(r, a) = (r * cos(a), r * sin(a))
end
```

where the interface `Complex` is defined as:

```
interface Complex
  abstype t
  make_complex : real * real -> t
```

```
  multiply : t * t -> t
  get_re_and_im : t -> real * real
end
```

Since the concrete type `real * real` of complex numbers is abstracted just as `t`, users of the modules cannot analyze values of this type by themselves. Thus, the difference of implementations is unobservable as far as the computed result of an entire program is concerned (ignoring lower-level phenomena such as elapsed time or consumed power, which are outside the scope of this study). Type abstraction is well-understood both in practice—via languages such as CLU [Liskov 1981], Alphard [Shaw 1981], Ada [Taft and Duff 1995], Modula [Wirth 1989], and ML [Milner, Tofte, Harper, and MacQueen 1997]—and in theory—via universal [Reynolds 1983; Reynolds 1974; Girard 1972] and existential [Mitchell and Plotkin 1988; Mitchell 1991] quantifications over types. It also forms the basis of more sophisticated mechanisms such as objects [Pierce and Turner 1993; Bruce, Cardelli, and Pierce 1999] and advanced module systems [Russo 1998; Dreyer, Crary, and Harper 2003]. Pierce [2002], Mitchell [1996], and Pierce [2005] offer comprehensive introductions to the relevant background.

Encryption is another form of information hiding that is predominant in communication systems. Primarily, it protects secrecy of data by obfuscating it when communicated on non-secret channels. Encryption is dynamic and more robust than type abstraction—in that it manipulates the real representation of secret data "on the fly" instead of checking their users *a priori*, and therefore works in uncooperative environments such as open networks as well (to the extent of the mathematical strength of the cryptographic scheme). For example, if the previous modules were implemented by means of symmetric encryption like

```
module Cartesian2
  secret_key k
  make_complex(x, y) = <encrypt (x, y) under k>
  multiply(c1, c2) =
    let (x1, y1) = <decrypt c1 under k> in
    let (x2, y2) = <decrypt c2 under k> in
    <encrypt (x1 * x2 - y1 * y2, x1 * y2 + y1 * x2) under k>
  get_re_and_im(c) = <decrypt c under k>
end
```

and

```
module Polar2
  secret_key k
  make_complex(x, y) =
    <encrypt (sqrt(x * x + y * y), atan2(y, x)) under k>
  multiply(c1, c2) =
    let (r1, a1) = <decrypt c1 under k> in
    let (r2, a2) = <decrypt c2 under k> in
    <encrypt (r1 * r2, a1 + a2) under k>
  get_re_and_im(c) =
    let (r, a) = <decrypt c under k> in
    (r * cos(a), r * sin(a))
end
```

then they could preserve abstraction even against untyped, arbitrary attackers in addition to well-typed, legitimate users. Such protection would remain effective in particular when abstract data structures are exported to the "outside" of the runtime system of a programming language, e.g., when they are stored in a file system or sent over an open network (as in Leifer, Peskine, Sewell, and Wansbrough [2003]).

In fact, the idea of using encryption-like operations for data abstraction is even older than type abstraction. Morris [1973a, 1973b] called this operation *sealing*: a fresh, secret seal is generated for each kind of abstract data structure, which are sealed when going out of a module and unsealed when coming back into it. Type abstraction evolved from a static variant of sealing [Morris 1973b; Liskov 1993]. The original, dynamic version of sealing coincides with *perfect encryption*—an ideal scheme where all the operations are symbolic (i.e., keys are atomic names and ciphertexts are formal terms) and decryption succeeds only if the keys match. We consider this form of encryption and regard it as synonymous with dynamic sealing throughout this thesis.

1.2 The Problem and Our Contributions

Although many studies have been carried out for cryptography and cryptographic protocols (Stinson [1995], Schneier [1996], and Menezes, van Oorshot, and Vanstone [1996] are standard textbooks on this area), little research has been carried out for "abstraction by encryption" in the context of programming languages and their theories. For instance, how can we prove that the two modules `Cartesian2` and `Polar2` above are indeed compatible? In general, can we develop a theory of abstraction by encryption—perhaps like *relational parametricity* [Reynolds 1983; Wadler 1989] (whose dual is sometimes called *representation independence* [Mitchell 1991]) for type abstraction? Can we use this theory to reason about cryptographic protocols as well? Conversely, can we learn from the theory and feed it back to type abstraction?

This thesis gives affirmative answers to these questions. To be specific, our contributions are threefold:

1. We define $\lambda^{\rightarrow}_{\mathtt{seal}}$, a simply typed call-by-value λ-calculus extended with primitives for perfect encryption/dynamic sealing and study *logical relations* as a theory of relational parametricity in this calculus. We prove them sound with respect to contextual equivalence and use them to show secrecy properties of cryptographic protocols via the notion of *non-interference*.

2. We develop a *bisimulation* proof method for an untyped version of the calculus ($\lambda_{\mathtt{seal}}$), prove it sound and complete with respect to contextual equivalence, and demonstrate its use through examples including abstract data structures and a cryptographic protocol.

3. We adapt this bisimulation method to λ-calculus with full universal, existential, and recursive types ($\lambda^{\forall\exists}_{\mu}$) and again prove its soundness and completeness. This is the first sound, complete, and yet elementary proof method for a language with general recursion and type abstraction.

1.3 Logical Relations for Type Abstraction: Review

As a preparation before outlining the technical developments of our main contributions, let us review the theory of relational parametricity or representation independence, i.e., logical relations for type abstraction. In general, logical relations are relations over some semantics of terms in typed λ-calculus, defined by induction on their types. See Mitchell [1996, Chapter 8] for more details on this general case. In this thesis, we focus our attention on operational semantics and logical relations over a term model—so-called *syntactic logical relations* [Pitts 1998; Pitts 2000; Birkedal and Harper 1999; Crary and Harper 2000]—to keep our development elementary.

Informally, syntactic logical relations for λ-calculus with abstract types are defined as follows by induction on the types of related terms (as is usual for any logical relations).

- Constants, such as integers and real numbers, are related if they are equal.
- Tuples are related if their elements are pairwise related.
- Functions are related if they map related arguments to related results.
- Data of abstract type α are related if they are related by $\varphi(\alpha)$, where φ is a *relation environment* mapping each abstract type to the relation between its implementations.

Then, the so-called "fundamental property" or "basic lemma" of logical relations guarantees that related terms are *contextually equivalent*, i.e., that they exhibit the same observable behavior under any context within the language.

For example, let us show the compatibility of the two modules `Cartesian` and `Polar` above by using these logical relations. We take

$$\varphi(\mathtt{t}) \quad = \quad \{((x,y),(r,\theta)) \mid x = r\cos\theta \text{ and } y = r\sin\theta \text{ for } r \geq 0\}.$$

That is, a pair (x,y) implementing `Cartesian.t` is related to another pair (r,θ) implementing `Polar.t` if $x = r\cos\theta$ and $y = r\cos\theta$. Then, we should check the functions `Cartesian.make_complex`, `Cartesian.multiply`, and `Cartesian.get_re_and_im` are logically related to the corresponding functions `Polar.make_complex`, `Polar.multiply`, and `Polar.get_re_and_im` at types $\mathtt{real} \times \mathtt{real} \to \mathtt{t}$, $\mathtt{t} \times \mathtt{t} \to \mathtt{t}$, and $\mathtt{t} \to \mathtt{real} \times \mathtt{real}$, respectively. This is straightforward by the definitions above along with high-school mathematics about complex numbers: for instance, to see that `Cartesian.make_complex` and `Polar.make_complex` are logically related, it suffices to check that they map arguments related at type $\mathtt{real} \times \mathtt{real}$ to results related at type `t`, which follows since $\mathtt{Cartesian.make\_complex}(x,y)$ and $\mathtt{Polar.make\_complex}(x,y)$ are related by $\varphi(\mathtt{t})$ for any real numbers x and y; similarly, `Cartesian.multiply` and `Polar.multiply` are logically related at type $\mathtt{t} \times \mathtt{t} \to \mathtt{t}$ since $\mathtt{Cartesian.multiply}(c_1,c_2)$ and $\mathtt{Polar.multiply}(c'_1,c'_2)$ are related by $\varphi(\mathtt{t})$ for any c_1 and c'_1 related by $\varphi(\mathtt{t})$ and for any c_2 and c'_2 again related by $\varphi(\mathtt{t})$.

1.4 Logical Relations for Perfect Encryption

The modules `Cartesian` and `Polar` above conceal the *type* of complex numbers and are indeed shown to be compatible by means of logical relations for type abstraction. How, then, can we prove the compatibility of `Cartesian2` and `Polar2`, which encrypt the *values* of complex numbers?

Our basic idea is simple: instead of relating terms of abstract type, let $\varphi(\mathtt{k})$ relate terms encrypted under key `k`. That is:

- Ciphertexts encrypted under secret key k are related if the plaintexts are related by $\varphi(\mathtt{k})$.

For example, to show the compatibility of `Cartesian2` and `Polar2`, let us take

$$\varphi(\mathtt{k}) \quad = \quad \{((x,y),(r,\theta)) \mid x = r\cos\theta \text{ and } y = r\sin\theta \text{ for } r \geq 0\}.$$

Then, all the functions in `Cartesian2` are logically related to those in `Polar2` at appropriate types—as was the case for `Cartesian` and `Polar`—and guaranteed by the fundamental property to be contextually equivalent.

A few tricks are necessary in addition to the basic idea above. First, in order for the fundamental property to hold, it turns out that we *must* treat non-secret keys as well as secret keys. Technically, this is because every well-typed term in the language has to be related to itself. Thus, we extend the definitions as follows. (The secrecy of a key is defined by whether it belongs to the domain of φ.)

- Ciphertexts (of type $\mathtt{bits}[\tau]$) encrypted under non-secret key k are related if the plaintexts (of type τ) are related at their type.

- A key is related to itself if it is non-secret.

The second definition reflects the intuition that terms are logically related when they can be revealed to a context *and yet* cannot be distinguished.

Second, in order to treat fresh key generation—which is essential both in cryptographic protocols and in module systems (e.g., for generative functors [Milner, Tofte, Harper, and MacQueen 1997])—the notion of relation environments needs drastic revision. Consider, for instance, the following two programs

```
let x = <generate_fresh_key> in
(<encrypt x under k>, <encrypt 123 under x>)
```

and

```
let x = <generate_fresh_key> in
(<encrypt x under k>, <encrypt 456 under x>)
```

which should be contextually equivalent provided that `k` is secret (and therefore `x` is kept secret as well). How can we specify $\varphi(\mathtt{k})$? Since the key bound to `x` does not exist before executing the programs and varies for each execution, it makes no sense—either intuitively or technically—to define $\varphi(\mathtt{k}) = \{(\mathtt{x},\mathtt{x})\}$ or $\varphi(\mathtt{x}) = \{(123, 456)\}$. We want to have something like

$$\varphi(\mathtt{k}) \quad = \quad \{(\mathtt{x},\mathtt{x}) \mid \varphi(\mathtt{x}) = \{(123,456)\}\}$$

though this definition is still ill-formed since it refers to φ while *defining* it. We solve this problem (in Section 2.6.2) by parametrizing relation environments and requiring some form of monotonicity on them. Roughly, each relation environment φ is parameterized by relation environments ψ "in the future," as in

$$\varphi^{\psi}(\mathtt{k}) \quad = \quad \{(\mathtt{x},\mathtt{x}) \mid \psi^{\psi}(\mathtt{x}) = \{(123,456)\}\}.$$

Naturally, the "type" (at the meta level) of such relation environments becomes recursive—as can be seen in the self application ψ^{ψ} above—but they can be constructed inductively from constant functions, just as functions of recursive type $\mu\alpha.(\alpha \to \mathtt{int})$ can be. Thanks to the monotonicity of parametrized relation environments, this does not lead to a paradox. Based on this idea, we prove the secrecy property of a complex cryptographic protocol: the Needham-Schroeder-Lowe public-key protocol.

1.5 Bisimulations for Perfect Encryption

Although logical relations are straightforward as far as simply typed languages with no recursion are concerned, their extensions to more expressive languages are harder. Recursive functions cause a difficulty in the proof of the fundamental property: roughly, this proof requires a form of induction on the syntax of terms parametrized by the values of their free variables, which does not work for a recursive function $f(x) = M$ since—by definition—its body M refers to f itself. Recursive *types* invalidate even the *definition* of logical relations, which proceeds by induction on the types of the terms to be related. Although it is possible to solve these problems by requiring a continuity property called *admissibility* [Wadler 1989; Birkedal and Harper 1999; Crary and Harper 2000], this property must be taken into account for every use of logical relations by the user (in various forms such as $\top\top$-closure [Pitts 2000; Pitts 1998; Abadi 2000]) and cannot be proved once and for all in their meta theory. This is rather unfortunate because types constrain the "attackers" (contexts) as well as the "defenders" (terms) in a language, while it is not always realistic to assign types—in particular simple ones—to all the participants in a system, some of which may even be malicious.

We circumvent these difficulties of logical relations by considering *bisimulations* instead. To illustrate the power of this approach, we choose an extreme case—that is, untyped λ-calculus (with perfect encryption).

Basically, bisimulations are relations defined by co-induction with a set of rules that exclude inequivalent terms. For example, bisimulations in untyped λ-calculus (with no encryption)—called applicative bisimulations [Abramsky 1990]—can be defined as follows: a binary relation $\mathcal{R}$ on closed values is called a bisimulation if it satisfies all of the following conditions.

- For any values v and v' related by $\mathcal{R}$, they are of the same kind: that is, both are constants, both are tuples, or both are functions.
- For any constants c and c' related by $\mathcal{R}$, we have $c = c'$.
- For any tuples $(v_1, \ldots, v_n)$ and $(v'_1, \ldots, v'_{n'})$ related by $\mathcal{R}$, they are of the same size (i.e., $n = n'$) and their elements v_i and v'_i (for any $1 \leq i \leq n$) are pairwise related by $\mathcal{R}$.
- For any functions f and f' related by $\mathcal{R}$, and for any argument v, the function applications $f v$ and $f' v$ both converge or both diverge. If they converge, their results are also related by $\mathcal{R}$.

Bisimilarity is defined as the largest bisimulation, which exists because the union of arbitrary bisimulations is also a bisimulation. Bisimilarity can be proved to coincide with contextual equivalence (for closed values) by a rather involved technique called Howe's method [Howe 1996].

It is not trivial to extend applicative bisimulations with perfect encryption. The main problem is the last condition above on functions, which applies two bisimilar functions f and f' to the *same* argument v. This does not make sense in the presence of perfect encryption (or, in fact, any form of information hiding): for instance, the functions `Cartesian2.get_re_and_im` and `Polar2.get_re_and_im` of course do *not* always give equivalent results when applied to an identical argument; still, the modules `Cartesian2` and `Polar2` should be compatible as a whole.

What is wrong here? Obviously, we should not apply these two functions—which belong to two different modules—to identical arguments. Rather, they should be applied to different but somehow related arguments. Thus, a revised definition could be:

- For any functions f and f' related by $\mathcal{R}$, and for any argument v and v' also related by $\mathcal{R}$, the function applications fv and $f'v'$ both converge or both diverge. If they converge, their results are also related by $\mathcal{R}$.

Unfortunately, this naive revision still does not work, because it breaks the crucial property that the union of two bisimulations is also a bisimulation: a counter-example is the union of the (hypothetical) bisimulation $\mathcal{R}$ between `Cartesian2` and `Polar2` and its inverse $\mathcal{R}^{-1}$. Intuitively, this is because we are confusing two different "worlds," one of which has `Cartesian2` on the left-hand side of an equation and `Polar2` on its right-hand side, while the other world has `Polar2` on the left and `Cartesian2` on the right. Similar problems arise from similar confusions for almost any different "worlds."

Our solution is to consider a *set* of such $\mathcal{R}$s to be a bisimulation, each of them representing a different world. (This also gives a very natural account for fresh key generation.) Then, it becomes straightforward again to check the property about unions of bisimulations, which guarantees the existence of bisimilarity.

One more invention is necessary concerning function arguments: since a context can carry out its own computation, they are not always values in $\mathcal{R}$ only; rather, they can also be complex values (such as tuples) synthesized from values in $\mathcal{R}$. We will find (in Section 3.4) that these function arguments are in general of the form $[v_1, \ldots, v_n/x_1, \ldots, x_n]e$ and $[v'_1, \ldots, v'_n/x_1, \ldots, x_n]e$ for $(v_1, v'_1), \ldots, (v_n, v'_n) \in \mathcal{R}$ where e is a term containing no secret key.

We did not mention the conditions of our bisimulations for ciphertexts and keys. Fortunately, they are just an untyped version of the similar conditions in the previous section:

- For any keys related by $\mathcal{R}$, they are non-secret (by definition).

- For any ciphertexts related by $\mathcal{R}$, either their keys are secret, or else the plaintexts are related by $\mathcal{R}$.

Then, finally, the following $\{\mathcal{R}\}$ can be established between `Cartesian2` and `Polar2`.

$$\begin{aligned}\mathcal{R} \;=\; & \{(\texttt{Cartesian2}, \texttt{Polar2})\} \\ \cup\; & \{(\texttt{Cartesian2.make\_complex}, \texttt{Polar2.make\_complex}), \\ & \;(\texttt{Cartesian2.multiply}, \texttt{Polar2.multiply}), \\ & \;(\texttt{Cartesian2.get\_re\_and\_im}, \texttt{Polar2.get\_re\_and\_im})\} \\ \cup\; & \{(\texttt{encrypt}\ (x,y)\ \texttt{under}\ k,\ \texttt{encrypt}\ (r,\theta)\ \texttt{under}\ k) \mid x = r\cos\theta \text{ and } y = r\sin\theta \text{ for } r \geq 0\} \\ \cup\; & \{((x,y),(x,y)) \mid x, y \text{ real numbers}\}\end{aligned}$$

The first part is the modules themselves, the second is their components, the third is encrypted representations of complex numbers, and the fourth is the results of `get_re_and_im`.

As an additional merit of considering sets of relations as bisimulations, it becomes much easier to prove that bisimilarity coincides with contextual equivalence (as generalized with the notion of multiple worlds): in short, it suffices to prove that terms of the form $[v_1, \ldots, v_n/x_1, \ldots, x_n]e$ and $[v'_1, \ldots, v'_n/x_1, \ldots, x_n]e$ with $(v_1, v'_1), \ldots, (v_n, v'_n) \in \mathcal{R}$ always evaluate to values of the same special form if their evaluations converge (where, in fact, the first term converges if and only if the second does), which follows by induction on the derivation of the evaluations. This simplicity is striking given the complexity of Howe's method, which has until now been the only syntactic technique for proving the soundness (with respect to contextual equivalence) of applicative bisimulations and their variants [Gordon 1995a; Gordon and Rees 1996; Gordon 1995b; Gordon and Rees 1995; Jeffrey and Rathke 1999; Jeffrey and Rathke 2004].

1.6 Bisimulations for Type Abstraction

Since bisimulations work better than logical relations for encryption in languages with recursive functions, recursive types, or no types at all, it is natural to wonder whether they also work better for type abstraction in a language with recursion—and the answer is yes. The basic ideas are similar to those in the previous section, except that we have to keep careful track of type information. Specifically, each value pair in relation $\mathcal{R}$ is now annotated with a type. For example, the relation $\mathcal{R}$ for modules `Cartesian` and `Polar` can be given as follows.

$$\begin{array}{rcl}\mathcal{R} & = & \{(\texttt{Cartesian},\ \texttt{Polar},\ \texttt{Complex})\} \\ & \cup & \{(\texttt{Cartesian.make\_complex},\ \texttt{Polar.make\_complex},\ \texttt{real} \times \texttt{real} \to \texttt{t}), \\ & & \ (\texttt{Cartesian.multiply},\ \texttt{Polar.multiply},\ \texttt{t} \times \texttt{t} \to \texttt{t}), \\ & & \ (\texttt{Cartesian.get\_re\_and\_im},\ \texttt{Polar.get\_re\_and\_im},\ \texttt{t} \to \texttt{real} \times \texttt{real})\} \\ & \cup & \{((x,y),(r,\theta),\texttt{t}) \mid x = r\cos\theta \text{ and } y = r\sin\theta \text{ for } r \geq 0\} \\ & \cup & \{((x,y),(x,y),\texttt{real} \times \texttt{real}) \mid x,y \text{ real numbers}\}\end{array}$$

Each element of $\mathcal{R}$ has the form (V, V', τ), which means that values V and V' are related at type τ.

In general, a bisimulation for type abstraction is a set X of pairs $(\Delta, \mathcal{R})$, where Δ is a *concretion environment* mapping each abstract type to their implementations. For example, $\Delta = \{(\texttt{t}, \texttt{real} \times \texttt{real}, \texttt{real} \times \texttt{real})\}$ in the case of `Cartesian` and `Polar`. The two concrete types coincide in this particular example, but they do not have to in general.

Naturally, each pair $(\Delta, \mathcal{R})$ in a bisimulation X has to satisfy type-annotated versions of the conditions in the previous section. Let $\Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_m, \sigma_m, \sigma'_m)\}$.

- For any $(V, V', \tau) \in \mathcal{R}$, V has type $[\sigma_1, \ldots, \sigma_m/\alpha_1, \ldots, \alpha_m]\tau$ and V' has type $[\sigma'_1, \ldots, \sigma'_m/\alpha_1, \ldots, \alpha_m]\tau$.
- For any $(c, c', \rho) \in \mathcal{R}$ with a primitive type ρ (such as `int` and `real`), we have $c = c'$.
- For any $((V_1, \ldots, V_n), (V'_1, \ldots, V_n), \tau_1 \times \cdots \times \tau_n) \in \mathcal{R}$, we have $(V_i, V'_i, \tau_i) \in \mathcal{R}$ for every $1 \leq i \leq n$.
- For any (possibly recursive) functions $(F, F', \tau \to \sigma) \in \mathcal{R}$, and for any arguments V and V' of the appropriate form[1], the function applications FV and $F'V'$ both converge or both diverge. If they converge, their results are also in $\mathcal{R}$ with type σ.

In addition, we have a simple condition for values of recursive types. Informally, it says

- For any $(V, V', \beta) \in \mathcal{R}$ with recursive type $\beta = \tau$ (where β possibly appears in τ), we have $(V, V', \tau) \in \mathcal{R}$.

[1] To be precise, $V = [V_1, \ldots, V_n/x_1, \ldots, x_n][\sigma_1, \ldots, \sigma_m/\alpha_1, \ldots, \alpha_m]M$ and $V' = [V'_1, \ldots, V'_n/x_1, \ldots, x_n][\sigma'_1, \ldots, \sigma'_m/\alpha_1, \ldots, \alpha_m]M$ for $(V_1, V'_1, \tau_1), \ldots, (V_n, V'_n, \tau_n) \in \mathcal{R}$ and $\alpha_1, \ldots, \alpha_m, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau$. Except for the type information, this is the same as in the bisimulations for encryption in the previous section.

though our technical development uses a more explicit form (called iso-recursive types). Then, the bisimilarity—the largest bisimulation as usual—can be proved to coincide with contextual equivalence (as generalized for multiple worlds when the language includes generative constructs such as `open` for existential packages [Mitchell and Plotkin 1988]). Again, this is surprisingly simple compared to logical relations with admissibility arguments. In fact, we believe that our development is the first result on a sound, complete, and elementary proof method for a language with full recursive and abstract types (with no restrictions to inductive or predicative types).

1.7 Structure of the Thesis

The thesis is structured as follows. Chapter 2 elaborates on our logical relations for perfect encryption (dynamic sealing) in $\lambda^{\rightarrow}_{\texttt{seal}}$, simply typed λ-calculus with perfect encryption, and shows how to reason about cryptographic protocols via encoding into this language. Chapter 3 details bisimulations for perfect encryption in untyped $\lambda_{\texttt{seal}}$ with examples of data abstraction as well as a cryptographic protocol. Chapter 4 expands on bisimulations for type abstraction in λ-calculus with full recursive, universal, and existential types, with examples of abstract types including an object encoding. Each of these chapters is self-contained (though closely related) and can be read independently. While some of the technicalities may seem similar, they are not the same—because of the presence or lack of types and key sets—and are necessarily repeated with slight differences. Chapter 5 concludes with general discussions on related and future work.

Chapter 2

Logical Relations for Perfect Encryption and Dynamic Sealing

Overview

The theory of *relational parametricity* and its *logical relations* proof technique are powerful tools for reasoning about information hiding in the polymorphic λ-calculus. We investigate the application of these tools in the security domain by defining a *cryptographic λ-calculus*—an extension of the standard simply typed λ-calculus with primitives for encryption, decryption, and key generation—and introducing syntactic logical relations (in the style of Pitts and Birkedal-Harper) for this calculus that can be used to prove behavioral equivalences between programs that use encryption.

We illustrate the framework by encoding some simple security protocols, including the Needham-Schroeder public-key protocol. We give a natural account of the well-known attack on the original protocol and a straightforward proof that the improved variant of the protocol is secure.

Results in this chapter have also been presented in Sumii and Pierce [2003].

2.1 Introduction

Information hiding is a central concern in both programming languages and computer security. In the security community, encryption is the fundamental means of hiding information from outsiders. In programming languages, mechanisms such as abstract data types, modules, and parametric polymorphism play an analogous role. Each community has developed a rich set of mathematical tools for reasoning about the hiding of information in applications built using its chosen primitives. Given the intuitive similarity of the notions of hiding in the two domains, it is natural to wonder whether some of these techniques can be transferred from the programming language setting and applied to security problems, or vice versa.

As a first step in this direction, we investigate the application of one well established tool from the theory of programming languages—the concept of *relational parametricity* [Reynolds 1983] and its accompanying *logical relations* proof method—in the domain of security protocols. (By logical relations, we mean *syntactic* ones [Pitts 2000; Birkedal and Harper 1999]—that is, relations over syntactic expressions in a term model—throughout this chapter.)

We begin by defining a *cryptographic λ-calculus*, an extension of the ordinary simply typed λ-calculus with primitives for encryption, decryption, and key generation. One can imagine a large family of different cryptographic λ-calculi, each based on a different set of encryption primitives. For the present study, we use the simplest member of this family—the one where the primitives are assumed to provide *perfect* shared-key encryption. This calculus offers a suitable mix of structures for our investigation: encryption primitives, since our goal is to reason about programs from the security domain, together with the type structure on which logical relations are built. (Some issues raised by introducing static types into the calculus will be discussed in Section 2.5 along with the details of the type system.) We now proceed in three steps:

1. We show how some simple security protocols can be modeled by expressions in the cryptographic λ-calculus. The essence of the encoding lies in representing each principal by (a pair of) the next message value it sends and/or a function representing its response to the next message it receives. Our main example is the Needham-Schroeder public-key protocol [Needham and Schroeder 1978]. The encoding of this protocol gives a clear account both of the well-known attack on the original protocol and of the resilience of the improved variant of this protocol to the same attack [Lowe 1995].

2. We formalize desired secrecy properties in terms of *behavioral equivalence*. Suppose, for instance, that we would like to prove that a program keeps some integer secret against all possible attacks. Let p_i be an instance of the program with the secret integer being i. If we encode each attacker as a function f that takes the program as an argument and returns an observable value (a boolean, say), then we want to show the equality $f(p_i) = f(p_j)$ for $i \neq j$. Since such a function is itself an expression in the cryptographic λ-calculus, we can use the same language to reason about the attacker and the program.

3. We introduce a proof technique for behavioral equivalence based on *logical relations*. The technique gives a method of "relating" (in a formal sense) two programs that differ only in their secrets and that behave equivalently in every observable respect. In particular, in its original form in the polymorphic λ-calculus, it gives a method of showing behavioral equivalence between different implementations of the same abstract type—so-called relational parametricity. We adapt the same ideas to the cryptographic λ-calculus, which enables us to prove the equivalence of p_i and p_j from the point of view of an arbitrary attacker f, without explicitly quantifying over all such attackers.

To illustrate these ideas, let us consider a simple system with two principals A and B sharing a secret key z, where A encrypts a secret integer i with z and sends it to B, and B replies by returning $i \bmod 2$. In the standard informal notation, this protocol can be written as:

$$\begin{array}{lll} 1.\quad A \to B & : & \{i\}_z \\ 2.\quad B \to A & : & i\,\mathtt{mod}\,2 \end{array}$$

In the cryptographic λ-calculus, this system can be encoded as a pair p of the ciphertext $\{i\}_z$, which represents the principal A sending the integer i encrypted with z, and a function $\lambda\{x\}_z.\, x\,\mathtt{mod}\,2$ representing the principal B, which publishes $x\,\mathtt{mod}\,2$ on receiving an integer x encrypted by z. Let p_i be an instance of the program with the secret integer being i.

$$p_i \quad = \quad \nu z.\, \langle \{i\}_z,\, \lambda\{x\}_z.\, x\,\mathtt{mod}\,2 \rangle$$

Here, the key generation construct $\nu z. \ldots$ guarantees that z is *fresh*, that is, different from any other keys and unknown to the attacker.

The network, scheduler, and attackers for this system are encoded as functions operating on this pair. We assume a standard model of "possible" attackers [Dolev and Yao 1983], who are able to intercept, forge, and forward messages, encrypt and decrypt them with any keys known to the attacker, and—in addition—schedule processes arbitrarily. (The last point is not usually emphasized, but is generally assumed by considering any possible scheduling when verifying protocols.) In short, it has full control of the network and process scheduler—or, to put it extremely, "the attacker *is* the network and scheduler." Then, the properties to prove are: (i) the system accomplishes its goal under a "good" network/scheduler and (ii) the system does not leak its secret to any possible attacker.

The "good" network/scheduler for this system can be represented as a function f that takes a pair p as an argument and applies its second element $\#_2(p)$, which is a function representing a principal receiving a message, to its first element $\#_1(p)$, which is a value representing a principal sending the message.

$$f \quad = \quad \lambda p.\, \#_2(p)\#_1(p)$$

The correctness of this network/scheduler can be checked by applying f to p, which yields $i \ \mathtt{mod}\ 2$ as expected. (Of course, this network/scheduler is designed to work with this particular system only. It is also possible to encode a generic network/scheduler that will work with a range of protocols, by including the intended receiver's name in each message and delivering messages accordingly.)

On the other hand, the property that the system keeps the concrete value of i secret against any possible attacker can be stated as a claim of *behavioral equivalence* between, say, p_3 and p_5. That is, $f(p_3)$ and $f(p_5)$ give the same result for any function f returning an observable value.

Why is this? The point here is that p_3 and p_5 differ only in the concrete values of the secret integers and behave equivalently in every other respect. That is, the correspondence between values encrypted by a secret key—i.e., the integers 3 and 5 encrypted by the key z—is preserved by every part of both programs. Indeed, the first elements of the pairs are $\{3\}_z$ and $\{5\}_z$, and the second elements of the pairs are functions that, given the arguments $\{3\}_z$ and $\{5\}_z$, return the same value 1. Since the key z itself is kept secret, no other values can be encrypted by it.

Our logical relation formalizes and generalizes this argument. It demonstrates behavioral equivalence between two programs which differ only in the concrete values of their secrets, i.e., the values encrypted by secret keys. It is defined as follows:

- Two integers are related if and only if they are equal.
- Two pairs are related if and only if their elements are related.
- Two functions are related if and only if they return related values when applied to related arguments.
- Two values encrypted by a secret key k are related if and only if they are related by $\varphi(k)$. Here, φ is a *relation environment* mapping each secret key to the relation between values encrypted by that key. Intuitively, it specifies what values are encrypted by each secret key (e.g., 3 and 5 are encrypted by z in the example above) and thereby keeps track of the correspondence between ciphertexts (e.g., between $\{3\}_z$ and $\{5\}_z$).

$$
\begin{array}{lcl}
e & ::= & i \quad | \quad int\_op_n(e_1, \ldots, e_n) \quad | \quad x \quad | \quad \lambda x.\, e \quad | \quad e_1 e_2 \quad | \\
& & \langle e_1, \ldots, e_n \rangle \quad | \quad \#_i(e) \quad | \quad \mathtt{in}_i(e) \quad | \quad \mathtt{case}\; e\; \mathtt{of}\; \mathtt{in}_1(x_1) \Rightarrow e_1 \; [\!] \ldots [\!] \; \mathtt{in}_n(x_n) \Rightarrow e_n \quad | \\
& & k \quad | \quad \nu x.\, e \quad | \quad \{e_1\}_{e_2} \quad | \quad \mathtt{let}\; \{x\}_{e_1} = e_2\; \mathtt{in}\; e_3\; \mathtt{else}\; e_4
\end{array}
$$

Figure 2.1: Syntax of $\lambda^{\rightarrow}_{\mathtt{seal}}$

The soundness theorem for the logical relation now tells us that, in order to prove behavioral equivalence of two programs, it suffices to find some φ such that the logical relation based on φ relates the programs we are interested in. For instance, in the example above, take $\varphi(z) = \{(3, 5)\}$. Then:

- The first elements of p_3 and p_5 are related by the definition of φ.
- The second elements of p_3 and p_5 are related since they return related values (i.e., 1) for any related arguments (i.e., $\{3\}_z$ and $\{5\}_z$).
- Therefore, the pairs p_3 and p_5 are related since their elements are related.

Thus, p_3 and p_5 are guaranteed to be behaviorally equivalent. In this way, the logical relation allows us to prove the behavioral equivalence of programs—which amounts to proving secrecy—in a compositional manner.

The contributions of this work are twofold: theoretically, it clarifies the intuitive similarity between two forms of information hiding in different domains, namely, encryption in security systems and type abstraction in programming languages; practically, it offers a method of proving behavioral equivalence (which implies secrecy) of programs that use encryption.

The rest of this chapter is structured as follows. Section 2.2 presents the syntax and informal semantics of the cryptographic λ-calculus and Section 2.3 demonstrates its use in encoding cryptographic protocols through examples. Section 2.4 shows its operational semantics, Section 2.5 gives a simple type system—a prerequisite for formalizing the logical relations—and Section 2.6 defines our logical relation and its variants to cope with more complex cases (such as "keys encrypting other keys" in security protocols). Section 2.7 discusses related work, and Section 2.8 future work.

2.2 Syntax and Informal Semantics

The cryptographic λ-calculus extends a standard λ-calculus with keys, fresh key generation, encryption, and decryption primitives. The formal syntax of the calculus is given in Figure 2.1. $int\_op_n(e_1, \ldots, e_n)$ is the syntax of primitive operators for integer arithmetics like $plus_2(e_1, e_2)$. (The subscript denotes the arity of each operator. This is necessary for the type system in Section 2.5.) We adopt infix notations such as $e_1 + e_2$ for binary operations. We use a tuple with no elements (i.e., $n = 0$) to represent a dummy value, written $\langle\rangle$. $\mathtt{in}_i(\,)$ denotes the i-th injection ("tagging") into a disjoint sum.

A key k is an element of the countably infinite set of keys K. The key generation form $\nu x.\, e$ generates a fresh key, binds it to the variable x, and evaluates the expression e (in which x may appear free). For example, the program $\nu x.\, \nu y.\, \langle x, y \rangle$ generates two fresh keys and yields a pair of them. The encryption expression $\{e_1\}_{e_2}$ encrypts the value obtained by evaluating the expression e_1 with the key obtained by evaluating the expression e_2. The decryption form $\mathtt{let}\; \{x\}_{e_1} = e_2\; \mathtt{in}$

e_3 `else` e_4 attempts to decrypt the ciphertext obtained by evaluating e_2, using the key obtained by evaluating e_1. If the decryption succeeds, it binds the plaintext thus obtained to the variable x and evaluates the expression e_3. If the decryption fails, it instead evaluates e_4. For example, the program $\texttt{let } \{x\}_{k'} = \{1\}_k \texttt{ in } x + 2 \texttt{ else } 0$ encrypts the integer 1 with the key k, tries to decrypt it with another key k'—which fails because the keys are different—and therefore gives 0. On the other hand, $\texttt{let } \{x\}_k = \{1\}_k \texttt{ in } x + 2 \texttt{ else } 0$ gives 3 because the decryption succeeds.

Several abbreviations are used in examples. We write `true` for $\texttt{in}_1(\langle\rangle)$, `false` for $\texttt{in}_2(\langle\rangle)$, and $\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$ for $\texttt{case } e \texttt{ of in}_1(\_) \Rightarrow e_1 \parallel \texttt{in}_2(\_) \Rightarrow e_2$, respectively, to represent booleans as a disjoint sum of dummy values. We use option values $\texttt{Some}(e)$, abbreviating $\texttt{in}_1(e)$, and `None`, abbreviating $\texttt{in}_2(\langle\rangle)$, to represent the return values of functions that may or may not actually return a meaningful value because of errors such as decryption failure. Finally, we use the pattern matching function syntax $\lambda\{x\}_{e_1}.\, e_2$ to abbreviate $\lambda y.\, \texttt{let } \{x\}_{e_1} = y \texttt{ in } e_2 \texttt{ else None}$ (where y does not appear free in e_1 and e_2), representing functions accepting arguments encrypted by a particular key only. For example, the function $\lambda\{x\}_k.\, \texttt{Some}(x+1)$ returns $\texttt{Some}(i+1)$ when applied to an integer i encrypted by the key k, and `None` for any ciphertext not encrypted by k.

Example 2.1. The expression $\lambda\{x\}_k.\, \texttt{Some}(x \texttt{ mod } 2)$, which is an abbreviation for $\lambda y.\, \texttt{let } \{x\}_k = y \texttt{ in in}_1(x \texttt{ mod } 2) \texttt{ else in}_2(\langle\rangle)$, represents a function f that accepts an integer x encrypted by the key k and returns its remainder when divided by 2, with the tag `Some` to denote success. For instance, the application $f(\{3\}_k)$ gives the result $\texttt{Some}(1)$ while $f(\{i\}_{k'})$ returns `None` for any i and any $k' \neq k$.

Example 2.2. Let $p_i = \nu z.\, \langle \{i\}_z, \lambda\{x\}_z.\, \texttt{Some}(x \texttt{ mod } 2)\rangle$ and $f = \lambda p.\, \#_2(p)\#_1(p)$. Then, $f(p_i)$ gives $\texttt{Some}(i \texttt{ mod } 2)$. This can be seen as a run of an encoding of the following simple system, in which two principals A and B share a key z (to be precise, a key bound to the variable z): first, A encrypts and sends i; then, B receives and decrypts it, and publishes its remainder when divided by 2. The function f plays the role of a "good" network and scheduler for this system.

Our cryptographic primitives directly model only shared-key encryption, but they can also be used to approximate public-key encryption: for any key k, the encryption function $\lambda x.\, \{x\}_k$ and decryption function $\lambda\{x\}_k.\, \texttt{Some}(x)$ can be passed around and used as encryption and decryption keys.

This encoding is somewhat tricky and not quite faithful to the real world: not only we are again assuming perfect encryption and taking no account of any algebraic or probabilistic properties of individual cryptography, but also it would be silly, in reality, to give an attacker the code of a function that includes any secret key, which a real attacker could presumably discover by disassembling the function. In addition, our encoding limits the capability of attackers: for instance, they cannot test equality of public keys. Nevertheless, it suffices for our goal in this chapter of giving an account of a few major "benchmark" examples. (In situations where this encoding does *not* suffice, we could try to reinforce it: for instance, in order to reason about a protocol in which equality of public keys is an essential issue, we could give an attacker a public key like $\lambda x.\, \{x\}_k$ along with a function like $\lambda e.\, \texttt{let } \{\_\}_k = e\langle\rangle \texttt{ in true else false}$ to test whether another (encryption) key is equal to the previous one.)

2.3 Applications

Now we demonstrate the use of our framework on some larger examples, in which concurrent principals communicate with one another by using encryption. Although the cryptographic λ-calculus has no built-in primitives for concurrency or communication, it can emulate a concurrent, communicating system by a reasonably straightforward encoding (recall the example in Section 2.1).

- The system as a whole is encoded as a tuple of the processes and their public keys (if any).
- An output process is encoded as the message itself.
- An input process is encoded as a function receiving a message.
- A network/scheduler/attacker for the system is encoded as a function that applies the input functions to the output messages in a certain order, possibly manipulating the messages using the keys that it knows.

Then, the following two properties are desired in general.

- Under a "correct" network and scheduler, i.e., a function applying appropriate messages to appropriate functions in some appropriate order, the program gives some correct result (*soundness*).
- Under *any* possible attacker, the program does not do anything "wrong" such as leaking a secret (*safety*).

More precise definitions of "correct" and "wrong" depend on the intention of each specific protocol, as we shall see below.

2.3.1 Encoding the Needham-Schroeder Public-Key Protocol

Consider the following system using the Needham-Schroeder public-key protocol [Needham and Schroeder 1978] in a network with a server A, a client B, and an attacker E. (1) B sends its own name B to A. (2) A generates a fresh nonce N_A, pairs it with its own name A, encrypts it with B's public key, and sends it to B. (3) B generates a fresh key N_B, pairs it with N_A, encrypts it with A's public key, and sends it to A. (4) A encrypts N_B with B's public key and sends it to B. (5) B encrypts some secret integer i with N_B and sends it to A.

$$
\begin{array}{lllll}
1. & B \to A & : & B \\
2. & A \to B & : & \{N_A, A\}_{k_B} \\
3. & B \to A & : & \{N_A, N_B\}_{k_A} \\
4. & A \to B & : & \{N_B\}_{k_B} \\
5. & B \to A & : & \{i\}_{N_B}
\end{array}
$$

Let us encode this system as an expression of the cryptographic λ-calculus. (The result is necessarily somewhat complex, because several actions implicitly assumed in the informal definition above—such as checks on the identity of names and keys—are made explicit in the encoding process.)

Recall that we encode such a concurrent, communicating system as a tuple of the principals and the public keys. So we begin the encoding by generating the system's keys and publishing their public portions, that is, A's encryption key, B's encryption key, and a key for E. (This key for E is necessary for encoding an attack to the protocol as we shall see soon, in which E is one of the clients of A and therefore A must know the key for E.)

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E, \ldots, \ldots\rangle\end{array}$$

Let us now encode B as the fourth element of the tuple. B starts by publishing its own name B, which we encode as a pair of B and a function representing B's next action, which we'll come back to in a moment. (We assume that names of principles are just integers—like IP addresses, for example—for the sake of simplicity.) The difference from the previous expression is underlined.

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E, \underline{\langle B, \ldots\rangle}, \ldots\rangle\end{array}$$

Next, let us encode A as the fifth element of the tuple. A receives a name X, encrypts the pair of a freshly generated nonce N_A and its own name A with X's key, and publishes it.

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E, \langle B, \ldots\rangle,\\ \;\underline{\lambda X.\,\nu N_A.\,\langle\{\langle N_A, A\rangle\}_{z_x}, \ldots\rangle}\rangle\end{array}$$

Here, z_x abbreviates $\texttt{if}\ X = A\ \texttt{then}\ z_A\ \texttt{else}\ \texttt{if}\ X = B\ \texttt{then}\ z_B\ \texttt{else}\ z_E$. The next action of B is to receive the pair of N_A and a name A' encrypted by z_B, check that $A' = A$, encrypt the pair of N_A and a freshly generated nonce N_B with z_A, and publish it:

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E,\\ \;\langle B, \underline{\lambda\{\langle N_A, A\rangle\}_{z_B}.\,\nu N_B.\,\texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A}, \ldots\rangle)}\rangle,\\ \;\lambda X.\,\nu N_A.\,\langle\{\langle N_A, A\rangle\}_{z_x}, \ldots\rangle\rangle\end{array}$$

Here, $\lambda\{\langle N_A, A\rangle\}_{z_B}.\ \ldots$ abbreviates a "pattern matching" expression $\lambda y.\,\texttt{let}\ \{p\}_{z_B} = y\ \texttt{in}\ (\texttt{if}\ \#_2(p) = A\ \texttt{then}\ (\lambda N_A.\ \ldots)\#_1(p)\ \texttt{else}\ \texttt{None})\ \texttt{else}\ \texttt{None}$. We will use similar abbreviations throughout this chapter. Next, A receives the pair of a nonce N'_A and N_B, checks that $N'_A = N_A$, encrypts N_B with z_B, and publishes it:

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E,\\ \;\langle B, \lambda\{\langle N_A, A\rangle\}_{z_B}.\,\nu N_B.\,\texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A}, \ldots\rangle)\rangle,\\ \;\langle \lambda X.\,\nu N_A.\,\langle\{\langle N_A, A\rangle\}_{z_x},\\ \qquad\qquad \underline{\lambda\{N_A, N_x\}_{z_A}.\,\texttt{Some}(\{N_x\}_{z_B})}\rangle\rangle\rangle\end{array}$$

Last, B receives a nonce N'_B encrypted by z_B, checks that $N'_B = N_B$, encrypts i with N_B, and publishes it:

$$\begin{array}{l}\nu z_A.\,\nu z_B.\,\nu z_E.\\ \langle \lambda x.\,\{x\}_{z_A}, \lambda x.\,\{x\}_{z_B}, z_E,\\ \;\langle B, \lambda\{\langle N_A, A\rangle\}_{z_B}.\,\nu N_B.\\ \qquad \texttt{Some}(\langle\{\langle N_A, N_B\rangle\}_{z_A},\ \underline{\lambda\{N_B\}_{z_B}.\,\texttt{Some}(\{i\}_{N_B})}\rangle)\rangle,\\ \;\langle \lambda X.\,\nu N_A.\,\langle\{\langle N_A, A\rangle\}_{z_x},\\ \qquad\qquad \lambda\{N_A, N_x\}_{z_A}.\,\texttt{Some}(\{N_x\}_{z_x})\rangle\rangle\rangle\end{array}$$

Let NS_i be the expression above.

A correct run of this system can be expressed by evaluation of this expression under the following function *Good*, which represents a "good" network/scheduler for this system.

$$
\begin{array}{l}
\lambda p.\, \texttt{let}\ \langle B, c_B\rangle = \#_4(p)\ \texttt{in} \\
\quad \texttt{let}\ \langle m_1, c_A\rangle = \#_5(p)B\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\langle m_2, c'_B\rangle) = c_B m_1\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(m_3) = c_A m_2\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(m_4) = c'_B m_3\ \texttt{in} \\
\quad \texttt{Some}(m_4)
\end{array}
$$

Indeed, $Good(NS_i)$ evaluates to $\texttt{Some}(\{i\}_{N_B})$ for some fresh N_B, which means a successful execution of the system.

It is well known that a use of the Needham-Schroeder public-key protocol such as the system above—namely, letting the server A accept a request not only from the friendly client B but also from the malicious attacker E—is vulnerable to the following man-in-the-middle attack, which allows E to impersonate A while interacting with B [Lowe 1995]. The attack goes as follows. (1) B sends its own name B to A, but E intercepts it. (1′) E sends its own name E to A. (2′) A generates a fresh nonce N_A, pairs it with A, encrypts it with E's public key, and sends it to E. (2) E encrypts the pair of N_A and A with B's public key and sends it to B, pretending to be A. (3,3′) B generates a fresh nonce N_B, pairs it with N_A, encrypts it with A's public key, and sends it to A. (4′) A encrypts N_B with E's public key and sends it to E. (4) E encrypts N_B with B's public key and sends it to B, pretending to be A. (5) B encrypts i with N_B and sends it to A, but E intercepts and decrypts it.

$$
\begin{array}{lllll}
1. & B \to E(A) & : & B \\
1'. & E \to A & : & E \\
2'. & A \to E & : & \{N_A, A\}_{k_E} \\
2. & E(A) \to B & : & \{N_A, A\}_{k_B} \\
3, 3'. & B \to A & : & \{N_A, N_B\}_{k_A} \\
4'. & A \to E & : & \{N_B\}_{k_E} \\
4. & E(A) \to B & : & \{N_B\}_{k_B} \\
5. & B \to E(A) & : & \{i\}_{N_B}
\end{array}
$$

This attack can be expressed in the cryptographic λ-calculus by the following function *Evil*.

$$
\begin{array}{l}
\lambda p.\, \texttt{let}\ \langle B, c_B\rangle = \#_4(p)\ \texttt{in} \\
\quad \texttt{let}\ \langle \{\langle N_A, A\rangle\}_{\#_3(p)}, c_A\rangle = \#_5(p)E\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\langle m, c'_B\rangle) = c_B(\#_2(p)\langle N_A, A\rangle)\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\{N_B\}_{\#_3(p)}) = c_A m\ \texttt{in} \\
\quad \texttt{let}\ \texttt{Some}(\{i\}_{N_B}) = c'_B(\#_2(p)N_B)\ \texttt{in} \\
\quad \texttt{Some}(i)
\end{array}
$$

$Evil(NS_i)$ indeed evaluates to $\texttt{Some}(i)$, which leaks the secret. In other words, if $i \neq j$, then $Evil(NS_i)$ and $Evil(NS_j)$ evaluate to different observable values, which shows that NS_i and NS_j are not behaviorally equivalent.

Of course, the example in this section is not the only use of the Needham-Schroeder protocol. Some uses are easy to reason about within our framework while others are not: e.g., it is straightforward to extend the example above with other clients besides just B (it suffices to duplicate the encoding of B, just replacing the name B with another), but changing the constant integer i to non-constant data like $\nu x.\, x$ leads to a challenge in even *defining* how to state the secrecy of such data. Indeed, this is the reason why we introduced the 5th message $\{i\}_{N_B}$ into the protocol—so that we can state the secrecy of N_B via the secrecy of i.

2.3.2 Encoding the Improved Needham-Schroeder Public-Key Protocol

Consider the following variant of the system above, using an improved version of the Needham-Schroeder public-key protocol [Lowe 1995]. (The difference from the original version is underlined.) (1) B sends its own name B to A. (2) A generates a fresh nonce N_A, pair it with its own name A, encrypts it with B's public key, and sends it to B. (3) B generates a fresh key N_B, tuples it with N_A <u>and B</u>, encrypts it with A's public key, and sends it to A. (4) A encrypts N_B with B's public key and sends it to B. (5) B encrypts some secret integer i with N_B and sends it to A.

$$\begin{array}{lccl}
1. & B \to A & : & B \\
2. & A \to B & : & \{N_A, A\}_{k_B} \\
3. & B \to A & : & \{N_A, N_B, \underline{B}\}_{k_A} \\
4. & A \to B & : & \{N_B\}_{k_B} \\
5. & B \to A & : & \{i\}_{N_B}
\end{array}$$

Following the same lines as the encoding of the original system, this improved system can be encoded as follows.

$$\begin{array}{l}
\nu z_A.\, \nu z_B.\, \nu z_E. \\
\langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \\
\;\langle B, \lambda\{\langle N_A, A\rangle\}_{z_B}.\, \nu N_B. \\
\qquad \mathtt{Some}(\langle\{\langle N_A, N_B, \underline{B}\rangle\}_{z_A},\ \lambda\{N_B\}_{z_B}.\, \mathtt{Some}(\{i\}_{N_B})\rangle)\rangle, \\
\;\langle \lambda X.\, \nu N_A.\, \langle\{\langle N_A, A\rangle\}_{z_x}, \\
\qquad\qquad\quad \lambda\{N_A, N_x, \underline{X}\}_{z_A}.\, \mathtt{Some}(\{N_x\}_{z_x})\rangle\rangle\rangle
\end{array}$$

Let NS'_i be the expression above.

How does this change prevent the attack? Recall that *Evil* was the following function.

$$\begin{array}{l}
\lambda p.\, \mathtt{let}\ \langle B, c_B\rangle = \#_4(p)\ \mathtt{in} \\
\quad \mathtt{let}\ \langle\{\langle N_A, A\rangle\}_{\#_3(p)}, c_A\rangle = \#_5(p)E\ \mathtt{in} \\
\quad \mathtt{let}\ \mathtt{Some}(\langle m, c'_B\rangle) = c_B(\#_2(p)\langle N_A, A\rangle)\ \mathtt{in} \\
\quad \mathtt{let}\ \mathtt{Some}(\{N_B\}_{\#_3(p)}) = c_A m\ \mathtt{in} \\
\quad \mathtt{let}\ \mathtt{Some}(\{i\}_{N_B}) = c'_B(\#_2(p)N_B)\ \mathtt{in} \\
\quad \mathtt{Some}(i)
\end{array}$$

When the attacker forwards the message $m = \{\langle N_A, N_B, B\rangle\}_{z_B}$ (which is encrypted by B's secret key and cannot be decrypted by the attacker) from B to A, A tries to match B against $X = E$, which fails. Thus, $\mathit{Evil}(NS'_i)$ reduces to None for any i and fails to leak the secret. In Section 2.6, we formally prove this secrecy property with respect to all possible attackers using our logical relation.

2.3.3 Encoding the ffgg Protocol

The ffgg protocol [Millen 1999] is an artificial protocol with an intentional flaw. It tries to communicate a secret name M in a rather strange manner. The point is that the protocol *does* keep the name secret as long as just one process runs for each principal, but fails to keep the secret *only* when more than one processes run for one of the principals! Although the cryptographic λ-calculus is sequential, it is actually expressive enough to encode this so-called "necessarily parallel attack" by interleaving.

To see this, let us encode the following exchange between two principals A and B using the ffgg protocol. (1) A sends its own name A to B. (2) B generates two fresh nonces N_1 and N_2 and sends them to A. (3) A tuples N_1, N_2, and some secret value M, encrypts them with a shared secret key k, and sends them to B. However, B does not check whether this N_2 is equal to the previous one which B already knows, and just lets x be the second element of the tuple and y be the third. (4) B tuples x, y and N_1, encrypts them with k, and sends them to A with N_1 and x.

$$
\begin{array}{llcl}
1. & A \to B & : & A \\
2. & B \to A & : & N_1, N_2 \\
3. & A \to B & : & \{N_1, N_2, M\}_k \text{ as } \{N_1, x, y\}_k \\
4. & B \to A & : & N_1, x, \{x, y, N_1\}_k
\end{array}
$$

This system can be encoded as the following expression ffgg_M.

$$
\begin{array}{l}
\nu z. \\
\langle\langle A, \lambda\langle N_1, N_2\rangle.\, \{\langle N_1, N_2, M\rangle\}_z\rangle, \\
\;\lambda A.\, \nu N_1.\, \nu N_2. \\
\;\langle\langle N_1, N_2\rangle, \lambda\{N_1, x, y\}_z.\, \langle N_1, x, \{\langle x, y, N_1\rangle\}_z\rangle\rangle\rangle
\end{array}
$$

The attack on this system is as follows. (1) A sends its own name A to B. (1′) Pretending to be A, the attacker E sends A to a parallel copy B′ of B. (2a) B generates two fresh nonces N_1 and N_2, and send them to A, but E intercepts them. (2′) B′ generates other two fresh nonces N_1' and N_2', and send them to A, but E again intercepts them. (2b) E sends N_1 and N_1' to A, pretending to be B. (3) A tuples N_1, N_1' and M, encrypts them with k, and sends them to B. (4) B tuples N_1', M and N_1, encrypts them with k, and send them to A with N_1 and N_1', but E intercepts them. (3′) E forwards the tuple of N_1', M and N_1 encrypted by k to B′, pretending to be A. (4′) B′ tuples M, N_1 and N_1', encrypts them with k, and send them to A with N_1' and M, but E intercepts them.

$$
\begin{array}{llcl}
1. & A \to B & : & A \\
1'. & E(A) \to B' & : & A \\
2a. & B \to E(A) & : & N_1, N_2 \\
2'. & B' \to E(A) & : & N_1', N_2' \\
2b. & E(B) \to A & : & N_1, N_1' \\
3. & A \to B & : & \{N_1, N_1', M\}_k \\
4. & B \to E(A) & : & N_1, N_1', \{N_1', M, N_1\}_k \\
3'. & E(A) \to B' & : & \{N_1', M, N_1\}_k \\
4'. & B' \to E(A) & : & N_1', M, \{M, N_1, N_1'\}_k
\end{array}
$$

This attack can be encoded as the following function, which takes the expression above as a parameter p.

$$\begin{array}{l}\lambda p.\,\texttt{let}\ \langle A, c_A\rangle = \#_1(p)\ \texttt{in}\\ \quad \texttt{let}\ \langle\langle N_1, N_2\rangle, c_B\rangle = \#_2(p)A\ \texttt{in}\\ \quad \texttt{let}\ \langle\langle N_1', N_2'\rangle, c_B'\rangle = \#_2(p)A\ \texttt{in}\\ \quad \texttt{let}\ m = c_A\langle N_1, N_1'\rangle\ \texttt{in}\\ \quad \texttt{let}\ \langle N_1, N_1', m'\rangle = c_B m\ \texttt{in}\\ \quad \texttt{let}\ \langle N_1', M, m''\rangle = c_B' m'\ \texttt{in}\\ \quad \texttt{Some}(M)\end{array}$$

This function indeed reveals the secret value M in the expression *ffgg*$_M$ above. Note that the function representing the principal B did not have to be replicated explicitly, because functions in λ-calculus can be applied any number of times. In this way, our framework can express the so-called "necessarily parallel attack" without any extra treatment.

By the way, in this encoding, there actually exists an even simpler function which leaks the secret.

$$\begin{array}{l}\lambda p.\,\texttt{let}\ \langle A, c_A\rangle = \#_1(p)\ \texttt{in}\\ \quad \texttt{let}\ \langle\langle N_1, N_2\rangle, c_B\rangle = \#_2(p)A\ \texttt{in}\\ \quad \texttt{let}\ m = c_A\langle N_1, N_2\rangle\ \texttt{in}\\ \quad \texttt{let}\ \langle N_1, N_2, m'\rangle = c_B m\ \texttt{in}\\ \quad \texttt{let}\ \langle N_2, M, m''\rangle = c_B m'\ \texttt{in}\\ \quad \texttt{Some}(M)\end{array}$$

This attack is usually considered impossible in reality, because it applies the "continuation" function c_B twice, which means exploiting one state of (a process running for) the principal B more than once. This kind of false attacks could perhaps be excluded in our framework by using *linear* types for continuation functions like c_B. See Section 2.8 for details.

2.4 Operational Semantics

In this section and the two that follow, we present the cryptographic λ-calculus, its type system, and the logical relations proof technique in detail.

The semantics of the calculus is defined by an *evaluation* relation mapping terms to results. For the ordinary λ-calculus, the evaluation relation has the form $e \Downarrow v$, read "evaluation of the (closed) expression e yields the value v." However, since the cryptographic λ-calculus includes a primitive for key generation, we need to represent "the set of keys generated so far" in some rigorous fashion. We do this by annotating the evaluation relation with a set s, representing the keys that have already been used when evaluation begins, and a set s', representing the keys that have been used when evaluation finishes. To be precise, we define the relation $(s)e \Downarrow V$ where V is either of the form $(s')v$ or *Error* (signalling a run-time type error). We maintain the invariant that $(s)e \Downarrow (s')v$ implies $s \subseteq s'$, that is, $s' \setminus s$ is the set of fresh keys generated during the evaluation of e. The evaluation relation is defined inductively by the rules in Figure 2.2.

Most of the evaluation rules are standard and straightforward; we explain just a few important points. In the rule for key generation, k is guaranteed to be "freshly generated" because $s \uplus \{k\}$ is defined and therefore $k \notin s$. (Here, $s \uplus s'$ is defined as $s \cup s'$ when $s \cap s' = \emptyset$, and undefined otherwise.) This is the only rule that increases the set of keys. In the rules for decryption, we first evaluate e_1 to obtain the decryption key k_1, then e_2 is evaluated to obtain a ciphertext of the form

$$
\begin{array}{rcl}
v & ::= & i \mid \lambda x.\, e \mid \langle v_1, \ldots, v_n \rangle \mid \mathtt{in}_i(v) \mid k \mid \{v\}_k \\
V & ::= & (s)v \mid \mathit{Error}
\end{array}
$$

$$
\frac{}{(s)i \Downarrow (s)i}
\qquad
\frac{(s_0)e_1 \Downarrow (s_1)i_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)i_n \quad \mathit{int\_op}_n(i_1, \ldots, i_n) = j}{(s_0)\mathit{int\_op}_n(e_1, \ldots, e_n) \Downarrow (s_n)j}
$$

$$
\frac{}{(s)\lambda x.\, e \Downarrow (s)\lambda x.\, e}
\qquad
\frac{(s)e_1 \Downarrow (s_1)\lambda x.\, e \quad (s_1)e_2 \Downarrow (s_2)v \quad (s_2)[v/x]e \Downarrow V}{(s)e_1 e_2 \Downarrow V}
$$

$$
\frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)v_n}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow (s_n)\langle v_1, \ldots, v_n \rangle}
\qquad
\frac{(s)e \Downarrow (s')\langle \ldots, v_i, \ldots \rangle}{(s)\#_i(e) \Downarrow (s')v_i}
$$

$$
\frac{(s)e \Downarrow (s')v}{(s)\mathtt{in}_i(e) \Downarrow (s')\mathtt{in}_i(v)}
\qquad
\frac{(s)e \Downarrow (s')\mathtt{in}_i(v) \quad (s')[v/x_i]e_i \Downarrow V}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow V}
$$

$$
\frac{}{(s)k \Downarrow (s)k}
\qquad
\frac{(s \uplus \{k\})[k/x]e \Downarrow V}{(s)\nu x.\, e \Downarrow V}
\qquad
\frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow (s_2)k}{(s)\{e_1\}_{e_2} \Downarrow (s_2)\{v\}_k}
$$

$$
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)k_1 \quad (s_1)e_2 \Downarrow (s_2)\{v\}_{k_2} \\ (s_2)[v/x]e_3 \Downarrow V \quad k_1 = k_2\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow V}
\qquad
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)k_1 \quad (s_1)e_2 \Downarrow (s_2)\{v\}_{k_2} \\ (s_2)e_4 \Downarrow V \quad k_1 \neq k_2\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow V}
$$

$$
\frac{(s_0)e_1 \Downarrow (s_1)i_1 \quad \ldots \quad (s_{j-1})e_j \Downarrow \mathit{Error}}{(s_0)\mathit{int\_op}_n(e_1, \ldots, e_n) \Downarrow \mathit{Error}}
\qquad
\frac{\begin{array}{c}(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{n-1})e_n \Downarrow (s_n)v_n \\ v_i \text{ is not of the form } j \text{ for some } 1 \le i \le n\end{array}}{(s_0)\mathit{int\_op}_n(e_1, \ldots, e_n) \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)e_1 e_2 \Downarrow \mathit{Error}}
\qquad
\frac{(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)e_1 e_2 \Downarrow \mathit{Error}}
\qquad
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_1 \text{ is not of the form } \lambda x.\, e\end{array}}{(s)e_1 e_2 \Downarrow \mathit{Error}}
$$

$$
\frac{(s_0)e_1 \Downarrow (s_1)v_1 \quad \ldots \quad (s_{i-1})e_i \Downarrow \mathit{Error}}{(s_0)\langle e_1, \ldots, e_n \rangle \Downarrow \mathit{Error}}
\qquad
\frac{(s)e \Downarrow \mathit{Error}}{(s)\#_i(e) \Downarrow \mathit{Error}}
\qquad
\frac{\begin{array}{c}(s)e \Downarrow (s')v \\ v \text{ is not of the form } \langle \ldots, v_i, \ldots \rangle\end{array}}{(s)\#_i(e) \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{in}_i(e) \Downarrow \mathit{Error}}
\qquad
\frac{(s)e \Downarrow \mathit{Error}}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e \Downarrow (s')v \quad v \text{ is not of the form } \mathtt{in}_i(v_i) \text{ for any } 1 \le i \le n}{(s)\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow e_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow e_n \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}}
\qquad
\frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}}
\qquad
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_2 \text{ is not of the form } k\end{array}}{(s)\{e_1\}_{e_2} \Downarrow \mathit{Error}}
$$

$$
\frac{(s)e_1 \Downarrow \mathit{Error}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}
\qquad
\frac{(s)e_1 \Downarrow (s_1)v \quad (s_1)e_2 \Downarrow \mathit{Error}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}
$$

$$
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_1 \text{ is not of the form } k\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}
\qquad
\frac{\begin{array}{c}(s)e_1 \Downarrow (s_1)v_1 \quad (s_1)e_2 \Downarrow (s_2)v_2 \\ v_2 \text{ is not of the form } \{v\}_k\end{array}}{(s)\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathit{Error}}
$$

Figure 2.2: Semantics of $\lambda^{\rightarrow}_{\mathtt{seal}}$

$\{v\}_{k_2}$. If e_1 does not evaluate to a key or e_2 does not evaluate to a ciphertext, then a type error occurs. Otherwise, if the two keys match ($k_1 = k_2$), the body e_3 is evaluated, with x bound to the decrypted plaintext v. Otherwise, the `else` clause e_4 is evaluated.

The following theorem and corollary state that the result of evaluating an expression is unique, modulo the names of freshly generated keys. (We write $Keys(e)$ for the set of keys syntactically appearing in e.)

Theorem 2.3. Let $s_1 \supseteq Keys(e)$ and let θ be a one-to-one substitution from s_1 to another set of keys s_2. If $(s_1)e \Downarrow (s_1 \uplus s_1')v_1$ and $(s_2)\theta e \Downarrow V$, then V has the form $(s_2 \uplus s_2')v_2$ and there exists some one-to-one substitution θ' from s_1' to s_2' such that $v_2 = (\theta \uplus \theta')v_1$.

Proof. By induction on the derivation of $(s_1)e \Downarrow (s_1 \uplus s_1')v_1$ with a lemma on monotonic increase of key sets. See Appendix A.4 for details. □

Corollary 2.4 (Uniqueness of Evaluation Result). Let $s \supseteq Keys(e)$. If $(s)e \Downarrow (s \uplus s_1')v_1$ and$(s)e \Downarrow V$, then V has the form $(s \uplus s_2')v_2$ and there exists some one-to-one substitution θ' from s_1' to s_2' such that $v_2 = \theta' v_1$.

Proof. Let $s_1 = s_2 = s$ and $\theta = id$ in Theorem 2.3. □

2.5 Type System

In this section, we define a simple type system for the cryptographic λ-calculus. Types in this setting play not only the traditional role of guaranteeing the absence of run-time type errors (a well-typed term cannot evaluate to *Error*), but, more importantly, provide a framework for the reasoning method we consider in the next section, in which the fundamental definition of the logical relations proceeds by induction on types.

In addition to the values found in the ordinary λ-calculus, the cryptographic λ-calculus has keys and ciphertexts. Therefore, besides the usual arrow, product, and sum types of the simply typed λ-calculus, we introduce a key type $\texttt{key}[\tau]$, whose elements are keys that can be used to encrypt values of type τ, and a ciphertext type $\texttt{bits}[\tau]$, whose elements are ciphertexts containing a plaintext value of type τ. Thus, keys of a given type cannot be used to encrypt values of different types, and ciphertexts of a given type cannot contain plaintext values of different types. This restriction is not particularly bothersome, since values of (finitely many) different types can always be injected into a common sum type. (Actually, in order to guarantee type safety, we do not need to annotate both key types and ciphertext types with their underlying plaintext types. However, doing so simplifies the definition of the logical relations in Section 2.6.)

The typing judgment has the form $\Gamma, \Delta \vdash e : \tau$, read "under the type environment Γ for variables and the type environment Δ for keys, the expression e has the type τ, i.e., e evaluates to a value of type τ." The typing rules (which are straightforward) are given in Figure 2.3. Here, $f \uplus f'$ for two mappings f and f' is defined as $(f \uplus f')(x) = f(x)$ for $x \in dom(f)$ and $(f \uplus f')(y) = f'(y)$ for $y \in dom(f')$ if $dom(f) \cap dom(f') = \emptyset$, and undefined otherwise. Note that the type environment Δ for keys is used in the rule (Key) in the same way the type environment Γ for variables is used in the rule (Var). For the sake of readability, we often write `bool` for $\texttt{unit} + \texttt{unit}$ and $\texttt{option}[\tau]$ for $\tau + \texttt{unit}$, where `unit` is the type of a tuple with no elements.

$$\tau \quad ::= \quad \texttt{int} \quad | \quad \tau_1 \to \tau_2 \quad | \quad \tau_1 \times \cdots \times \tau_n \quad | \quad \tau_1 + \cdots + \tau_n \quad | \quad \texttt{key}[\tau] \quad | \quad \texttt{bits}[\tau]$$

$$\frac{}{\Gamma, \Delta \vdash i : \texttt{int}}(\text{Const}) \quad \frac{\Gamma, \Delta \vdash e_1 : \texttt{int} \quad \ldots \quad \Gamma, \Delta \vdash e_n : \texttt{int}}{\Gamma, \Delta \vdash \mathit{int\_op}_n(e_1, \ldots, e_n) : \texttt{int}}(\text{Op})$$

$$\frac{}{\Gamma, \Delta \vdash x : \Gamma(x)}(\text{Var}) \quad \frac{\Gamma \uplus \{x \mapsto \tau_1\}, \Delta \vdash e : \tau_2}{\Gamma, \Delta \vdash \lambda x.\, e : \tau_1 \to \tau_2}(\text{Abs}) \quad \frac{\Gamma, \Delta \vdash e_1 : \tau' \to \tau \quad \Gamma, \Delta \vdash e_2 : \tau'}{\Gamma, \Delta \vdash e_1 e_2 : \tau}(\text{App})$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma, \Delta \vdash e_n : \tau_n}{\Gamma, \Delta \vdash \langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n}(\text{Pair}) \quad \frac{\Gamma, \Delta \vdash e : \tau_1 \times \cdots \times \tau_i \times \cdots \times \tau_n}{\Gamma, \Delta \vdash \#_i(e) : \tau_i}(\text{Proj})$$

$$\frac{\Gamma, \Delta \vdash e : \tau_i}{\Gamma, \Delta \vdash \texttt{in}_i(e) : \tau_1 + \cdots + \tau_i + \cdots + \tau_n}(\text{In})$$

$$\frac{\Gamma, \Delta \vdash e : \tau_1 + \cdots + \tau_n \quad \Gamma \uplus \{x_1 \mapsto \tau_1\}, \Delta \vdash e_1 : \tau \quad \ldots \quad \Gamma \uplus \{x_n \mapsto \tau_n\}, \Delta \vdash e_n : \tau}{\Gamma, \Delta \vdash \texttt{case}\ e\ \texttt{of}\ \texttt{in}_1(x_1) \Rightarrow e_1 \,[\!]\, \ldots \,[\!]\, \texttt{in}_n(x_n) \Rightarrow e_n : \tau}(\text{Case})$$

$$\Gamma, \Delta \vdash k : \texttt{key}[\Delta(k)]\ (\text{Key}) \quad \frac{\Gamma \uplus \{x \mapsto \texttt{key}[\tau']\}, \Delta \vdash e : \tau}{\Gamma, \Delta \vdash \nu x.\, e : \tau}(\text{New})$$

$$\frac{\Gamma, \Delta \vdash e_1 : \tau \quad \Gamma, \Delta \vdash e_2 : \texttt{key}[\tau]}{\Gamma, \Delta \vdash \{e_1\}_{e_2} : \texttt{bits}[\tau]}(\text{Enc})$$

$$\frac{\Gamma, \Delta \vdash e_1 : \texttt{key}[\tau'] \quad \Gamma, \Delta \vdash e_2 : \texttt{bits}[\tau'] \quad \Gamma \uplus \{x \mapsto \tau'\}, \Delta \vdash e_3 : \tau \quad \Gamma, \Delta \vdash e_4 : \tau}{\Gamma, \Delta \vdash \texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4 : \tau}(\text{Dec})$$

Figure 2.3: Type system of $\lambda^{\rightarrow}_{\texttt{seal}}$

In what follows, we often abbreviate a sequence of the form $X_1, \ldots, X_n$ as $\tilde{X}$ and a proposition of the form $\bigwedge_{1 \le j \le m} P(Y_{1j}, \ldots, Y_{nj})$ as $P(\tilde{Y}_1, \ldots, \tilde{Y}_n)$. For example, $\tilde{k} \in \tilde{s}$ abbreviates $(k_1 \in s_1) \wedge \ldots \wedge (k_n \in s_n)$.

The following theorem and corollary state that the evaluation of a well-typed program never causes a type error.

Theorem 2.5. Suppose $\Gamma, \Delta \vdash e : \tau$ and $\emptyset, \Delta \vdash \tilde{v} : \tilde{\tau}$ for $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$. If $(s)[\tilde{v}/\tilde{x}]e \Downarrow V$ for $s = \mathit{dom}(\Delta)$, then there exist some v and Δ' such that $V = (s \uplus s')v$ and $\emptyset, \Delta \uplus \Delta' \vdash v : \tau$ for $s' = \mathit{dom}(\Delta')$.

Proof. By induction on the derivation of $(s)[\tilde{v}/\tilde{x}]e \Downarrow V$. See Appendix A.5 for details. □

Corollary 2.6 (Type Safety). If $\emptyset, \emptyset \vdash e : \tau$, then $(\emptyset)e \not\Downarrow \mathit{Error}$.

Proof. Immediate from Theorem 2.5. □

One subtle point deserves mention, concerning the relation between types and the modeling of security protocols. Since we intend to represent both principals and attackers as terms of the cryptographic λ-calculus, if we restrict our attention to only well-typed terms, we seem to run the risk of artificially (and unrealistically) restricting the power of the attackers we can model. In particular, since the calculus under this type system is strongly normalizing (i.e., every well-typed program terminates), the attackers are not Turing-complete. Moreover, there exists a specific kind of attacks—so-called "type attacks"—whose *essence* is to deceive principals into confusing values of different types.

Nevertheless, we believe that the present simple type system is flexible enough to allow typical attacks: indeed, all of the attacks we have seen so far are well-typed in the type system. As for type attacks, they are either (1) actually well-typed in the present type system, which does not distinguish nonces from keys, or (2) easily prevented using standard dynamic type checking techniques (see e.g. [Heather, Lowe, and Schneider 2000] for details).

2.6 Logical Relations for Encryption

Recall the family of expressions p_i from Example 2.2:

$$p_i = \nu z.\, \langle \{i\}_z, \lambda\{x\}_z.\, \mathtt{Some}(x \;\mathtt{mod}\; 2) \rangle$$

Suppose we want to argue that each p_i keeps its concrete value of i secret from any possible attacker. Intuitively, this is so because the only capabilities p_i provides to an attacker (at least, if that attacker can be represented as an expression of the cryptographic λ-calculus) are a ciphertext encrypting i under a key that the attacker cannot learn plus a function that will return just the least significant bit of a number encrypted with this key.

The intuition that the concrete value of i is kept secret can be formulated more precisely as a *non-interference* condition: for any i and j such that $i \;\mathtt{mod}\; 2 = j \;\mathtt{mod}\; 2$ (i.e., such that the part of the information that we do allow p_i and p_j to reveal is the same), we want to prove that p_i and p_j are behaviorally equivalent, in the following sense.

Definition 2.7 (Extensional Equivalence). We say that $\vdash e \equiv e' : \tau$, pronounced "the expressions e and e' are extensionally equivalent at type τ," if and only if both of the following conditions hold:

- $\emptyset, \emptyset \vdash e : \tau$ and $\emptyset, \emptyset \vdash e' : \tau$
- For any f with $\emptyset, \emptyset \vdash f : \tau \rightarrow \texttt{bool}$, there exist some s and s' such that one of the following conditions holds:
  - $(\emptyset) f e \Downarrow (s)\texttt{true}$ and $(\emptyset) f e' \Downarrow (s')\texttt{true}$
  - $(\emptyset) f e \Downarrow (s)\texttt{false}$ and $(\emptyset) f e' \Downarrow (s')\texttt{false}$

Essentially, this says that two expressions e and e' yield the same result under any observer function f. Although this extensional equivalence is defined for closed expressions only, it can be used to prove the more general property of *contextual equivalence* for open expressions as follows. Take any expressions e and e' of type τ and any context $C[\,]$ of type $\texttt{bool}$ with a hole of type τ. Let $\tilde{x}$ be the free variables of e and e', and let $f = \lambda x_0.\, C[x_0 \tilde{x}]$, $e_0 = \lambda \tilde{x}.\, e$, and $e'_0 = \lambda \tilde{x}.\, e'$. Then $f e_0 = f e'_0$ implies $C[e] = C[e']$. Thus, contextual equivalence of e and e' follows from extensional equivalence of e_0 and e'_0.

In the following subsections, we define three variants of the logical relation proof technique for extensional equivalence. The first one shows the basic ideas, but it is not powerful enough to prove secrecy properties of realistic programs, such as (the encoding of) the improved Needham-Schroeder public-key protocol in Section 2.3. The others are extensions of the basic logical relation, the second for addressing the issue of "a key encrypting another key" (as in Needham-Schroeder) and the third for accommodating discrepancies in the number of keys used in the programs being compared.

2.6.1 Basic Logical Relation

Extensional equivalence is difficult to prove directly because it involves a quantification over all functions f of type $\tau \rightarrow \texttt{bool}$, which are infinitely many in general. Instead, we would like prove it in a compositional manner, by showing that each part of two programs behaves equivalently. However, this approach will not suffice to prove any interesting case of extensional equivalence if we do not consider the correspondence between ciphertexts. Consider, for example, the expressions $e = \nu z.\, \langle \{\texttt{true}\}_z, \{\texttt{false}\}_z, \lambda\{x\}_z.\, \texttt{Some}(x)\rangle$ and $e' = \nu z.\, \langle \{\texttt{false}\}_z, \{\texttt{true}\}_z, \lambda\{x\}_z.\, \texttt{Some}(\texttt{not}(x))\rangle$. Although these tuples are equivalent, it cannot be shown that the third elements $\lambda\{x\}_z.\, \texttt{Some}(x)$ and $\lambda\{x\}_z.\, \texttt{Some}(\texttt{not}(x))$ are "equivalent" in this context without knowing (1) the fact that (the key bound to) z is kept secret throughout the whole programs and (2) the relation between values encrypted by z, that is, $\{\texttt{true}\}_z$ in e corresponds to $\{\texttt{false}\}_z$ in e' and $\{\texttt{false}\}_z$ to $\{\texttt{true}\}_z$. (Recall the correspondence between $\{3\}_z$ and $\{5\}_z$ in the example in Section 2.1.)

Thus, we generalize $\vdash e \equiv e' : \tau$ to the *logical relation* $\varphi \vdash e \sim e' : \tau$, in which the parameter φ is a *relation environment*: a mapping from keys to relations, associating to each secret key k a relation $\varphi(k)$ between the values that may be encrypted by k. Given φ, the family of relations $\varphi \vdash e \sim e' : \tau$ is defined by induction on τ as follows:

- Two functions are related if and only if they map any related arguments to related results.
- Two pairs are related if and only if their corresponding elements are related.

$$
\begin{array}{lcl}
\varphi \vdash (s)i \sim (s')i' : \mathtt{int} & \iff & i = i' \\
\varphi \vdash (s)f \sim (s')f' : \tau_1 \to \tau_2 & \iff & f = \lambda x.\, e \text{ and } f' = \lambda x.\, e' \text{ where} \\
& & \varphi \uplus \psi \vdash (s \uplus t)[v/x]e \approx (s' \uplus t')[v'/x]e' : \tau_2 \text{ for any} \\
& & \varphi \uplus \psi \vdash (s \uplus t)v \sim (s' \uplus t')v' : \tau_1 \text{ with } dom(\psi) \subseteq t \cap t' \\
\varphi \vdash (s)p \sim (s')p' : \tau_1 \times \cdots \times \tau_n & \iff & p = \langle v_1, \ldots, v_n \rangle \text{ and } p' = \langle v'_1, \ldots, v'_n \rangle \text{ where} \\
& & \varphi \vdash (s)\tilde{v} \sim (s')\tilde{v}' : \tilde{\tau} \\
\varphi \vdash (s)t \sim (s')t' : \tau_1 + \cdots + \tau_n & \iff & t = \mathtt{in}_i(v) \text{ and } t' = \mathtt{in}_i(v') \text{ where} \\
& & \varphi \vdash (s)v \sim (s')v' : \tau_i \\
\varphi \vdash (s)k \sim (s')k' : \mathtt{key}[\tau] & \iff & k = k' \text{ and } k \in s \cap s' \text{ and } k \notin dom(\varphi) \\
\varphi \vdash (s)c \sim (s')c' : \mathtt{bits}[\tau] & \iff & c = \{v\}_k \text{ and } c' = \{v'\}_k \text{ where either} \\
& & k \in dom(\varphi) \text{ and } k \in s \cap s' \text{ and } (v, v') \in \varphi(k), \text{ or else} \\
& & k \notin dom(\varphi) \text{ and } k \in s \cap s' \text{ and } \varphi \vdash (s)v \sim (s')v' : \tau \\
\\
\varphi \vdash (s)e \approx (s')e' : \tau & \iff & (s)e \Downarrow (s \uplus t)v \text{ and } (s')e' \Downarrow (s' \uplus t')v' \text{ where} \\
& & \varphi \uplus \psi \vdash (s \uplus t)v \sim (s' \uplus t')v' : \tau \text{ with } dom(\psi) \subseteq t \cap t'
\end{array}
$$

Figure 2.4: Basic logical relation for perfect encryption

- Two tagged values are related if and only if their tags are equal and their bodies are related.
- Two keys are related if and only if they are identical and not secret. Here, the set of secret keys is identified with the domain of φ (see below).
- Two ciphertexts $\{v\}_k$ and $\{v'\}_{k'}$ are related if and only if $k = k'$ and either:
  - k is secret and $(v, v') \in \varphi(k)$, or else
  - k is not secret and v and v' are related.

Intuitively, $\varphi \vdash v \sim v' : \tau$ means "under any possible attackers, the values v and v' behave equivalently and furthermore preserve the invariant that values encrypted by any secret key k are related by $\varphi(k)$." It is this invariant which makes the logical relation work at all: as is often the case in inductive proofs, requiring this extra condition helps us in proving the final goal, i.e., extensional equivalence. Note that, in the definition above, secret keys are not related even if they are identical, because if they were related, an attacker would be able to encrypt arbitrary values under the keys and break the invariance. In other words, φ represents the restriction on the attackers' knowledge that each $k \in dom(\varphi)$ is unknown to them and, furthermore, for each $(v, v') \in \varphi(k)$, the ciphertexts $\{v\}_k$ and $\{v'\}_k$ are indistinguishable to the attackers. (See Section 2.8 for some discussion of the issue of equality for ciphertexts.)

As for expressions, arbitrary expressions are related if and only if they evaluate to values that, in turn, are related under a relation environment extended with the fresh keys that were generated during evaluation.

The formal definition of the logical relation is given in Figure 2.4. $\varphi \vdash (s)v \sim (s')v' : \tau$ and $\varphi \vdash (s)e \approx (s')e' : \tau$ are logical relations for values and expressions, respectively. The sets s and s', respectively, denote the keys generated so far on the left and right hand sides.

Strictly speaking, the relation environment φ should take s, s' and a (partial) mapping Δ from keys to types as parameters. Then, for each $k \in s \cap s'$, $\varphi(k)$ is a relation on two values v, v' of type

$\Delta(k)$ such that $Keys(v) \subseteq s$ and $Keys(v') \subseteq s'$. In order to simplify the notations, however, we omit s, s' and Δ since they are obvious from the context.

Example 2.8. For the e and e' in the previous example, let $\tau = \texttt{bits}[\texttt{bool}] \times \texttt{bits}[\texttt{bool}] \times (\texttt{bits}[\texttt{bool}] \to \texttt{option}[\texttt{bool}])$. Then, $\emptyset \vdash (\emptyset)e \approx (\emptyset)e' : \tau$. To prove this, let $t = t' = \{k\}$ and $\psi = \{k \mapsto \{(\texttt{true}, \texttt{false}), (\texttt{false}, \texttt{true})\}\}$ in the definition of $\emptyset \vdash (\emptyset)e \approx (\emptyset)e' : \tau$.

Example 2.9. For the p_i in Example 2.2, let $\tau = \texttt{bits}[\texttt{int}] \times (\texttt{bits}[\texttt{int}] \to \texttt{option}[\texttt{int}])$. Then, $\emptyset \vdash (\emptyset)p_i \approx (\emptyset)p_j : \tau$ for any i and j with $i \mathbin{\texttt{mod}} 2 = j \mathbin{\texttt{mod}} 2$. (Here, we define $\varphi \vdash (s)i \sim (s')i' : \texttt{int} \iff i = i'$.) To prove this, let $t = t' = \{k\}$ and $\psi(k) = \{(i, j)\}$ in the definition of $\emptyset \vdash (\emptyset)p_i \approx (\emptyset)p_j : \tau$.

The following theorem and corollary state that the logical relation indeed implies extensional equivalence.

Theorem 2.10. Let $\Gamma, \Delta \vdash e : \tau$ for $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$, and suppose that $\varphi \vdash (s)\tilde{v} \sim (s')\tilde{v}' : \tilde{\tau}$ with $dom(\varphi) \cap dom(\Delta) = \emptyset$ and $s, s' \supseteq dom(\varphi) \uplus dom(\Delta)$. Then, $\varphi \vdash (s)[\tilde{v}/\tilde{x}]e \approx (s')[\tilde{v}'/\tilde{x}]e : \tau$. That is, any expression is related to itself when its free variables are substituted with related values.

Proof. By induction on the structure of e. See Appendix A.6 for details. □

Corollary 2.11 (Soundness of Logical Relation). If $\emptyset \vdash (\emptyset)e \approx (\emptyset)e' : \tau$, then $\vdash e \equiv e' : \tau$.

Proof. Straightforward since any observer function f is related to itself by Theorem 2.10. See Appendix A.7 for details. □

2.6.2 Extended Logical Relation

In the basic logical relation above, a relation between values encrypted by each secret key k is given by the relation environment φ. However, φ gives us no information about the relations that should be associated with fresh keys that are still to be generated in the future. As a result, the basic logical relation technique fails to prove the equivalence of some important examples that are, in fact, equivalent: in particular, we cannot prove the security of the improved version of the Needham-Schroeder public-key protocol from Section 2.3.2.

For a simpler example showing where the proof technique goes wrong, consider a program $q_i = \nu x.\, \langle \lambda\_.\, \nu y.\, \{y\}_x,\ \lambda\{y'\}_x.\, \texttt{Some}(\{i\}_{y'})\rangle$ for some secret integer i. Since the key x (to be precise, the key bound to the variable x) is kept secret, the key $y = y'$ is also kept secret, so i is kept secret. Therefore, q_3 and q_5, say, should be equivalent. But in order to prove this by using the basic logical relation above, we would have to give a relation between values encrypted by the key k bound to x. Since the key k' that will be bound to y is not yet determined, we cannot specify a relation like $\varphi(k) = \{(k', k')\}$. Thus, q_3 and q_5 cannot be related.

This problem can be addressed by refining the definition of the logical relation a little, i.e., parameterizing the relation environment φ with respect to sets s and s' of keys—representing the sets of keys that will have been generated at some point of interest in the future—as well as the relation environment ψ that will be in effect at that time. (The definition of "a relation environment parametrized by another relation environment" is recursive, but such entities can be constructed inductively, just as elements of a recursive type can be.) Then, in the example above, for instance, we can specify the needed relation as $\varphi^{\psi}_{s,s'}(k) = \{(k',k') \mid \psi^{\chi}_{t,t'}(k') = \{(3,5)\}$ for any t,t' and $\chi\}$. Accordingly, we extend the definition of the logical relation for ciphertext types to:

$$\begin{array}{l}
\varphi \vdash (s)c \sim (s')c' : \mathtt{bits}[\tau] \iff \\
c = \{v\}_k \text{ and } c' = \{v'\}_k \text{ for some } v, v', k \text{ such that either} \\
k \in \mathit{dom}(\varphi) \text{ and } k \in s \cap s' \text{ and } (v,v') \in \varphi^{\varphi}_{s,s'}(k)\text{, or else} \\
k \notin \mathit{dom}(\varphi) \text{ and } k \in s \cap s' \text{ and } \varphi \vdash (s)v \sim (s')v' : \tau
\end{array}$$

Interestingly, even after this extension, the propositions in Section 2.6.1 (and their proofs!) continue to hold *without change*—as long as we impose the condition that φ in $\varphi^{\psi}_{s,s'}(k)$ is monotonic with respect to extension of s, s', and ψ. Intuitively, this condition guarantees that values related once do not become unrelated as fresh keys are generated in the future. This is not the case if we take $\varphi^{\psi}_{s,s'}(k) = \{(k',k') \mid k' \notin s \cup s'\}$, for example. The monotonicity condition excludes such anomalies. Formally, we require that each φ satisfies

$$\varphi^{\psi}_{s,s'}(k) \subseteq \varphi^{\psi \uplus \chi}_{s \uplus t, s' \uplus t'}(k)$$

for any s, s', t and t' with $s \cap t = \emptyset$ and $s' \cap t' = \emptyset$, and for any ψ and χ with $\mathit{dom}(\psi) \subseteq s \cap s'$ and $\mathit{dom}(\chi) \subseteq t \cap t'$. Technically, this condition is needed in the proof of Lemma A.7 (weakening of logical relation) when τ is a ciphertext type. We refer to this condition as "φ is monotonic."

Example 2.12. For the previous q_i, let $\tau = (\mathtt{unit} \to \mathtt{bits}[\mathtt{key}[\mathtt{int}]]) \times (\mathtt{bits}[\mathtt{key}[\mathtt{int}]] \to \mathtt{option}$ $[\mathtt{bits}[\mathtt{int}]])$. Then, $\emptyset \vdash (\emptyset)q_i \approx (\emptyset)q_j : \tau$ for any i and j. To prove this, let $t = t' = \{k\}$ and

$$\psi^{\varphi}_{s,s'}(k) = \big\{(k',k') \;\big|\; \varphi^{\chi}_{t,t'}(k') = \{(i,j)\} \text{ for any } t,t' \text{ and } \chi\big\}$$

in the definition of $\emptyset \vdash (\emptyset)q_i \approx (\emptyset)q_j : \tau$. It is straightforward to check that ψ is monotonic. Hence $\vdash q_i \equiv q_j : \tau$.

Example 2.13. Let us see how to prove the correctness of the encoding in Section 2.3.2 of the improved Needham-Schroeder public-key protocol, using the extended logical relation.

First, in order for the encoding NS'_i to be well-typed at all, values encrypted by the keys z_B and z_x need to be tagged. (The tags are underlined.)

$$\begin{array}{l}
\nu z_A.\, \nu z_B.\, \nu z_E. \\
\langle \lambda x.\, \{x\}_{z_A}, \lambda x.\, \{x\}_{z_B}, z_E, \\
\;\langle B, \lambda\{\underline{\mathtt{in}_1}(\langle N_A, A\rangle)\}_{z_B}.\, \nu N_B. \\
\qquad \mathtt{Some}(\langle\{\langle N_A, N_B, B\rangle\}_{z_A},\, \lambda\{\underline{\mathtt{in}_2}(N_B)\}_{z_B}.\, \mathtt{Some}(\{i\}_{N_B})\rangle)\rangle, \\
\;\langle \lambda X.\, \nu N_A.\, \langle\{\underline{\mathtt{in}_1}(\langle N_A, A\rangle)\}_{z_x}, \\
\qquad\qquad \lambda\{N_A, N_x, X\}_{z_A}.\, \mathtt{Some}(\{\underline{\mathtt{in}_2}(N_x)\}_{z_x})\rangle\rangle\rangle
\end{array}$$

Call this expression NS''_i. It can be given the type

$$\begin{array}{l}
(\tau_1 \to \mathtt{bits}[\tau_1]) \times (\tau_2 \to \mathtt{bits}[\tau_2]) \times \mathtt{key}[\tau_2] \times \\
(\mathtt{nam} \times (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\tau_1] \times \\
\qquad\qquad\qquad (\mathtt{bits}[\tau_2] \to \mathtt{option}[\mathtt{bits}[\mathtt{int}]])])) \times \\
(\mathtt{nam} \to (\mathtt{bits}[\tau_2] \times (\mathtt{bits}[\tau_1] \to \mathtt{option}[\mathtt{bits}[\tau_2]])))
\end{array}$$

where nam is actually just int and

$$
\begin{array}{rcl}
\tau_1 & = & \mathtt{key}[\sigma] \times \mathtt{key}[\mathtt{int}] \times \mathtt{nam} \\
\tau_2 & = & \mathtt{key}[\sigma] \times \mathtt{nam} + \mathtt{key}[\mathtt{int}]
\end{array}
$$

for some σ. Call this type τ.

Now, NS''_i and NS''_j can be related (and are therefore extensionally equivalent) for any i and j by letting $t = t' = \{k_A, k_B, k_E\}$ and

$$
\begin{array}{rcl}
\psi^{\varphi}_{s,s'}(k_A) & = & \big\{(v, v') \mid \varphi \vdash (s)v \sim (s')v' : \tau_1\big\} \\
& \cup & \big\{(\langle N_A, N_B, B\rangle, \langle N_A, N_B, B\rangle) \mid \\
& & \quad \varphi^{\chi}_{t,t'}(N_A) = r \text{ and } \varphi^{\chi}_{t,t'}(N_B) = \{(i,j)\} \\
& & \quad \text{for any } t, t' \text{ and } \chi\big\} \\
\psi^{\varphi}_{s,s'}(k_B) & = & \big\{(v, v') \mid \varphi \vdash (s)v \sim (s')v' : \tau_2\big\} \\
& \cup & \big\{(\mathtt{in}_1(\langle N_A, A\rangle), \mathtt{in}_1(\langle N_A, A\rangle)) \mid \\
& & \quad \varphi^{\chi}_{t,t'}(N_A) = r \text{ for any } t, t' \text{ and } \chi\big\} \\
& \cup & \big\{(\mathtt{in}_2(N_B), \mathtt{in}_2(N_B)) \mid \\
& & \quad \varphi^{\chi}_{t,t'}(N_B) = \{(i,j)\} \text{ for any } t, t' \text{ and } \chi\big\}
\end{array}
$$

for some r in the definition of $\emptyset \vdash (\emptyset)NS''_i \approx (\emptyset)NS''_j : \tau$.

It is straightforward, by the way, to check that $\mathit{Good}(NS''_i)$ evaluates to $\mathtt{Some}(\{i\}_{N_B})$ for some fresh N_B. So this system is indeed both safe (from attacks that can be modeled in our setting) and sound.

2.6.3 Another Extended Logical Relation

Another way of extending the logical relation is to let a relation environment φ map a *pair* of secret keys—rather than one secret key—to a relation between values encrypted by those keys. Consider, for example, the following two expressions.

$$
\begin{array}{rcl}
e & = & \nu x.\, \langle \{1\}_x, \{2\}_x, \\
& & \quad \lambda z.\, \mathtt{let}\ \{i\}_x = z\ \mathtt{in}\ \mathtt{Some}(i\ \mathtt{mod}\ 2)\ \mathtt{else}\ \mathtt{None}\rangle \\
e' & = & \nu x.\, \nu y.\, \langle \{3\}_x, \{4\}_y, \\
& & \quad \lambda z.\, \mathtt{let}\ \{i\}_x = z\ \mathtt{in}\ \mathtt{Some}(i\ \mathtt{mod}\ 2)\ \mathtt{else} \\
& & \qquad \mathtt{let}\ \{j\}_y = z\ \mathtt{in}\ \mathtt{Some}(j\ \mathtt{mod}\ 2)\ \mathtt{else}\ \mathtt{None}\rangle
\end{array}
$$

They should be extensionally equivalent because, in both expressions, the keys x and y are kept secret, and therefore the only way to use the first and second elements of the tuples is to apply the third elements, which return the same value. However, this extensional equivalence cannot be proved by using either of the logical relations above, because the second elements are encrypted by different keys.

This problem can be solved by letting a relation environment φ take a *pair* of secret keys, like $\varphi(k_x, k_x) = \{(1,3)\}$ and $\varphi(k_x, k_y) = \{(2,4)\}$ for example, and extending the definition of the logical relation accordingly, letting

$$\begin{array}{l}
\varphi \vdash (s)c \sim (s')c' : \mathtt{bits}[\tau] \iff \\
c = \{v\}_k \text{ and } c' = \{v'\}_{k'} \text{ for some } v, v', k, k' \text{ such that either} \\
(k, k') \in dom(\varphi) \text{ and } (k, k') \in s \times s' \text{ and } (v, v') \in \varphi(k)\text{, or else} \\
(k, k') \notin dom(\varphi) \text{ and } (k, k') \in s \times s' \text{ and } \varphi \vdash (s)v \sim (s')v' : \tau \\
\\
\varphi \vdash (s)k \sim (s')k' : \mathtt{key}[\tau] \iff \\
k = k' \text{ and } (k, k) \in s \times s' \text{ and} \\
(k, k'') \notin dom(\varphi) \text{ and } (k'', k) \notin dom(\varphi) \text{ for any } k''
\end{array}$$

and so forth. Again, it is straightforward to adapt the results in Section 2.6.1 for this extension. (It may seem somewhat surprising that the results in Section 2.6.1 are so easily adapted to different definitions of logical relations. This stems from the fact that the proofs of the propositions do not actually depend on the internal structure of relation environments.)

2.7 Related Work

Numerous approaches to formal verification of security protocols have been explored in the literature (e.g. [Meadows 2000; Meadows 1995; Millen 2004; Heintze and Clarke 1999]). Of these, the spi-calculus [Abadi and Gordon 1999] is one of the most powerful; it comes equipped with useful techniques such as bisimulation [Abadi and Gordon 1998; Boreale, De Nicola, and Pugliese 2002] for proving behavioral equivalences and static typing for guaranteeing secrecy [Abadi 1999] and authenticity [Gordon and Jeffrey 2001]. We are not in a position yet to claim that our approach is superior to the spi-calculus (or any other existing approach); rather, our goal has been simply to explore how standard techniques for reasoning about type abstraction can be adapted to the task of reasoning about encryption, in particular about security protocols. For this study, λ-calculus offers an attractive starting point, since it is in this setting that relational parametricity is best understood. Of course, the cost of this choice is that we depend on the ability of the λ-calculus to encode communication and concurrency by function application and interleaving. Since this encoding is not fully abstract (processes are linear by default while functions are not), a process that is actually secure is not always encoded as a secure λ-term. Any attacks that we discover for the encoded term must be reality-checked against the original process (cf. the false attack on the encoding of the ffgg protocol in Section 2.3). However, if the encoding of a process *can* be proved secure, then the process itself should also be secure, at least against our notion of attackers (cf. the correctness proof of the improved Needham-Schroeder public-key protocol in Section 2.3 and the discussion in Section 2.8).

Formalizing and proving secrecy as non-interference—i.e., equivalence between instances of a program with different secret values—has been a popular approach both in the security community and in the programming language community. Non-interference reasoning in protocol verification can be found in [Volpano 1999; Ryan and Schneider 1999; Durante, Focardi, and Gorrieri 1999], among others.

There have also been many proposals for using techniques from programming languages—in particular, static typing—to guarantee security of programs. For example, Heintze and Riecke [1998] proposed λ-calculus with type-based information flow control, and proved a non-interference property—that a value of high security does not leak to any context of low security—using a logical relation. Other work in this line includes [Pottier 2002; Pottier and Simonet 2002; Hennessy and Riely 2000; Honda and Yoshida 2002; Smith and Volpano 1998]. Most of those approaches aim to statically exclude attackers coming into a system, rather than to dynamically protect a program from attackers outside the system. An exception is the work cited above on static typing for secrecy and authenticity in spi-calculus.

Originally, logical relations were developed in the domain of denotational semantics for the purpose of establishing various kinds of correspondence in mathematical models of typed λ-calculi (see [Mitchell 1996, Chapter 8], for example). In this setting, defining or using a logical relation requires establishing or understanding the denotational model(s) on which the logical relation is defined. In addition, the soundness of such relational reasoning (with respect to the operational semantics) depends on that of the denotational model. We circumvented these issues by adopting the approach of *syntactic* logical relations [Pitts 2000; Birkedal and Harper 1999], i.e., (a variant of) logical relations based on a term model of a language.

Since the cryptographic λ-calculus has a key generation primitive, we must be able to reason about generative names. For this purpose, we adapted Stark's work on λ-calculus with name generation [Stark 1994] in formulating both the semantics in Section 2.4 and the logical relation in Section 2.6.1. A technical difference of our adaptation from Stark's original is that he introduced bijections while we rely on α-conversion in order to manage the possible differences between names generated by each of the related terms. In addition, the combination of the logical relation for name generation and that for type abstraction [Reynolds 1983] gave rise to a new problem—namely, how to specify fresh keys that have not yet been generated. This issue is critical when a fresh key is encrypted by another key, which is often the case in programs exchanging keys. We addressed this problem in Section 2.6.2 by extending the logical relation in a non-trivial way. The same technique would also apply for other purposes such as treating "references to references" in establishing logical relations for ML-like references [Pitts and Stark 1998].

Harper and Lillibridge [personal communication, July 2000] have independently developed a *typed seal calculus* that is closely related to our cryptographic λ-calculus. Their work mainly focuses on encoding *sealing* [Morris 1973a] primitives in terms of other mechanisms such as exceptions and references and vice versa, rather than establishing techniques for reasoning about secrecy properties of programs using sealing.

Recently, Goubault-Larrecq, Lasota, Nowak, and Zhang [2004] gave an elegant reformulation of our logical relations—based on Stark's categorical semantics of fresh name generation [Stark 1996] and Plotkin et al.'s lax logical relations [Plotkin, Power, Sannella, and Tennent 2000]—and proved it to be *complete* with respect to contextual equivalence. Besides completeness, their development is parametric with respect to addition of primitive types and values: that is, all the results are proved to remain valid even when the language is extended, for example, with asymmetric encryption obeying some algebraic laws.

2.8 Future Work

Recursive Functions and Recursive Types. It can be shown (from Theorem 2.10 and the definition of $\varphi \vdash (s)e \approx (s')e' : \tau$) that, under our simple type system, the evaluation of a well-typed expression *always* terminates. Therefore, recursive functions cannot be written. Indeed, introducing recursive functions breaks the soundness proof of the logical relations; introducing recursive *types* breaks the very definition of the logical relations. This problem is of concern because it suggests that our approach would not be *sound* with respect to attacks that rely on recursion. (If, indeed, there are any such attacks in reality: observe that, for each particular λ-term, if there exists an attack that uses recursion to reveal a secret within a finite amount of time, then the same attack should also be possible without using recursion.) We expect that this limitation can be removed by incorporating the theory of logical relation for λ-calculus with recursive functions and/or recursive types (e.g., [Birkedal and Harper 1999]).

Equality for Ciphertexts. In our calculus, we did not introduce any construct to test ciphertexts for equality. This lack of equality for ciphertexts can be a weakness of our development in this chapter for the same reason as the lack of recursion may be. For example, $\nu x.\, \{3\}_x$ and $\nu x.\, \{5\}_x$ are equivalent in our calculus, but an attacker may discover the difference just by comparing the ciphertexts as bit strings. Using non-deterministic encryption (a.k.a. random encryption and probabilistic encryption) for implementation does not solve this problem: for instance, $\langle \{123\}_k, \{123\}_k \rangle$ and $\texttt{let}\ x = \{123\}_k\ \texttt{in}\ \langle x, x \rangle$ would be inequivalent under non-deterministic encryption, but they *are* equivalent in the present calculus.

The issue of ciphertext equality is closely related to that of polymorphic equality in the standard theory of relational parametricity for type abstraction [Wadler 1989, Section 3.4]. The solution would also be similar—i.e., require the corresponding relation $\varphi(k)$ to respect equality—but it remains to see what effects it will have on reasoning about information hiding by encryption.

State and Linearity. Although real programs often have some kind of state or linearity (in the sense of linear logic that some of the "resources" which they offer can be exploited only once), our framework does not take them into account. Thus, it cannot prove the security of a program depending on them.

For example, consider an expression $p_i = \nu z.\, \lambda x.\, \texttt{let}\ \{\_\}_z = x\ \texttt{in}\ \texttt{in}_1(i)\ \texttt{else}\ \texttt{in}_2(z)$ for some secret integer i. Although this program leaks the secret integer i under the attacker $f = \lambda p.\, \texttt{let}\ \texttt{in}_2(z) = p\{0\}_k\ \texttt{in}\ \texttt{let}\ \texttt{in}_1(i) = p\{0\}_z\ \texttt{in}\ \texttt{Some}(i)$, it is actually secure if we impose the constraint that the function $\lambda x.\, \ldots$ is used linearly (i.e., applied only once). A similar example can be given using an ML-like reference cell.

Although we have not yet come across a realistic program whose security depends on its state or linearity in a crucial manner (maybe because such a "dangerous" design is avoided *a priori* by engineering practice?), we expect that this issue can be addressed, too, by incorporating the theory of logical relation for λ-calculus with state or linearity [Pitts and Stark 1998; Bierman, Pitts, and Russo 2000].

Moreover, pursuing this direction of adapting logical relations for linearity and state might lead to a theory of relational parametricity for process calculi with some form of information hiding, such as polymorphic π-calculus [Turner 1995] and spi-calculus [Abadi and Gordon 1999]. We conjecture that, in combination with a big-step, evaluation semantics of process calculi [Pitts and Ross 1998], this approach might lead to a more systematic, structural method than the bisimulation-based techniques that have been explored in the past [Pierce and Sangiorgi 2000; Boreale, De Nicola, and Pugliese 2002] for proving equivalence between concurrent programs with information hiding.

Beyond Secrecy. We were able to prove secrecy properties of security protocols by means of logical relations because (1) our logical relations are a means of proving (contextual) equivalence and (2) equivalence leads to secrecy via non-interference. A natural question is whether we might be able to prove security properties *other* than secrecy—such as authenticity, anonymity, etc.—in a similar fashion via logical relations. The general idea would be to prove the equivalence between (encodings of) a real system and an ideal one whose "security" is obvious (cf. [Durante, Focardi, and Gorrieri 2000], for instance); further study is needed to see what kinds of problems can be addressed in this manner using our framework.

Type Abstraction via Encryption. While we have focused here on adapting the theory of type abstraction into encryption, it is also interesting to think of using the techniques of encryption for type abstraction. Specifically, it may be possible to implement type abstraction by means of encryption, in order to protect secrets not only from well-typed programs, but also from arbitrary attackers—in other words, to combine polymorphism with dynamic typing without losing type abstraction. That would enable us to write programs in a high-level language using type abstraction and translate them into a lower-level code using encryption. Then, the problem is whether and how such translation is possible, preserving the abstraction. In an earlier version of this work, we suggested one possibility for such a translation [Pierce and Sumii 2000, Section 4], but proved nothing about it. The results in the present chapter—in particular, the logical relations in Section 2.6—may help improve our understanding of this issue.

Chapter 3

A Bisimulation for Perfect Encryption and Dynamic Sealing

Overview

We define $\lambda_{\texttt{seal}}$, an untyped call-by-value λ-calculus with primitives for protecting abstract data by *sealing*, and develop a bisimulation proof method that is sound and complete with respect to contextual equivalence. This provides a formal basis for reasoning about data abstraction in open, dynamic settings where static techniques such as type abstraction and logical relations are not applicable.

Results in this chapter are to appear in Sumii and Pierce [2004a].

3.1 Introduction

Dynamic sealing: Birth, death, and rebirth

Sealing is a linguistic mechanism for protecting abstract data. As originally proposed by Morris [1973a, 1973b], it consists of three constructs: seal creation, sealing, and unsealing. A fresh seal is created for each module that defines abstract data. Data is sealed when it is passed out of the module, so that it cannot be inspected or modified by outsiders who do not know the seal; the data is unsealed again when it comes back into the module, so that it can be manipulated concretely. Data abstraction is preserved as long as the seal is kept local to the module.

Originally, sealing was a dynamic mechanism. Morris [1973b] also proposed a static variant, in which the creation and use of seals at module boundaries follow a restricted pattern that can be verified by the compiler, removing the need for run-time sealing and unsealing. Other researchers found that a similar effect could be obtained by enriching a static type system with mechanisms for *type abstraction* (see CLU [Liskov 1993], for example). Type abstraction became the primary method for achieving data abstraction in languages from CLU to the present day. It is also well understood via the theory of existential types [Mitchell and Plotkin 1988].

Recently, however, as programming languages and the environments in which they operate become more and more open—e.g., addressing issues of persistence and distribution—dynamic sealing is being rediscovered. For example, Rossberg [2003] proposes to use a form of dynamic sealing to allow type abstraction to coexist with dynamic typing; Leifer, Peskine, Sewell, and

Wansbrough [2003] use hashes of implementations of abstract types to protect abstractions among different programs running on different sites; Dreyer, Crary, and Harper [2003] use a variant of sealing (somewhere between static and dynamic) to give a type-theoretic account of ML-like modules and functors; finally, we [Pierce and Sumii 2000] have proposed a translation (conjectured to be fully abstract) of System-F-style type abstraction into dynamic sealing.

Another reason for the renewal of interest in sealing is that it happens to coincide with *perfect encryption* (under shared-key cryptography), that is, with an ideal encryption scheme where a ciphertext can be decrypted only if the key under which it was encrypted is known explicitly. Perfect encryption is a common abstraction in current research on both systems security and programming languages, for example in modeling and reasoning about cryptographic protocols (e.g., the spi-calculus [Abadi and Gordon 1999]).

Problem

Although interest in dynamic sealing is reviving, there remains a significant obstacle to its extensive study: the lack of sufficiently powerful methods for reasoning about sealing. First, to the best of our knowledge, there has been no work at all on proof techniques for sealing in untyped sequential languages. There are several versions of bisimulation for the spi-calculus, but encoding other languages such as λ-calculus into spi-calculus raises the question of what abstraction properties are preserved by the encoding itself. Indeed, standard encodings of λ-calculus into π-calculus [Milner 1999] are not *fully abstract*, i.e., do not preserve equivalence. Second, even in statically typed settings, the published techniques for obtaining abstraction properties are in general very weak. For instance, the first two [Rossberg 2003; Leifer, Peskine, Sewell, and Wansbrough 2003] of the works cited above use (variants of) the colored brackets of Zdancewic et al. [Grossman, Morrisett, and Zdancewic 2000] but only prove (or even state) abstraction properties for cases where abstract data is published *by itself* with no interface functions provided (i.e., once sealed, data is never unsealed).

Abstraction as equivalence

We aim to establish a method for proving abstraction of programs using dynamic sealing in an untyped setting. To this end, let us first consider how to state the property of abstraction in the first place. Take, for example, the following module implementing complex numbers in an imaginary ML-like language.

```
module PolarComplex =
  abstype t = real * real
  let from_re_and_im : real * real -> t =
    fun (x, y) ->
      (sqrt(x * x + y * y), atan2(y, x))
  let to_re_and_im : t -> real * real =
    fun (r, t) ->
      (r * cos(t), r * sin(t))
  let multiply : t * t -> t =
    fun ((r1, t1), (r2, t2)) ->
      (r1 * r2, t1 + t2)
```

```
end
```

Using dynamic sealing instead of type abstraction, this module can be written as follows for some secret seal `k`.

```
module PolarComplex =
  let from_re_and_im =
    fun (x, y) ->
      let z = (sqrt(x * x + y * y), atan2(y, x)) in
      <seal z under k>
  let to_re_and_im =
    fun z ->
      let (r, t) = <unseal z under k> in
      (r * cos(t), r * sin(t))
  let multiply =
    fun (z1, z2) ->
      let (r1, t1) = <unseal z1 under k> in
      let (r2, t2) = <unseal z2 under k> in
      let z = (r1 * r2, t1 + t2) in
      <seal z under k>
end
```

Now, the question is: Is this use of sealing correct? That is, does it really protect data abstraction? In particular, can we show that this module has the same external behavior as another sealed module that also implements complex numbers, e.g., the following module with another secret seal `k'`?

```
module CartesianComplex =
  let from_re_and_im =
    fun (x, y) ->
      <check that x and y are real numbers>;
      let z = (x, y) in <seal z under k'>
  let to_re_and_im =
    fun z ->
      let (x, y) = <unseal z under k'> in (x, y)
  let multiply =
    fun (z1, z2) ->
      let (x1, y1) = <unseal z1 under k'> in
      let (x2, y2) = <unseal z2 under k'> in
      let z = (x1 * x2 - y1 * y2,
               x1 * y2 + x2 * y2) in
      <seal z under k'>
end
```

Formally, we want to show the contextual equivalence [Morris 1968] of the two modules `PolarComplex` and `CartesianComplex`. In general, however, it is difficult to *directly* prove contextual equivalence because it demands that we consider an infinite number of contexts.

Equivalence by bisimulation

To overcome this difficulty, we define a notion of *bisimulation* for our language (by extending applicative bisimulation [Abramsky 1990]) and use it as a tool for proving contextual equivalence. Essentially, a bisimulation records a set of pairs of "corresponding values" of two different programs. In the example of `PolarComplex` and `CartesianComplex`, the bisimulation is (roughly):

$$
\begin{aligned}
& \{(\texttt{PolarComplex}, \texttt{CartesianComplex})\} \\
\cup\ & \{(\texttt{PolarComplex.from\_re\_and\_im}, \texttt{CartesianComplex.from\_re\_and\_im}), \\
& \ (\texttt{PolarComplex.to\_re\_and\_im}, \texttt{CartesianComplex.to\_re\_and\_im}), \\
& \ (\texttt{PolarComplex.multiply}, \texttt{CartesianComplex.multiply})\} \\
\cup\ & \{((x, y), (x, y)) \mid x, y \text{ real numbers}\} \\
\cup\ & \{(\{(r, \theta)\}_k, \{(r \cos\theta, r \sin\theta)\}_{k'}) \mid r \geq 0\}
\end{aligned}
$$

The first part is the modules themselves. The second part is the individual elements of the modules. The third is arguments of `from_re_and_im` as well as results of `to_re_and_im`. The last is the representations of complex numbers sealed under k or k', where $\{\ \}$ denotes sealing.

From the soundness of bisimulation, we obtain the contextual equivalence of the two modules. Furthermore, our bisimulation is *complete*: if two programs are contextually equivalent, then there always exists a bisimulation between them. This means that (at least in theory) we can use bisimulation to prove *any* valid contextual equivalence.

Contribution

The main contribution of this work is a sound and complete bisimulation proof method for contextual equivalence in an untyped functional language with dynamic sealing. Along the way, we are led to refine the usual contextual equivalence to account for the variations in observing power induced by the context's knowledge (or ignorance) of the seals used in observed terms.

Parts of our theory are analogous to bisimulation techniques developed for the spi-calculus [Abadi and Gordon 1998; Borgström and Nestmann 2002; Boreale, De Nicola, and Pugliese 2002; Abadi and Fournet 2001]. However, our bisimulation is technically simpler and thus more suitable for reasoning about dynamic sealing for data abstraction in sequential languages. Furthermore, our setting requires us to extend even the definition of contextual equivalence in a natural but significant way, as discussed in Section 3.3.

Structure of the chapter

The rest of this chapter is structured as follows. Section 3.2 formalizes the syntax and the semantics of our language, $\lambda_{\texttt{seal}}$. Section 3.3 defines a suitable notion of contextual equivalence. Section 3.4 presents our bisimulation and gives several examples, including the complex number packages discussed above and an encoding of the Needham-Schroeder-Lowe key exchange protocol. Section 3.5 proves soundness and completeness of the bisimulation with respect to contextual equivalence. Section 3.7 discusses related work, and Section 3.8 sketches future work.

$d, e ::=$	term
x	variable
$\lambda x.\, e$	function
$e_1\, e_2$	application
c	constant
$op(e_1, \ldots, e_n)$	primitive
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditional branch
$\langle e_1, \ldots, e_n \rangle$	tupling
$\#_i(e)$	projection
k	seal
$\nu x.\, e$	fresh seal generation
$\{e_1\}_{e_2}$	sealing
$\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$	unsealing
$u, v, w ::=$	value
$\lambda x.\, e$	function
c	constant
$\langle v_1, \ldots, v_n \rangle$	tuple
k	seal
$\{v\}_k$	sealed value

Figure 3.1: Syntax of $\lambda_{\texttt{seal}}$

Notation

Throughout the chapter, we use overbars as shorthands for sequences—e.g., we write $\overline{x}$ and $(\overline{v}, \overline{v}')$ instead of $x_1, \ldots, x_n$ and $(v_1, v_1'), \ldots, (v_n, v_n')$ where $n \geq 0$. Thus, $\overline{k} \in s$ stands for $k_1, \ldots, k_n \in s$ and $(\overline{v}, \overline{v}') \in \mathcal{R}$ for $(v_1, v_1'), \ldots, (v_n, v_n') \in \mathcal{R}$. Similarly, $\{\overline{k}\}$ is a shorthand for the set $\{k_1, \ldots, k_n\}$ where $k_i \neq k_j$ for any $i \neq j$. When s and t are sets, $s \uplus t$ is defined to be $s \cup t$ if $s \cap t = \emptyset$, and undefined otherwise.

3.2 Syntax and Semantics

$\lambda_{\texttt{seal}}$ is the standard untyped, call-by-value λ-calculus extended with sealing. Its syntax is given in Figure 3.1. Seal k is an element of the countably infinite set $\mathcal{K}$ of all seals. We use meta-variables s and t for finite subsets of $\mathcal{K}$. Fresh seal generation $\nu x.\, e$ generates a fresh seal k, binds it to x and evaluates e. The meaning of freshness will soon be clarified below. Sealing $\{e_1\}_{e_2}$ evaluates e_1 to value v and e_2 to seal k, and seals v under k. Unsealing $\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$ evaluates e_1 to seal k_1 and e_2 to sealed value $\{v\}_{k_2}$. If $k_1 = k_2$, the unsealing succeeds and e_3 is evaluated with x bound to v. Otherwise, the unsealing fails and e_4 is evaluated.

The calculus is also parametrized by first-order constants and primitives (involving no seals) such as real numbers and their arithmetics. We use infix notations for binary primitives like $e_1 + e_2$. We assume that constants include booleans $\texttt{true}$ and $\texttt{false}$. We also assume that op includes the equality $=$ for constants. Note that we do not have equality for sealed values yet (cf. Section 3.6), though equality for seals is easy to implement as in $\texttt{let}\ \{x\}_{k_1} = \{c\}_{k_2}\ \texttt{in}\ \texttt{true}\ \texttt{else}\ \texttt{false}$.

$$\frac{\mathit{Seals}(e) \subseteq s}{(s)\,\lambda x.\,e \Downarrow (s)\,\lambda x.\,e}\text{(E-Lam)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,\lambda x.\,e \quad (s_1)\,e_2 \Downarrow (s_2)\,v \quad (s_2)\,[v/x]e \Downarrow (t)\,w}{(s)\,e_1\,e_2 \Downarrow (t)\,w}\text{(E-App)}$$

$$\frac{}{(s)\,c \Downarrow (s)\,c}\text{(E-Const)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,c_1 \quad \ldots \quad (s_{n-1})\,e_n \Downarrow (s_n)\,c_n \quad [\![op(c_1,\ldots,c_n)]\!] = c}{(s)\,op(e_1,\ldots,e_n) \Downarrow (s_n)\,c}\text{(E-Prim)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,\texttt{true} \quad (s_1)\,e_2 \Downarrow (t)\,v}{(s)\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow (t)\,v}\text{(E-Cond-True)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,\texttt{false} \quad (s_1)\,e_3 \Downarrow (t)\,v}{(s)\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow (t)\,v}\text{(E-Cond-False)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,v_1 \quad \ldots \quad (s_{n-1})\,e_n \Downarrow (s_n)\,v_n}{(s)\,\langle e_1,\ldots,e_n\rangle \Downarrow (s_n)\,\langle v_1,\ldots,v_n\rangle}\text{(E-Tuple)}$$

$$\frac{(s)\,e \Downarrow (t)\,\langle v_1,\ldots,v_n\rangle \quad 1 \le i \le n}{(s)\,\#_i(e) \Downarrow (t)\,v_i}\text{(E-Proj)}$$

$$\frac{k \in s}{(s)\,k \Downarrow (s)\,k}\text{(E-Seal)} \qquad \frac{(s \uplus \{k\})\,[k/x]e \Downarrow (t)\,v}{(s)\,\nu x.\,e \Downarrow (t)\,v}\text{(E-New)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,v \quad (s_1)\,e_2 \Downarrow (s_2)\,k}{(s)\,\{e_1\}_{e_2} \Downarrow (s_2)\,\{v\}_k}\text{(E-Do-Seal)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,k \quad (s_1)\,e_2 \Downarrow (s_2)\,\{v\}_k \quad (s_2)\,[v/x]e_3 \Downarrow (t)\,w}{(s)\,\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4 \Downarrow (t)\,w}\text{(E-Unseal-Succ)}$$

$$\frac{(s)\,e_1 \Downarrow (s_1)\,k_1 \quad (s_1)\,e_2 \Downarrow (s_2)\,\{v\}_{k_2} \quad k_1 \neq k_2 \quad (s_2)\,e_4 \Downarrow (t)\,w}{(s)\,\texttt{let}\ \{x\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4 \Downarrow (t)\,w}\text{(E-Unseal-Fail)}$$

Figure 3.2: Semantics of $\lambda_{\texttt{seal}}$

We adopt the standard notion of variable binding and write $FV(e)$ for the set of free variables in e. We also write $\mathit{Seals}(e)$ for the set of seals that appear in term e.

We write $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ for $(\lambda x.\, e_2)e_1$. We also write $\bot$ for $(\lambda x.\, xx)(\lambda x.\, xx)$ and $\lambda\{x\}_k.\, e$ for $\lambda y.\, \texttt{let}\ \{x\}_k = y\ \texttt{in}\ e\ \texttt{else}\ \bot$ where $y \notin FV(e)$. Furthermore, we write $\lambda\langle x, y\rangle.\, e$ for $\lambda z.\, \texttt{let}\ x = \#_1(z)\ \texttt{in}\ \texttt{let}\ y = \#_2(z)\ \texttt{in}\ e$ where $z \notin FV(e)$. We use similar notations of pattern matching throughout the chapter.

The semantics of $\lambda_{\texttt{seal}}$ is given in Figure 3.2 by big-step evaluation $(s)\, e \Downarrow (t)\, v$ annotated with the set s of seals before the evaluation and the set t of seals after the evaluation. It is parametrized by the meaning $[\![op(c_1, \dots, c_n)]\!]$ of primitives. For example, $[\![1.23 + 4.56]\!] = 5.79$. For simplicity, we adopt the left-to-right evaluation order. As usual, substitutions $[e/x]$ avoid capturing free variables by implicit α-conversion. The meaning of freshness is formalized by requiring $k \notin s$ in (E-New). We write $(s)\, e \Downarrow$ if $(s)\, e \Downarrow (t)\, v$ for some t and v.

Because of fresh seal generation, our evaluation is not quite deterministic. For instance, we have both $(\emptyset)\, \nu x.\, x \Downarrow (\{k_1\})\, k_1$ and $(\emptyset)\, \nu x.\, x \Downarrow (\{k_2\})\, k_2$ for $k_1 \neq k_2$. Nevertheless, we have the following property:

Property 3.1. Evaluation is deterministic modulo the names of freshly generated seals. That is, for any $(s)\, e \Downarrow (t)\, v$ and $(s)\, e \Downarrow (t')\, v'$ with $\mathit{Seals}(e) \subseteq s$, we have $v = [\overline{k}/\overline{x}]e_0$ and $v' = [\overline{k}'/\overline{x}]e_0$ for some e_0 with $\mathit{Seals}(e_0) \subseteq s$, some $\overline{k}$ with $\{\overline{k}\} \subseteq t \setminus s$, and some $\overline{k}'$ with $\{\overline{k}'\} \subseteq t' \setminus s$.

Proof. Straightforward induction on the derivation of $(s)\, e \Downarrow (t)\, v$. □

In what follows, we implicitly use the following properties of evaluation without explicitly referring to them.

Property 3.2. Every value evaluates only to itself. That is, for any s and v with $s \supseteq \mathit{Seals}(v)$, we have $(s)\, v \Downarrow (s)\, v$. Furthermore, if $(s)\, v \Downarrow (t)\, w$, then $t = s$ and $w = v$.

Proof. Straightforward induction on the syntax of values. □

Property 3.3. Evaluation never decreases the seal set. That is, for any $(s)\, e \Downarrow (t)\, v$, we have $s \subseteq t$.

Proof. Straightforward induction on the derivation of $(s)\, e \Downarrow (t)\, v$. □

3.3 Generalized Contextual Equivalence

In standard untyped λ-calculus, contextual equivalence for closed values[1] can be defined by saying that v and v' are contextually equivalent if $[v/x]e \Downarrow \iff [v'/x]e \Downarrow$ for any term e. In $\lambda_{\texttt{seal}}$, however, contextual equivalence cannot be defined for two values in isolation. For instance, consider $\lambda\{x\}_k.\, x + 1$ and $\lambda\{x\}_{k'}.\, x + 2$. Whether these values are equivalent or not depends on what values sealed under k or k' are known to the context. If the original terms which created k and k' were $\nu z.\, \langle \{2\}_z, \lambda\{x\}_z.\, x + 1\rangle$ and $\nu z.\, \langle \{1\}_z, \lambda\{x\}_z.\, x + 2\rangle$, for example, then the only values sealed under k or k' are 2 and 1, respectively. Thus, the equivalence above does hold. On the other

[1]For the sake of simplicity, we focus on equivalence of closed values (as opposed to open expressions) in this chapter. For open expressions e and e' with free variables $x_1, \dots, x_n$, it suffices to consider the equivalence of $\lambda x_1. \dots \lambda x_n.\, e$ and $\lambda x_1. \dots \lambda x_n.\, e'$ instead. See also Section 4.6 for more formal discussion on this issue.

hand, it does not hold if the terms were, say, $\nu z.\, \langle \{3\}_z, \lambda\{x\}_z.\, x+1 \rangle$ and $\nu z.\, \langle \{4\}_z, \lambda\{x\}_z.\, x+2 \rangle$. This observation that we have to consider multiple pairs of values at once leads to the following definition of contextual equivalence.

Definition 3.4. A *value relation* $\mathcal{R}$ is a set of pairs of values.

Definition 3.5. The *contextual equivalence* $\equiv$ is the set of all triples $(s, s', \mathcal{R})$ such that for any $(\overline{v}, \overline{v}') \in \mathcal{R}$, we have the following properties.

1. $\mathit{Seals}(\overline{v}) \subseteq s$ and $\mathit{Seals}(\overline{v}') \subseteq s'$.
2. $(s)\, [\overline{v}/\overline{x}]e \Downarrow \iff (s')\, [\overline{v}'/\overline{x}]e \Downarrow$ for any e with $\mathit{Seals}(e) = \emptyset$.

The intuition is that $\overline{v}$ and $\overline{v}'$ are indistinguishable for any observer within the language (unless it somehow knows any of the seals in s or s' *a priori*). We write $(s)\, v_1, \ldots, v_n \equiv (s')\, v'_1, \ldots, v'_n$ for $(s, s', \{(v_1, v'_1), \ldots, (v_n, v'_n)\}) \in \equiv$. In order to lighten the notation, we do not enclose these $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ in parentheses. We also write $(s)\, v \equiv_{\mathcal{R}} (s')\, v'$ when $(v, v') \in \mathcal{R}$ and $(s, s', \mathcal{R}) \in \equiv$. Intuitively, it can be read as "value v with seal set s and value v' with seal set s' are contextually equivalent under contexts' knowledge $\mathcal{R}$."

Note that no generality is lost by requiring $\mathit{Seals}(e) = \emptyset$ in the definition above: if e needs its own seals, it can freshly generate an arbitrary number of them by using ν; if e knows some $\overline{k} \in s$ in the left-hand side and corresponding $\overline{k}' \in s'$ in the right-hand side of contextual equivalence, it suffices to require $(\overline{k}, \overline{k}') \in \mathcal{R}$ so that these seals can be substituted for some free $\overline{x}$ in e by Condition (2) above. Thus, our contextual equivalence subsumes standard contextual equivalence where a context knows none, all, or part of the seals (or, more generally, values involving the seals). Conversely, the standard contextual equivalence (for closed values) is *implied* by the generalized one in the following sense: if $(v, v') \in \mathcal{R}$ for some $(s, s', \mathcal{R}) \in \equiv$, then it is immediate by definition that $K[v] \Downarrow \iff K[v'] \Downarrow$ for any context K with a hole $[\,]$.

Example 3.6. Let $s = \{k\}$ and $s' = \{k'\}$. We have $(s)\, \{2\}_k \equiv (s')\, \{1\}_{k'}$ since the context has no means to unseal the values sealed under k or k'. (A formal proof of this claim based on bisimulation will be given later in Example 3.11 with Corollary 3.24.) We also have $(s)\, \lambda\{x\}_k.\, x+1 \equiv (s')\, \lambda\{x\}_{k'}.\, x+2$ since the context cannot make up any values sealed under k or k'.

Furthermore, we have

$$(s)\, \{2\}_k, \lambda\{x\}_k.\, x+1 \quad \equiv \quad (s')\, \{1\}_{k'}, \lambda\{x\}_{k'}.\, x+2$$

because applications of the functions to the sealed values yield the same integer 3. Similarly,

$$(s)\, \{4\}_k, \lambda\{x\}_k.\, x+1 \quad \equiv \quad (s')\, \{5\}_{k'}, \lambda\{x\}_{k'}.\, x$$

holds. However,

$$\begin{array}{rl} & (s)\, \{2\}_k, \lambda\{x\}_k.\, x+1, \{4\}_k, \lambda\{x\}_k.\, x+1 \\ \equiv & (s')\, \{1\}_{k'}, \lambda\{x\}_{k'}.\, x+2, \{5\}_{k'}, \lambda\{x\}_{k'}.\, x \end{array}$$

does not hold, because applications of the last functions to the first sealed values yield different integers 3 and 1.

As the last example shows, even if $(s, s', \mathcal{R}_1) \in \equiv$ and $(s, s', \mathcal{R}_2) \in \equiv$, we do not always have $(s, s', \mathcal{R}_1 \cup \mathcal{R}_2) \in \equiv$. Intuitively, this means that we should not confuse two worlds where the uses of seals are different. This is the reason why we defined $\equiv$ as a *set* of triples $(s, s', \mathcal{R})$ rather than just a *function* that takes a pair (s, s') of seal sets and returns the set $\mathcal{R}$ of all pairs of equivalent values.

Conversely, again as the examples above suggest, there are cases where $(s, s', \mathcal{R}_1) \in \equiv$ and $(s, s', \mathcal{R}_2) \in \equiv$ for $\mathcal{R}_1 \subseteq \mathcal{R}_2$. This implies that there is a partial order among the value relations $\mathcal{R}$ in contextual equivalence. We could alternatively define contextual equivalence only with such value relations that are maximal in this ordering, but this would just complicate the technicalities that follow.

We will use the following lemmas about contextual equivalence in what follows.

Lemma 3.7. Application, projection, fresh seal generation, and unsealing preserve contextual equivalence. That is:

1. For any $(u, u') \in \mathcal{R}$ and $(v, v') \in \mathcal{R}$ with $(s, s', \mathcal{R}) \in \equiv$, if $(s)\, uv \Downarrow (t)\, w$ and $(s)\, u'v' \Downarrow (t')\, w'$, then $(t, t', \mathcal{R} \cup \{(w, w')\}) \in \equiv$.
2. For any $(\langle v_1, \ldots, v_n \rangle, \langle v'_1, \ldots, v'_n \rangle) \in \mathcal{R}$ with $(s, s', \mathcal{R}) \in \equiv$, we have $(s, s', \mathcal{R} \cup \{(v_i, v'_i)\}) \in \equiv$ for any $1 \leq i \leq n$.
3. For any $(s, s', \mathcal{R}) \in \equiv$, we have $(s \uplus \{k\}, s' \uplus \{k'\}, \mathcal{R} \uplus \{(k, k')\}) \in \equiv$ for any $k \notin s$ and $k' \notin s'$.
4. For any $(\{v\}_k, \{v'\}_{k'}) \in \mathcal{R}$ and $(k, k') \in \mathcal{R}$ with $(s, s', \mathcal{R}) \in \equiv$, we have $(s, s', \mathcal{R} \cup \{(v, v')\}) \in \equiv$.

Proof. To prove the case of application, let us assume $(u, u') \in \mathcal{R}$ and $(v, v') \in \mathcal{R}$ with $(s, s', \mathcal{R}) \in \equiv$ as well as $(s)\, uv \Downarrow (t)\, w$ and $(s)\, u'v' \Downarrow (t')\, w'$, and prove $(t, t', \mathcal{R} \cup \{(w, w')\}) \in \equiv$. The first condition of contextual equivalence in Definition 3.5 follows immediately from (part of) Property 3.1. To show the second, take any $(\overline{w}, \overline{w}') \in \mathcal{R} \cup \{(w, w')\}$, take any e with $\mathit{Seals}(e) = \emptyset$, and prove $(t)\, [\overline{w}/\overline{x}]e \Downarrow \iff (t')\, [\overline{w}'/\overline{x}]e \Downarrow$. Without loss of generality, let $u_1 = u$ and $u'_1 = u'$. Then, from the second condition in the definition of $(s, s', \mathcal{R}) \in \equiv$, we have

$$
\begin{aligned}
& (s)\, [u, v, w_2, \ldots, w_n / y, z, x_2, \ldots, x_n](\texttt{let}\ x_1 = yz\ \texttt{in}\ e) \Downarrow \\
\iff \quad & (s')\, [u', v', w'_2, \ldots, w'_n / y, z, x_2, \ldots, x_n](\texttt{let}\ x_1 = yz\ \texttt{in}\ e) \Downarrow
\end{aligned}
$$

from which the conclusion follows with the assumptions $(s)\, uv \Downarrow (t)\, w$ and $(s)\, u'v' \Downarrow (t')\, w'$.

The other cases follow similarly by taking $\texttt{let}\ x_1 = \#_i(y)\ \texttt{in}\ e$ or $\nu x_1.\, e$ or $\texttt{let}\ \{x_1\}_y = z\ \texttt{in}\ e$, respectively, instead of $\texttt{let}\ x_1 = yz\ \texttt{in}\ e$ above. □

Lemma 3.8. Contextually equivalent values put in the same value context yield contextually equivalent values. That is, for any $(s, s', \mathcal{R}) \in \equiv$ and $(\overline{v}, \overline{v}') \in \mathcal{R}$, and for any $w = [\overline{v}/\overline{x}]e_0$ and $w' = [\overline{v}'/\overline{x}]e_0$ with $\mathit{Seals}(e_0) = \emptyset$, we have $(s, s', \mathcal{R} \cup \{(w, w')\}) \in \equiv$.

Proof. Immediate from the definition of contextual equivalence, using the property of substitution that $[[\overline{v}/\overline{x}]e_0/x]e = [\overline{v}/\overline{x}]([e_0/x]e)$ when $\{\overline{x}\} \cap FV(e) = \emptyset$. □

Lemma 3.9. Any subset of contextually equivalent values are contextually equivalent. That is, for any $(s, s', \mathcal{R}) \in \equiv$, we have $(s, s', \mathcal{S}) \in \equiv$ for any $\mathcal{S} \subseteq \mathcal{R}$.

Proof. Immediate from the definition of contextual equivalence. □

3.4 Bisimulation

Giving a direct proof of contextual equivalence is generally difficult, because the definition involves universal quantification over an infinite number of contexts. Thus, we want a more convenient tool for proving contextual equivalence. For this purpose, we define the notion of bisimulation as follows.

Definition 3.10. A *bisimulation* is a set X of triples $(s, s', \mathcal{R})$ such that:

1. For each $(v, v') \in \mathcal{R}$, we have $\mathit{Seals}(v) \subseteq s$ and $\mathit{Seals}(v') \subseteq s'$.

2. For each $(v, v') \in \mathcal{R}$, v and v' are of the same kind. That is, both are functions, both are constants, both are tuples, both are seals, or both are sealed values.

3. For each $(c, c') \in \mathcal{R}$, we have $c = c'$.

4. For each $(\langle v_1, \ldots, v_n \rangle, \langle v'_1, \ldots, v'_{n'} \rangle) \in \mathcal{R}$, we have $n = n'$ and $(s, s', \mathcal{R} \cup \{(v_i, v'_i)\}) \in X$ for every $1 \leq i \leq n$.

5. For each $(k_1, k'_1) \in \mathcal{R}$ and $(k_2, k'_2) \in \mathcal{R}$, we have $k_1 = k_2 \iff k'_1 = k'_2$.

6. For each $(\{v\}_k, \{v'\}_{k'}) \in \mathcal{R}$, we have either $(k, k') \in \mathcal{R}$ and $(s, s', \mathcal{R} \cup \{(v, v')\}) \in X$, or else $k \notin \mathit{fst}(\mathcal{R})$ and $k' \notin \mathit{snd}(\mathcal{R})$. Here, $\mathit{fst}(\mathcal{R})$ is the set of the first elements of all pairs in $\mathcal{R}$ and $\mathit{snd}(\mathcal{R})$ the second.

7. Take any $(\lambda x.\, e, \lambda x.\, e') \in \mathcal{R}$. Take also any $\overline{k}$ and $\overline{k}'$ with $s \cap \{\overline{k}\} = s' \cap \{\overline{k}'\} = \emptyset$. Moreover, let $v = [\overline{u}/\overline{x}]d$ and $v' = [\overline{u}'/\overline{x}]d$ for any $(\overline{u}, \overline{u}') \in \mathcal{R} \uplus \{(\overline{k}, \overline{k}')\}$ and $\mathit{Seals}(d) = \emptyset$. Then, we have $(s \uplus \{\overline{k}\})\, (\lambda x.\, e)v \Downarrow \iff (s' \uplus \{\overline{k}'\})\, (\lambda x.\, e')v' \Downarrow$. Furthermore, if $(s \uplus \{\overline{k}\})\, (\lambda x.\, e)v \Downarrow (t)\, w$ and $(s' \uplus \{\overline{k}'\})\, (\lambda x.\, e')v' \Downarrow (t')\, w'$, then $(t, t', \mathcal{R} \uplus \{(\overline{k}, \overline{k}')\} \cup \{(w, w')\}) \in X$.

For any bisimulation X, we write $(s)\, v\, X_{\mathcal{R}}\, (s')\, v'$ when $(v, v') \in \mathcal{R}$ and $(s, s', \mathcal{R}) \in X$. This can be read "values v and v' with seal sets s and s' are bisimilar under contexts' knowledge $\mathcal{R}$."

The intuitions behind the definition of bisimulation are as follows. Each of the conditions excludes pairs of values that are distinguishable by a context (except for Condition 1, which just restricts the scoping of seals). Condition 2 excludes pairs of values of different kinds, e.g., 123 and $\lambda x.\, x$. Condition 3 excludes pairs of different constants. Condition 4 excludes pairs of tuples with distinguishable elements. Condition 5 excludes cases such as $(k, k') \in \mathcal{R}$ and $(k, k'') \in \mathcal{R}$ with $k' \neq k''$, for which contexts like $\texttt{let}\ \{x\}_y = \{()\}_z\ \texttt{in}\ x\ \texttt{else}\ \bot$ can distinguish the left-hand side (setting $y = z = k$) and the right-hand side (setting $y = k'$ and $z = k''$). Condition 6 excludes cases where (i) the context can unseal both of two sealed values whose contents are distinguishable, or (ii) the context can unseal only one of the two sealed values.

Condition 7, the most interesting one, is about what a context can do to distinguish two functions. Obviously, this will involve applying them to some arguments—but what arguments? Certainly not arbitrary terms, because in general a context has only a partial knowledge of (values involving) the seals in s and s'. All that a context can do for making up the arguments is to carry out some computation d using values $\overline{u}$ and $\overline{u}'$ from its knowledge. Therefore, the arguments have forms $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$.

An important and perhaps surprising point here is that it actually suffices to consider cases where these arguments are *values*. This restriction is useful and even crucial for proving bisimulation of functions: if the arguments $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ were not values, we should evaluate them before applying the functions; in particular, if evaluation of one argument converges, then evaluation of the other argument must converge as well; proving this property amounts to proving the contextual equivalence of $\overline{u}$ and $\overline{u}'$, which was the whole purpose of our bisimulation!

Fortunately, our restriction of the arguments to values can be justified by the "fundamental property" proved in the next section, which says that the special forms $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ are preserved by evaluation. The only change required as a result of this restriction is the addition of $\{(\overline{k}, \overline{k}')\}$ to knowledge $\mathcal{R}$ in Condition 7: it compensates for the fact that d can no longer be a fresh seal generation, while the context can still generate its own fresh seals $\overline{k}$ and $\overline{k}'$ when making up the arguments. Without such a change, our bisimulation would indeed be unsound: a counter-example would be $(\emptyset, \emptyset, \{(\lambda x.\, \{\mathtt{true}\}_x, \lambda x.\, \{\mathtt{false}\}_x)\})$, which would satisfy all the conditions of bisimulation (including Condition 7, in particular, because the arguments v and v' could not contain any seal), while contexts like $\nu y.\, \mathtt{let}\ \{z\}_y = [\,]\, y\ \mathtt{in}\ \mathtt{if}\ z\ \mathtt{then}\ ()\ \mathtt{else}\ \bot$ can distinguish the two functions.

The rest of Condition 7 is straightforward: the results w and w' of function application should also be bisimilar.

Example 3.11. Let $s = \{k\}$, $s' = \{k'\}$, and $\mathcal{R} = \{(\{2\}_k, \{1\}_{k'})\}$. Then $\{(s, s', \mathcal{R})\}$ is a bisimulation, as can be seen by a straightforward check of the conditions above.

Example 3.12. Let $s = \{k_1, k_2\}$, $s' = \{k'\}$, and

$$\begin{array}{rcl}\mathcal{R} & = & \{(\langle\{2\}_{k_1}, \{4\}_{k_2}\rangle, \\ & & \quad \langle\{1\}_{k'}, \{5\}_{k'}\rangle), \\ & & (\{2\}_{k_1}, \{1\}_{k'}), \\ & & (\{4\}_{k_2}, \{5\}_{k'})\}.\end{array}$$

Then $\{(s, s', \mathcal{R})\}$ is a bisimulation. This example illustrates the fact that the number of seals may differ in the left-hand side and in the right-hand side of bisimulation. Note that the closure condition (Condition 4) in the definition of bisimulation demands that we include not only the original pairs, but also their corresponding components.

Example 3.13. Suppose we want to show that the pair $\langle\{2\}_k,\ \lambda\{x\}_k.\, x+1\rangle$ is bisimilar to $\langle\{1\}_{k'},\ \lambda\{x\}_{k'}.\, x+2\rangle$, assuming that seals k and k' are not known to the context. Again, the closure conditions in the definition force us to include the corresponding components of the pairs (Condition 4), as well as the results of evaluating the second components applied to the first components (Condition 7); moreover, since Condition 7 allows the context to enrich the set of seals with arbitrary seals of its own, our bisimulation will consist of an infinite collection of similar sets, differing in the context's choice of seals.

Formally, let $\mathcal{G}$ be the following function on sets of pairs of seals:

$$\begin{array}{rcl}\mathcal{G}\{(\overline{k}_0, \overline{k}'_0)\} & = & \{(\langle\{2\}_k,\ \lambda\{x\}_k.\, x+1\rangle, \\ & & \quad \langle\{1\}_{k'},\ \lambda\{x\}_{k'}.\, x+2\rangle), \\ & & (\{2\}_k,\ \{1\}_{k'}), \\ & & (\lambda\{x\}_k.\, x+1,\ \lambda\{x\}_{k'}.\, x+2), \\ & & (3,\ 3)\} \\ & \cup & \{(\overline{k}_0,\ \overline{k}'_0)\}\end{array}$$

Then

$$X \quad = \quad \{(\{k,\overline{k}_0\},\ \{k',\overline{k}'_0\},\ \mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\}) \mid k \notin \{\overline{k}_0\} \wedge k' \notin \{\overline{k}'_0\}\}$$

is a bisimulation. The only non-trivial work required to show this is checking Condition 7 for the pair $(\lambda\{x\}_k.\, x+1,\ \lambda\{x\}_{k'}.\, x+2) \in \mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\}$, for each $\overline{k}_0$ and $\overline{k}'_0$ with $k \notin \{\overline{k}_0\}$ and $k' \notin \{\overline{k}'_0\}$.

Consider any $v = [\overline{u}/\overline{x}]d$ and $v' = [\overline{u}'/\overline{x}]d$ with $(\overline{u},\overline{u}') \in \mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\}$ and $\mathit{Seals}(d) = \{k,\overline{k}_0\} \cap \{\overline{k}_1\} = \{k',\overline{k}'_0\} \cap \{\overline{k}'_1\} = \emptyset$. If the evaluations of $(\lambda\{x\}_k.\, x+1)v$ and $(\lambda\{x\}_{k'}.\, x+2)v'$ diverge, then the condition holds.

Let us focus on cases where the evaluation of $(\lambda\{x\}_k.\, x+1)v$ converges (without loss of generality, thanks to symmetry), that is, where v is of the form $\{w\}_k$. Then, either d is of the form $\{d_0\}_{x_i}$ and $u_i = k$, or else d is a variable x_i and $u_i = \{w\}_k$. However, the former case is impossible: k is not in the first projection of $\mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\}$ or $\{(\overline{k}_1,\overline{k}'_1)\}$ by their definitions. So we must be in the latter case.

Since the only element of the form $\{w\}_k$ in the first projection of $\mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\}$ is $\{2\}_k$ where the corresponding element in its second projection is $\{1\}_{k'}$, we have $v = \{2\}_k$ and $v' = \{1\}_{k'}$. Then, the only evaluations of $(\lambda\{x\}_k.\, x+1)v$ and $(\lambda\{x\}_{k'}.\, x+2)v'$ are

$$(\{k,\overline{k}_0\} \uplus \{\overline{k}_1\})\,(\lambda\{x\}_k.\, x+1)v \quad \Downarrow \quad (\{k,\overline{k}_0,\overline{k}_1\})\,3$$

and

$$(\{k',\overline{k}'_0\} \uplus \{\overline{k}'_1\})\,(\lambda\{x\}_{k'}.\, x+2)v' \quad \Downarrow \quad (\{k',\overline{k}'_0,\overline{k}'_1\})\,3.$$

Thus, the condition follows from

$$\mathcal{G}\{(\overline{k}_0,\overline{k}'_0)\} \uplus \{(\overline{k}_1,\overline{k}'_1)\} \cup \{(3,3)\} \quad = \quad \mathcal{G}\{(\overline{k}_0,\overline{k}'_0),(\overline{k}_1,\overline{k}'_1)\}$$

and

$$(\{k,\overline{k}_0,\overline{k}_1\},\{k',\overline{k}'_0,\overline{k}'_1\},\mathcal{G}\{(\overline{k}_0,\overline{k}'_0),(\overline{k}_1,\overline{k}'_1)\}) \in X.$$

Example 3.14 (Complex Numbers). Now let us show a bisimulation relating the two implementations of complex numbers in Section 3.1. First, let

$$\begin{array}{rcl}
v & = & \langle \lambda\langle x,y\rangle.\, \{\langle x+0.0, y+0.0\rangle\}_k, \\
 & & \lambda\{\langle x,y\rangle\}_k.\, \langle x,y\rangle, \\
 & & \lambda\langle\{\langle x_1,y_1\rangle\}_k, \{\langle x_2,y_2\rangle\}_k\rangle. \\
 & & \quad \{\langle x_1 \times x_2 - y_1 \times y_2, x_1 \times y_2 + y_1 \times x_2\rangle\}_k\rangle \\
v' & = & \langle \lambda\langle x,y\rangle.\, \{\langle \mathtt{sqrt}\langle x \times x + y \times y\rangle, \mathtt{atan2}\langle y,x\rangle\rangle\}_{k'}, \\
 & & \lambda\{\langle r,\theta\rangle\}_{k'}.\, \langle r \times \mathtt{cos}\,\theta, r \times \mathtt{sin}\,\theta\rangle, \\
 & & \lambda\langle\{\langle r_1,\theta_1\rangle\}_{k'}, \{\langle r_2,\theta_2\rangle\}_{k'}\rangle.\, \{\langle r_1 \times r_2, \theta_1 + \theta_2\rangle\}_{k'}\rangle.
\end{array}$$

The first component of each triple corresponds to the `from_re_and_im` functions in 3.1. The implementation in v just seals the x and y coordinates provided as arguments, after checking that they are indeed real numbers by attempting to add them to 0.0. The implementation in v' performs an appropriate change of representation before sealing. The second components correspond to the `to_re_and_im` functions in 3.1, and the third components to the `multiply` functions.

The construction of the bisimulation follows the same pattern as Example 3.13, except that the operator $\mathcal{G}$ is more interesting:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{G}\{(\overline{k}_0, \overline{k}'_0)\} =} \\
& \{(v, v')\} \\
\cup & \{(\lambda\langle x, y\rangle.\, \{\langle x + 0.0, y + 0.0\rangle\}_k, \\
& \quad \lambda\langle x, y\rangle.\, \{\langle \mathtt{sqrt}(x \times x + y \times y), \mathtt{atan2}(y, x)\rangle\}_{k'}), \\
& \ (\lambda\{\langle x, y\rangle\}_k.\, \langle x, y\rangle, \\
& \quad \lambda\{\langle r, \theta\rangle\}_{k'}.\, \langle r \times \mathtt{cos}\, \theta, r \times \mathtt{sin}\, \theta\rangle), \\
& \ (\lambda\langle\{\langle x_1, y_1\rangle\}_k, \{\langle x_2, y_2\rangle\}_k\rangle. \\
& \qquad \{\langle x_1 \times x_2 - y_1 \times y_2, x_1 \times y_2 + y_1 \times x_2\rangle\}_k, \\
& \quad \lambda\langle\{\langle r_1, \theta_1\rangle\}_{k'}, \{\langle r_2, \theta_2\rangle\}_{k'}\rangle.\, \{\langle r_1 \times r_2, \theta_1 + \theta_2\rangle\}_{k'})\} \\
\cup & \{(\langle x, y\rangle, \langle x, y\rangle) \mid x \text{ and } y \text{ are arbitrary real numbers}\} \\
\cup & \{(\{\langle r\cos\theta, r\sin\theta\rangle\}_k, \{\langle r, \theta\rangle\}_{k'}) \mid r \geq 0\} \\
\cup & \{(\overline{k}_0, \overline{k}'_0)\}
\end{array}
$$

Example 3.15 (Generative vs. Non-Generative Functors). In this example, we use bisimulation to show the equivalence of two instantiations of a generative functor, where generativity is modeled by fresh seal generation and the equivalence really depends on the generativity.

A functor is a parameterized module—a function from modules to modules. For example, a module implementing sets by binary trees can be parameterized by the type of elements and their comparison function. In the same imaginary ML-like language as in Section 3.1, such a functor might be written as follows:

```
functor Set(module Element : sig
              type t
              val less_than : t -> t -> bool
            end) =
  type elt = Element.t
  abstype set = Element.t tree
  let empty : set = Leaf
  let rec add : elt -> set -> set =
    fun x ->
      fun Leaf -> Node(x, Leaf, Leaf)
        | Node(y, l, r) ->
            if Element.less_than x y then
              Node(y, add x l, r)
            else if Element.less_than y x then
              Node(y, l, add x r)
            else Node(y, l, r)
  let rec is_elt_of : elt -> set -> bool =
    fun x ->
      fun Leaf -> false
        | Node(y, l, r) ->
            if Element.less_than x y then
              is_elt_of x l
```

```
                else if Element.less_than y x then
                  is_elt_of x r
                else true
    end
```

Now, consider the following three applications of this functor:

```
    module IntSet1 =
      Set(module Element =
        type t = int
        let less_than : t -> t -> bool =
          fun x -> fun y -> (x <_int y)
      end)
    module IntSet2 =
      Set(module Element =
        type t = int
        let less_than : t -> t -> bool =
          fun x -> fun y -> (x <_int y)
      end)
    module IntSet3 =
      Set(module Element =
        type t = int
        let less_than : t -> t -> bool =
          fun x -> fun y -> (x >_int y)
      end)
```

If the functor `Set` is non-generative,[2] the abstract type `IntSet3.set` becomes compatible with `IntSet1.set` and `IntSet2.set`, even though the comparison function of `IntSet3` is not compatible with that of `IntSet1` or `IntSet2`. As a result, (part of) their abstraction as sets of integers is lost: for instance, `IntSet2` and `IntSet3` are distinguished by a context like

$$C[\,] \;=\; \texttt{let s = }[\,]\texttt{.add 7 (}[\,]\texttt{.add 3 }[\,]\texttt{.empty) in}$$
$$\texttt{IntSet1.is\_elt\_of 7 s}$$

while they *should* be equivalent if considered just as two different implementations of integer sets.

This situation can be translated into $\lambda_{\texttt{seal}}$ as follows. First, the non-generative functor can be implemented by the following function f, using a standard call-by-value fixed-point operator `fix`

[2]We intentionally avoid calling it "applicative" since the original design [Leroy 1995] of applicative functors carefully prevents the problem which follows here.

(which is definable since the language is untyped).

$$
\begin{array}{l}
\lambda \mathtt{lt}. \\
\quad \langle \{\mathtt{nil}\}_k, \\
\quad\; \mathtt{fix}(\lambda \mathtt{add}.\ \lambda \langle x, \{y\}_k \rangle. \\
\quad\quad \mathtt{if}\ \mathtt{lt}\langle x, x\rangle\ \mathtt{then}\ \bot\ \mathtt{else}\ \text{(* check that x has type } \mathtt{elt} \text{ *)} \\
\quad\quad \mathtt{if}\ y = \mathtt{nil}\ \mathtt{then}\ \{\langle x, \mathtt{nil}, \mathtt{nil}\rangle\}_k\ \mathtt{else} \\
\quad\quad \mathtt{if}\ \mathtt{lt}\langle x, \#_1(y)\rangle\ \mathtt{then}\ \{\langle \#_1(y), \mathtt{add}(x, \#_2(y)), \#_3(y)\rangle\}_k\ \mathtt{else} \\
\quad\quad \mathtt{if}\ \mathtt{lt}\langle \#_1(y), x\rangle\ \mathtt{then}\ \{\langle \#_1(y), \#_2(y), \mathtt{add}(x, \#_3(y))\rangle\}_k\ \mathtt{else} \\
\quad\quad \{y\}_k), \\
\quad\; \mathtt{fix}(\lambda \mathtt{is\_elt\_of}.\ \lambda \langle x, \{y\}_k \rangle. \\
\quad\quad \mathtt{if}\ y = \mathtt{nil}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else} \\
\quad\quad \mathtt{if}\ \mathtt{lt}\langle x, \#_1(y)\rangle\ \mathtt{then}\ \mathtt{is\_elt\_of}\langle x, \#_2(y)\rangle\ \mathtt{else} \\
\quad\quad \mathtt{if}\ \mathtt{lt}\langle \#_1(y), x\rangle\ \mathtt{then}\ \mathtt{is\_elt\_of}\langle x, \#_3(y)\rangle\ \mathtt{else} \\
\quad\quad \mathtt{true})\rangle
\end{array}
$$

Next, we translate the three applications of the functor into three applications of f to appropriate comparison functions:

$$
\begin{array}{rcl}
(\{k\})\, f(\lambda\langle x, y\rangle.\, x <_{\mathtt{int}} y) & \Downarrow & (\{k\})\, v_1 \\
(\{k\})\, f(\lambda\langle x, y\rangle.\, x <_{\mathtt{int}} y) & \Downarrow & (\{k\})\, v_2 \\
(\{k\})\, f(\lambda\langle x, y\rangle.\, x >_{\mathtt{int}} y) & \Downarrow & (\{k\})\, v_3
\end{array}
$$

The values v_2 and v_3 are *not* contextually equivalent when the context knows v_1. That is, $(\{k\}, \{k\}, \{(v_1, v_1), (v_2, v_3)\}) \notin \equiv$. To see this, take $e = \#_3(x)\ \langle 7,\ \#_2(y)\ \langle 7,\ \#_2(y)\ \langle 3,\ \#_1(y)\rangle\rangle\rangle$, setting $x = v_1$, $y = v_2$ in the left hand side and $x = v_1$, $y = v_3$ in the right hand side.

Note that v_2 and v_3 *are* contextually equivalent if the context knows neither v_1, f, nor any other values involving the seal k. That is, $(\{k\}, \{k\}, \{(v_2, v_3)\}) \in \equiv$. Indeed, the context $C[\,]$ above uses `IntSet1` to distinguish `IntSet2` and `IntSet3`. Our definition of contextual equivalence as a set of relations (annotated with seal sets) gives a precise account for such subtle variations of contexts' knowledge.

On the other hand, if we take the `Set` functor to be generative, then `IntSet2` and `IntSet3` are contextually equivalent even if the context also knows `IntSet1`, since all the abstract types are incompatible with one another. This case can be modeled in $\lambda_{\mathtt{seal}}$ by the following function g, which generates a fresh seal for each application instead of using the same seal k for all instantiations.

$$
\begin{array}{l}
\lambda \mathtt{lt}.\, \nu z. \\
\quad \langle \{\mathtt{nil}\}_z, \\
\quad\; \mathtt{fix}(\lambda \mathtt{add}.\ \lambda \langle x, \{y\}_z \rangle.\ \ldots), \\
\quad\; \mathtt{fix}(\lambda \mathtt{is\_elt\_of}.\ \lambda \langle x, \{y\}_z \rangle.\ \ldots)\rangle
\end{array}
$$

Consider the following three applications of g.

$$
\begin{array}{rcl}
(\emptyset)\, g(\lambda\langle x, y\rangle.\, x <_{\mathtt{int}} y) & \Downarrow & (\{k_1\})\, w_1 \\
(\{k_1\})\, g(\lambda\langle x, y\rangle.\, x <_{\mathtt{int}} y) & \Downarrow & (\{k_1, k_2\})\, w_2 \\
(\{k_1\})\, g(\lambda\langle x, y\rangle.\, x >_{\mathtt{int}} y) & \Downarrow & (\{k_1, k_3\})\, w_3
\end{array}
$$

Now w_2 and w_3 are bisimilar even if the context knows w_1. That is, there exists a bisimulation X such that $(\{k_1, k_2\}, \{k_1, k_3\}, \mathcal{R}) \in X$ with $\{(w_1, w_1), (w_2, w_3)\} \subseteq \mathcal{R}$. It is straightforward to construct this bisimulation in the same manner as Examples 3.13 and 3.14.

Example 3.16. Let us show that $\lambda x.\, \langle 3, x\rangle$ is bisimilar to itself. This example is technically trickier than previous ones, because arbitrary values provided by the context can appear verbatim within results. These results can again be passed as arguments and thus appear within yet larger results, etc. To achieve the required closure conditions, we need to reach a limit of this process. This can be accomplished by defining a bisimulation X inductively.

We require $(\emptyset, \emptyset, \emptyset) \in X$ as the (trivial) base case. The induction rule is as follows. Take any $(s, s', \mathcal{R}) \in X$. Take any $\overline{w} = [\overline{v}/\overline{x}]\overline{e}$ and $\overline{w}' = [\overline{v}'/\overline{x}]\overline{e}$ with $(\overline{v}, \overline{v}') \in \mathcal{R}$ and $\mathit{Seals}(\overline{e}) = \emptyset$. Take any $t \supseteq s$ and $t' \supseteq s'$ of the forms $\{\overline{k}\}$ and $\{\overline{k}'\}$. Let

$$\begin{array}{rcl}\mathcal{S} & = & \{(\lambda x.\, \langle 3, x\rangle, \lambda x.\, \langle 3, x\rangle), \\ & & (\langle 3, \overline{w}\rangle, \langle 3, \overline{w}'\rangle), \\ & & (3, 3), \\ & & (\overline{w}, \overline{w}'), \\ & & (\overline{k}, \overline{k}')\}.\end{array}$$

We then require that $(t, t', \mathcal{T}) \in X$ for any $\mathcal{T} \subseteq \mathcal{S}$. The bisimulation we want is the least X satisfying these conditions.

Intuitively, we have defined X so that the conditions of bisimulation—Condition 7, in particular—are immediately satisfied. The final technical twist $\mathcal{T} \subseteq \mathcal{S}$ is needed because the closure conditions in the definition of bisimulation add individual pairs of elements rather than adding their whole "deductive closures" at once.

Example 3.17 (Protocol Encoding). As a final illustration of the power of our bisimulation technique (and $\lambda_{\texttt{seal}}$ itself), let us consider a more challenging example. This example is an encoding of the protocol below, which is based on the key exchange protocol of Needham, Schroeder, and Lowe [Needham and Schroeder 1978; Lowe 1995].

$$\begin{array}{lrcl}1. & \mathrm{B} \to \mathrm{A} & : & B \\ 2. & \mathrm{A} \to \mathrm{B} & : & \{N_A, A\}_{k_B} \\ 3. & \mathrm{B} \to \mathrm{A} & : & \{N_A, N_B, B\}_{k_A} \\ 4. & \mathrm{A} \to \mathrm{B} & : & \{N_B\}_{k_B} \\ 5. & \mathrm{B} \to \mathrm{A} & : & \{i\}_{N_B}\end{array}$$

In this protocol, A is a server accepting requests from good B and evil E. It is supposed to work as follows. (1) B sends its own name B to A. (2) A generates a fresh nonce N_A, pair it with its own name A, encrypts the pair with B's public key, and sends it to B. (3) B generates a fresh key N_B, tuples it with N_A and B, encrypts the tuple with A's public key, and sends it to A. (4) A encrypts N_B with B's public key and sends it to B. (5) B encrypts some secret integer i with N_B and sends it to A.

The idea of the encoding is as follows. We use sealing, unsealing and fresh seal generation as (perfect) encryption, decryption, and fresh key generation. The whole system is expressed as a tuple of (functions representing) keys known to the attacker and terms U and V representing principals B and A.

$$W \quad = \quad \langle \lambda x.\, \{x\}_{k_A}, \lambda x.\, \{x\}_{k_B}, k_E, U, V\rangle$$

Each principal is encoded as a pair of the last value it sent (if any) and a continuation function waiting to receive a next message. When the message is received, the function returns the next

$$\mathcal{S} \;=\; \{(U,U'),(V,V'),(W,W'),$$

— *corresponding keys and constants known to the attacker*

$$(\overline{k},\overline{k}'),(A,A),(B,B),(E,E),$$
$$(\lambda x.\,\{x\}_{k_A},\lambda x.\,\{x\}_{k'_A}),(\lambda x.\,\{x\}_{k_B},\lambda x.\,\{x\}_{k'_B}),(k_E,k'_E),$$
$$(\overline{w},\overline{w}'),(\{\overline{w}\}_{k_A},\{\overline{w}'\}_{k'_A}),(\{\overline{w}\}_{k_B},\{\overline{w}'\}_{k'_B}),$$

— *corresponding components from principal B at Step 1*

$$(\lambda\{\langle x,y\rangle\}_{k_B}.\,\mathtt{assert}(y=A);\nu z.\,(\{\langle x,z,B\rangle\}_{k_A},\lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0=z);\{i\}_z),$$
$$\lambda\{\langle x,y\rangle\}_{k'_B}.\,\mathtt{assert}(y=A);\nu z.\,(\{\langle x,z,B\rangle\}_{k'_A},\lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0=z);\{j\}_z)),$$

— *corresponding components from principal A at Step 2, communicating with B*

$$(\langle\{\langle\overline{k}_{AB},A\rangle\}_{k_B},\lambda\{\langle y_0,z,x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AB});\mathtt{assert}(x_0=B);\{z\}_{k_B}\rangle,$$
$$\langle\{\langle\overline{k}'_{AB},A\rangle\}_{k'_B},\lambda\{\langle y_0,z,x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AB});\mathtt{assert}(x_0=B);\{z\}_{k'_B}\rangle),$$
$$(\{\langle\overline{k}_{AB},A\rangle\}_{k_B},\{\langle\overline{k}'_{AB},A\rangle\}_{k'_B}),$$
$$(\lambda\{\langle y_0,z,x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AB});\mathtt{assert}(x_0=B);\{z\}_{k_B},$$
$$\lambda\{\langle y_0,z,x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AB});\mathtt{assert}(x_0=B);\{z\}_{k'_B}),$$

— *corresponding components from principal B at step 3, communicating with A*

$$(\langle\{\langle\overline{k}_{AB},\overline{k}_B,B\rangle\}_{k_A},\lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0=\overline{k}_{AB});\{i\}_{\overline{k}_B}\rangle,$$
$$\langle\{\langle\overline{k}'_{AB},\overline{k}'_B,B\rangle\}_{k'_A},\lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0=\overline{k}'_{AB});\{j\}_{\overline{k}'_B}\rangle),$$
$$(\{\langle\overline{k}_{AB},\overline{k}_B,B\rangle\}_{k_A},\{\langle\overline{k}'_{AB},\overline{k}'_B,B\rangle\}_{k'_A}),$$
$$(\lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0=\overline{k}_{AB});\{i\}_{\overline{k}_B},$$
$$\lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0=\overline{k}'_{AB});\{j\}_{\overline{k}'_B}),$$

— *corresponding components from principal A at step 4, communicating with B*

$$(\{\overline{k}_B\}_{k_B},\{\overline{k}'_B\}_{k'_B}),$$

— *corresponding components from principal B at step 5, communicating with A*

$$(\{i\}_{\overline{k}_B},\{j\}_{\overline{k}'_B}),$$

— *corresponding components from principal A at Step 2, communicating with E*

$$(\langle\{\langle\overline{k}_{AE},A\rangle\}_{k_E},\lambda\{\langle y_0,z,x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AE});\mathtt{assert}(x_0=E);\{z\}_{k_E}\rangle,$$
$$\langle\{\langle\overline{k}'_{AE},A\rangle\}_{k'_E},\lambda\{\langle y_0,z,x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AE});\mathtt{assert}(x_0=E);\{z\}_{k'_E}\rangle),$$
$$(\{\langle\overline{k}_{AE},A\rangle\}_{k_E},\{\langle\overline{k}'_{AE},A\rangle\}_{k'_E}),$$
$$(\lambda\{\langle y_0,z,x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AE});\mathtt{assert}(x_0=E);\{z\}_{k_E},$$
$$\lambda\{\langle y_0,z,x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AE});\mathtt{assert}(x_0=E);\{z\}_{k'_E}),$$
$$(\langle\overline{k}_{AE},A\rangle,\langle\overline{k}'_{AE},A\rangle),$$
$$(\overline{k}_{AE},\overline{k}'_{AE}),$$

— *corresponding components from principal B at Step 3, communicating with E*

$$(\langle\{\langle\overline{w},\overline{k}_B,B\rangle\}_{k_A},\lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0=\overline{k}_B);\{i\}_{\overline{k}_B}\rangle,$$
$$\langle\{\langle\overline{w}',\overline{k}'_B,B\rangle\}_{k'_A},\lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0=\overline{k}'_B);\{j\}_{\overline{k}'_B}\rangle),$$
$$(\{\langle\overline{w},\overline{k}_B,B\rangle\}_{k_A},\{\langle\overline{w}',\overline{k}'_B,B\rangle\}_{k'_A}),$$
$$(\lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0=\overline{k}_B);\{i\}_{\overline{k}_B},$$
$$\lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0=\overline{k}'_B);\{j\}_{\overline{k}'_B})),$$

— *corresponding components from principal A at Step 4, communicating with E*

$$(\{\overline{w}\}_{k_E},\{\overline{w}'\}_{k'_E})\}$$

Figure 3.3: Bisimulation for the Needham-Schroeder-Lowe protocol

state of the principal. Communication occurs by a context applying these functions in an appropriate order (when the environment is behaving normally) or perhaps in some strange, arbitrary order (when the environment is under the control of a malicious attacker). Thus, contexts play the role of the network, scheduler, and attackers. More details about the encoding—including a more detailed justification of the claim that it *is* a reasonable encoding of the protocol above—can be found in the previous chapter. We write $\mathtt{assert}(e_1); e_2$ as syntactic sugar for $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ \bot$.

$$\begin{array}{rcl}
U & = & \langle B, \lambda\{\langle x, y\rangle\}_{k_B}.\,\mathtt{assert}(y = A); \\
 & & \qquad \nu z.\,\langle\{\langle x, z, B\rangle\}_{k_A}, \\
 & & \qquad\quad \lambda\{z_0\}_{k_B}.\,\mathtt{assert}(z_0 = z); \\
 & & \qquad\qquad \{i\}_z\rangle\rangle \\
V & = & \lambda x.\,\mathtt{let}\ k_x = (\mathtt{if}\ x = B\ \mathtt{then}\ k_B\ \mathtt{else} \\
 & & \qquad \mathtt{if}\ x = E\ \mathtt{then}\ k_E\ \mathtt{else}\ \bot)\ \mathtt{in} \\
 & & \quad \nu y.\,\langle\{(y, A)\}_{k_x}, \\
 & & \qquad \lambda\{\langle y_0, z, x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0 = y); \\
 & & \qquad\qquad \mathtt{assert}(x_0 = x); \\
 & & \qquad\qquad \{z\}_{k_x}\rangle
\end{array}$$

Now, take any integers i and j. We prove that the system W above (where the secret value sent from B to A is i) and the system W' below (where the secret is j) are bisimilar, which means that the protocol keeps i and j secret against attackers.

$$\begin{array}{rcl}
U' & = & \langle B, \lambda\{\langle x, y\rangle\}_{k'_B}.\,\mathtt{assert}(y = A); \\
 & & \qquad \nu z.\,\langle\{\langle x, z, B\rangle\}_{k'_A}, \\
 & & \qquad\quad \lambda\{z_0\}_{k'_B}.\,\mathtt{assert}(z_0 = z); \\
 & & \qquad\qquad \{j\}_z\rangle\rangle \\
V' & = & \lambda x.\,\mathtt{let}\ k_x = (\mathtt{if}\ x = B\ \mathtt{then}\ k'_B\ \mathtt{else} \\
 & & \qquad \mathtt{if}\ x = E\ \mathtt{then}\ k'_E\ \mathtt{else}\ \bot)\ \mathtt{in} \\
 & & \quad \nu y.\,\langle\{(y, A)\}_{k_x}, \\
 & & \qquad \lambda\{\langle y_0, z, x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0 = y); \\
 & & \qquad\qquad \mathtt{assert}(x_0 = x); \\
 & & \qquad\qquad \{z\}_{k_x}\rangle \\
W' & = & \langle \lambda x.\,\{x\}_{k'_A}, \lambda x.\,\{x\}_{k'_B}, k'_E, U', V'\rangle
\end{array}$$

The construction of the bisimulation X is by induction, following the same basic pattern as Example 3.16. The base case is $(\emptyset, \emptyset, \emptyset) \in X$. The induction rule is as follows. Take any $(s, s', \mathcal{R}) \in X$. Take any $\overline{w} = [\overline{v}/\overline{x}]\overline{e}$ and $\overline{w}' = [\overline{v}'/\overline{x}]\overline{e}$ with $(\overline{v}, \overline{v}') \in \mathcal{R}$ and $\mathit{Seals}(\overline{e}) = \emptyset$. Take any $t \supseteq s$ and $t' \supseteq s'$ of the forms $\{k_A, k_B, k_E, \overline{k}_{AB}, \overline{k}_{AE}, \overline{k}_B, \overline{k}\}$ and $\{k'_A, k'_B, k'_E, \overline{k}'_{AB}, \overline{k}'_{AE}, \overline{k}'_B, \overline{k}'\}$. Then, $(t, t', \mathcal{T}) \in X$ for any subset $\mathcal{T}$ of the set $\mathcal{S}$ given in Figure 3.3. It is routine to check the conditions of bisimulation for this X.

It is well known that the secrecy property does not hold for the original version of this protocol (i.e., without Lowe's fix), in which the third message is $\{N_A, N_B\}_{k_A}$ instead of $\{N_A, N_B, B\}_{k_A}$ (i.e., the B is missing). This flaw is mirrored in our setting as well: if we tried to construct a bisimulation for this version in the same way as above, it would fail to be a bisimulation for the following reason. Since we would have

$$(\{\langle\overline{w}, \overline{k}_B\rangle\}_{k_A}, \{\langle\overline{w}', \overline{k}'_B\rangle\}_{k'_A}) \in \mathcal{S}$$

instead of

$$(\{\langle\overline{w},\overline{k}_B,B\rangle\}_{k_A},\{\langle\overline{w}',\overline{k}'_B,B\rangle\}_{k'_A})\in\mathcal{S}$$

along with $(\overline{k}_{AE},\overline{k}'_{AE})\in\mathcal{S}$, we would have $(\{\langle\overline{k}_{AE},\overline{k}_B\rangle\}_{k_A},\{\langle\overline{k}'_{AE},\overline{k}'_B\rangle\}_{k'_A})\in\mathcal{S}$ by taking $\overline{w}=\overline{k}_{AE}$ and $\overline{w}'=\overline{k}'_{AE}$ in the definition of X above. Since we would have

$$\begin{aligned}(&\lambda\{\langle y_0,z\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AE});\{z\}_{k_E},\\ &\lambda\{\langle y_0,z\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AE});\{z\}_{k'_E})\in\mathcal{S}\end{aligned}$$

as well instead of

$$\begin{aligned}(&\lambda\{\langle y_0,z,x_0\rangle\}_{k_A}.\,\mathtt{assert}(y_0=\overline{k}_{AE});\mathtt{assert}(x_0=E);\{z\}_{k_E},\\ &\lambda\{\langle y_0,z,x_0\rangle\}_{k'_A}.\,\mathtt{assert}(y_0=\overline{k}'_{AE});\mathtt{assert}(x_0=E);\{z\}_{k'_E})\in\mathcal{S}\end{aligned}$$

we should also have $(\{\overline{k}_B\}_{k_E},\{\overline{k}'_B\}_{k'_E})\in\mathcal{S}$ by applying these functions to the previous ciphertexts, according the condition of bisimulation for functions (Condition 7). Furthermore, since $(k_E,k'_E)\in\mathcal{S}$, we would need $(\overline{k}_B,\overline{k}'_B)\in\mathcal{S}$ as well, according to the condition of bisimulation for sealed values (Condition 6). Then, since $(\{i\}_{\overline{k}_B},\{j\}_{\overline{k}'_B})\in\mathcal{S}$, we should require $(i,j)\in\mathcal{S}$. This contradicts with the condition of bisimulation for constants (Condition 3) if $i\neq j$. Observe how the same attack is prevented in the fixed version of this protocol: the assertion $\mathtt{assert}(x_0=E)$ fails since x_0 is bound to B.

3.5 Soundness and Completeness

Bisimilarity, written $\sim$, is the largest bisimulation. It exists because the union of two bisimulations is always a bisimulation. We will need several simple lemmas about bisimulation in the development that follows.

Lemma 3.18 (Monotonicity). Take any bisimulation X. For any $(s,s',\mathcal{R})\in X$ and $(t,t',\mathcal{S})\in X$ with $\mathcal{R}\subseteq\mathcal{S}$, if $(s)\,v\;X_{\mathcal{R}}\;(s')\,v'$, then $(t)\,v\;X_{\mathcal{S}}\;(t')\,v'$.

Proof. Immediate from the definitions of $(s)\,v\;X_{\mathcal{R}}\;(s')\,v'$ and $(t)\,v\;X_{\mathcal{S}}\;(t')\,v'$. □

Lemma 3.19 (Addition of Fresh Seals). Take any bisimulation X and $(s,s',\mathcal{R})\in X$. Then, $X\cup\{(s\uplus\{k\},s'\uplus\{k'\},\mathcal{R}\uplus\{(k,k')\})\}$ is a bisimulation for any $k\notin s$ and $k'\notin s'$.

Proof. Straightforward by checking the conditions of bisimulation. □

We want to show that the bisimilarity $\sim$ coincides with the contextual equivalence $\equiv$. Since we defined $\sim$ by co-induction, the easy direction is showing that contextual equivalence implies bisimilarity.

Lemma 3.20 (Completeness of Bisimilarity). $\equiv\,\subseteq\,\sim$.

Proof. Since $\sim$ is the greatest bisimulation, it suffices to check that $\equiv$ is a bisimulation. Condition 1 is immediate since it is the same as Condition (1) in the definition of contextual equivalence. Condition 2 follows by considering contexts which destruct v and v', i.e., a context applying them as functions, a context projecting them as tuples, etc. Condition 3 follows by considering a context like $\texttt{if}\ [\,] = c\ \texttt{then}\ ()\ \texttt{else}\ \bot$. Condition 4 follows from Lemma 3.7 (2). Condition 5 follows by considering a context like $\texttt{let}\ \{x\}_y = \{()\}_z\ \texttt{in}\ x\ \texttt{else}\ \bot$, setting $y = k_1$ and $z = k_2$ in the left-hand side, and $y = k_1'$ and $z = k_2'$ in the right-hand side. Condition 6 follows by considering contexts of the form $\texttt{let}\ \{x\}_y = z\ \texttt{in}\ e$—setting $y = k$ and $z = \{v\}_k$ in the left-hand side, and $y = k'$ and $z = \{v'\}_{k'}$ in the right-hand-side—and by Lemma 3.7 (4). Condition 7 follows by Lemma 3.7 (1) together with Lemma 3.7 (3) to add the fresh seals $\overline{k}$ and $\overline{k}'$, Lemma 3.8 to make the arguments v and v', and Lemma 3.9 to remove them after the applications. □

Next, we need to prove soundness, i.e., that bisimilarity implies contextual equivalence. For this purpose, we define the following relation.

Definition 3.21 (Bisimilarity in Context). We define $\cong$ as

$$\{(s, s', \mathcal{R}, [\overline{v}/\overline{x}]e_0, [\overline{v}'/\overline{x}]e_0) \mid (s)\,\overline{v} \sim_{\mathcal{R}} (s')\,\overline{v}' \wedge \mathit{Seals}(e_0) = \emptyset\}$$

where $(s)\,\overline{v} \sim_{\mathcal{R}} (s')\,\overline{v}'$ is a shorthand for $(s)\,v_1 \sim_{\mathcal{R}} (s')\,v_1' \wedge \cdots \wedge (s)\,v_n \sim_{\mathcal{R}} (s')\,v_n'$.

We write $(s)\,e \cong_{\mathcal{R}} (s')\,e'$ for $(s, s', \mathcal{R}, e, e') \in \cong$. The intuition of this definition is: $\cong$ relates bisimilar values $\overline{v}$ and $\overline{v}'$ put in context e_0.

The two lemmas below are the key properties of our bisimulation. The first states that evaluation preserves $\cong$, the second that $\cong$ implies observational equivalence (i.e., if evaluation of one expression converges, then evaluation of the other expression also converges).

Lemma 3.22 (Fundamental Property, Part I). Suppose $(s_0)\,e \cong_{\mathcal{R}_0} (s_0')\,e'$. If $(s_0)\,e \Downarrow (t)\,w$ and $(s_0')\,e' \Downarrow (t')\,w'$, then $(t)\,w \cong_{\mathcal{R}} (t')\,w'$ for some $\mathcal{R} \supseteq \mathcal{R}_0$.

Proof. By induction on the derivation of $(s_0)\,e \Downarrow (t)\,w$. See Appendix B.1 for details. □

Lemma 3.23 (Fundamental Property, Part II). If $(s_0)\,e \cong_{\mathcal{R}_0} (s_0')\,e'$, then $(s_0)\,e \Downarrow \iff (s_0')\,e' \Downarrow$.

Proof. By induction on the derivation of $(s_0)\,e \Downarrow$. Details are found in Appendix B.2. □

An immediate consequence of the previous property is that bisimulation implies contextual equivalence.

Corollary 3.24 (Soundness of Bisimilarity). $\sim \subseteq \equiv$.

Proof. Suppose $(s, s', \mathcal{R}) \in \sim$ and we shall prove $(s, s', \mathcal{R}) \in \equiv$. The first condition of contextual equivalence is immediate since it is the same as the first condition of bisimulation. The second condition of contextual equivalence is proved as follows. Take any $(\overline{v}, \overline{v}') \in \mathcal{R}$ and any e with $\mathit{Seals}(e) = \emptyset$. By definition, $(s)\,[\overline{v}/\overline{x}]e \cong_{\mathcal{R}} (s')\,[\overline{v}'/\overline{x}]e$. Thus, by Lemma 3.23, $(s)\,[\overline{v}/\overline{x}]e \Downarrow \iff (s')\,[\overline{v}'/\overline{x}]e \Downarrow$. □

Combining soundness and completeness, we obtain the main theorem about our bisimulation: that bisimilarity coincides with contextual equivalence.

Theorem 3.25. $\sim = \equiv$.

Proof. By Lemma 3.20 and Corollary 3.24. □

3.6 Extension with Equality for Sealed Values

A number of variants of $\lambda_{\texttt{seal}}$ can be considered. For example, the version of $\lambda_{\texttt{seal}}$ in this chapter does not allow a context to test two sealed values for equality. This is reasonable if the environment is a safe runtime system (where sealing can be implemented just by tagging) which disallows comparison of sealed values. It is unrealistic, however, to expect such a restriction in an arbitrary (perhaps hostile) environment, where sealing must be implemented by encryption. Fortunately, our technique extends directly to such a modest change as adding equality for sealed values. For instance, it is straightforward to extend $\lambda_{\texttt{seal}}$ with syntactic equality $=_1$ for first-order values (including sealed values) along with an additional condition of bisimulation: $v_1 =_1 v_2 \iff v_1' =_1 v_2'$ for every $(v_1, v_1') \in \mathcal{R}$ and $(v_2, v_2') \in \mathcal{R}$. Then, it is also straightforward to prove the soundness and completeness of bisimilarity under this extension, with an additional lemma that $\cong$ respects $=_1$ (which can be proved by induction on the syntax of values being compared).

Of course, the more observations we allow, the more difficult it becomes to establish the equivalence of two given modules. For example, the two implementations of complex numbers given in the introduction are no longer equivalent (or bisimilar) under the extension above, because there are many polar representations of $0 + 0i$ while there is only one Cartesian representation. So, for example, a context like

$$\begin{aligned} C[\,] \;=\;& \texttt{let x = [].from\_re\_and\_im(0.0, 0.0) in} \\ & \texttt{let y = [].from\_re\_and\_im(-1.0, 0.0) in} \\ & \texttt{x } =_1 \texttt{ [].multiply x y} \end{aligned}$$

would distinguish `CartesianComplex` and `PolarComplex`. To recover the equivalence, the polar representation of $0+0i$ must be standardized and checks inserted wherever it can be created:

```
let from_re_and_im =
  fun (x, y) ->
    let z =
      if x = 0.0 && y = 0.0 then (0.0, 0.0) else
      (sqrt(x * x + y * y), atan2(y, x)) in
    <seal z under k>
let multiply =
  fun (z1, z2) ->
    let (r1, t1) = <unseal z1 under k> in
    let (r2, t2) = <unseal z2 under k> in
    let z =
      if r1 = 0.0 || r2 = 0.0 then (0.0, 0.0) else
      (r1 * r2, t1 + t2) in
    <seal z under k>
```

3.7 Related Work

As discussed in the introduction, sealing was first proposed by Morris [1973b, 1973a] and has been revisited in more recent work on extending the "scope" (in both informal and technical senses) of type abstraction in various forms [Dreyer, Crary, and Harper 2003; Rossberg 2003; Leifer, Peskine, Sewell, and Wansbrough 2003; Sewell 2001].

Bisimulations have been studied extensively in process calculi. In particular, bisimulations for the spi-calculus [Abadi and Gordon 1998; Borgström and Nestmann 2002; Boreale, De Nicola, and Pugliese 2002; Abadi and Fournet 2001] are the most relevant to this work, because the perfect encryption in spi-calculus is very similar to dynamic sealing in our calculus. Our bisimulation is analogous to bisimulations for spi-calculus in that both keep track of the environment's knowledge.

However, since processes and messages are different entities in spi-calculus, all the technicalities—i.e., definitions and proofs—must be developed separately for processes and messages. By contrast, our bisimulation is monolithic and more straightforward. In particular, the condition of our bisimulation for functions (Condition 7 in Definition 3.10) is simpler than conditions of bisimulations in spi-calculus for the case of input, where the received messages are defined by another large set of separate rules and/or not so much restricted as function arguments in Condition 7, leading to more complex bisimulations.

Furthermore, it is possible even to encode and verify some (though not all) security protocols in our framework. The encoding naturally models the concurrency among principals and attackers (including so-called "necessarily parallel" attacks) by means of interleaving. Thanks to higher-order functions, we can also simulate asymmetric encryption and can thereby express public-key protocols such as Needham-Schroeder-Lowe (unlike the spi-calculus) without extending the calculus. See the previous chapter for further discussion about this encoding of security protocols.

While our bisimulation is complete with respect to contextual equivalence, no completeness proof is available for spi-calculus bisimulations: the original [Abadi and Gordon 1998] is known to be incomplete; others [Borgström and Nestmann 2002; Boreale, De Nicola, and Pugliese 2002] are proved to be complete only for some subset of processes (called *structurally image-finite* processes) despite the claim of completeness for one of them [Borgström and Nestmann 2002]; proof of (soundness and) completeness for bisimilarity in applied π-calculus [Abadi and Fournet 2001, Theorem 1] has not been written down in a form accessible to others [personal communication, August 2004].

Another line of work on bisimulations in process calculi concerns techniques for lightening the burden of constructing a bisimulation—e.g., "bisimulation up to" [Sangiorgi and Milner 1992]. It remains to be seen whether these techniques would be useful in our setting. Note that our operational semantics is built upon big-step evaluation (as opposed to small-step reduction) in the first place, which cuts down the intermediate terms and reduces the size of a bisimulation.

Abramsky [1990] studied *applicative bisimulation* for the λ-calculus. For functions $\lambda x.\, e$ and $\lambda x.\, e'$ to be bisimilar, it requires that $(\lambda x.\, e)d$ and $(\lambda x.\, e')d$ are observationally equivalent for any closed d, and that they evaluate to bisimilar values if the evaluations converge. Thus, it requires the two arguments to be the same, which actually makes the soundness proof harder [Howe 1996]. We avoided this problem by allowing some difference between the arguments of functions in our bisimulation.

Jeffrey and Rathke [1999] defined bisimulation for λ-calculus with name generation, of which our seal generation is an instance. Although their theory does distinguish private and public names, it lacks a proper mechanism to keep track of contexts' knowledge of name-involving values in general, such as functions containing names inside the bodies. As a result, they had to introduce additional language constructs—such as global references [Jeffrey and Rathke 1999] or communication channels [Jeffrey and Rathke 2004]—for the bisimulation to be sound. We solved this problem by using a set of relations (rather than a single relation) between values as a bisimulation, i.e., by considering multiple pairs of values at once.

A well-known method of proving the abstraction obtained by type abstraction is logical relations [Reynolds 1983; Mitchell 1991]. Although they are traditionally defined on denotational models, they have recently been studied in the syntactic setting of term models as well [Pitts 1998; Pitts 2000]. In the previous chapter, we have defined syntactic logical relations for perfect encryption and used them to prove secrecy properties of security protocols. Although logical relations are analogous to bisimulations in that both relate corresponding values between two different programs, logical relations are defined by induction on types and cannot be applied in untyped settings. Moreover, logical relations in more sophisticated settings (such as recursive functions and recursive types) than simply typed λ-calculus tend to become rather complicated. Indeed, "keys encrypting keys" (as in security protocols) required non-trivial extension in the logical relations above, while they imposed no difficulty to our bisimulation in this chapter.

3.8 Future Work

We have defined a bisimulation for $\lambda_{\texttt{seal}}$ and proved its soundness and completeness with respect to contextual equivalence.

There are several directions for future work. One is to apply our bisimulation to more examples, e.g., to prove the full abstraction of our translation of type abstraction into dynamic sealing—indeed, this was actually the original motivation for the present work. When the target language is untyped, the translation of source term $\vdash M : \tau$ can be given as $\texttt{let}\ x = \mathit{erase}(M)\ \texttt{in}\ \mathcal{E}^{+}_{\emptyset}(x, \tau)$, where $\mathcal{E}^{+}$ is defined like Figure 3.4 along with its dual $\mathcal{E}^{-}$ in a type-directed manner. Intuitively, $\mathcal{E}^{+}$ is a "firewall" that protects terms from contexts, where $\mathcal{E}^{-}$ is a "sandbox" that protects contexts from terms. Bisimulation would help proving properties of this translation. We may also be able to use such an interpretation of type abstraction by dynamic sealing as a (both formal and informal) basis for reasoning about type abstraction in broader settings.

It would also be interesting to extend our developments for more general operations on sealing/encryption as in applied pi-calculus [Abadi and Fournet 2001]. Recall that the syntax and semantics in Section 3.2 did not allow any primitive op (or constant c) to involve seals.

Mechanical support for bisimulation proofs is of natural interest as well. Full automation is hopeless, since general cases subsume the halting problem (i.e., whether the evaluation of a λ-expression converges or diverges), but many of the conditions of bisimulation are easy to check or satisfy by adding elements to the bisimulation. One challenging point would be the case analysis on function arguments $[\overline{u}/\overline{x}]d$ and $[\overline{u}'/\overline{x}]d$ in Condition 7, shown in detail in Example 3.13.

$$
\begin{array}{lcll}
\mathcal{E}^+_\rho(x, \texttt{bool}) & = & x & \\
\mathcal{E}^+_\rho(x, \tau_1 \times \cdots \times \tau_n) & = & \texttt{let}\ \langle y_1, \ldots, y_n \rangle = x\ \texttt{in} & \\
 & & \langle \mathcal{E}^+_\rho(y_1, \tau_1), \ldots, \mathcal{E}^+_\rho(y_n, \tau_n) \rangle & \\
\mathcal{E}^+_\rho(x, \tau \to \sigma) & = & \lambda y.\, \texttt{let}\ z = x\mathcal{E}^-_\rho(y, \tau)\ \texttt{in}\ \mathcal{E}^+_\rho(z, \sigma) & \\
\mathcal{E}^+_\rho(x, \forall \alpha.\, \tau) & = & \lambda y.\, \texttt{let}\ z = x()\ \texttt{in}\ \mathcal{E}^+_\rho(z, \tau) & \\
\mathcal{E}^+_\rho(x, \exists \alpha.\, \tau) & = & \nu z.\, \mathcal{E}^+_{\rho, \alpha \mapsto z}(x, \tau) & \\
\mathcal{E}^+_\rho(x, \alpha) & = & \{x\}_{\rho(\alpha)} & \\
\mathcal{E}^+_\rho(x, \alpha) & = & x & \text{if } \alpha \notin Dom(\rho) \\
 & & & \\
\mathcal{E}^-_\rho(x, \texttt{bool}) & = & \texttt{if}\ x\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false} & \\
\mathcal{E}^-_\rho(x, \tau_1 \times \cdots \times \tau_n) & = & \texttt{let}\ \langle y_1, \ldots, y_n \rangle = x\ \texttt{in} & \\
 & & \langle \mathcal{E}^-_\rho(y_1, \tau_1), \ldots, \mathcal{E}^-_\rho(y_n, \tau_n) \rangle & \\
\mathcal{E}^-_\rho(x, \tau \to \sigma) & = & \lambda y.\, \texttt{let}\ z = x\mathcal{E}^+_\rho(y, \tau)\ \texttt{in}\ \mathcal{E}^-_\rho(z, \sigma) & \\
\mathcal{E}^-_\rho(x, \forall \alpha.\, \tau) & = & \lambda y.\, \nu z.\, \mathcal{E}^-_{\rho, \alpha \mapsto z}(x, \tau) & \\
\mathcal{E}^-_\rho(x, \exists \alpha.\, \tau) & = & \mathcal{E}^-_\rho(x, \tau) & \\
\mathcal{E}^-_\rho(x, \alpha) & = & \texttt{let}\ \{y\}_{\rho(\alpha)} = x\ \texttt{in}\ y\ \texttt{else}\ \bot & \\
\mathcal{E}^-_\rho(x, \alpha) & = & x & \text{if } \alpha \notin Dom(\rho)
\end{array}
$$

Figure 3.4: Translation of type abstraction into dynamic sealing

Chapter 4

A Bisimulation for Type Abstraction and Recursion

Overview

We present a sound, complete, and elementary proof method, based on bisimulation, for contextual equivalence in a λ-calculus with full universal, existential, and recursive types. Unlike logical relations (either semantic or syntactic), our development is elementary, using only sets and relations and avoiding advanced machinery such as domain theory, admissibility, and $\top\top$-closure. Unlike other bisimulations, ours is complete even for existential types. The key idea is to consider *sets* of relations—instead of just relations—as bisimulations.

Results in this chapter are to appear in Sumii and Pierce [2004b].

4.1 Introduction

Proving the equivalence of computer programs is important not only for verifying the correctness of program transformations such as compiler optimizations, but also for showing the compatibility of program modules. Consider two modules M and M' implementing the same interface I; if these different implementations are equivalent under this common interface, then they are indeed compatible, correctly hiding their differences from outside view.

Contextual equivalence is a natural definition of program equivalence: two programs are called contextually equivalent if they exhibit the same observable behavior when put in any legitimate context of the language. However, direct proofs of contextual equivalence are typically infeasible, because its definition involves a universal quantification over an infinite number of contexts (and naive approaches such as structural induction on the syntax of contexts do not work). This has led to a search for alternative methods for proving contextual equivalence, whose fruits can be grouped into two categories: *logical relations* and *bisimulations*.

Logical relations (and their shortcomings). Logical relations were first developed for denotational semantics of typed λ-calculi (see, e.g., [Mitchell 1996, Chapter 8] for details) and can also be adapted [Pitts 2000; Pitts 1998] to their term models; this adaptation is sometimes called syntactic logical relations [Crary and Harper 2000]. Logical relations are relations on terms defined by induction on their types: for instance, two pairs are related when their elements are pairwise related; two tagged terms $\mathtt{in}_i(M)$ and $\mathtt{in}_j(N)$ of a sum type are related when the tags i and j are equal and the contents M and N are also related; and, crucially, two functions are related when they map related arguments to related results. The soundness of logical relations is proved via the Fundamental Property (or Basic Lemma), which states that any well-typed term is related to itself.

Logical relations are pleasantly straightforward, as long as we stick to the simply typed λ-calculus (or even the polymorphic λ-calculus) without recursion. However, their extension with recursion is challenging. Recursive functions cause a problem in the proof of the fundamental property that must be addressed by introducing additional "unwinding properties" [Pitts 2000; Pitts 1998; Birkedal and Harper 1999; Crary and Harper 2000]. Recursive *types* are even more difficult (in particular with negative occurrences): since logical relations are defined by induction on types, recursive types require topological properties even in the *definition* of logical relations [Birkedal and Harper 1999; Crary and Harper 2000]. Worse, these difficulties are not confined to meta-theorems, but are visible to the users of logical relations: in order to prove contextual equivalence using logical relations, one often has to prove the admissibility, compute the limit, or calculate the $\top\top$-closure of particular logical relations.

Bisimulations (and their shortcomings). Bisimulations were originally developed for process calculi [Milner 1980; Milner 1995; Milner 1999] and state transition systems in general. Abramsky [1990] adapted bisimulations to untyped λ-calculus and called them *applicative bisimulations*. Briefly, two functions $\lambda x.\, M$ and $\lambda x.\, M'$ are bisimilar when $(\lambda x.\, M)N \Downarrow \iff (\lambda x.\, M')N \Downarrow$ for any N and the results are also bisimilar if these evaluations converge. Gordon and Rees [Gordon 1995a; Gordon and Rees 1996; Gordon 1995b; Gordon and Rees 1995] extended applicative bisimulations to calculi with objects, subtyping, universal polymorphism, and recursive types. Sangiorgi [Sangiorgi 1992] has defined *context bisimulation*, which is a variant of applicative bisimulation for higher-order π-calculus [Sangiorgi 1992].

Unlike logical relations, bisimulations have no difficulty with recursion (or even concurrency). However, existing bisimulation methods for typed λ-calculi are very weak in the presence of existential polymorphism; that is, they are useless for proving interesting equivalence properties of existential packages. For instance, consider the two packages

$$\begin{aligned} M &= \mathtt{pack}\ \mathtt{int}, \langle 1, \lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0\rangle\ \mathtt{as}\ \tau \\ M' &= \mathtt{pack}\ \mathtt{bool}, \langle \mathtt{true}, \lambda x\!:\!\mathtt{bool}.\, \neg x\rangle\ \mathtt{as}\ \tau \end{aligned}$$

where $\tau = \exists\alpha.\, \alpha \times (\alpha \to \mathtt{bool})$. Existing bisimulation methods cannot prove the contextual equivalence $\vdash M \equiv M' : \tau$ of these simple packages, because they cannot capture the fact that the only values of type α are 1 in the "left-hand world" and $\mathtt{true}$ in the right-hand world. The same observation applies to context bisimulation.

The only exceptions to the problem above are bisimulations for polymorphic π-calculi [Pierce and Sangiorgi 2000; Berger, Honda, and Yoshida 2003]. However, π-calculus is name-based and low-level. As a result, it is rather difficult to encode polymorphic λ-calculus into polymorphic π-calculus while preserving equivalence (though there are some results [Berger, Honda, and Yoshida 2003] for the case without recursion), so it is at least as difficult to use π-calculus for reasoning about abstraction in λ-calculus or similar languages with (in particular higher-order) functions and recursion. In addition to the problem of encoding, existing bisimulations for polymorphic

π-calculi are incomplete [Pierce and Sangiorgi 2000] and complex [Berger, Honda, and Yoshida 2003].

Encoding existential polymorphism in terms of universal polymorphism does not help either. Consider the following encodings of M and M'

$$\begin{array}{rcl} N & = & \lambda f:\sigma.\, f[\mathtt{int}]\langle 1, \lambda x:\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0\rangle \\ N' & = & \lambda f:\sigma.\, f[\mathtt{bool}]\langle \mathtt{true}, \lambda x:\mathtt{bool}.\, \neg x\rangle \end{array}$$

where $\sigma = \forall\alpha.\, \alpha \times (\alpha \to \mathtt{bool}) \to \mathtt{ans}$ and $\mathtt{ans}$ is some answer type. In order to establish the bisimulation between N and N', one has at least to prove

$$f[\mathtt{int}]\langle 1, \lambda x:\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0\rangle \Downarrow \quad\Longleftrightarrow\quad f[\mathtt{bool}]\langle \mathtt{true}, \lambda x:\mathtt{bool}.\, \neg x\rangle \Downarrow$$

for any observer function f of type σ, which is almost the same as the *definition* of $\vdash M \equiv M' : \tau$.

Our solution. We address these problems—and thereby obtain a sound and complete bisimulation for existential types (as well as universal and recursive types)—by adapting key ideas from the previous chapter on bisimulation for *sealing* [Morris 1973a; Morris 1973b], a dynamic form of data abstraction. The crucial insight is that we should define bisimulations as *sets* of relations—rather than just relations—annotated with type information.

For instance, a bisimulation X showing the contextual equivalence of M and M' above can be defined (roughly) as

$$X \quad = \quad \{(\emptyset, \mathcal{R}_0), (\Delta, \mathcal{R}_1), (\Delta, \mathcal{R}_2), (\Delta, \mathcal{R}_3)\}$$

where

$$\begin{array}{rcl} \mathcal{R}_0 & = & \{(M, M', \tau)\} \\ \mathcal{R}_1 & = & \mathcal{R}_0 \cup \{(\langle 1, \lambda x:\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0\rangle,\ \langle \mathtt{true}, \lambda x:\mathtt{bool}.\, \neg x\rangle,\ \alpha \times (\alpha \to \mathtt{bool}))\} \\ \mathcal{R}_2 & = & \mathcal{R}_1 \cup \{(1, \mathtt{true}, \alpha), \\ & & \qquad (\lambda x:\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0,\ \lambda x:\mathtt{bool}.\, \neg x,\ \alpha \to \mathtt{bool})\} \\ \mathcal{R}_3 & = & \mathcal{R}_2 \cup \{(\mathtt{false}, \mathtt{false}, \mathtt{bool})\} \\ \Delta & = & \{(\alpha, \mathtt{int}, \mathtt{bool})\}. \end{array}$$

Because we are ultimately interested in the equivalence of M and M', we begin by including $(\emptyset, \mathcal{R}_0)$ in X. (The role of the first element $\emptyset$ of this pair will be explained in a moment.) Next, since a context can open those packages and examine their contents, we add $(\Delta, \mathcal{R}_1)$ to X, where Δ is a *concretion environment* mapping the abstract type α to its respective concrete types in the left-hand side and the right-hand side. Then, since the contents of the packages are pairs, a context can examine their elements, so we add $(\Delta, \mathcal{R}_2)$ to X. Last, since the second elements of the pairs are functions of type $\alpha \to \mathtt{bool}$, a context can apply them to any arguments of type α; the only such arguments are, in fact, 1 in the left-hand world and $\mathtt{true}$ in the right-hand world, so we add $(\Delta, \mathcal{R}_3)$ to X. Since the results of these applications are equal as booleans, there is nothing else that a context can do to distinguish the values in R_3.

Conceptually, each $\mathcal{R}$ occurring in a pair $(\Delta, \mathcal{R}) \in X$ represents the *knowledge* of a context at some point in time, which increases via new observations by the context. In order to prove contextual equivalence, it suffices to find a bisimulation X that is closed under this increase of contexts' knowledge. (Thus, in fact, not only X but also the singleton set $\{(\Delta, \mathcal{R}_3)\}$ is a bisimulation in our definition.)

Why do we consider a bisimulation X to be a set of $\mathcal{R}$s (with corresponding Δs) instead of taking their union in the first place? Because the latter does not exist in general! In other words, the union of two "valid" $\mathcal{R}$s is not always a valid $\mathcal{R}$. For instance, consider the union of $\mathcal{R}_3$ and its inverse $\mathcal{R}_3^{-1} = \{(V', V, \tau) \mid (V, V', \tau) \in \mathcal{R}_3\}$. Although each of them makes perfect sense by itself, taking their union is nonsensical because it confuses two different worlds (which, in fact, is not even type-safe). This observation is absolutely fundamental in the presence of type abstraction (or other forms of information hiding such as sealing), and it forms the basis of many technicalities in the present chapter (as well as the previous chapter). By considering a set of relations instead of taking their union, it becomes straightforward to define bisimilarity to be the largest bisimulation and thereby apply standard co-inductive arguments—in order to prove the completeness of bisimilarity, for instance. (In addition, this also gives a natural account to the *generativity* of existential types, i.e., to the fact that opening the same package twice gives incompatible contents.) Thus, for example, both $\{(\Delta, \mathcal{R}_3)\}$ and $\{(\Delta^{-1}, \mathcal{R}_3^{-1})\}$ are bisimulations (where $\Delta^{-1} = \{(\alpha, \tau, \tau') \mid (\alpha, \tau', \tau) \in \Delta\}$) and so is their union $\{(\Delta, \mathcal{R}_3), (\Delta^{-1}, \mathcal{R}_3^{-1})\}$, but neither $\{(\Delta, \mathcal{R}_3 \cup \mathcal{R}_3^{-1})\}$ nor $\{(\Delta^{-1}, \mathcal{R}_3 \cup \mathcal{R}_3^{-1})\}$ is.

This decision does not incur any significant difficulty for users of our bisimulation: we devise a trick—explained below, in the definition of bisimulation for packages—that keeps the set of relations finite in many cases; even where this trick does not apply, it is not very difficult to define the infinite set of relations (e.g., by set comprehension or by induction) and check it against our definition of bisimulation (as we will do in Example 4.4.3 for generative functors or as we did in the previous chapter (Examples 3.16 and 3.17) for security protocols).

Contributions. This is the first sound, complete, and elementary proof method for contextual equivalence in a language with higher-order functions, impredicative polymorphism (both universal and existential), and full recursive types. As discussed above, previous results in this area were (1) limited to recursive types with no negative occurrence, (2) incomplete for existential types, and/or (3) technically involved.

Many of the ideas used here are drawn from the previous chapter on a sound and complete bisimulation for untyped λ-calculus with *dynamic sealing* (also known as *perfect encryption*). This form of information hiding is very different from static type abstraction. Given the difference, it is surprising (and interesting) in itself to find that similar ideas can be adapted to both settings. Furthermore, the language in the present chapter is typed (unlike in the previous chapter), requiring many refinements to take type information into account throughout the technical development. In general, typed equivalence is much coarser than untyped equivalence—in particular with polymorphism—because not only terms but also contexts have to respect types. Accordingly, our bisimulation keeps careful track of the mapping of abstract type variables to concrete types, substituting the former with the latter if and only if appropriate.

The rest of this chapter is structured as follows. Section 4.2 presents our language and its contextual equivalence, generalized in a non-trivial way for open types as required by the technicalities which follow. Section 4.3 defines our bisimulation. Section 4.4 gives examples to illustrate

$M, N, C, D ::=$	term
x	variable
$\texttt{fix}\ f(x:\tau):\sigma = M$	recursive function
$M\,N$	application
$\Lambda\alpha.\,M$	type function
$M[\tau]$	type application
$\texttt{pack}\ \tau, M\ \texttt{as}\ \exists\alpha.\,\sigma$	packing
$\texttt{open}\ M\ \texttt{as}\ \alpha, x\ \texttt{in}\ N$	opening
$\langle M_1, \ldots, M_n\rangle$	tupling
$\#_i(M)$	projection
$\texttt{in}_i(M)$	injection
$\texttt{case}\ M\ \texttt{of}\ \texttt{in}_1(x_1) \Rightarrow M_1 \parallel \ldots \parallel \texttt{in}_n(x_n) \Rightarrow M_n$	case branch
$\texttt{fold}(M)$	folding
$\texttt{unfold}(M)$	unfolding
$U, V, W ::=$	value
$\texttt{fix}\ f(x:\tau):\sigma = M$	recursive function
$\Lambda\alpha.\,M$	type function
$\texttt{pack}\ \tau, V\ \texttt{as}\ \exists\alpha.\,\sigma$	package
$\langle V_1, \ldots, V_n\rangle$	tuple
$\texttt{in}_i(V)$	injected value
$\texttt{fold}(V)$	folded value
$\pi, \rho, \sigma, \tau ::=$	type
α	type variable
$\tau \to \sigma$	function type
$\forall\alpha.\,\tau$	universal type
$\exists\alpha.\,\tau$	existential type
$\tau_1 \times \ldots \times \tau_n$	product type
$\tau_1 + \ldots + \tau_n$	sum type
$\mu\alpha.\,\tau$	recursive type

Figure 4.1: Syntax of $\lambda_\mu^{\forall\exists}$

its uses and Section 4.5 proves soundness and completeness of the bisimulation with respect to the generalized contextual equivalence. Section 4.6 generalizes these results, which have been restricted to closed values for simplicity, to non-values and open terms. Section 4.7 discusses a limitation of our bisimulation concerning higher-order functions. Section 4.8 discusses related work, and Section 4.9 concludes with future work.

Throughout the chapter, we use overbars as shorthands for sequences—e.g., we write $\overline{x}$, $[\overline{V}/\overline{x}]$, $(\overline{\alpha}, \overline{\sigma}, \overline{\sigma}')$ and $\overline{x}:\overline{\tau}$ instead of $x_1, \ldots, x_n$, $[V_1, \ldots, V_n/x_1, \ldots, x_n]$, $(\alpha_1, \sigma_1, \sigma_1'), \ldots, (\alpha_n, \sigma_n, \sigma_n')$ and $x_1:\tau_1, \ldots, x_n:\tau_n$ where $n \geq 0$.

$$\frac{}{(\texttt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M) \Downarrow (\texttt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M)}\text{(E-Fix)}$$

$$\frac{M \Downarrow (\texttt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M) \quad N \Downarrow V \quad [V/x][(\texttt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M)/f]M \Downarrow W}{M N \Downarrow W}\text{(E-App)}$$

$$\frac{}{\Lambda\alpha.\, M \Downarrow \Lambda\alpha.\, M}\text{(E-TAbs)} \qquad \frac{M \Downarrow \Lambda\alpha.\, N \quad [\tau/\alpha]N \Downarrow V}{M[\tau] \Downarrow V}\text{(E-TApp)}$$

$$\frac{M \Downarrow V}{\texttt{pack}\, \tau, M\, \texttt{as}\, \exists\alpha.\, \sigma \Downarrow \texttt{pack}\, \tau, V\, \texttt{as}\, \exists\alpha.\, \sigma}\text{(E-Pack)}$$

$$\frac{M \Downarrow \texttt{pack}\, \sigma, V\, \texttt{as}\, \exists\alpha.\, \tau \quad [V/x][\sigma/\alpha]N \Downarrow W}{\texttt{open}\, M\, \texttt{as}\, \alpha, x\, \texttt{in}\, N \Downarrow W}\text{(E-Open)}$$

$$\frac{M_1 \Downarrow V_1 \quad \ldots \quad M_n \Downarrow V_n}{\langle M_1, \ldots, M_n\rangle \Downarrow \langle V_1, \ldots, V_n\rangle}\text{(E-Tuple)} \qquad \frac{M \Downarrow \langle V_1, \ldots, V_i, \ldots, V_n\rangle}{\#_i(M) \Downarrow V_i}\text{(E-Proj)}$$

$$\frac{M \Downarrow V}{\texttt{in}_i(M) \Downarrow \texttt{in}_i(V)}\text{(E-Inj)}$$

$$\frac{M \Downarrow \texttt{in}_i(V) \quad [V/x_i]M_i \Downarrow W}{\texttt{case}\, M\, \texttt{of}\, \texttt{in}_1(x_1) \Rightarrow M_1 \parallel \ldots \parallel \texttt{in}_i(x_i) \Rightarrow M_i \parallel \ldots \parallel \texttt{in}_n(x_n) \Rightarrow M_n \Downarrow W}\text{(E-Case)}$$

$$\frac{M \Downarrow V}{\texttt{fold}(M) \Downarrow \texttt{fold}(V)}\text{(E-Fold)} \qquad \frac{M \Downarrow \texttt{fold}(V)}{\texttt{unfold}(M) \Downarrow V}\text{(E-Unfold)}$$

Figure 4.2: Semantics of $\lambda_\mu^{\forall\exists}$

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\text{(T-Var)}$$

$$\frac{FTV(\tau) \subseteq \Gamma \quad \Gamma, f : \tau \to \sigma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\texttt{fix}\ f(x : \tau) : \sigma = M) : \tau \to \sigma}\text{(T-Fix)} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}\text{(T-App)}$$

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\, M : \forall\alpha.\, \tau}\text{(T-TAbs)} \qquad \frac{\Gamma \vdash M : \forall\alpha.\, \sigma \quad FTV(\tau) \subseteq \Gamma}{\Gamma \vdash M[\tau] : [\tau/\alpha]\sigma}\text{(T-TApp)}$$

$$\frac{FTV(\tau) \subseteq \Gamma \quad \Gamma \vdash M : [\tau/\alpha]\sigma}{\Gamma \vdash \texttt{pack}\ \tau, M\ \texttt{as}\ \exists\alpha.\, \sigma : \exists\alpha.\, \sigma}\text{(T-Pack)}$$

$$\frac{\Gamma \vdash M : \exists\alpha.\, \tau \quad \Gamma, \alpha, x : \tau \vdash N : \sigma \quad \alpha \notin FTV(\sigma)}{\Gamma \vdash \texttt{open}\ M\ \texttt{as}\ \alpha, x\ \texttt{in}\ N : \sigma}\text{(T-Open)}$$

$$\frac{\Gamma \vdash V_1 : \tau_1 \quad \ldots \quad \Gamma \vdash V_n : \tau_n}{\Gamma \vdash \langle M_1, \ldots, M_n \rangle : \tau_1 \times \ldots \times \tau_n}\text{(T-Tuple)} \qquad \frac{\Gamma \vdash M : \tau_1 \times \ldots \times \tau_i \times \ldots \times \tau_n}{\Gamma \vdash \#_i(M) : \tau_i}\text{(T-Proj)}$$

$$\frac{\Gamma \vdash M : \tau_i \quad FTV(\tau_1) \subseteq \Gamma \quad \ldots \quad FTV(\tau_n) \subseteq \Gamma}{\Gamma \vdash \texttt{in}_i(M) : \tau_1 + \ldots + \tau_i + \ldots + \tau_n}\text{(T-Inj)}$$

$$\frac{\Gamma \vdash M : \tau_1 + \ldots + \tau_n \quad \Gamma, x_1 : \tau_1 \vdash M_1 : \tau \quad \ldots \quad \Gamma, x_n : \tau_n \vdash M_n : \tau}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \texttt{in}_1(x_1) \Rightarrow M_1 \parallel \ldots \parallel \texttt{in}_n(x_n) \Rightarrow M_n : \tau}\text{(T-Case)}$$

$$\frac{\Gamma \vdash M : [\mu\alpha.\, \tau/\alpha]\tau}{\Gamma \vdash \texttt{fold}(M) : \mu\alpha.\, \tau}\text{(T-Fold)} \qquad \frac{\Gamma \vdash M : \mu\alpha.\, \tau}{\Gamma \vdash \texttt{unfold}(M) : [\mu\alpha.\, \tau/\alpha]\tau}\text{(T-Unfold)}$$

Figure 4.3: Typing rules of $\lambda_\mu^{\forall\exists}$

4.2 Generalized Contextual Equivalence

Our language is a standard call-by-value λ-calculus with polymorphic and recursive types. (We conjecture that it would also be straightforward to adapt our method to a call-by-name setting.) Its syntax, semantics, and typing rules are given in Figures 4.1, 4.2, and 4.3. We include recursive functions $\mathtt{fix}\ f(x:\tau):\sigma = M$ as a primitive for the sake of exposition; alternatively, they can be implemented in terms of a fixed-point operator, which is typable using recursive types. We adopt the standard notion of variable binding with implicit α-conversion and write $\lambda x:\tau.\, M$ for $\mathtt{fix}\ f(x:\tau):\sigma = M$ when f is not free in M. We will write $\mathtt{let}\ x:\tau = M\ \mathtt{in}\ N$ for $(\lambda x:\tau.\, N)M$. We sometimes omit type annotations—as in $\lambda x.\, M$ and $\mathtt{let}\ x = M\ \mathtt{in}\ N$—when they are obvious from the context. The semantics is defined by the evaluation $M \Downarrow V$ of term M to value V.

For simplicity, we consider the equivalence of closed values only. (This restriction entails no loss of generality: see Section 4.6.) However, in order to formalize the soundness and completeness of our bisimulation with respect to contextual equivalence, it helps to extend the definition of contextual equivalence to values of open *types*. For instance, we will have to consider whether $\lambda x:\mathtt{int}.\, x$ is contextually equivalent to $\lambda x:\mathtt{int}.\, x-1$ at type $\alpha \to \mathtt{int}$, where the implementation of abstract type α is $\mathtt{int}$ in fact. But this clearly depends on what values of type α (or, more generally, what values involving type α) exist in the context: for instance, if the only values of type α are 2 in the left-hand world and 3 in the right-hand world, then the equivalence does hold; however, if some integers i on the left and j on the right have type α where $i \neq j-1$, then it does not hold. In order to capture at once all such values in the context involving type α, we consider the equivalence of *multiple* pairs of values—annotated with their types—such as $\{(2,3,\alpha), ((\lambda x:\mathtt{int}.\, x), (\lambda x:\mathtt{int}.\, x-1), \alpha \to \mathtt{int})\}$ and $\{(i,j,\alpha), ((\lambda x:\mathtt{int}.\, x), (\lambda x:\mathtt{int}.\, x-1), \alpha \to \mathtt{int})\}$; the former should be included in the equivalence while the latter should not, provided that $i \neq j-1$. For this reason, we generalize and define contextual equivalence as follows.

Definition 4.1. A *concretion environment* Δ is a finite set of triples of the form $(\alpha, \sigma, \sigma')$ with σ and σ' closed and $(\alpha, \tau, \tau') \in \Delta \wedge (\alpha, \sigma, \sigma') \in \Delta \Rightarrow \tau = \sigma \wedge \tau' = \sigma'$.

The intuition is that, under Δ, abstract type α is implemented by concrete type σ in the left-hand side and by another concrete type σ' in the right-hand side (of an equivalence). For instance, in the example in Section 4.1, the concrete implementations of abstract type α were $\mathtt{int}$ in the left-hand world and $\mathtt{bool}$ in the right-hand world, so Δ was $\{(\alpha, \mathtt{int}, \mathtt{bool})\}$. We write $Dom(\Delta)$ for $\{\alpha_1, \ldots, \alpha_n\}$ when $\Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_n, \sigma_n, \sigma'_n)\}$ and write $\Delta_1 \uplus \Delta_2$ for $\Delta_1 \cup \Delta_2$ when $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$.

Definition 4.2. A *typed value relation* $\mathcal{R}$ is a (either finite or infinite) set of triples of the form (V, V', τ).

The intuition is that $\mathcal{R}$ relates value V in the left-hand side and value V' in the right-hand side at type τ.

Definition 4.3. Let $\Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_m, \sigma_m, \sigma'_m)\}$. We write $\Delta \vdash \mathcal{R}$ if, for any $(V, V', \tau) \in \mathcal{R}$, we have $\vdash V : [\overline{\sigma}/\overline{\alpha}]\tau$ and $\vdash V' : [\overline{\sigma'}/\overline{\alpha}]\tau$.

Definition 4.4 (Typed Value Relation in Context). We write $(\Delta, \mathcal{R})^\circ$ for the relation

$$\begin{array}{l}\{([\overline{U}/\overline{y}][\overline{\sigma}/\overline{\alpha}]D,\ [\overline{U}'/\overline{y}][\overline{\sigma}'/\overline{\alpha}]D,\ \tau) \mid \\ \ \Delta = \{(\alpha_1, \sigma_1, \sigma_1'), \dots, (\alpha_m, \sigma_m, \sigma_m')\}, \\ \ (U_1, U_1', \rho_1), \dots, (U_n, U_n', \rho_n) \in \mathcal{R}, \\ \ \alpha_1, \dots, \alpha_m, y_1 : \rho_1, \dots, y_n : \rho_n \vdash D : \tau\}.\end{array}$$

Intuitively, this relation represents contexts into which values related by $\mathcal{R}$ have been put.

Definition 4.5. *Generalized contextual equivalence* is the set $\equiv$ of all pairs $(\Delta, \mathcal{R})$ such that:

A. $\Delta \vdash \mathcal{R}$.

B. For any $(M, M', \tau) \in (\Delta, \mathcal{R})^\circ$, we have $M \Downarrow \iff M' \Downarrow$.

Note that the standard contextual equivalence—between two closed values of a closed type—is subsumed by the case where each Δ is empty and each $\mathcal{R}$ is a singleton. Conversely, the standard contextual equivalence is *implied* by the generalized one in the following sense: if $(V, V', \tau) \in \mathcal{R}$ for some $(\Delta, \mathcal{R}) \in \equiv$ where V, V', and τ are closed, then it is immediate by definition that $K[V] \Downarrow \iff K[V'] \Downarrow$ for any context K with a hole $[\,]$ for terms of type τ. See also Section 4.6 for discussions on non-values and open terms.

We write

$$\Delta \ \vdash \ V_1, V_2, \dots \ \equiv \ V_1', V_2', \dots \ : \ \tau_1, \tau_2, \dots$$

for

$$(\Delta, \{(V_1, V_1', \tau_1), (V_2, V_2', \tau_2), \dots\}) \ \in \ \equiv.$$

We also write $\Delta \vdash V \equiv_{\mathcal{R}} V' : \tau$ for $(V, V', \tau) \in \mathcal{R}$ with $(\Delta, \mathcal{R}) \in \equiv$. Intuitively, this can be read, "values V and V' have type τ under concretion environment Δ and are contextually equivalent under knowledge $\mathcal{R}$."

The following properties follow immediately from the definition above.

Corollary 4.6 (Reflexivity). If $\vdash V_1 : [\overline{\sigma}/\overline{\alpha}]\tau_1, \vdash V_2 : [\overline{\sigma}/\overline{\alpha}]\tau_2, \dots$, then

$$\{(\overline{\alpha}, \overline{\sigma}, \overline{\sigma})\} \ \vdash \ V_1, V_2, \dots \ \equiv \ V_1, V_2, \dots \ : \ \tau_1, \tau_2, \dots.$$

Corollary 4.7 (Symmetry). If

$$\{(\overline{\alpha}, \overline{\sigma}, \overline{\sigma}')\} \ \vdash \ V_1, V_2, \dots \ \equiv \ V_1', V_2', \dots \ : \ \tau_1, \tau_2, \dots$$

then

$$\{(\overline{\alpha}, \overline{\sigma}', \overline{\sigma})\} \ \vdash \ V_1', V_2', \dots \ \equiv \ V_1, V_2, \dots \ : \ \tau_1, \tau_2, \dots.$$

Corollary 4.8 (Transitivity). If

$$\{(\overline{\alpha}, \overline{\sigma}, \overline{\sigma}')\} \ \vdash \ V_1, V_2, \dots \ \equiv \ V_1', V_2', \dots \ : \ \tau_1, \tau_2, \dots$$

and

$$\{(\overline{\alpha}, \overline{\sigma}', \overline{\sigma}'')\} \ \vdash \ V_1', V_2', \dots \ \equiv \ V_1'', V_2'', \dots \ : \ \tau_1, \tau_2, \dots$$

then

$$\{(\overline{\alpha}, \overline{\sigma}, \overline{\sigma}'')\} \ \vdash \ V_1, V_2, \dots \ \equiv \ V_1'', V_2'', \dots \ : \ \tau_1, \tau_2, \dots.$$

Example 4.9. Suppose that our language is extended in the obvious way with integers and booleans (these are, of course, definable in the language we have already given, but we prefer not to clutter examples with encodings), and let $\Delta = \{(\alpha, \mathtt{int}, \mathtt{int})\}$. Then we have:

$$\Delta \ \vdash \ 2,\ (\lambda x\!:\!\mathtt{int}.\, x) \ \equiv \ 3,\ (\lambda x\!:\!\mathtt{int}.\, x-1) \ : \ \alpha,\ (\alpha \to \mathtt{int})$$

More generally,

$$\Delta \ \vdash \ i,\ (\lambda x\!:\!\mathtt{int}.\, x) \ \equiv \ j,\ (\lambda x\!:\!\mathtt{int}.\, x-1) \ : \ \alpha,\ (\alpha \to \mathtt{int})$$

if and only if $i = j - 1$.

Example 4.10. Let $\Delta = \{(\alpha, \mathtt{int}, \mathtt{bool})\}$. We have

$$\Delta \ \vdash \ 1,\ (\lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0) \ \equiv \ \mathtt{true},\ (\lambda x\!:\!\mathtt{bool}.\, \neg x) \ : \ \alpha,\ (\alpha \to \mathtt{bool})$$

$$\Delta \ \vdash \ 1,\ (\lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0) \ \equiv \ \mathtt{false},\ (\lambda x\!:\!\mathtt{bool}.\, x) \ : \ \alpha,\ (\alpha \to \mathtt{bool})$$

but

$$\begin{array}{rl} \Delta \ \vdash & 1,\ (\lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0),\ 1,\ (\lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0) \\ \not\equiv & \mathtt{true},\ (\lambda x\!:\!\mathtt{bool}.\, \neg x),\ \mathtt{false},\ (\lambda x\!:\!\mathtt{bool}.\, x) \\ : & \alpha,\ (\alpha \to \mathtt{bool}),\ \alpha,\ (\alpha \to \mathtt{bool}). \end{array}$$

The last example shows that, even if $(\Delta, \mathcal{R}_1) \in \equiv$ and $(\Delta, \mathcal{R}_2) \in \equiv$, the union $(\Delta, \mathcal{R}_1 \cup \mathcal{R}_2)$ does not always belong to $\equiv$. In other words, one should not confuse two different implementations of an abstract type, even if each of them is correct in itself.

4.3 Bisimulation

Contextual equivalence is difficult to prove directly, because it involves a universal quantification over arbitrary contexts. Fortunately, we can avoid considering all contexts by observing that there are actually only a few "primitive" operations that contexts can perform on the values they have access to: for instance, if a context is comparing a pair $\langle v, w\rangle$ with another pair $\langle v', w'\rangle$, all it can do is to project the first elements v and v' or the second elements w and w' (and add them to its knowledge for later use). Similarly, in order to compare functions $\lambda x.\, M$ and $\lambda x.\, M'$, a context has to apply them to some arguments it can make up from its knowledge. Intuitively, our bisimulations are sets of relations representing such contextual knowledge, closed under increase of knowledge via primitive operations like projection and application.

Based on the ideas above, our bisimulation is defined as follows. More detailed technical intuitions will be given after the definition.

Definition 4.11 (Bisimulation). A *bisimulation* is a set X of pairs $(\Delta, \mathcal{R})$ such that:

1. $\Delta \vdash \mathcal{R}$.

2. For each

$$(\mathtt{fix}\ f(x\!:\!\pi)\!:\!\rho = M,\ \mathtt{fix}\ f(x\!:\!\pi')\!:\!\rho' = M',\ \tau \to \sigma) \in \mathcal{R}$$

and for any $(V, V', \tau) \in (\Delta, \mathcal{R})^\circ$, we have

$$(\texttt{fix}\, f(x : \pi) : \rho = M)V \Downarrow \quad \Longleftrightarrow \quad (\texttt{fix}\, f(x : \pi') : \rho' = M')V' \Downarrow.$$

Furthermore, if $(\texttt{fix}\, f(x : \pi) : \rho = M)V \Downarrow W$ and $(\texttt{fix}\, f(x : \pi') : \rho' = M')V' \Downarrow W'$, then

$$(\Delta, \mathcal{R} \cup \{(W, W', \sigma)\}) \in X.$$

3. Let $\Delta = \{(\alpha_1, \sigma_1, \sigma_1'), \ldots, (\alpha_m, \sigma_m, \sigma_m')\}$. For each

$$(\Lambda\alpha.\, M, \Lambda\alpha.\, M', \forall\alpha.\, \tau) \in \mathcal{R}$$

and for any ρ with $FTV(\rho) \subseteq Dom(\Delta)$, we have

$$(\Lambda\alpha.\, M)[[\overline{\sigma}/\overline{\alpha}]\rho] \Downarrow \quad \Longleftrightarrow \quad (\Lambda\alpha.\, M')[[\overline{\sigma}'/\overline{\alpha}]\rho] \Downarrow.$$

Furthermore, if $(\Lambda\alpha.\, M)[[\overline{\sigma}/\overline{\alpha}]\rho] \Downarrow W$ and $(\Lambda\alpha.\, M')[[\overline{\sigma}'/\overline{\alpha}]\rho] \Downarrow W'$, then

$$(\Delta, \mathcal{R} \cup \{(W, W', [\rho/\alpha]\tau)\}) \in X.$$

4. For each

$$(\texttt{pack}\, \sigma, V \texttt{ as } \exists\alpha.\, \tau,\ \texttt{pack}\, \sigma', V' \texttt{ as } \exists\alpha.\, \tau',\ \exists\alpha.\, \tau'') \in \mathcal{R},$$

we have either

$$(\Delta \uplus \{(\alpha, \sigma, \sigma')\}, \mathcal{R} \cup \{(V, V', \tau'')\}) \in X,$$

or else $(\beta, \sigma, \sigma') \in \Delta$ and $(V, V', [\beta/\alpha]\tau'') \in \mathcal{R}$ for some β.

5. For each $(\langle V_1, \ldots, V_n\rangle, \langle V_1', \ldots, V_n'\rangle, \tau_1 \times \ldots \times \tau_n) \in \mathcal{R}$ and for any $1 \leq i \leq n$, we have $(\Delta, \mathcal{R} \cup (V_i, V_i', \tau_i)) \in X$.

6. For each $(\texttt{in}_i(V), \texttt{in}_j(V'), \tau_1 + \ldots + \tau_n) \in \mathcal{R}$, we have $i = j$ and $(\Delta, \mathcal{R} \cup (V, V', \tau_i)) \in X$.

7. For each $(\texttt{fold}(V), \texttt{fold}(V'), \mu\alpha.\, \tau) \in \mathcal{R}$, we have $(\Delta, \mathcal{R} \cup (V, V', [\mu\alpha.\, \tau/\alpha]\tau)) \in X$.

As usual, *bisimilarity*, written $\sim$, is the largest bisimulation; it exists because the union of two bisimulations is again a bisimulation.

We write

$$\Delta \ \vdash \ V_1, \ldots, V_n \quad X \quad V_1', \ldots V_n' \ : \ \tau_1, \ldots, \tau_n$$

for

$$(\Delta, \{(V_1, V_1', \tau_1), \ldots, (V_n, V_n', \tau_n)\}) \quad \in \quad X.$$

We also write $\Delta \vdash V \ X_{\mathcal{R}}\ V' : \tau$ for $(V, V', \tau) \in \mathcal{R}$ with $(\Delta, \mathcal{R}) \in X$. Intuitively, it can be read: values V and V' of type τ with concretion environment Δ are bisimilar under knowledge $\mathcal{R}$.

We now elaborate the intuitions behind the definition of bisimulation. Condition 1 ensures that bisimilar values V and V' are well typed under the concretion environment Δ. The other conditions are concerned with the things that a context can do with the values it knows to gain more knowledge.

Condition 2 deals with the case where a context applies two functions it knows ($\mathtt{fix}\, f(x\!:\!\pi)\!:\!\rho = M$ and $\mathtt{fix}\, f(x\!:\!\pi')\!:\!\rho' = M'$) to some arguments V and V'. To make up these arguments, the context can make use of any values it already knows ($\overline{U}$ and $\overline{U}'$ in Definition 4.4) and assemble them using a term D with free variables $\overline{y}$, where the abstract types $\overline{\alpha}$ are kept abstract.

The crucial observation here is that it suffices to consider value arguments only, i.e., only the cases where the assembled terms $[\overline{U}/\overline{y}][\overline{\sigma}/\overline{\alpha}]D$ and $[\overline{U}'/\overline{y}][\overline{\sigma}'/\overline{\alpha}]D'$ are values. This simplification is essential for proving the bisimilarity of functions—indeed, it is the "magic" that makes our whole approach tractable. Intuitively, it can be understood via the fact that any *terms* of the form $[\overline{U}/\overline{y}][\overline{\sigma}/\overline{\alpha}]D$ and $[\overline{U}'/\overline{y}][\overline{\sigma}'/\overline{\alpha}]D$ evaluate to *values* of the same form, as proved in Lemma 4.14 below.

Then, to avoid exhibiting an observable difference in behaviors, the function applications should either both diverge or else both converge; in the latter case, the resulting values become part of the context's knowledge and can be used for further experiments.[1]

Condition 3 is similar to Condition 2, but for type application rather than term application.

Condition 4 is for packages defining an abstract type α. Essentially, a context can open the two packages and examine their contents only abstractly, as expressed in the first half of this condition. However, if the context happens to know another abstract type β whose implementations coincide with α's, there is no need for us to consider them twice. The second half of the condition expresses this simplification. It is not so crucial as the previous simplification in Condition 2, but it is useful for proving the bisimulation of packages, keeping X finite in many cases despite the generativity of `open`, as we mentioned in the introduction.

Conditions 5, 6, and 7 are for tuples, injected values, and folded values, respectively. They capture the straightforward increase of the context's knowledge via projection, case branch, or unfolding.

4.4 Examples

Before presenting our main technical result—that bisimulation is sound and complete for contextual equivalence—we develop several examples illustrating concrete applications of the bisimulation method. The first three examples involve existential packages, whose equivalence cannot be proved by other bisimulations for λ-calculi. The fourth example involves recursive types with negative occurrences, for which logical relations have difficulties. Our bisimulation technique yields a straightforward proof of equivalence for each of the examples.

4.4.1 Warm-Up

Consider the following simple packages

$$\begin{array}{rcl} U & = & \mathtt{pack}\,\mathtt{int}, \langle 1, \lambda x\!:\!\mathtt{int}.\, x \overset{\mathtt{int}}{=} 0\rangle\ \mathtt{as}\ \tau \\ U' & = & \mathtt{pack}\,\mathtt{bool}, \langle \mathtt{true}, \lambda x\!:\!\mathtt{bool}.\, \neg x\rangle\ \mathtt{as}\ \tau \end{array}$$

[1] Another technical point may deserve mentioning here: instead of $(\Delta, \mathcal{R} \cup \{(W, W', \sigma)\}) \in X$, we could require $(W, W', \sigma) \in \mathcal{R}$ to reduce the number of $\mathcal{R}$s required to be in X by "predicting" the increase of contexts' knowledge *a priori*. We rejected this alternative for the sake of uniformity with Condition 4, which anyway requires the concretion environment Δ to be extended. This decision does *not* make it difficult to construct a bisimulation, as we will see soon in the examples.

where $\tau = \exists\alpha.\, \alpha \times (\alpha \to \texttt{bool})$. We aim to prove that U and U' are contextually equivalent at type τ. To this end, let

$$X \quad = \quad \{(\emptyset, \mathcal{R}_0), (\Delta, \mathcal{R}_1), (\Delta, \mathcal{R}_2), (\Delta, \mathcal{R}_3), (\Delta, \mathcal{R}_4), (\Delta, \mathcal{R}_5)\},$$

where

$$\begin{array}{rcl}
\Delta & = & (\alpha, \texttt{int}, \texttt{bool}) \\
\mathcal{R}_0 & = & \{(U, U', \tau)\} \\
\mathcal{R}_1 & = & \mathcal{R}_0 \cup \{(\langle 1, \lambda x\!:\!\texttt{int}.\, x \overset{\texttt{int}}{=} 0\rangle,\ \langle \texttt{true}, \lambda x\!:\!\texttt{bool}.\, \neg x\rangle,\ \alpha \times (\alpha \to \texttt{bool}))\} \\
\mathcal{R}_2 & = & \mathcal{R}_1 \cup \{(1, \texttt{true}, \alpha)\} \\
\mathcal{R}_3 & = & \mathcal{R}_1 \cup \{(\lambda x\!:\!\texttt{int}.\, x \overset{\texttt{int}}{=} 0,\ \lambda x\!:\!\texttt{bool}.\, \neg x,\ \alpha \to \texttt{bool})\} \\
\mathcal{R}_4 & = & \mathcal{R}_2 \cup \mathcal{R}_3 \\
\mathcal{R}_5 & = & \mathcal{R}_4 \cup \{(\texttt{false}, \texttt{false}, \texttt{bool})\}.
\end{array}$$

Then, X is a bisimulation. To prove it, we must check each condition in Definition 4.11 for every $(\Delta, \mathcal{R}) \in X$. Most of the checks are trivial, except the following cases:

- Condition 4 on $(U, U', \tau) \in \mathcal{R}_i$ for $i \geq 1$, where the second half of the condition holds.

- Condition 2 on
$$(\lambda x\!:\!\texttt{int}.\, x \overset{\texttt{int}}{=} 0,\ \lambda x\!:\!\texttt{bool}.\, \neg x,\ \alpha \to \texttt{bool}) \in \mathcal{R}_i$$
  for $i \geq 3$. Since V and V' are values, the D in Definition 4.4 is either a value or a variable. However, if D is a value, it can never satisfy the assumption $\alpha, y_1\!:\!\rho_1, \ldots, y_n\!:\!\rho_n \vdash D : \alpha$ (easy case analysis on the syntax of D). Thus, D must be a variable. Without loss of generality, let $D = y_1$. Then, by inversion of (T-Var), $\rho_1 = \alpha$. Since $(U_1, U_1', \rho_1) \in \mathcal{R}_n$, we have $U_1 = 1$ and $U_1' = \texttt{true}$. Thus, $V = 1$ and $V' = \texttt{true}$, from which the rest of this condition is obvious.

Alternatively, in this particular example, we can just take $X = \{(\Delta, \mathcal{R}_5)\}$ in the first place and prove it to be a bisimulation by the same arguments as above. Since $(U, U', \tau) \in \mathcal{R}_5$, this still suffices for showing the contextual equivalence of U and U', thanks to the soundness of bisimilarity (Corollary 4.16) and the generalized definition of contextual equivalence (Definition 4.5).

4.4.2 Complex Numbers

Suppose now that we have real numbers and operations in the language. Then the following two implementations U and U' of complex numbers should be contextually equivalent at the appropriate type $\exists\alpha.\, \tau$.

$$\begin{array}{rcl}
U & = & \texttt{pack}\,(\texttt{real} \times \texttt{real}), \langle id, id, cmul\rangle\ \texttt{as}\ \exists\alpha.\, \tau \\
U' & = & \texttt{pack}\,(\texttt{real} \times \texttt{real}), \langle ctop, ptoc, pmul\rangle\ \texttt{as}\ \exists\alpha.\, \tau \\
\tau & = & (\texttt{real} \times \texttt{real} \to \alpha) \times (\alpha \to \texttt{real} \times \texttt{real}) \times (\alpha \to \alpha \to \alpha)
\end{array}$$

$$id \quad = \quad \lambda c\!:\!\texttt{real} \times \texttt{real}.\, c$$

$$
\begin{aligned}
cmul &= \lambda c_1 : \mathtt{real} \times \mathtt{real}.\, \lambda c_2 : \mathtt{real} \times \mathtt{real}. \\
&\qquad \langle \#_1(c_1) \times \#_1(c_2) - \#_2(c_1) \times \#_2(c_2),\ \#_2(c_1) \times \#_1(c_2) + \#_1(c_1) \times \#_2(c_2) \rangle
\end{aligned}
$$

$$
\begin{aligned}
ctop &= \lambda c : \mathtt{real} \times \mathtt{real}.\, \langle \sqrt{(\#_1(c))^2 + (\#_2(c))^2},\ \mathrm{atan2}(\#_2(c), \#_1(c)) \rangle \\
ptoc &= \lambda p : \mathtt{real} \times \mathtt{real}.\, \langle \#_1(p) \times \cos(\#_2(p)),\ \#_1(p) \times \sin(\#_2(p)) \rangle \\
pmul &= \lambda p_1 : \mathtt{real} \times \mathtt{real}.\, \lambda p_2 : \mathtt{real} \times \mathtt{real}.\, \langle \#_1(p_1) \times \#_1(p_2),\ \#_2(p_1) + \#_2(p_2) \rangle
\end{aligned}
$$

The first functions in these packages make a complex number from its real and imaginary parts, and the second functions perform the converse conversion. The third functions multiply complex numbers.

To prove the contextual equivalence of U and U', consider $X = \{(\Delta, \mathcal{R})\}$ where

$$
\begin{aligned}
\Delta &= \{(\alpha, \mathtt{real} \times \mathtt{real}, \mathtt{real} \times \mathtt{real})\} \\
\mathcal{R} &= \{(U, U', \exists \alpha.\, \tau), \\
&\qquad (\langle id, id, cmul \rangle, \langle ctop, ptoc, pmul \rangle, \tau), \\
&\qquad (id, ctop, \mathtt{real} \times \mathtt{real} \to \alpha), \\
&\qquad (id, ptoc, \alpha \to \mathtt{real} \times \mathtt{real}), \\
&\qquad (cmul, pmul, \alpha \to \alpha \to \alpha)\} \\
&\quad \cup\ \{(v, w, \alpha) \mid w = \langle r, t \rangle, \\
&\qquad\qquad \langle r \times \cos(t), r \times \sin(t) \rangle \Downarrow v, \\
&\qquad\qquad r \geq 0\} \\
&\quad \cup\ \{(c, c, \mathtt{real} \times \mathtt{real}) \mid\ \vdash c : \mathtt{real} \times \mathtt{real}\} \\
&\quad \cup\ \{(r, r, \mathtt{real}) \mid\ \vdash r : \mathtt{real}\}.
\end{aligned}
$$

Then X is a bisimulation, as can be checked in the same manner as in the previous example.

4.4.3 Functions Generating Packages

The following functions U and U' *generate* packages. (I.e., they behave a bit like *functors* in ML-style module systems.)

$$
\begin{aligned}
U &= \lambda y : \mathtt{int}.\, M \\
U' &= \lambda y : \mathtt{int}.\, M' \\
M &= \mathtt{pack}\ \mathtt{int}, \langle y, \lambda x : \mathtt{int}.\, x \rangle\ \mathtt{as}\ \tau \\
M' &= \mathtt{pack}\ \mathtt{int}, \langle y + 1, \lambda x : \mathtt{int}.\, x - 1 \rangle\ \mathtt{as}\ \tau \\
\tau &= \exists \alpha.\, \alpha \times (\alpha \to \mathtt{int})
\end{aligned}
$$

To prove that U is contextually equivalent to U' at type $\mathtt{int} \to \tau$, it suffices to consider the following infinite bisimulation.

$$
\begin{aligned}
X &= \{(\Delta, \mathcal{R}) \mid \\
&\qquad \Delta = \{(\beta_i, \mathtt{int}, \mathtt{int}) \mid -n \leq i \leq n\}, \\
&\qquad \mathcal{R} \subseteq \cup_{-n \leq i \leq n} \mathcal{R}_i,
\end{aligned}
$$

$$
\begin{aligned}
& \qquad n = 0,1,2,\dots\} \\
\mathcal{R}_i \quad = \quad & \{(U, U', \mathtt{int} \to \tau), \\
& ([i/y]M, [i/y]M', \tau), \\
& (\langle i, \lambda x\!:\!\mathtt{int}.\, x\rangle, \langle i+1, \lambda x\!:\!\mathtt{int}.\, x-1\rangle, \beta_i \times (\beta_i \to \mathtt{int})), \\
& (i, i+1, \beta_i), \\
& (\lambda x\!:\!\mathtt{int}.\, x, \lambda x\!:\!\mathtt{int}.\, x-1, \beta_i \to \mathtt{int}), \\
& (i, i, \mathtt{int})\}
\end{aligned}
$$

The generativity of U and U' is given a simple account by having a different abstract type β_i for each instantiation of U and U' with $y = i$.

The inclusion of all $\mathcal{R} \subseteq \cup_{-n \le i \le n} \mathcal{R}_i$ in the definition of X simplifies the definition of this bisimulation; although it admits some $\mathcal{R}$s that are not strictly relevant to the proof (such as those with only the elements of tuples, but without the tuples themselves), they are not a problem since they do not violate any of the conditions of bisimulation. In other words, to prove the contextual equivalence of two values, one has only to find *some* bisimulation including the values rather than the minimal one.

4.4.4 Recursive Types with Negative Occurrence

Consider the packages C and C' implementing counter objects as follows: each counter is implemented as a pair of its state part (of abstract type `st`) and its method part; the latter is implemented as a function that takes a state and returns the tuple of methods[2]; in this example, there are two methods in the tuple: one returns a new counter object with the state incremented (or, in the second implementation, decremented) by 1, while the other tells whether another counter object has been incremented (or decremented) the same number of times as the present one.

$$
\begin{aligned}
\tau &= \exists \mathtt{st}.\, \sigma \\
\sigma &= \mu \mathtt{self}.\, \mathtt{st} \times (\mathtt{st} \to \rho) \\
\rho &= \mathtt{self} \times (\mathtt{self} \to \mathtt{bool})
\end{aligned}
$$

$$
\begin{aligned}
C &= \mathtt{pack}\ \mathtt{int}, \mathtt{fold}(\langle 0, M\rangle)\ \mathtt{as}\ \tau \\
C' &= \mathtt{pack}\ \mathtt{int}, \mathtt{fold}(\langle 0, M'\rangle)\ \mathtt{as}\ \tau
\end{aligned}
$$

$$
\begin{aligned}
M &= \mathtt{fix}\ f(s\!:\!\mathtt{int})\!:\![\mathtt{int}/\mathtt{st}][\sigma/\mathtt{self}]\rho = \\
&\qquad \langle \mathtt{fold}(\langle s+1, f\rangle), \\
&\qquad\ \ \lambda c\!:\![\mathtt{int}/\mathtt{st}]\sigma.\, (s \overset{\mathtt{int}}{=} \#_1(\mathtt{unfold}(c)))\rangle \\
M' &= \mathtt{fix}\ f(s\!:\!\mathtt{int})\!:\![\mathtt{int}/\mathtt{st}][\sigma/\mathtt{self}]\rho = \\
&\qquad \langle \mathtt{fold}(\langle s-1, f\rangle), \\
&\qquad\ \ \lambda c\!:\![\mathtt{int}/\mathtt{st}]\sigma.\, (s \overset{\mathtt{int}}{=} \#_1(\mathtt{unfold}(c)))\rangle
\end{aligned}
$$

[2]This implementation can be viewed as a variant of the so-called recursive existential encoding of objects (see [Bruce, Cardelli, and Pierce 1999] for details), but our goal here is to illustrate the power of our bisimulation with existential recursive types, rather than to discuss the object encoding itself.

Let us prove the contextual equivalence of C and C' at type τ. To do so, we consider the bisimulation $X = \{(\Delta, \mathcal{R})\}$ where:

$$\begin{array}{rcl} \Delta & = & \{(\texttt{st}, \texttt{int}, \texttt{int})\} \\ \mathcal{R} & = & \{(C, C', \tau), \\ & & \quad (\texttt{fold}(\langle n, M\rangle), \texttt{fold}(\langle -n, M'\rangle), \sigma), \\ & & \quad (\langle n, M\rangle, \langle -n, M'\rangle, \texttt{st} \times (\texttt{st} \to [\sigma/\texttt{self}]\rho)), \\ & & \quad (n, -n, \texttt{st}), \\ & & \quad (M, M', \texttt{st} \to [\sigma/\texttt{self}]\rho), \\ & & \quad (\langle \texttt{fold}(\langle n+1, M\rangle),\ \lambda c : [\texttt{int}/\texttt{st}]\sigma.\,(n \overset{\texttt{int}}{=} \#_1(\texttt{unfold}(c)))\rangle, \\ & & \quad\ \langle \texttt{fold}(\langle -n-1, M'\rangle),\ \lambda c : [\texttt{int}/\texttt{st}]\sigma.\,(-n \overset{\texttt{int}}{=} \#_1(\texttt{unfold}(c)))\rangle, \\ & & \quad\ \sigma \times (\sigma \to \texttt{bool})), \\ & & \quad (\lambda c : [\texttt{int}/\texttt{st}]\sigma.\,(n \overset{\texttt{int}}{=} \#_1(\texttt{unfold}(c))), \\ & & \quad\ \lambda c : [\texttt{int}/\texttt{st}]\sigma.\,(-n \overset{\texttt{int}}{=} \#_1(\texttt{unfold}(c))), \\ & & \quad\ \sigma \to \texttt{bool}), \\ & & \quad (\texttt{true}, \texttt{true}, \texttt{bool}), \\ & & \quad (\texttt{false}, \texttt{false}, \texttt{bool}) \mid \\ & & \quad n = 0, 1, 2, \ldots\} \end{array}$$

It can indeed be shown to be a bisimulation just as the bisimulations in previous examples. That is, unlike logical relations, our bisimulation incurs no difficulty at all for recursive functions or recursive types even with negative occurrence.

4.4.5 Higher-Order Functions

The following higher-order functions represent the "dual" of the example in Section 4.4.1.

$$\begin{array}{rcl} U & = & \lambda f : \sigma.\, f[\texttt{int}]\langle 1, \lambda x : \texttt{int}.\, x \overset{\texttt{int}}{=} 0\rangle \\ U' & = & \lambda f : \sigma.\, f[\texttt{bool}]\langle \texttt{true}, \lambda x : \texttt{bool}.\, \neg x\rangle \\ \sigma & = & \forall \alpha.\, \alpha \times (\alpha \to \texttt{bool}) \to \texttt{unit} \end{array}$$

It is surprisingly easy to prove the contextual equivalence of U and U' at type $\sigma \to \texttt{unit}$, i.e.,

$$[U/x]C \Downarrow \quad \Longleftrightarrow \quad [U'/x]C \Downarrow$$

for any $x : \sigma \to \texttt{unit} \vdash C : \tau$. Since

$$\begin{array}{rcl} [U/x]C & = & [1, (\lambda x : \texttt{int}.\, x \overset{\texttt{int}}{=} 0)/y, z][\texttt{int}/\beta]D_0 \\ [U'/x]C & = & [\texttt{true}, (\lambda x : \texttt{bool}.\, \neg x)/y, z][\texttt{bool}/\beta]D_0 \end{array}$$

for $D_0 = [(\lambda f : \sigma.\, f[\beta]\langle y, z\rangle)/x]C$, it suffices to prove

$$[1, (\lambda x : \texttt{int}.\, x \overset{\texttt{int}}{=} 0)/y, z][\texttt{int}/\beta]D \Downarrow \quad \Longleftrightarrow \quad [\texttt{true}, (\lambda x : \texttt{bool}.\, \neg x)/y, z][\texttt{bool}/\beta]D \Downarrow$$

for every $\beta, y:\beta, z:\beta \to \texttt{bool} \vdash D : \tau$. (Note that D_0 has the same typing as D thanks to the standard substitution lemma.) However, this follows immediately from the bisimulation $\{(\Delta, \mathcal{R})\}$ where

$$\begin{array}{rcl} \Delta & = & \{(\beta, \texttt{int}, \texttt{bool})\} \\ \mathcal{R} & = & \{(1, \texttt{true}, \beta), \\ & & \quad (\lambda x : \texttt{int}.\, x \overset{\texttt{int}}{=} 0,\ \lambda x : \texttt{bool}.\, \neg x,\ \beta \to \texttt{bool}), \\ & & \quad (\texttt{false}, \texttt{false}, \texttt{bool})\} \end{array}$$

along with the soundness of bisimilarity in the next section.

4.5 Soundness and Completeness

We prove that bisimilarity coincides with contextual equivalence (in the generalized form presented in Section 4.2). That is, two values can be proved to be bisimilar if and only if they are contextually equivalent.

First, we prove the "if" part, i.e., that contextual equivalence is included in bisimilarity. This direction is easier because our bisimulation is defined co-inductively: it suffices simply to prove that contextual equivalence is a bisimulation.

Lemma 4.12 (Completeness of Bisimulation). $\equiv \subseteq \sim$.

Proof. By checking that $\equiv$ satisfies each condition of bisimulation. Details can be found in Appendix C.1. □

Next, we show that bisimilarity is included in contextual equivalence. To do so, we need to consider the question: When we put bisimilar values into a context and evaluate them, what changes? The answer is: Nothing! I.e., evaluating a pair of expressions, each consisting of some set of bisimilar values placed in some context, results again in a pair of expressions that can be described by some set of bisimilar values placed in some context. Furthermore, this evaluation converges in the left-hand side if and only if it converges in the right-hand side. Since the proof of the latter property requires the former property, we formalize the observations above in the following order.

Definition 4.13 (Bisimilarity in Context). We write $\Delta \vdash N \sim^{\circ}_{\mathcal{R}} N' : \tau$ if $(N, N', \tau) \in (\Delta, \mathcal{R})^{\circ}$ and $(\Delta, \mathcal{R}) \in \sim$.

The intuition is that $\sim^{\circ}$ relates bisimilar values put in contexts.

Lemma 4.14 (Fundamental Property, Part I). Suppose $\Delta_0 \vdash N \sim^{\circ}_{\mathcal{R}_0} N' : \tau$. If $N \Downarrow W$ and $N' \Downarrow W'$, then $\Delta \vdash W \sim^{\circ}_{\mathcal{R}} W' : \tau$ for some $\Delta \supseteq \Delta_0$ and $\mathcal{R} \supseteq \mathcal{R}_0$.

Proof. By induction on the derivation of $N \Downarrow W$. Details are found in Appendix C.2. □

Lemma 4.15 (Fundamental Property, Part II). If $\Delta_0 \vdash N \sim^{\circ}_{\mathcal{R}_0} N' : \tau$, then $N \Downarrow \iff N' \Downarrow$.

Proof. By induction on the derivation of $N \Downarrow$ together with Lemma 4.14. Details are in Appendix C.3. $\square$

Corollary 4.16 (Soundness of Bisimilarity). $\sim \subseteq \equiv$.

Proof. By the definitions of $\equiv$ and $\sim^\circ$ together with Lemma 4.15. $\square$

Combining soundness and completeness, we obtain the main theorem about our bisimulation: that bisimilarity coincides with contextual equivalence.

Theorem 4.17. $\sim = \equiv$.

Proof. By Corollary 4.16 and Lemma 4.12. $\square$

Note that these proofs are much simpler than soundness proofs of applicative bisimulations in previous work [Howe 1996; Gordon 1995a; Gordon and Rees 1996; Gordon 1995b; Gordon and Rees 1995; Abramsky 1990] thanks to the generalized condition on functions (Condition 2), which is anyway required in the presence of existential polymorphism as discussed in the introduction.

4.6 Non-Values and Open Terms

So far, we have restricted ourselves to the equivalence of closed values for the sake of simplicity. In this section, we show how our method can be used for proving the standard contextual equivalence of non-values and open terms as well. (Although our approach here may seem *ad hoc*, it suffices for the present purpose of proving the contextual equivalence of open terms. For other studies on different equivalences for open terms, see [Pitts 2000; Pitts 1998] for instance.)

A *context* K in the standard sense is a term with some subterm replaced by a *hole* $[\,]$. We write $K[M]$ for the term obtained by substituting the hole in K with M (which does *not* apply α-conversion and may capture free variables). Then, the standard contextual equivalence

$$\alpha_1, \ldots, \alpha_m, x_1 : \tau_1, \ldots, x_n : \tau_n \quad \vdash \quad M \quad \overset{\text{std}}{\equiv} \quad M' \quad : \quad \tau$$

for well-typed terms $\overline{\alpha}, \overline{x} : \overline{\tau} \vdash M : \tau$ and $\overline{\alpha}, \overline{x} : \overline{\tau} \vdash M' : \tau$ can be defined as: $K[M] \Downarrow \iff K[M'] \Downarrow$ for every context K with $\vdash K[M] : \texttt{unit}$ and $\vdash K[M'] : \texttt{unit}$, where $\texttt{unit}$ is the nullary tuple type. (In fact, any closed type works in place of $\texttt{unit}$.)

We will show that the standard contextual equivalence above holds if and only if the closed values $V = \Lambda\overline{\alpha}.\, \lambda\overline{x} : \overline{\tau}.\, M$ and $V' = \Lambda\overline{\alpha}.\, \lambda\overline{x} : \overline{\tau}.\, M'$ are bisimilar, i.e.,

$$\begin{array}{rl}
\emptyset \ \vdash & \Lambda\alpha_1. \ldots \Lambda\alpha_m. \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n. M \\
\sim & \Lambda\alpha_1. \ldots \Lambda\alpha_m. \lambda x_1 : \tau_1. \ldots \lambda x_n : \tau_n. M' \\
: & \forall\alpha_1. \ldots \forall\alpha_m. \tau_1 \to \ldots \to \tau_n \to \tau.
\end{array}$$

(If M and M' have no free term/type variables at all, it suffices just to take $V = \Lambda\alpha.\, M$ and $V' = \Lambda\alpha.\, M'$ for any type variable α.) The "only if" direction is obvious from the definitions of contextual equivalences—both the standard one above and the generalized one in Section 4.2—and from the completeness of bisimulation. To prove the "if" direction, suppose $\emptyset \vdash V \sim V' : \forall\overline{\alpha}$.

$$\frac{\Gamma \vdash M \preceq M' : \rho \quad \{\overline{\alpha}\} \subseteq Dom(\Gamma) \quad \Gamma \vdash \overline{x} : \overline{\tau}}{\Gamma \vdash M \preceq (\Lambda\overline{\alpha}.\, \lambda\overline{x}\!:\!\overline{\tau}.\, M')[\overline{\alpha}]\overline{x} : \rho}\text{(B-Exp)}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x \preceq x : \tau}\text{(B-Var)} \qquad \frac{\Gamma, f\!:\!\tau \to \sigma, x\!:\!\tau \vdash M \preceq M' : \sigma}{\Gamma \vdash (\mathtt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M) \preceq (\mathtt{fix}\, f(x\!:\!\tau)\!:\!\sigma = M') : \tau \to \sigma}\text{(B-Fix)}$$

$$\frac{\Gamma \vdash M \preceq M' : \tau \to \sigma \quad \Gamma \vdash N \preceq N' : \tau}{\Gamma \vdash MN \preceq M'N' : \sigma}\text{(B-App)}$$

$$\frac{\Gamma, \alpha \vdash M \preceq M' : \tau}{\Gamma \vdash \Lambda\alpha.\, M \preceq \Lambda\alpha.\, M' : \forall\alpha.\, \tau}\text{(B-TAbs)} \qquad \frac{\Gamma \vdash M \preceq M' : \forall\alpha.\, \sigma \quad FTV(\tau) \subseteq \Gamma}{\Gamma \vdash M[\tau] \preceq M'[\tau] : [\tau/\alpha]\sigma}\text{(B-TApp)}$$

$$\frac{\Gamma \vdash M \preceq M' : [\tau/\alpha]\sigma \quad FTV(\tau) \subseteq \Gamma}{\Gamma \vdash (\mathtt{pack}\, \tau, M\ \mathtt{as}\ \exists\alpha.\, \sigma) \preceq (\mathtt{pack}\, \tau, M'\ \mathtt{as}\ \exists\alpha.\, \sigma) : \exists\alpha.\, \sigma}\text{(B-Pack)}$$

$$\frac{\Gamma \vdash M \preceq M' : \exists\alpha.\, \tau \quad \Gamma, \alpha, x\!:\!\tau \vdash N \preceq N' : \sigma \quad \alpha \notin FTV(\sigma)}{\Gamma \vdash (\mathtt{open}\ M\ \mathtt{as}\ \alpha, x\ \mathtt{in}\ N) \preceq (\mathtt{open}\ M'\ \mathtt{as}\ \alpha, x\ \mathtt{in}\ N') : \sigma}\text{(B-Open)}$$

$$\frac{\Gamma \vdash M_1 \preceq M_1' : \tau_1 \quad \ldots \quad \Gamma \vdash M_n \preceq M_n' : \tau_n}{\Gamma \vdash \langle M_1, \ldots, M_n\rangle \preceq \langle M_1', \ldots, M_n'\rangle : \tau_1 \times \ldots \times \tau_n}\text{(B-Tuple)}$$

$$\frac{\Gamma \vdash M \preceq M' : \tau_1 \times \ldots \times \tau_i \times \ldots \times \tau_n}{\Gamma \vdash \#_i(M) \preceq \#_i(M') : \tau_i}\text{(B-Proj)}$$

$$\frac{\Gamma \vdash M \preceq M' : \tau_i \quad FTV(\tau_1) \subseteq \Gamma \quad \ldots \quad FTV(\tau_n) \subseteq \Gamma}{\Gamma \vdash \mathtt{in}_i(M) \preceq \mathtt{in}_i(M') : \tau_1 + \ldots + \tau_i + \ldots + \tau_n}\text{(B-Inj)}$$

$$\frac{\Gamma \vdash M \preceq M' : \tau_1 + \ldots + \tau_n \quad \Gamma, x_1\!:\!\tau_1 \vdash M_1 \preceq M_1' : \tau \quad \ldots \quad \Gamma, x_n\!:\!\tau_n \vdash M_n \preceq M_n' : \tau}{\begin{array}{c}\Gamma \vdash (\mathtt{case}\ M\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow M_1 \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow M_n) \\ \preceq (\mathtt{case}\ M'\ \mathtt{of}\ \mathtt{in}_1(x_1) \Rightarrow M_1' \parallel \ldots \parallel \mathtt{in}_n(x_n) \Rightarrow M_n') : \tau\end{array}}\text{(B-Case)}$$

$$\frac{\Gamma \vdash M \preceq M' : [\mu\alpha.\, \tau/\alpha]\tau}{\Gamma \vdash \mathtt{fold}(M) \preceq \mathtt{fold}(M') : \mu\alpha.\, \tau}\text{(B-Fold)}$$

$$\frac{\Gamma \vdash M \preceq M' : \mu\alpha.\, \tau}{\Gamma \vdash \mathtt{unfold}(M) \preceq \mathtt{unfold}(M') : [\mu\alpha.\, \tau/\alpha]\tau}\text{(B-Unfold)}$$

Figure 4.4: β-expansion

$\overline{\tau} \to \tau$. By the soundness of bisimulation, we have $\emptyset \vdash V \equiv V' : \forall\overline{\alpha}.\overline{\tau} \to \tau$. Given any K with $\vdash K[M] : \texttt{unit}$ and $\vdash K[M'] : \texttt{unit}$, take $C = K[z[\alpha_1]\dots[\alpha_m]x_1 \dots x_n]$ for fresh z. Then, it suffices to prove $K[M] \Downarrow \iff [V/z]C \Downarrow$ and $K[M'] \Downarrow \iff [V'/z]C \Downarrow$.

To this end, we prove the more general lemma below in order for induction to work. The intuition is that a term M and its β-expanded version $(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M)[\overline{\alpha}]\overline{x}$ should behave equivalently under any context. Since the free type/term variables $\overline{\alpha}$ and $\overline{x}$ are to be substituted by some types/values during evaluation under a context, this "β-expansion" relation needs to be generalized to allow nesting. Thus, we define:

Definition 4.18 (β-Expansion). $\Gamma \vdash M \preceq M' : \tau$ is the smallest relation on pairs of λ-terms M and M' (annotated with a type environment Γ and a type τ) satisfying all the rules in Figure 4.4.

The main rule is (B-Exp). The other rules are just for preserving the relation $\preceq$ under any context.

Then, we can prove:

Lemma 4.19. For any

$$\alpha_1, \dots, \alpha_m, x_1\!:\!\tau_1, \dots, x_n\!:\!\tau_n \quad\vdash\quad M \quad\preceq\quad M' \quad:\quad \tau,$$

for any closed $\sigma_1, \dots, \sigma_m$, and for any $(\vdash V_1 \preceq V_1' : [\overline{\sigma}/\overline{\alpha}]\tau_1) \wedge \dots \wedge (\vdash V_n \preceq V_n' : [\overline{\sigma}/\overline{\alpha}]\tau_n)$, we have

$$[\overline{V}/\overline{x}][\overline{\sigma}/\overline{\alpha}]M \Downarrow \quad\iff\quad [\overline{V}'/\overline{x}][\overline{\sigma}/\overline{\alpha}]M' \Downarrow.$$

Furthermore, if $[\overline{V}/\overline{x}][\overline{\sigma}/\overline{\alpha}]M \Downarrow W$ and $[\overline{V}'/\overline{x}][\overline{\sigma}/\overline{\alpha}]M' \Downarrow W'$, then $\vdash W \preceq W' : [\overline{\sigma}/\overline{\alpha}]\tau$.

Proof. Straightforward induction on the derivation of $\overline{\alpha}, \overline{x}\!:\!\overline{\tau} \vdash M \preceq M' : \tau$. □

Theorem 4.20. For any $\overline{\alpha}, \overline{x}\!:\!\overline{\tau} \vdash M : \tau$ and $\overline{\alpha}, \overline{x}\!:\!\overline{\tau} \vdash M' : \tau$, if $\vdash \Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M \sim \Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M' : \forall\overline{\alpha}.\,\overline{\tau} \to \tau$, then $\overline{\alpha}, \overline{x}\!:\!\overline{\tau} \vdash M \overset{\text{std}}{\equiv} M' : \tau$.

Proof. By the soundness of bisimulation, we have $[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M)/z]C \Downarrow \iff [(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M')/z]C$ for any well-typed C. Thus, given K, take $C = K[z[\overline{\alpha}]\overline{x}]$ and we get $K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M)[\overline{\alpha}]\overline{x}] \Downarrow \iff K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M')[\overline{\alpha}]\overline{x}] \Downarrow$. Meanwhile, by the definition of $\preceq$, we have $\vdash K[M] \preceq K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M)[\overline{\alpha}]\overline{x}] : \texttt{unit}$ and $\vdash K[M'] \preceq K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M')[\overline{\alpha}]\overline{x}] : \texttt{unit}$. By the lemma above, we obtain $K[M] \Downarrow \iff K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M)[\overline{\alpha}]\overline{x}] \Downarrow$ and $K[M'] \Downarrow \iff K[(\Lambda\overline{\alpha}.\,\lambda\overline{x}\!:\!\overline{\tau}.\,M')[\overline{\alpha}]\overline{x}] \Downarrow$. Hence $K[M] \Downarrow \iff K[M'] \Downarrow$. □

Example 4.21. We have $x\!:\!\texttt{int} \vdash x+1 \overset{\text{std}}{\equiv} 1+x : \texttt{int}$. That is, $x+1$ and $1+x$ are contextually equivalent (in the standard sense above) at type $\texttt{int}$ provided that x has type $\texttt{int}$. To show this, it suffices to prove $\emptyset \vdash \lambda x\!:\!\texttt{int}.\,x+1 \sim \lambda x\!:\!\texttt{int}.\,1+x : \texttt{int} \to \texttt{int}$, which is trivial.

Example 4.22. The packages

$$\begin{aligned} M &= \texttt{pack int}, \langle y, \lambda x\!:\!\texttt{int}.\,x\rangle \texttt{ as } \tau \\ M' &= \texttt{pack int}, \langle y+1, \lambda x\!:\!\texttt{int}.\,x-1\rangle \texttt{ as } \tau \end{aligned}$$

are contextually equivalent (again in the standard sense above) at type

$$\tau \quad=\quad \exists\alpha.\,\alpha \times (\alpha \to \texttt{int})$$

provided that y has type $\texttt{int}$. This follows from the bisimilarity of $\lambda y\!:\!\texttt{int}.\,M$ and $\lambda y\!:\!\texttt{int}.\,M'$, which was shown in Section 4.4.3.

4.7 Limitations (Or: The Return of Higher-Order Functions)

Although the proof of contextual equivalence in Section 4.4.5 was strikingly simple, the trick used there does not apply in general. For example, consider the following implementations of integer multisets with a higher-order function to compute a weighed sum of all elements. (We assume standard definitions of lists and binary trees.)

$$
\begin{array}{rcl}
\texttt{IntSet} & = & \texttt{pack}\,\texttt{intList}, \texttt{Nil}, \texttt{add}, \texttt{weigh}\ \texttt{as}\ \exists\alpha.\,\tau \\
\texttt{IntSet}' & = & \texttt{pack}\,\texttt{intTree}, \texttt{Lf}, \texttt{add}', \texttt{weigh}'\ \texttt{as}\ \exists\alpha.\,\tau \\
\tau & = & \alpha \times (\texttt{int} \to \alpha \to \alpha) \times ((\texttt{int} \to \texttt{real}) \to \alpha \to \texttt{real}) \\
\texttt{add} & = & \lambda i : \texttt{int}.\, \lambda s : \texttt{intList}.\, \texttt{Cons}(i, s) \\
\texttt{add}' & = & \lambda i : \texttt{int}.\, \texttt{fix}\ f(s : \texttt{intTree}) : \texttt{intTree} = \\
 & & \qquad \texttt{case}\ s\ \texttt{of}\ \texttt{Lf} \Rightarrow \texttt{Nd}(i, \texttt{Lf}, \texttt{Lf}) \\
 & & \qquad \talloblong\ \texttt{Nd}(j, s_1, s_2) \Rightarrow \texttt{if}\ i < j\ \texttt{then}\ \texttt{Nd}(j, f s_1, s_2)\ \texttt{else}\ \texttt{Nd}(j, s_1, f s_2) \\
\texttt{weigh} & = & \lambda g : \texttt{int} \to \texttt{real}.\, \texttt{fix}\ f(s : \texttt{intList}) : \texttt{real} = \\
 & & \qquad \texttt{case}\ s\ \texttt{of}\ \texttt{Nil} \Rightarrow 0 \ \talloblong\ \texttt{Cons}(j, s_0) \Rightarrow gj + f s_0 \\
\texttt{weigh}' & = & \lambda g : \texttt{int} \to \texttt{real}.\, \texttt{fix}\ f(s : \texttt{intTree}) : \texttt{real} = \\
 & & \qquad \texttt{case}\ s\ \texttt{of}\ \texttt{Lf} \Rightarrow 0 \ \talloblong\ \texttt{Nd}(j, s_1, s_2) \Rightarrow gj + f s_1 + f s_2
\end{array}
$$

Unlike the previous example, these implementations have no syntactic similarity, which disables the simple proof. Instead, we have to put the whole packages into the bisimulation along with their elements. Then, by Condition 2 of bisimulation, we need at least to prove $\texttt{weigh}\ V\ W \Downarrow \iff \texttt{weigh}'\ V'\ W' \Downarrow$ for a certain class of V, W, V', and W'. In particular, V and V' can be of the forms $\lambda z : \texttt{int}.\, [\texttt{IntSet}/y]D$ and $\lambda z : \texttt{int}.\, [\texttt{IntSet}'/y]D$ for any D of appropriate type. Thus, because of the function application gj in weigh and weigh$'$, we must prove

$$[\texttt{IntSet}, j/y, z]D \Downarrow \quad \iff \quad [\texttt{IntSet}', j/y, z]D \Downarrow$$

for every D (and j). We are stuck, however, since this subsumes the definition of $\texttt{IntSet} \equiv \texttt{IntSet}'$ and is *harder* to prove!

Resolving this problem requires weakening Condition 2. By a close examination of the soundness proof (in Appendix C.2 and C.3), we find the following candidate.

2$'$. Take any

$$(\texttt{fix}\ f(x : \pi) : \rho = M,\ \texttt{fix}\ f(x : \pi') : \rho' = M',\ \tau \to \sigma) \in \mathcal{R}$$

and any $(V, V', \tau) \in (\Delta, \mathcal{R})^\circ$. Assume that, for any

$$
\begin{array}{rcl}
N & \sqsubset & (\texttt{fix}\ f(x : \pi) : \rho = M)V \\
N' & \sqsubset & (\texttt{fix}\ f(x : \pi') : \rho' = M')V'
\end{array}
$$

with $(N, N', \sigma) \in (\Delta_0, \mathcal{R}_0)^\circ$ and $(\Delta_0, \mathcal{R}_0) \in X$, we have $N \Downarrow \iff N' \Downarrow$. Assume furthermore that, if $N \Downarrow U$ and $N' \Downarrow U'$, then $(U, U', \sigma) \in (\Delta_1, \mathcal{R}_1)^\circ$ for some $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$ with $(\Delta_1, \mathcal{R}_1) \in X$. Then, we have

$$(\texttt{fix}\ f(x : \pi) : \rho = M)V \Downarrow \quad \iff \quad (\texttt{fix}\ f(x : \pi') : \rho' = M')V' \Downarrow.$$

Furthermore, if $(\texttt{fix}\ f(x : \pi) : \rho = M)V \Downarrow W$ and $(\texttt{fix}\ f(x : \pi') : \rho' = M')V \Downarrow W'$, then $(W, W', \sigma) \in (\Delta_2, \mathcal{R}_2)^\circ$ for some $\Delta_2 \supseteq \Delta$ and $\mathcal{R}_2 \supseteq \mathcal{R}$ with $(\Delta_2, \mathcal{R}_2) \in X$.

Here, $N_1 \sqsubset N_2$ means that, if $N_2 \Downarrow$, then $N_1 \Downarrow$ and the former evaluation derivation tree is strictly taller than the latter. (This is reminiscent of *indexed models* [Appel and McAllester 2001; Ahmed, Appel, and Virga 2003], but it is unclear how they extend to a relational setting with existential types.)

This generalization seems quite powerful: for instance, it allows us to conclude that gj in `weigh` and `weigh`$'$ gives the same result when g is substituted by V or V'. Unfortunately, however, the condition above has X in a negative position ($(\Delta_1, \mathcal{R}_1) \in X$) and breaks the property that the union of two bisimulations is a bisimulation. Although it is still possible to prove soundness (for an arbitrary bisimulation X instead of $\sim$) and completeness ($\equiv$ is still a bisimulation because Condition 2$'$ is *weaker* than Condition 2), the new condition is rather technical and hard to understand. We leave it for future work to find a more intuitive principle behind Condition 2$'$ that addresses this issue.

4.8 Related Work

Semantic logical relations. Originally, logical relations were devised in denotational semantics for relating models of λ-calculus. Although they are indeed useful for this purpose (e.g., relating CPS semantics and direct-style semantics), they are not as useful for proving contextual equivalence or abstraction properties, for the following reasons. First, denotational semantics tend to require more complex mathematics (such as CPOs and categories) than operational semantics. Second, it is hard—though not impossible [Hughes 1997]—to define a *fully abstract* model of polymorphic λ-calculus, i.e., a model that always preserves equivalence. Without full abstraction, not all equivalent terms can be proved to be equivalent.

Logical relations for polymorphic λ-calculus are also useful for proving *parametricity* properties [Wadler 1989], e.g., that all functions of type $\forall\alpha.\, \alpha \to \alpha$ behave like the polymorphic identity function (or diverge, if there is recursion in the language). By contrast, our bisimulation is only useful for proving the equivalence of two given λ-terms and cannot be employed for predicting such properties based on only types.

Syntactic logical relations. Pitts [2000] proposed *syntactic logical relations*, which use only the term model of polymorphic λ-calculus to prove contextual equivalence. He introduced the notion of $\top\top$-closure (application closure of the two functions in a Galois connection between terms and contexts) in order to treat recursive functions without using denotational semantics. He proved that his syntactic logical relations are complete with respect to contextual equivalence in call-by-name polymorphic λ-calculus with recursive functions and universal types (and lists).

Pitts [1998] also proposed syntactic logical relations for a variant of call-by-value polymorphic λ-calculus with recursive functions, universal types, and existential types, where type abstraction is restricted to values like $\Lambda\alpha.\, V$ instead of $\Lambda\alpha.\, M$. Although he showed (by a counter-example) that these logical relations are incomplete in this language and attributed the incompleteness to the presence of recursive functions, we have shown that a similar counter-example can be given without using recursive functions [personal communication, June 2000]. However, both of the counter-examples depend on the fact that type abstraction is restricted to values. It remains unclear whether his syntactic logical relations can be made complete in a setting without this restriction.

Birkedal and Harper [1999] and Crary and Harper [2000] extended syntactic logical relations with recursive *types* by requiring certain unwinding properties. This extension is conjectured to be complete with respect to contextual equivalence [personal communication, March 2004].

Compared to syntactic logical relations, our bisimulation is even more syntactic and elementary, liberating its user from the burden of calculating $\top\top$-closure or proving unwinding properties even with arbitrary recursive types (and functions).

Applicative bisimulations. Abramsky [1990] proposed *applicative bisimulations* for proving contextual equivalence of untyped λ-terms. Gordon and Rees [Gordon 1995a; Gordon and Rees 1996; Gordon 1995b; Gordon and Rees 1995] adapted applicative bisimulations to calculi with objects, subtyping, universal polymorphism, and recursive types. As discussed in Section 4.1, however, these results do not apply to type abstraction using existential types. We solved this issue by considering sets of relations as bisimulations.

As a byproduct, it has become much *easier* to prove the soundness of our bisimulation: technically, this simplification is due to the generalization in the condition of bisimulation for functions (Condition 2 in Definition 4.11), where our bisimulation allows different arguments V and V' while applicative bisimulation requires them to be the same.

Bisimulations for polymorphic π-calculi. Pierce and Sangiorgi [2000] developed a bisimulation proof technique for polymorphic π-calculus, using a separate environment for representing contexts' knowledge. In a sense, our bisimulation unifies the environmental knowledge with the bisimulation itself by generalizing the latter as a set of relations. Because of the imperative nature of π-calculus, their bisimulation is far from complete—in particular, aliasing of names is problematic.

Berger, Honda, and Yoshida [2003] defined two equivalence proof methods for linear π-calculi, one based on the syntactic logical relations of Pitts [1998, 2000] and the other based on the bisimulations of Pierce and Sangiorgi [2000]. Their main goal is to give a generic account for various features such as functions, state and concurrency by encoding them into appropriate versions of linear π-calculi. They proved soundness and completeness of their logical relations for one of the linear π-calculi, which directly corresponds to polymorphic λ-calculus (without recursion). They also proved full abstraction of the call-by-value and call-by-name encodings of the polymorphic λ-calculus to this version of linear π-calculus. However, for the other settings (e.g., with recursive functions or types), full abstraction of encodings and completeness of their logical relations are unclear. Completeness of their bisimulations is not discussed either. In addition, their developments are much heavier than ours for the purpose of just proving the equivalence of typed λ-terms.

Bisimulations for cryptographic calculi. Various bisimulations [Abadi and Gordon 1998; Abadi and Fournet 2001; Boreale, De Nicola, and Pugliese 2002; Borgström and Nestmann 2002] have been proposed for extensions of π-calculus with cryptographic primitives [Abadi and Gordon 1999; Abadi and Fournet 2001]. Their main idea is similar to Pierce and Sangiorgi's: using a separate environment to represent attackers' knowledge. In the previous chapter, we have applied our idea of using sets of relations as bisimulations to an extension of λ-calculus with perfect encryption (also known as dynamic sealing) and obtained a sound and complete proof method for contextual equivalence in this setting. Although this extension was untyped, it is straightforward to combine the present work with the previous one and obtain a bisimulation for typed λ-calculus with perfect encryption. The fact that our idea applies to such apparently different forms of information hiding as encryption and type abstraction suggests that it is successful in capturing the essence of "information hiding" in programming languages and computation models.

4.9 Future Work

We have presented the first sound, complete, and elementary bisimulation proof method for λ-calculus with full universal, existential, and recursive types.

Although full automation is impossible because equivalence of λ-terms (with recursion) is undecidable, some mechanical support would be useful. The technique of "bisimulation up to" [Sangiorgi and Milner 1992] would also be useful to reduce the size of a bisimulation in some cases, though our bisimulations tend to be smaller than bisimulations in process calculi in the first place, since ours are based on big-step evaluation rather than small-step reduction.

Another direction is to extend the calculus with more complex features such as state (cf. [Pitts and Stark 1998; Bierman, Pitts, and Russo 2000]). For example, it would be possible to treat state by passing around the state throughout the evaluation of terms and their bisimulation. More ambitiously, one could imagine generalizing this state-passing approach to more general "monadic bisimulation" by formalizing effects via monads [Moggi 1991].

Yet another possibility is to adopt our idea of "sets of relations as bisimulations" to other higher-order calculi with information hiding—such as higher-order π-calculus [Sangiorgi 1992], where restriction hides names and complicates equivalence—and compare the outcome with context bisimulation.

Finally, as suggested above, the idea of considering sets of relations as bisimulations may be useful for other forms of information hiding such as secrecy typing [Heintze and Riecke 1998]. It would be interesting to see whether such an adaptation is indeed possible and, furthermore, to consider if these variations can be generalized into a unified theory of information hiding.

Chapter 5

Conclusions

5.1 Summary of Results

Theories of information hiding were presented in versions of λ-calculus: syntactic logical relations for simply typed λ-calculus with perfect encryption (i.e., dynamic sealing), bisimulations for untyped λ-calculus with perfect encryption, and bisimulations for λ-calculus with general recursion and impredicative polymorphism (both universal and existential). All of the three theories were proved to be sound with respect to a natural notion of program equivalence, and the latter two were proved complete as well (while the first can also be made complete [Goubault-Larrecq, Lasota, Nowak, and Zhang 2004]). Our thesis was that these theories can be used for reasoning about programs involving information hiding. It was shown both by the proofs above and by examples including abstract data structures such as complex numbers, set functors, and counter objects as well as cryptographic protocols such as Needham-Schroeder-Lowe [Needham and Schroeder 1978; Lowe 1995] and ffgg [Millen 1999].

5.2 Related Work in Perspective

The relationships of our results to other work have already been discussed in each chapter. From a broader viewpoint, the major portion of related work can be classified into three categories.

5.2.1 Semantic and Syntactic Logical Relations

Originally, logical relations were developed for comparing denotational models of λ-calculus such as complete partial orders and Cartesian closed categories. Mitchell [1996, Chapter 8] offers an introduction to this area. The problem of this approach for the purpose of reasoning about abstraction is twofold: first, very few semantic models of information hiding (in a general form like impredicative polymorphism) are fully abstract and preserve equivalence; second, these models are far too complex to be useful for reasoning about realistic examples such as ours.

To avoid these difficulties, Pitts [2000] proposed logical relations based on a syntactic model for λ-calculus with universal types and recursive functions. He also developed an existential version [Pitts 1998], though it suffers from a strange problem concerning completeness (see Section 4.8). Birkedal and Harper [1999] adapted the approach of Pitts to a language with a single arbitrary recursive type and Crary and Harper [2000] extended it to general multiple recursive types. All of them relied on some notion of continuity to account for recursion: $\top\top$-closure in Pitts and admissibility in Harper et al.

On one hand, it is possible to adapt our logical relations for encryption (in Chapter 2) to a denotational setting and thereby obtain a beautiful and complete theory [Goubault-Larrecq, Lasota, Nowak, and Zhang 2004] based on a categorical formalism of fresh name generation [Stark 1996]. On the other hand, however, our bisimulations for encryption (in Chapter 3) gives a much simpler, operational, and still complete theory with no category, $\top\top$-closure, or admissibility.

5.2.2 Extensions of Pi-Calculus and their Bisimulations

As our calculi are extensions of λ-calculus with information hiding, polymorphic π-calculus [Turner 1995] and spi-calculus [Abadi and Gordon 1999] are extensions of π-calculus with type abstraction and (perfect and symmetric) encryption. They have their own theories of bisimulations: Pierce and Sangiorgi [2000] developed bisimulations for polymorphic π-calculus by annotating them with separate environments describing possible values of abstract types; inspired by these bisimulations, Abadi and Gordon [1999] proposed bisimulations for spi-calculus, also annotated with separate environments to describe possible values involving secret keys; Boreale, De Nicola, and Pugliese [2002] and Borgström and Nestmann [2002] improved Abadi and Gordon's bisimulations for spi-calculus in more appropriate forms so that they provide better technicalities such as so-called "up-to" properties; Abadi and Fournet [2001] generalized spi-calculus with arbitrary algebraic operations including imperfect and asymmetric encryption, and developed a theory of bisimulations which represents attackers' knowledge through explicit substitutions.

Compared to these results, our bisimulations benefit—and suffer, albeit to a lesser degree—from the simplicity of λ-calculus against π-calculus. Since functions are values in λ-calculus while processes are not messages in π-calculus, our bisimulations do not need separate environments (or explicit substitutions) to account for attackers' knowledge; instead, it leads to very natural formulation to consider *sets* of relations as bisimulations (and contextual equivalence). Moreover, π-calculus is imperative and much lower-level than λ-calculus: as a result, completeness is trickier and none of the bisimulations above seem to be proved complete[1], while our completeness proof is straightforward. The "big-step" evaluation semantics of λ-calculus also helps to reduce the size of bisimulations in contrast to "small-step" reduction semantics of π-calculus. A price of these simplicities is that we cannot model the linearity or state of processes; in fact, however, this price is often low and we are indeed able to verify many cryptographic protocols (and even so-called "necessarily parallel" attacks as in Section 2.3.3) with no problem.

5.2.3 Applicative Bisimulations and their Variants

Bisimulations for functional languages were first studied by Abramsky [1990] for untyped call-by-name λ-calculus and called *applicative bisimulations*. Their soundness with respect to contextual equivalence is proved via a rather mysterious proof technique [Howe 1996]. Gordon [1995a] studied bisimulations for a simply typed call-by-name λ-calculus with streams and recursive functions, Gordon and Rees [1996] for a first-order language with objects and subtyping, Gordon and Rees [1995] for λ-calculus with universal types and subtyping, Gordon [1995b] for a language with objects and universal types (as well as an untyped language with objects), Jeffrey and Rathke [1999]

[1]Despite the completeness claims in certain papers [Borgström and Nestmann 2002; Abadi and Fournet 2001], no proof is published or written down in accessible forms [personal communications, August 2004].

$$\begin{tikzcd}
\text{contextual equivalence in } \lambda_\mu^{\forall\exists} \arrow[d, "\substack{\text{completeness of bisimulations}\\ \text{for type abstraction}}"'] & \text{contextual equivalence in } \lambda_{\texttt{seal}} \\
\text{bisimulations in } \lambda_\mu^{\forall\exists} \arrow[r, "\text{the translation}"'] & \text{bisimulations in } \lambda_{\texttt{seal}} \arrow[u, "\substack{\text{soundness of bisimulations}\\ \text{for perfect encryption}}"']
\end{tikzcd}$$

Figure 5.1: Conjectured outline of full abstraction

for a language with fresh name generation, and Jeffrey and Rathke [2004] for a language with generative communication channels. All of them use Howe's method for proving their soundness and none of them can be used for showing non-trivial contextual equivalence induced by encryption or existential types. By contrast, thanks to sets of relations, our bisimulations for information hiding are complete and their soundness proof is much easier.

5.3 Directions for Future Work

Among the possibilities of future work discussed in previous chapters, the most interesting is to prove full abstraction for the translation (outlined in Section 3.8) of type abstraction into perfect encryption. Roughly, the proof should be possible by translating the bisimulations for type abstraction in Chapter 3 into those for perfect encryption in Chapter 4, as illustrated in Figure 5.1. Although first-order cases would be straightforward by induction on types, higher-order functions (specifically, arrow types in negative positions) are a challenge since they reverse the roles of terms and contexts, requiring some dual of contextual equivalence—which we might call "contextual co-equivalence"—as an induction hypothesis. Roughly, a term M is "contextually co-equivalent" if $C[M]$ and $C'[M]$ give the same observable result for any equivalent contexts C and C' sharing some secret. Non-trivial work seems necessary concerning how to show such properties during the proof. Intuitively, it is not surprising that full abstraction is so hard: in fact, no fully abstract semantics is known at all for λ-calculus with impredicative polymorphism, except for the trivial term model and a complex (and—arguably—unsatisfactory) one based on game theory [Hughes 1997].

Another interesting direction is to design and implement a real programming language environment based on the theoretical connection between type abstraction and encryption (or sealing) studied in this thesis. Such an environment can accommodate not only statically checked code (as in ML and Java), but also dynamically checked code (as in Scheme and Perl) by means of symbolic sealing—as well as not-at-all-checked code (as in Assembly and C) by means of real encryption—without losing protection for data abstraction. Figure 5.2 illustrates one possible architecture for this scheme. Naturally, more research and development are necessary for carrying out this rather ambitious project.

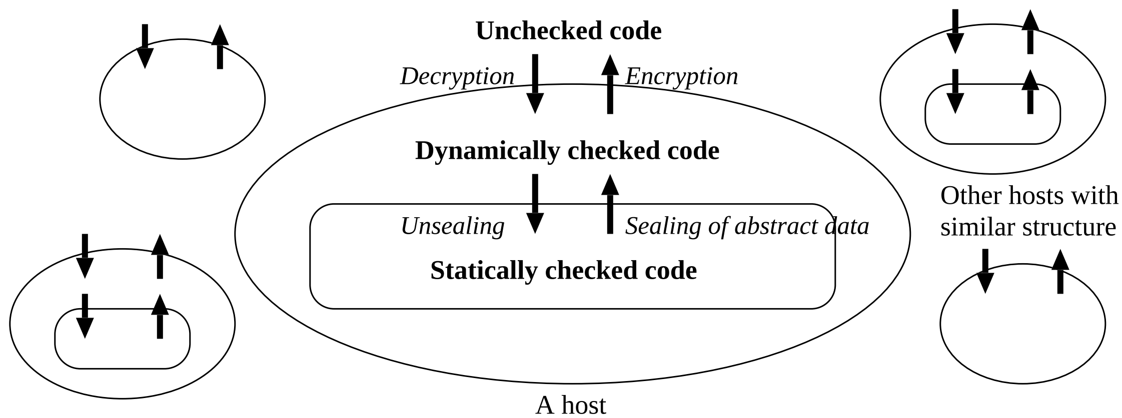

Figure 5.2: Data abstraction with three levels of checked code

Appendix A

Proofs for Chapter 2

For the sake of conciseness, we write $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi$ and $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, respectively, for the sets $\{(v, v') \mid \varphi \vdash (s)v \sim (s')v' : \tau\}$ and $\{(e, e') \mid \varphi \vdash (s)e \approx (s')e' : \tau\}$. That is, $(v, v') \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi$ means $\varphi \vdash (s)v \sim (s')v' : \tau$ and $(e, e') \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ means $\varphi \vdash (s)e \approx (s')e' : \tau$.

In the proofs, we use the following lemmas.

A.1 Lemmas about Evaluation

Lemma A.1 (Monotonic Increase of Key Set). If $(s)e \Downarrow (s')v$, then $s \subseteq s'$. If $s \supseteq \mathit{Keys}(e)$ furthermore, then $s' \supseteq \mathit{Keys}(v)$.

Proof. By induction on the derivation of $(s)e \Downarrow (s')v$. □

Lemma A.2 (Evaluation of Value). $(s)v \Downarrow (s)v$ for any s and v. In addition, if $(s)v \Downarrow V$, then $V = (s)v$.

Proof. Immediately follows by induction on the structure of v. □

Lemma A.3 (Weakening of Key Set). If $(s)e \Downarrow (s')v$, then $(s \uplus t)e \Downarrow (s' \uplus t)v$ for any t with $t \cap s = \emptyset$ and $t \cap s' = \emptyset$.

Proof. By induction on the derivation of $(s)e \Downarrow (s')v$. □

A.2 Lemmas about Typing

Lemma A.4 (Weakening of Type Judgment). If $\Gamma, \Delta \vdash e : \tau$, then $\Gamma \uplus \Gamma', \Delta \uplus \Delta' \vdash e : \tau$ for any Γ' and Δ' with $\mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma') = \emptyset$ and $\mathit{dom}(\Delta) \cap \mathit{dom}(\Delta') = \emptyset$.

Proof. By induction on the derivation of $\Gamma, \Delta \vdash e : \tau$. □

Lemma A.5 (Substitution Lemma). Let $\Gamma = \{\tilde{x} \mapsto \tilde{\tau}\}$. If $\Gamma, \Delta \vdash e : \tau$ and $\emptyset, \Delta \vdash \tilde{v} : \tilde{\tau}$, then $\Gamma, \Delta \vdash [\tilde{v}/\tilde{x}]e : \tau$.

Proof. By induction on the derivation of $\Gamma, \Delta \vdash e : \tau$. We use Lemma A.4 when the last rule used for the derivation is (Var). □

A.3 Lemmas about Logical Relation

Lemma A.6 (Coincidence of Logical Relations). $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ for any s, s', τ, φ.

Proof. Immediately follows from the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$ and from Lemma A.2. $\square$

Lemma A.7 (Weakening of Logical Relation). Suppose $s \cap s_0 = \emptyset$ and $s' \cap s'_0 = \emptyset$ where $dom(\varphi) \subseteq s \cap s'$ and $dom(\varphi_0) \subseteq s_0 \cap s'_0$. Then, $\mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{val}}_{s \uplus s_0, s' \uplus s'_0}(\tau)(\varphi \uplus \varphi_0)$ and $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s \uplus s_0, s' \uplus s'_0}$ $(\tau)(\varphi \uplus \varphi_0)$.

Proof. By induction on the structure of τ. In the proof of the latter half, note that, by Lemma A.3, if $(s)e \Downarrow (s \uplus t)v$ and $(s')e' \Downarrow (s' \uplus t')v'$, then $(s \uplus s_0)e \Downarrow (s \uplus s_0 \uplus t)v$ and $(s' \uplus s'_0)e' \Downarrow (s' \uplus s'_0 \uplus t')v'$. $\square$

A.4 Proof of Theorem 2.3

By induction on the derivation of $(s_1)e \Downarrow (s_1 \uplus s'_1)v_1$. The latter half of Lemma A.1 is used when the last rule used for the derivation is either (1) the evaluation rule for function application or (2) the evaluation rule for decryption. $\square$

A.5 Proof of Theorem 2.5

By induction on the derivation of $(s)[\tilde{v}/\tilde{x}]e \Downarrow V$. We perform case analysis on the form of e. We show the following four cases (the other cases are similar).

Case $e = x$**.** By (Var), $x = x_i$ and $\tau = \tau_i$ for some i. Thus, by Lemma A.2, we have $V = (s)v_i$. So the theorem follows by letting $v = v_i$ and $\Delta' = \emptyset$.

Case $e = \lambda x.\, e_0$**.** By the evaluation rule for λ-abstraction, we have $V = (s)\lambda x.\, [\tilde{v}/\tilde{x}]e_0$. Meanwhile, by (Abs), τ is of the form $\sigma_1 \to \sigma_2$ and we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash e_0 : \sigma_2$. Therefore, by Lemma A.5, we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash [\tilde{v}/\tilde{x}]e_0 : \sigma_2$. Then, by Lemma (Abs), we have $\Gamma, \Delta \vdash \lambda x.$ $[\tilde{v}/\tilde{x}]e_0 : \sigma_1 \to \sigma_2$. Thus, the theorem follows by letting $v = \lambda x.\, [\tilde{v}/\tilde{x}]e_0$ and $\Delta' = \emptyset$.

Case $e = \nu x.\, e_0$**.** By the evaluation rule for key generation, we have $(s \uplus \{k\})[k/x][\tilde{v}/\tilde{x}]e_0 \Downarrow V$ for some k. Meanwhile, by (Key), we have $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \vdash k : \mathtt{key}[\tau']$. In addition, by Lemma A.4, we have $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \vdash \tilde{v} : \tilde{\tau}$, Furthermore, by (New) we have $\Gamma \uplus \{x \mapsto \mathtt{key}[\tau']\}, \Delta \vdash e_0 : \tau$, so by Lemma A.4 we have $\Gamma \uplus \{x \mapsto \mathtt{key}[\tau']\}, \Delta \uplus \{k \mapsto \tau'\} \vdash e_0 : \tau$. Therefore, by the induction hypothesis, there exist some v'_0 and Δ_0 such that $V = (s \uplus \{k\} \uplus s_0)v'_0$ and $\emptyset, \Delta \uplus \{k \mapsto \tau'\} \uplus \Delta_0 \vdash$ $v'_0 : \tau$ for $s_0 = dom(\Delta_0)$. Thus, the theorem follows by letting $v = v'_0$ and $\Delta' = \{k \mapsto \tau'\} \uplus \Delta_0$.

Case $e = (\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4)$**.** By (Dec), we have $\Gamma, \Delta \vdash e_1 : \mathtt{key}[\tau']$, $\Gamma, \Delta \vdash e_2 : \mathtt{bits}$ $[\tau']$, $\Gamma \uplus \{x \mapsto \tau'\}, \Delta \vdash e_3 : \tau$, and $\Gamma, \Delta \vdash e_4 : \tau$ for some τ'. In addition, by the evaluation rules for decryption, we have $(s)[\tilde{v}/\tilde{x}]e_1 \Downarrow V_1$ for some V_1. Therefore, by the induction hypothesis, there exist some v'_1 and Δ_1 such that $V_1 = (s \uplus s_1)v'_1$ and $\emptyset, \Delta \uplus \Delta_1 \vdash v'_1 : \mathtt{key}[\tau']$ for $s_1 = dom(\Delta_1)$. Then, since v'_1 is a value of a key type, v'_1 is of the form k_1 by (Key).

Furthermore, by the evaluation rules for decryption, we have $(s \uplus s_1)[\tilde{v}/\tilde{x}]e_2 \Downarrow V_2$ for some V_2. Meanwhile, by Lemma A.4, we have $\Gamma, \Delta \uplus \Delta_1 \vdash e_2 : \mathtt{bits}[\tau']$. Furthermore, by Lemma A.4, we have $\emptyset, \Delta \uplus \Delta_1 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some v_2' and Δ_2 such that $V_2 = (s \uplus s_1 \uplus s_2)v_2'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash v_2' : \mathtt{bits}[\tau']$ for $s_2 = dom(\Delta_2)$. Then, since v_2' is a value of a ciphertext type, v_2' is of the form $\{v'\}_{k_2}$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash v' : \tau'$ by (Enc).

Now we perform the following case analysis.

Sub-case $k_1 = k_2$. Let $k_1 = k_2 = k$. By the evaluation rules for decryption, we have $(s \uplus s_1 \uplus s_2)[v'/x][\tilde{v}/\tilde{x}]e_3 \Downarrow V$. In addition, by Lemma A.4, we have $\Gamma \uplus \{x \mapsto \tau'\}, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash e_3 : \tau$. Furthermore, by Lemma A.4, we have $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some v_3' and Δ_3 such that $V = (s \uplus s_1 \uplus s_2 \uplus s_3)v_3'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \uplus \Delta_3 \vdash v_3' : \tau$ for $s_3 = dom(\Delta_3)$. Thus, the theorem follows by letting $v = v_3'$ and $\Delta' = \Delta_1 \uplus \Delta_2 \uplus \Delta_3$.

Sub-case $k_1 \neq k_2$. By the evaluation rules for decryption, we have $(s \uplus s_1 \uplus s_2)[\tilde{v}/\tilde{x}]e_4 \Downarrow V$. In addition, by Lemma A.4, we have $\Gamma, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash e_4 : \tau$. Furthermore, by Lemma A.4, we have $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \vdash \tilde{v} : \tilde{\tau}$. Therefore, by the induction hypothesis, there exist some v_4' and Δ_4 such that $V = (s \uplus s_1 \uplus s_2 \uplus s_4)v_4'$ and $\emptyset, \Delta \uplus \Delta_1 \uplus \Delta_2 \uplus \Delta_4 \vdash v_4' : \tau$ for $s_4 = dom(\Delta_4)$. Thus, the theorem follows by letting $v = v_4'$ and $\Delta' = \Delta_1 \uplus \Delta_2 \uplus \Delta_4$. □

A.6 Proof of Theorem 2.10

By induction on the structure of e. We perform case analysis on the form of e. We show the following four cases (the other cases are similar).

Case $e = x$. By (Var), we have $x = x_i$ and $\tau = \tau_i$ for some i. Thus, by Lemma A.6, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (v_i, v_i') \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\tau)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau_i)\varphi$.

Case $e = \lambda x.\, e_0$. By (Abs), τ is of the form $\sigma_1 \to \sigma_2$ and we have $\Gamma \uplus \{x \mapsto \sigma_1\}, \Delta \vdash e_0 : \sigma_2$. Assume $(v, v') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t, s' \uplus t'}(\sigma_1)(\varphi \uplus \psi)$ with $dom(\psi) \subseteq t \cap t'$. By Lemma A.7, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t, s \uplus t'}(\tilde{\tau})(\varphi \uplus \psi)$. Therefore, by the induction hypothesis, we have $([v/x][\tilde{v}/\tilde{x}]e_0, [v'/x][\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s \uplus t, s' \uplus t'}(\sigma_2)\varphi \uplus \psi$. Thus, by the definition of $\mathcal{R}^{\mathrm{val}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi$ and by Lemma A.6, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\lambda x.\, [\tilde{v}/\tilde{x}]e_0, \lambda x.\, [\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{val}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi \subseteq \mathcal{R}^{\mathrm{exp}}_{s,s'}(\sigma_1 \to \sigma_2)\varphi = \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$.

Case $e = \nu x.\, e_0$. By (New), we have $\Gamma \uplus \{x \mapsto \mathtt{key}[\sigma]\}, \Delta \vdash e_0 : \tau$ for some σ. By Lemma A.7, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tilde{\tau})\varphi$ for any $k \notin s \cup s'$. In addition, by the definition of $\mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\mathtt{key}[\sigma])\varphi$, we have $(k, k) \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\}, s' \uplus \{k\}}(\mathtt{key}[\sigma])\varphi$. Therefore, by the induction hypothesis, we have $([k/x][\tilde{v}/\tilde{x}]e_0, [k/x][\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tau)\varphi$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s \uplus \{k\}, s' \uplus \{k\}}(\tau)\varphi$, there exist some t_0, w_0, t_0', w_0', and ψ_0 with $dom(\psi_0) \subseteq t_0 \cap t_0'$ such that $(s \uplus \{k\})[k/x][\tilde{v}/\tilde{x}]e_0 \Downarrow (s \uplus \{k\} \uplus t_0)w_0$, $(s' \uplus \{k\})[k/x][\tilde{v}'/\tilde{x}]e_0 \Downarrow (s' \uplus \{k\} \uplus t_0')w_0'$, and $(w_0, w_0') \in \mathcal{R}^{\mathrm{val}}_{s \uplus \{k\} \uplus t_0, s' \uplus \{k\} \uplus t_0'}(\tau)(\varphi \uplus \psi_0)$. Then, by the evaluation rule for key generation, we have $(s)\nu x.\, [\tilde{v}/\tilde{x}]e_0 \Downarrow (s \uplus \{k\} \uplus t_0)w_0$ and $(s')\nu x.\, [\tilde{v}'/\tilde{x}]e_0 \Downarrow (s' \uplus \{k\} \uplus t_0')w_0'$. Thus, by letting $t = \{k\} \uplus t_0$, $v = w_0$, $t' = \{k\} \uplus t_0'$, $v' = w_0'$, and $\psi = \psi_0$ in the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\nu x.\, [\tilde{v}/\tilde{x}]e_0, \nu x.\, [\tilde{v}'/\tilde{x}]e_0) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\tau)\varphi$.

Case $e = (\mathtt{let}\ \{x\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4)$. By (Dec), we have $\Gamma, \Delta \vdash e_1 : \mathtt{key}[\sigma]$, $\Gamma, \Delta \vdash e_2 : \mathtt{bits}[\sigma]$, $\Gamma \uplus \{x \mapsto \sigma\}, \Delta \vdash e_3 : \tau$, and $\Gamma, \Delta \vdash e_4 : \tau$ for some σ. Then, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_1, [\tilde{v}'/\tilde{x}]e_1) \in \mathcal{R}^{\mathrm{exp}}_{s,s'}(\mathtt{key}[\sigma])\varphi$. Thus, by the definition of $\mathcal{R}^{\mathrm{exp}}_{s,s'}(\mathtt{key}[\sigma])\varphi$, there exist some t_1, w_1, t_1', w_1', and ψ_1 with $dom(\psi_1) \subseteq t_1 \cap t_1'$ such that $(s)[\tilde{v}/\tilde{x}]e_1 \Downarrow (s \uplus t_1)w_1$, $(s')[\tilde{v}'/\tilde{x}]e_1 \Downarrow (s' \uplus t_1')w_1'$, and $(w_1, w_1') \in \mathcal{R}^{\mathrm{val}}_{s \uplus t_1, s' \uplus t_1'}(\mathtt{key}[\sigma])(\varphi \uplus \psi_1)$. Then, by the definition of $\mathcal{R}^{\mathrm{val}}_{s \uplus t_1, s' \uplus t_1'}(\mathtt{key}[\sigma])(\varphi \uplus \psi_1)$, we have $w_1 = w_1'$ is of the form k_1 where $k_1 \in (s \uplus t_1) \cap (s' \uplus t_1')$ and $k_1 \notin dom(\varphi \uplus \psi_1)$.

Furthermore, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_2, [\tilde{v}'/\tilde{x}]e_2) \in \mathcal{R}^{\text{exp}}_{s \uplus t_1, s' \uplus t'_1}(\texttt{bits}[\sigma])$ $(\varphi \uplus \psi_1)$. Thus, by the definition of $\mathcal{R}^{\text{exp}}_{s \uplus t_1, s' \uplus t'_1}(\texttt{bits}[\sigma])(\varphi \uplus \psi_1)$, there exist some t_2, w_2, t'_2, w'_2, and ψ_2 with $dom(\psi_2) \subseteq t_2 \cap t'_2$ such that $(s \uplus t_1)[\tilde{v}/\tilde{x}]e_2 \Downarrow (s \uplus t_1 \uplus t_2)w_2$, $(s' \uplus t'_1)[\tilde{v}'/\tilde{x}]e_2 \Downarrow$ $(s' \uplus t'_1 \uplus t'_2)w'_2$, and $(w_2, w'_2) \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\texttt{bits}[\sigma])(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the definition of $\mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2, s \uplus t'_1 \uplus t'_2}(\texttt{bits}[\sigma])(\varphi \uplus \psi_1 \uplus \psi_2)$, w_2 and w'_2 are respectively of the form $\{w\}_{k_2}$ and $\{w'\}_{k_2}$ where $k_2 \in (s \uplus t_1 \uplus t_2) \cap (s' \uplus t'_1 \uplus t'_2)$.

Now we perform the following case analysis.

Sub-case $k_1 = k_2$. Since $k_1 \notin dom(\varphi \uplus \psi_1 \uplus \psi_2)$, we have $(w, w') \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\sigma)(\varphi \uplus \psi_1 \uplus \psi_2)$. In addition, by Lemma A.7, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tilde{\tau})(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the induction hypothesis, we have $([w/x][\tilde{v}/\tilde{x}]e_3, [w'/x][\tilde{v}'/\tilde{x}]e_3) \in \mathcal{R}^{\text{exp}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$. Thus, by the definition of $\mathcal{R}^{\text{exp}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$, there exist some t_3, w_3, t'_3, w'_3, and ψ_3 with $dom(\psi_3) \subseteq t_3 \cap t'_3$ such that $(s \uplus t_1 \uplus t_2)[w/x][\tilde{v}/\tilde{x}]e_3 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_3)w_3$, $(s' \uplus t'_1 \uplus t'_2)[w'/x][\tilde{v}'/\tilde{x}]e_3 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_3)w'_3$, and $(w_3, w'_3) \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2 \uplus t_3, s' \uplus t'_1 \uplus t'_2 \uplus t'_3}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2 \uplus \psi_3)$. Therefore, by the evaluation rules for decryption, $(s)\texttt{let } \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2 \texttt{ in } [\tilde{v}/\tilde{x}]e_3 \texttt{ else } [\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_3)w_3$ and $(s')\texttt{let }$ $\{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2 \texttt{ in } [\tilde{v}'/\tilde{x}]e_3 \texttt{ else } [\tilde{v}'/\tilde{x}]e_4 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_3)w'_3$. Thus, by letting $t = t_1 \uplus t_2 \uplus t_3$, $v = w_3$, $t' = t'_1 \uplus t'_2 \uplus t'_3$, $v' = w'_3$, and $\psi = \psi_1 \uplus \psi_2 \uplus \psi_3$ in the definition of $\mathcal{R}^{\text{exp}}_{s,s'}(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\texttt{let } \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2 \texttt{ in } [\tilde{v}/\tilde{x}]e_3 \texttt{ else } [\tilde{v}/\tilde{x}]e_4, \texttt{let } \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2 \texttt{ in }$ $[\tilde{v}'/\tilde{x}]e_3 \texttt{ else } [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\text{exp}}_{s,s'}(\tau)\varphi$.

Sub-case $k_1 \neq k_2$. By Lemma A.7, we have $(\tilde{v}, \tilde{v}') \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tilde{\tau})(\varphi \uplus \psi_1 \uplus \psi_2)$. Then, by the induction hypothesis, we have $([\tilde{v}/\tilde{x}]e_4, [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\text{exp}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$. Thus, by the definition of $\mathcal{R}^{\text{exp}}_{s \uplus t_1 \uplus t_2, s' \uplus t'_1 \uplus t'_2}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2)$, there exist some t_4, w_4, t'_4, w'_4, and ψ_4 with $dom(\psi_4) \subseteq t_4 \cap t'_4$ such that $(s \uplus t_1 \uplus t_2)[\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_4)w_4$, $(s' \uplus t'_1 \uplus t'_2)[\tilde{v}'/\tilde{x}]e_4 \Downarrow (s' \uplus t'_1 \uplus t'_2 \uplus t'_4)w'_4$ and $(w_4, w'_4) \in \mathcal{R}^{\text{val}}_{s \uplus t_1 \uplus t_2 \uplus t_4, s' \uplus t'_1 \uplus t'_2 \uplus t'_4}(\tau)(\varphi \uplus \psi_1 \uplus \psi_2 \uplus \psi_4)$. Therefore, by the evaluation rules for decryption, we have $(s)\texttt{let } \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2 \texttt{ in } [\tilde{v}/\tilde{x}]e_4 \texttt{ else } [\tilde{v}/\tilde{x}]e_4 \Downarrow (s \uplus t_1 \uplus t_2 \uplus t_4)w_4$ and $(s')\texttt{let } \{x\}_{[\tilde{v}'/\tilde{x}]e_1} = [\tilde{v}'/\tilde{x}]e_2 \texttt{ in } [\tilde{v}'/\tilde{x}]e_4 \texttt{ else } [\tilde{v}'/\tilde{x}]e_4 \Downarrow (s \uplus t'_1 \uplus t'_2 \uplus t'_4)w'_4$. Thus, by letting $t = t_1 \uplus t_2 \uplus t_4$, $v = w_4$, $t' = t'_1 \uplus t'_2 \uplus t'_4$, $v' = w'_4$, and $\psi = \psi_1 \uplus \psi_2 \uplus \psi_4$ in the definition of $\mathcal{R}^{\text{exp}}_{s,s'}$ $(\tau)\varphi$, we have $([\tilde{v}/\tilde{x}]e, [\tilde{v}'/\tilde{x}]e) = (\texttt{let } \{x\}_{[\tilde{v}/\tilde{x}]e_1} = [\tilde{v}/\tilde{x}]e_2 \texttt{ in } [\tilde{v}/\tilde{x}]e_4 \texttt{ else } [\tilde{v}/\tilde{x}]e_4, \texttt{let } \{x\}_{[\tilde{v}'/\tilde{x}]e_1} =$ $[\tilde{v}'/\tilde{x}]e_2 \texttt{ in } [\tilde{v}'/\tilde{x}]e_4 \texttt{ else } [\tilde{v}'/\tilde{x}]e_4) \in \mathcal{R}^{\text{exp}}_{s,s'}(\tau)\varphi$. □

A.7 Proof of Corollary 2.11

By Theorem 2.10, we have $(f, f) \in \mathcal{R}^{\text{exp}}_{\emptyset,\emptyset}(\tau \to \texttt{bool})\emptyset$ for any f with $\emptyset, \emptyset \vdash f : \tau \to \texttt{bool}$. Then, by the definition of $\mathcal{R}^{\text{exp}}_{\emptyset,\emptyset}(\tau \to \texttt{bool})\emptyset$, there exist some s, w, s', w', φ with $dom(\varphi) \subseteq s \cap s'$ such that $(\emptyset)f \Downarrow (s)w$, $(\emptyset)f \Downarrow (s')w'$, and $(w, w') \in \mathcal{R}^{\text{val}}_{s,s'}(\tau \to \texttt{bool})\varphi$. Then, by the definition of $\mathcal{R}^{\text{val}}_{s,s'}(\tau \to$ $\texttt{bool})\varphi$, w and w' are respectively of the form $\lambda x.\, e_0$ and $\lambda x.\, e'_0$ where $([v/x]e_0, [v'/x]e'_0) \in \mathcal{R}^{\text{exp}}_{s \uplus t, s' \uplus t'}$ $(\texttt{bool})(\varphi \uplus \psi)$ for any v, v', t, t', ψ with $dom(\psi) \subseteq t \cap t'$ such that $(v, v') \in \mathcal{R}^{\text{val}}_{s \uplus t, s' \uplus t'}(\tau)(\varphi \uplus \psi)$.

Meanwhile, by Lemma A.7, we have $(e, e') \in \mathcal{R}^{\text{exp}}_{s,s'}(\tau)\varphi$. Then, by the definition of $\mathcal{R}^{\text{exp}}_{s,s'}(\tau)\varphi$, there exist some t, v, t', v', ψ with $dom(\psi) \subseteq t \cap t'$ such that $(s)e \Downarrow (s \uplus t)v$, $(s')e' \Downarrow (s' \uplus t')v'$, and $(v, v') \in \mathcal{R}^{\text{val}}_{s \uplus t, s' \uplus t'}(\tau)(\varphi \uplus \psi)$.

Therefore, $([v/x]e_0, [v'/x]e'_0) \in \mathcal{R}^{\text{exp}}_{s \uplus t, s' \uplus t'}(\texttt{bool})(\varphi \uplus \psi)$. Then, by the definition of $\mathcal{R}^{\text{exp}}_{s \uplus t, s' \uplus t'}$ $(\texttt{bool})(\varphi \uplus \psi)$, there exist some $s_0, w_0, s'_0, w'_0, \varphi_0$ with $dom(\varphi_0) \subseteq s_0 \cap s'_0$ such that $(s \uplus t)[v/x]e_0 \Downarrow$

$(s \uplus t \uplus s_0)w_0$, $(s' \uplus t')[v'/x]e'_0 \Downarrow (s' \uplus t' \uplus s'_0)w_0$, and $(w_0, w'_0) \in \mathcal{R}^{\text{val}}_{s \uplus t \uplus s_0, s' \uplus t' \uplus s'_0}(\texttt{bool})(\varphi \uplus \psi \uplus \varphi_0)$. Then, by the definition of $\mathcal{R}^{\text{val}}_{s \uplus t \uplus s_0, s' \uplus t' \uplus s'_0}(\texttt{bool})(\varphi \uplus \psi \uplus \varphi_0)$, we have $w_0 = w'_0 = \texttt{true}$ or $w_0 = w'_0 = \texttt{false}$.

By the way, by the evaluation rules for function application, we have $(\emptyset) f e \Downarrow (s \uplus t \uplus s_0)w_0$ and $(\emptyset) f e' \Downarrow (s' \uplus t' \uplus s'_0)w'_0$. Thus, by Definition 2.7, we have $\vdash e \equiv e' : \tau$. $\square$

Appendix B

Proofs for Chapter 3

B.1 Proof of Lemma 3.22

By induction on the derivation of $(s_0)\, e \Downarrow (t)\, w$.

By the definition of $\cong$, we have $e = [\overline{v}_0/\overline{x}_0]e_0$ and $e' = [\overline{v}'_0/\overline{x}_0]e_0$ for some $(s_0)\, \overline{v}_0 \sim_{\mathcal{R}_0} (s'_0)\, \overline{v}'_0$ with $Seals(e_0) = \emptyset$. If e is a value, then e' is also a value (easy case analysis on the syntax of e_0) and the result is immediate, because every value evaluates only to itself. We consider the remaining possibilities in detail, assuming that e_0 is neither a value nor a variable; there is one case for each of the non-value evaluation rules.

Case (E-Tuple), (E-Do-Seal). Straightforward induction.

Case (E-Proj). Straightforward induction, using Condition 4 of the definition of bisimulation.

Case (E-Prim), (E-Cond-True) and (E-Cond-False). Straightforward induction, using Condition 3 of the definition of bisimulation.

Case (E-New). Straightforward induction, using Lemma 3.19.

Case (E-App). Then e_0 has the form $e_1 e_2$ and the final step in the derivation of $(s_0)\, e \Downarrow (t)\, w$ has the following form:

$$\frac{(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\, w_1 \quad (s_1)\, [\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\, w_2 \quad \ldots}{(s_0)\, [\overline{v}_0/\overline{x}_0](e_1 e_2) \Downarrow (t)\, w}$$

The third premise is elided; we will come back to it in a minute. Since (E-App) is the only rule for evaluating an application, the final step in the derivation of $(s'_0)\, e' \Downarrow (t')\, w'$ has a similar form:

$$\frac{(s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\, w'_1 \quad (s'_1)\, [\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\, w'_2 \quad \ldots}{(s'_0)\, [\overline{v}'_0/\overline{x}_0](e_1 e_2) \Downarrow (t')\, w'}$$

By the definition of $\cong$, we have $(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \cong_{\mathcal{R}_0} (s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1$. Thus, by the induction hypothesis on the subderivations for $[\overline{v}_0/\overline{x}_0]e_1$ and $[\overline{v}'_0/\overline{x}_0]e_1$, we obtain $(s_1)\, w_1 \cong_{\mathcal{R}_1} (s'_1)\, w'_1$ for some $\mathcal{R}_1 \supseteq \mathcal{R}_0$. By the definition of $\cong$, we have $w_1 = [\overline{v}_1/\overline{x}_1]e_3$ and $w'_1 = [\overline{v}'_1/\overline{x}_1]e_3$ for some $(s_1)\, \overline{v}_1 \sim_{\mathcal{R}_1} (s'_1)\, \overline{v}'_1$ with $Seals(e_3) = \emptyset$. Since $(s_0)\, \overline{v}_0 \sim_{\mathcal{R}_0} (s'_0)\, \overline{v}'_0$ and $\mathcal{R}_0 \subseteq \mathcal{R}_1$, we have $(s_1)\, \overline{v}_0 \sim_{\mathcal{R}_1} (s'_1)\, \overline{v}'_0$ by Lemma 3.18.

Applying the definition of $\cong$ again, we have $(s_1)\, [\overline{v}_0/\overline{x}_0]e_2 \cong_{\mathcal{R}_1} (s'_1)\, [\overline{v}'_0/\overline{x}_0]e_2$. So we may apply the induction hypothesis to the subderivations for $[\overline{v}_0/\overline{x}_0]e_2$ and $[\overline{v}'_0/\overline{x}_0]e_2$, obtaining $(s_2)\, w_2 \cong_{\mathcal{R}_2}$

$(s'_2)\, w'_2$ for some $\mathcal{R}_2 \supseteq \mathcal{R}_1$. By the definition of $\cong$, we have $w_2 = [\overline{v}_2/\overline{x}_2]e_4$ and $w'_2 = [\overline{v}'_2/\overline{x}_2]e_4$ for some $(s_2)\, \overline{v}_2 \sim_{\mathcal{R}_2} (s'_2)\, \overline{v}'_2$ with $\mathit{Seals}(e_4) = \emptyset$. Since $(s_1)\, \overline{v}_1 \sim_{\mathcal{R}_1} (s'_1)\, \overline{v}'_1$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$, we have (s_2) $\overline{v}_1 \sim_{\mathcal{R}_2} (s'_2)\, \overline{v}'_1$ by Lemma 3.18.

Now we need to deal with the third premises. Since $w_1 = [\overline{v}_1/\overline{x}_1]e_3$ and $w'_1 = [\overline{v}'_1/\overline{x}_1]e_3$ must both be functions, e_3 itself must be either a function or a variable; we consider these cases in turn.

Sub-case $e_3 = \lambda y.\, e_5$**.** Then the final steps in the evaluation derivations for e and e' are:

$$\frac{\begin{array}{c}(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\, [\overline{v}_1/\overline{x}_1](\lambda y.\, e_5) \\ (s_1)\, [\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\, [\overline{v}_2/\overline{x}_2]e_4 \\ (s_2)\, [\overline{v}_1, \overline{v}_2/\overline{x}_1, \overline{x}_2][e_4/y]e_5 \Downarrow (t)\, w\end{array}}{(s_0)\, [\overline{v}_0/\overline{x}_0](e_1 e_2) \Downarrow (t)\, w}$$

$$\frac{\begin{array}{c}(s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\, [\overline{v}'_1/\overline{x}_1](\lambda y.\, e_5) \\ (s'_1)\, [\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\, [\overline{v}'_2/\overline{x}_2]e_4 \\ (s'_2)\, [\overline{v}'_1, \overline{v}'_2/\overline{x}_1, \overline{x}_2][e_4/y]e_5 \Downarrow (t')\, w'\end{array}}{(s'_0)\, [\overline{v}'_0/\overline{x}_0](e_1 e_2) \Downarrow (t')\, w'}$$

Since $(s_2)\, \overline{v}_1, \overline{v}_2 \sim_{\mathcal{R}_2} (s'_2)\, \overline{v}'_1, \overline{v}'_2$ and $\mathit{Seals}(e_4) = \mathit{Seals}(e_5) = \emptyset$, we have $(s_2)\, [\overline{v}_1, \overline{v}_2/\overline{x}_1, \overline{x}_2][e_4/y]e_5$ $\cong_{\mathcal{R}_2} (s'_2)\, [\overline{v}'_1, \overline{v}'_2/\overline{x}_1, \overline{x}_2][e_4/y]e_5$ by the definition of $\cong$. So we may apply the induction hypothesis a third time, yielding $(t)\, w \cong_{\mathcal{R}} (t')\, w'$ for some $\mathcal{R} \supseteq \mathcal{R}_2 \supseteq \mathcal{R}_1 \supseteq \mathcal{R}_0$, as required.

Sub-case $e_3 = x_{0i}$**, with** $v_{1i} = \lambda y.\, e_5$ **and** $v'_{1i} = \lambda y.\, e'_5$ **.** Then the evaluation derivations for e and e' are:

$$\frac{\begin{array}{c}(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\, \lambda y.\, e_5 \\ (s_1)\, [\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\, [\overline{v}_2/\overline{x}_2]e_4 \\ (s_2)\, [[\overline{v}_2/\overline{x}_2]e_4/y]e_5 \Downarrow (t)\, w\end{array}}{(s_0)\, [\overline{v}_0/\overline{x}_0](e_1 e_2) \Downarrow (t)\, w}$$

$$\frac{\begin{array}{c}(s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\, \lambda y.\, e'_5 \\ (s'_1)\, [\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\, [\overline{v}'_2/\overline{x}_2]e_4 \\ (s'_2)\, [[\overline{v}'_2/\overline{x}_2]e_4/y]e_5 \Downarrow (t')\, w'\end{array}}{(s'_0)\, [\overline{v}'_0/\overline{x}_0](e_1 e_2) \Downarrow (t')\, w'}$$

Since $(s_2)\, [[\overline{v}_2/\overline{x}_2]e_4/y]e_5 \Downarrow (t)\, w$ and $(s'_2)\, [[\overline{v}'_2/\overline{x}_2]e_4/y]e_5 \Downarrow (t')\, w'$, we have $(s_2)\, (\lambda y.\, e_5)[\overline{v}_2/\overline{x}_2]e_4 \Downarrow$ $(t)\, w$ and $(s_2)\, (\lambda y.\, e'_5)[\overline{v}'_2/\overline{x}_2]e_4 \Downarrow (t')\, w'$ by (E-App). Then, since $(s_2)\, \lambda y.\, e_5 \sim_{\mathcal{R}_2} (s'_2)\, \lambda y.\, e'_5$ and $(s_2)\, \overline{v}_2 \sim_{\mathcal{R}_2} (s'_2)\, \overline{v}'_2$ and since $\mathit{Seals}(e_4) = \emptyset$, we have $(t, t', \mathcal{R}_2 \cup \{(w, w')\}) \in \sim$ by Condition 7 of the definition of bisimulation. Thus, by the definition of $\cong$, we have $(t)\, [w/z]z \cong_{\mathcal{R}_2 \cup \{(w,w')\}} (t')$ $[w'/z]z$. That is, $(t)\, w \cong_{\mathcal{R}} (t')\, w'$ for $\mathcal{R} = \mathcal{R}_2 \cup \{(w, w')\} \supseteq \mathcal{R}_2 \supseteq \mathcal{R}_1 \supseteq \mathcal{R}_0$, as required.

Case (E-Unseal-Succ). Then e_0 is of the form $\mathtt{let}\ \{y\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4$ and the given evaluation derivations have the forms:

$$\frac{(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\, w_1 \quad (s_1)\, [\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\, w_2 \quad \ldots}{(s_0)\, [\overline{v}_0/\overline{x}_0](\mathtt{let}\ \{y\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4) \Downarrow (t)\, w}$$

$$\frac{(s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\, w'_1 \quad (s'_1)\, [\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\, w'_2 \quad \ldots}{(s'_0)\, [\overline{v}'_0/\overline{x}_0](\mathtt{let}\ \{y\}_{e_1} = e_2\ \mathtt{in}\ e_3\ \mathtt{else}\ e_4) \Downarrow (t')\, w'}$$

By the definition of $\cong$, we have $(s_0)\, [\overline{v}_0/\overline{x}_0]e_1 \cong_{\mathcal{R}_0} (s'_0)\, [\overline{v}'_0/\overline{x}_0]e_1$. Thus, by the induction hypothesis, we have $(s_1)\, w_1 \cong_{\mathcal{R}_1} (s'_1)\, w'_1$ for some $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Then, by the definition of $\cong$, we have

$w_1 = [\overline{v}_1/\overline{x}_1]e_5$ and $w_1' = [\overline{v}_1'/\overline{x}_1]e_5$ for some $(s_1)\,\overline{v}_1 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_1'$ and $\mathit{Seals}(e_5) = \emptyset$. Since (s_0) $\overline{v}_0 \sim_{\mathcal{R}_0} (s_0')\,\overline{v}_0'$ and $\mathcal{R}_0 \subseteq \mathcal{R}_1$, we have $(s_1)\,\overline{v}_0 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_0'$ by Lemma 3.18.

Now, again by the definition of $\cong$, we have $(s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \cong_{\mathcal{R}_1} (s_1')\,[\overline{v}_0'/\overline{x}_0]e_2$. Thus, by the induction hypothesis, we have $(s_2)\,w_2 \cong_{\mathcal{R}_2} (s_2')\,w_2'$ for some $\mathcal{R}_2 \supseteq \mathcal{R}_1$. Then, by the definition of $\cong$, we have $w_2 = [\overline{v}_2/\overline{x}_2]e_6$ and $w_2' = [\overline{v}_2'/\overline{x}_2]e_6$ for some $(s_2)\,\overline{v}_2 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_2'$ and $\mathit{Seals}(e_6) = \emptyset$. Since $(s_1)\,\overline{v}_0 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_0'$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$, we have $(s_2)\,\overline{v}_0 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_0'$ by Lemma 3.18. Furthermore, since $(s_1)\,\overline{v}_1 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_1'$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$, we have $(s_2)\,\overline{v}_1 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_1'$ by Lemma 3.18.

Since $w_1 = [\overline{v}_1/\overline{x}_1]e_5$ and $w_1' = [\overline{v}_1'/\overline{x}_1]e_5$ must be seals while $w_2 = [\overline{v}_2/\overline{x}_2]e_6$ and $w_2' = [\overline{v}_2'/\overline{x}_2]e_6$ must be values sealed under these seals, there are two possible forms for e_5 and e_6.

Sub-case $e_5 = x_{1i}$ and $e_6 = \{e_7\}_{x_{2j}}$. The evaluation derivation for e is

$$\frac{\begin{array}{c}(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\,v_{1i}\\ (s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\,\{[\overline{v}_2/\overline{x}_2]e_7\}_{v_{2j}}\\ (s_2)\,[\overline{v}_0,\overline{v}_2/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \Downarrow (t)\,w\end{array}}{(s_0)\,[\overline{v}_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t)\,w}$$

where $v_{1i} = v_{2j}$. Since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,v_{1i}'$ and $(s_2)\,v_{2j} \sim_{\mathcal{R}_2} (s_2')\,v_{2j}'$, we have $v_{1i}' = v_{2j}'$ by Condition 5 of bisimulation. Then, the evaluation derivation for e' is:

$$\frac{\begin{array}{c}(s_0')\,[\overline{v}_0'/\overline{x}_0]e_1 \Downarrow (s_1')\,v_{1i}'\\ (s_1')\,[\overline{v}_0'/\overline{x}_0]e_2 \Downarrow (s_2')\,\{[\overline{v}_2'/\overline{x}_2]e_7\}_{v_{2j}'}\\ (s_2')\,[\overline{v}_0',\overline{v}_2'/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \Downarrow (t')\,w'\end{array}}{(s_0')\,[\overline{v}_0'/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t')\,w'}$$

Since $(s_2)\,\overline{v}_0,\overline{v}_2 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_0',\overline{v}_2'$ and $\mathit{Seals}(e_3) = \mathit{Seals}(e_7) = \emptyset$, we have $(s_2)\,[\overline{v}_0,\overline{v}_2/\overline{x}_0,\overline{x}_2][e_7/y]e_3$ $\cong_{\mathcal{R}_2} (s_2')\,[\overline{v}_0',\overline{v}_2'/\overline{x}_0,\overline{x}_2][e_7/y]e_3$ by the definition of $\cong$. Then, by the induction hypothesis, we have $(t)\,w \cong_{\mathcal{R}} (t')\,w'$ for some $\mathcal{R} \supseteq \mathcal{R}_2 \supseteq \mathcal{R}_1 \supseteq \mathcal{R}_0$.

Sub-case $e_5 = x_{1i}$ and $e_6 = x_{2j}$. The evaluation derivation for e is

$$\frac{\begin{array}{c}(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\,v_{1i} \quad (s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\,v_{2j}\\ (s_2)\,[\overline{v}_0,v/\overline{x}_0,y]e_3 \Downarrow (t)\,w\end{array}}{(s_0)\,[\overline{v}_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t)\,w}$$

where $v_{2j} = \{v\}_{v_{1i}}$ for some v. Since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,v_{1i}'$ and $(s_2)\,v_{2j} \sim_{\mathcal{R}_2} (s_2')\,v_{2j}'$, we have $v_{2j}' = \{v'\}_{k'}$ for some $(s_2)\,v \sim_{\mathcal{R}_2} (s_2')\,v'$ and $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,k'$ by Condition 6 of bisimulation. Furthermore, since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,k'$ and $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,v_{1i}'$, we have $k' = v_{1i}'$ by Condition 5 of bisimulation. Then, the evaluation derivation for e' is:

$$\frac{\begin{array}{c}(s_0')\,[\overline{v}_0'/\overline{x}_0]e_1 \Downarrow (s_1')\,v_{1i}' \quad (s_1')\,[\overline{v}_0'/\overline{x}_0]e_2 \Downarrow (s_2')\,v_{2j}'\\ (s_2')\,[\overline{v}_0',v'/\overline{x}_0,y]e_3 \Downarrow (t')\,w'\end{array}}{(s_0')\,[\overline{v}_0'/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t')\,w'}$$

Since $(s_2)\,\overline{v}_0,v \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_0',v'$ and $\mathit{Seals}(e_3) = \emptyset$, we have $(s_2)\,[\overline{v}_0,v/\overline{x}_0,y]e_3 \cong_{\mathcal{R}_2} (s_2')\,[\overline{v}_0',v'/\overline{x}_0,y]e_3$ by the definition of $\cong$. Then, by the induction hypothesis, we have $(t)\,w \cong_{\mathcal{R}} (t')\,w'$ for some $\mathcal{R} \supseteq \mathcal{R}_2 \supseteq \mathcal{R}_1 \supseteq \mathcal{R}_0$.

Case (E-Unseal-Fail). Similar to the case of (E-Unseal-Succ).

B.2 Proof of Lemma 3.23

We assume $(s_0)\,e \Downarrow (t)\,w$ and prove $(s_0')\,e' \Downarrow$ by induction on the derivation of $(s_0)\,e \Downarrow (t)\,w$. The other direction follows by symmetry.

The argument is very similar to the proof of Lemma 3.22, except that we are proving the *existence* of an evaluation derivation for e' by using the given evaluation derivation for e, instead of proving a property of given evaluation derivations for e and e'. We show just the most interesting case: the one for (E-Unseal-Succ).

By the definition of $\cong$, we have $e = [\overline{v}_0/\overline{x}_0]e_0$ and $e' = [\overline{v}_0'/\overline{x}_0]e_0$ for some $(s_0)\,\overline{v}_0 \sim_{\mathcal{R}_0} (s_0')\,\overline{v}_0'$ and $\mathit{Seals}(e_0) = \emptyset$. In the case of (E-Unseal-Succ), e_0 is of the form $\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4$ and the evaluation derivation for e has the following form:

$$\frac{(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\,w_1 \quad (s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\,w_2 \quad \ldots}{(s_0)\,[\overline{v}_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t)\,w}$$

We aim to derive an evaluation of e' of a similar form:

$$\frac{(s_0')\,[\overline{v}_0'/\overline{x}_0]e_1 \Downarrow (s_1')\,w_1' \quad (s_1')\,[\overline{v}_0'/\overline{x}_0]e_2 \Downarrow (s_2')\,w_2' \quad \ldots}{(s_0')\,[\overline{v}_0'/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t')\,w'}$$

By the definition of $\cong$, we have $(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \cong_{\mathcal{R}_0} (s_0')\,[\overline{v}_0'/\overline{x}_0]e_1$. Since $(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow$, we have $(s_0')\,[\overline{v}_0'/\overline{x}_0]e_1 \Downarrow (s_1')\,w_1'$ for some s_1' and w_1' by the induction hypothesis. Furthermore, by Lemma 3.22, we have $(s_1)\,w_1 \cong_{\mathcal{R}_1} (s_1')\,w_1'$ for some $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Then, by the definition of $\cong$, we have $w_1 = [\overline{v}_1/\overline{x}_1]e_5$ and $w_1' = [\overline{v}_1'/\overline{x}_1]e_5$ for some $(s_1)\,\overline{v}_1 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_1'$ and $\mathit{Seals}(e_5) = \emptyset$. Since $(s_0)\,\overline{v}_0 \sim_{\mathcal{R}_0} (s_0')\,\overline{v}_0'$ and $\mathcal{R}_0 \subseteq \mathcal{R}_1$, we have $(s_1)\,\overline{v}_0 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_0'$ by Lemma 3.18.

Now, again by the definition of $\cong$, we have $(s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \cong_{\mathcal{R}_1} (s_1')\,[\overline{v}_0'/\overline{x}_0]e_2$. Since $(s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow$, we have $(s_1')\,[\overline{v}_0'/\overline{x}_0]e_2 \Downarrow (s_2')\,w_2'$ for some s_2' and w_2' by the induction hypothesis. Furthermore, by Lemma 3.22, we have $(s_2)\,w_2 \cong_{\mathcal{R}_2} (s_2')\,w_2'$ for some $\mathcal{R}_2 \supseteq \mathcal{R}_1$. Then, by the definition of $\cong$, we have $w_2 = [\overline{v}_2/\overline{x}_2]e_6$ and $w_2' = [\overline{v}_2'/\overline{x}_2]e_6$ for some $(s_2)\,\overline{v}_2 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_2'$ and $\mathit{Seals}(e_6) = \emptyset$. Since $(s_1)\,\overline{v}_0 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_0'$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$, we have $(s_2)\,\overline{v}_0 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_0'$ by Lemma 3.18. Furthermore, since $(s_1)\,\overline{v}_1 \sim_{\mathcal{R}_1} (s_1')\,\overline{v}_1'$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$, we have $(s_2)\,\overline{v}_1 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_1'$ by Lemma 3.18.

Since $w_1 = [\overline{v}_1/\overline{x}_1]e_5$ must be a seal while $w_2 = [\overline{v}_2/\overline{x}_2]e_6$ must be a value sealed under this seal, there are two possible forms for e_5 and e_6.

Sub-case $e_5 = x_{1i}$ and $e_6 = \{e_7\}_{x_{2j}}$. The evaluation derivation for e is

$$\frac{\begin{array}{c}(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\,v_{1i} \\ (s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\,\{[\overline{v}_2/\overline{x}_2]e_7\}_{v_{2j}} \\ (s_2)\,[\overline{v}_0,\overline{v}_2/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \Downarrow (t)\,w\end{array}}{(s_0)\,[\overline{v}_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t)\,w}$$

where $v_{1i} = v_{2j}$. Since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s_2')\,v_{1i}'$ and $(s_2)\,v_{2j} \sim_{\mathcal{R}_2} (s_2')\,v_{2j}'$ where v_{1i} and v_{2j} are seals, v_{1i}' and v_{2j}' are also seals by Condition 2 of bisimulation. Furthermore, $v_{1i}' = v_{2j}'$ by Condition 5 of bisimulation.

Meanwhile, since $(s_2)\,\overline{v}_0,\overline{v}_2 \sim_{\mathcal{R}_2} (s_2')\,\overline{v}_0',\overline{v}_2'$ and $\mathit{Seals}(e_3) = \mathit{Seals}(e_7) = \emptyset$, we have $(s_2)\,[\overline{v}_0,\overline{v}_2/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \cong_{\mathcal{R}_2} (s_2')\,[\overline{v}_0',\overline{v}_2'/\overline{x}_0,\overline{x}_2][e_7/y]e_3$ by the definition of $\cong$. Then, since $(s_2)\,[\overline{v}_0,\overline{v}_2/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \Downarrow$, we have $(s_2')\,[\overline{v}_0',\overline{v}_2'/\overline{x}_0,\overline{x}_2][e_7/y]e_3 \Downarrow (t')\,w'$ for some t' and w' by the induction hypothesis.

Therefore, since $v'_{1i} = v'_{2j}$, we can derive an evaluation of e' as follows:

$$\frac{\begin{array}{c}(s'_0)\,[\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\,v'_{1i} \\ (s'_1)\,[\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\,\{[\overline{v}'_2/\overline{x}_2]e_7\}_{v'_{2j}} \\ (s'_2)\,[\overline{v}'_0, \overline{v}'_2/\overline{x}_0, \overline{x}_2][e_7/y]e_3 \Downarrow (t')\,w'\end{array}}{(s'_0)\,[\overline{v}'_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t')\,w'}$$

Sub-case $e_5 = x_{1i}$ **and** $e_6 = x_{2j}$**.** The evaluation derivation for e is

$$\frac{\begin{array}{c}(s_0)\,[\overline{v}_0/\overline{x}_0]e_1 \Downarrow (s_1)\,v_{1i} \quad (s_1)\,[\overline{v}_0/\overline{x}_0]e_2 \Downarrow (s_2)\,v_{2j} \\ (s_2)\,[\overline{v}_0, v/\overline{x}_0, y]e_3 \Downarrow (t)\,w\end{array}}{(s_0)\,[\overline{v}_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t)\,w}$$

where $v_{2j} = \{v\}_{v_{1i}}$ for some v. Since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s'_2)\,v'_{1i}$ and $(s_2)\,v_{2j} \sim_{\mathcal{R}_2} (s'_2)\,v'_{2j}$ where v_{1i} is a seal and v_{2j} is a sealed value, v'_{1i} is also a seal and v'_{2j} is also a sealed value by Condition 2 of bisimulation. Furthermore, we have $v'_{2j} = \{v'\}_{k'}$ for some $(s_2)\,v \sim_{\mathcal{R}_2} (s'_2)\,v'$ and $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s'_2)$ k' by Condition 6 of bisimulation. Moreover, since $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s'_2)\,k'$ and $(s_2)\,v_{1i} \sim_{\mathcal{R}_2} (s'_2)\,v'_{1i}$, we have $k' = v'_{1i}$ by Condition 5 of bisimulation.

Meanwhile, since $(s_2)\,\overline{v}_0, v \sim_{\mathcal{R}_2} (s'_2)\,\overline{v}'_0, v'$ and $\mathit{Seals}(e_3) = \emptyset$, we have $(s_2)\,[\overline{v}_0, v/\overline{x}_0, y]e_3 \cong_{\mathcal{R}_2} (s'_2)\,[\overline{v}'_0, v'/\overline{x}_0, y]e_3$ by the definition of $\cong$. Then, since $(s_2)\,[\overline{v}_0, v/\overline{x}_0, y]e_3 \Downarrow$, we have $(s'_2)\,[\overline{v}'_0, v'/\overline{x}_0, y]$ $e_3 \Downarrow (t')\,w'$ for some t' and w' by the induction hypothesis.

Therefore, since $v'_{2j} = \{v'\}_{v'_{1i}}$, we can derive an evaluation of e'

$$\frac{\begin{array}{c}(s'_0)\,[\overline{v}'_0/\overline{x}_0]e_1 \Downarrow (s'_1)\,v'_{1i} \quad (s'_1)\,[\overline{v}'_0/\overline{x}_0]e_2 \Downarrow (s'_2)\,v'_{2j} \\ (s'_2)\,[\overline{v}'_0, v'/\overline{x}_0, y]e_3 \Downarrow (t')\,w'\end{array}}{(s'_0)\,[\overline{v}'_0/\overline{x}_0](\texttt{let}\ \{y\}_{e_1} = e_2\ \texttt{in}\ e_3\ \texttt{else}\ e_4) \Downarrow (t')\,w'}$$

as required.

Appendix C

Proofs for Chapter 4

C.1 Proof of Lemma 4.12

Since $\sim$ is the greatest bisimulation, it suffices to check that $\equiv$ is a bisimulation by checking each condition of bisimulation. Take any $(\Delta, \mathcal{R}) \in \equiv$ with $\Delta = \{(\overline{\alpha}, \overline{\sigma}, \overline{\sigma}')\}$. Then, from the definition of $\equiv$, we have:

A$_0$. $\vdash V_0 : [\overline{\sigma}/\overline{\alpha}]\tau_0$ and $\vdash V_0' : [\overline{\sigma}'/\overline{\alpha}]\tau_0$ for any $(V_0, V_0', \tau_0) \in \mathcal{R}$, and

B$_0$. $[\overline{V}_0/\overline{x}_0][\overline{\sigma}/\overline{\alpha}]C_0 \Downarrow \iff [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'/\overline{\alpha}]C_0 \Downarrow$ for any $(\overline{V}_0, \overline{V}'_0, \overline{\tau}_0) \in \mathcal{R}$ and for any $\overline{\alpha}, \overline{x}_0 : \overline{\tau}_0 \vdash C_0 : \tau_0$.

We now check the conditions in the definition of bisimulation.

Condition 1: Immediate, since it is just the same as A$_0$.

Condition 2: Suppose that

$$((\mathtt{fix}\, f(x:\pi):\rho = M), (\mathtt{fix}\, f(x:\pi'):\rho' = M'), \tau \to \sigma) \quad \in \quad \mathcal{R}$$

and that $V = [\overline{U}/\overline{y}][\overline{\sigma}/\overline{\alpha}]D$ and $V' = [\overline{U}'/\overline{y}][\overline{\sigma}'/\overline{\alpha}]D$ with $(\overline{U}, \overline{U}', \overline{\rho}) \in \mathcal{R}$ and $\overline{\alpha}, \overline{y} : \overline{\rho} \vdash D : \tau$. Then,

$$(\mathtt{fix}\, f(x:\pi):\rho = M)V \Downarrow \iff (\mathtt{fix}\, f(x:\pi'):\rho' = M')V' \Downarrow$$

follows from B$_0$ by taking (for fresh g):

$$\begin{array}{rcl} C_0 & = & gD \\ \overline{x}_0 & = & g, \overline{y} \\ \overline{\tau}_0 & = & \tau \to \sigma, \overline{\rho} \\ \overline{V}_0 & = & (\mathtt{fix}\, f(x:\pi):\rho = M), \overline{U} \\ \overline{V}'_0 & = & (\mathtt{fix}\, f(x:\pi'):\rho' = M'), \overline{U}' \end{array}$$

Suppose furthermore that $(\mathtt{fix}\, f(x:\pi):\rho = M)V \Downarrow W$ and $(\mathtt{fix}\, f(x:\pi'):\rho' = M')V' \Downarrow W'$. Then

$$(\Delta, \mathcal{R} \cup \{(W, W', \sigma)\}) \quad \in \quad \equiv$$

will follow from the definition of $\equiv$ if we can prove:

A$_2$. $\vdash V_2 : [\overline{\sigma}/\overline{\alpha}]\tau_2$ and $\vdash V_2' : [\overline{\sigma}'/\overline{\alpha}]\tau_2$ for any $(V_2, V_2', \tau_2) \in \mathcal{R} \cup \{(W, W', \sigma)\}$, and

B$_2$. $[\overline{V}_2/\overline{x}_2][\overline{\sigma}/\overline{\alpha}]C_2 \Downarrow \iff [\overline{V}'_2/\overline{x}_2][\overline{\sigma}'/\alpha]C_2 \Downarrow$ for any $(\overline{V}_2, \overline{V}'_2, \overline{\tau}_2) \in \mathcal{R} \cup \{(W, W', \sigma)\}$ and for any $\overline{\alpha}, \overline{x}_2 : \overline{\tau}_2 \vdash C_2 : \tau_2$.

But A$_2$ follows from A$_0$ in the case where (V_2, V_2', τ_2) is drawn from $\mathcal{R}$ and from type preservation in the case where $(V_2, V_2', \tau_2) = (W, W', \sigma)$. B$_2$ holds as follows: without loss of generality, let $(V_{2_1}, V'_{2_1}, \tau_{2_1}) = (W, W', \sigma)$ and $(V_{2_i}, V'_{2_i}, \tau_{2_i}) \in \mathcal{R}$ for $2 \leq i \leq n$; then, it suffices to take in B$_0$ (for fresh g)

$$\begin{array}{rcl}
C_0 & = & \mathtt{let}\; x_{2_1} = gD \;\mathtt{in}\; C_2 \\
\overline{x}_0 & = & g, \overline{y}, x_{2_2}, \ldots, x_{2_n} \\
\overline{\tau}_0 & = & \tau \to \sigma, \overline{\rho}, \tau_{2_2}, \ldots, \tau_{2_n} \\
\overline{V}_0 & = & (\mathtt{fix}\; f(x : \pi) : \rho = M), \overline{U}, V_{2_2}, \ldots, V_{2_n} \\
\overline{V}'_0 & = & (\mathtt{fix}\; f(x : \pi') : \rho' = M'), \overline{U}', V'_{2_2}, \ldots, V'_{2_n}
\end{array}$$

so that the evaluations of $[\overline{V}_0/\overline{x}_0][\overline{\sigma}/\overline{\alpha}]C_0$ and $[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'/\overline{\alpha}]C_0$ amount to the evaluations of $[\overline{V}_2/\overline{x}_2]$ $[\overline{\sigma}/\overline{\alpha}]C_2$ and $[\overline{V}'_2/\overline{x}_2][\overline{\sigma}'/\overline{\alpha}]C_2$ as below.

$$\frac{\begin{array}{c}[(\mathtt{fix}\; f(x : \pi) : \rho = M), \overline{U}/g, \overline{y}][\overline{\sigma}/\overline{\alpha}](gD) \Downarrow W \\ [W, V_{2_2}, \ldots, V_{2_n}/x_{2_1}, x_{2_2}, \ldots, x_{2_n}][\overline{\sigma}/\overline{\alpha}]C_2 \Downarrow\end{array}}{[(\mathtt{fix}\; f(x : \pi) : \rho = M), \overline{U}, V_{2_2}, \ldots, V_{2_n}/g, \overline{y}, x_{2_2}, \ldots, x_{2_n}][\overline{\sigma}/\overline{\alpha}](\mathtt{let}\; x_{2_1} = gD \;\mathtt{in}\; C_2) \Downarrow}$$

$$\frac{\begin{array}{c}[(\mathtt{fix}\; f(x : \pi') : \rho' = M'), \overline{U}'/g, \overline{y}][\overline{\sigma}'/\overline{\alpha}](gD) \Downarrow W' \\ [W', V'_{2_2}, \ldots, V'_{2_n}/x_{2_1}, x_{2_2}, \ldots, x_{2_n}][\overline{\sigma}'/\overline{\alpha}]C_2 \Downarrow\end{array}}{[(\mathtt{fix}\; f(x : \pi') : \rho' = M'), \overline{U}', V'_{2_2}, \ldots, V'_{2_n}/g, \overline{y}, x_{2_2}, \ldots, x_{2_n}][\overline{\sigma}'/\overline{\alpha}](\mathtt{let}\; x_{2_1} = gD \;\mathtt{in}\; C_2) \Downarrow}$$

Condition 3: Suppose that

$$(\Lambda\alpha.\, M, \Lambda\alpha.\, M', \forall\alpha.\, \tau) \quad \in \quad \mathcal{R}$$

and that $FTV(\rho) \subseteq Dom(\Delta)$. Then

$$(\Lambda\alpha.\, M)[[\overline{\sigma}/\overline{\alpha}]\rho] \Downarrow \iff (\Lambda\alpha.\, M')[[\overline{\sigma}'/\overline{\alpha}]\rho] \Downarrow$$

follows from B$_0$ by taking (for fresh g):

$$\begin{array}{rcl}
C_0 & = & g[\rho] \\
\overline{x}_0 & = & g \\
\overline{\tau}_0 & = & \forall\alpha.\, \tau \\
\overline{V}_0 & = & \Lambda\alpha.\, M \\
\overline{V}'_0 & = & \Lambda\alpha.\, M'
\end{array}$$

Suppose furthermore that $(\Lambda\alpha.\, M)[[\overline{\sigma}/\overline{\alpha}]\rho] \Downarrow W$ and $(\Lambda\alpha.\, M')[[\overline{\sigma}'/\overline{\alpha}]\rho] \Downarrow W'$. Then

$$(\Delta, \mathcal{R} \cup \{(W, W', [\rho/\alpha]\tau)\}) \quad \in \quad \equiv$$

will follow from the definition of $\equiv$ if we can prove:

A$_3$. $\vdash V_3 : [\overline{\sigma}/\overline{\alpha}]\tau_3$ and $\vdash V_3' : [\overline{\sigma}'/\overline{\alpha}]\tau_3$ for any $(V_3, V_3', \tau_3) \in \mathcal{R} \cup \{(W, W', [\rho/\alpha]\tau)\}$, and

B$_3$. $[\overline{V}_3/\overline{x}_3][\overline{\sigma}/\overline{\alpha}]C_3 \Downarrow \iff [\overline{V}_3'/\overline{x}_3][\overline{\sigma}'/\overline{\alpha}]C_3 \Downarrow$ for any $(\overline{V}_3, \overline{V}_3', \overline{\tau}_3) \in \mathcal{R} \cup \{(W, W', [\rho/\alpha]\tau)\}$ and for any $\overline{\alpha}, \overline{x}_3 : \overline{\tau}_3 \vdash C_3 : \tau_3$.

But A$_3$ follows from A$_0$ in the case where (V_3, V_3', τ_3) is drawn from $\mathcal{R}$ and from type preservation in the case where $(V_3, V_3', \tau_3) = (W, W', [\rho/\alpha]\tau)$. B$_3$ holds as follows: without loss of generality, let $(V_{3_1}, V_{3_1}', \tau_{3_1}) = (W, W', [\rho/\alpha]\tau)$ and $(V_{3_i}, V_{3_i}', \tau_{3_i}) \in \mathcal{R}$ for $2 \leq i \leq n$; then, it suffices to take in B$_0$ (for fresh g)

$$\begin{array}{rcl} C_0 & = & \texttt{let}\ x_{3_1} = g[\rho]\ \texttt{in}\ C_3 \\ \overline{x}_0 & = & g, x_{3_2}, \ldots, x_{3_n} \\ \overline{\tau}_0 & = & \forall\alpha.\,\tau, \tau_{3_2}, \ldots, \tau_{3_n} \\ \overline{V}_0 & = & \Lambda\alpha.\,M, V_{3_2}, \ldots, V_{3_n} \\ \overline{V}_0' & = & \Lambda\alpha.\,M', V_{3_2}', \ldots, V_{3_n}' \end{array}$$

so that the evaluations of $[\overline{V}_0/\overline{x}_0][\overline{\sigma}/\overline{\alpha}]C_0$ and $[\overline{V}_0'/\overline{x}_0][\overline{\sigma}'/\overline{\alpha}]C_0$ amount to the evaluations of $[\overline{V}_3/\overline{x}_3]$ $[\overline{\sigma}/\overline{\alpha}]C_3$ and $[\overline{V}_3'/\overline{x}_3][\overline{\sigma}'/\overline{\alpha}]C_3$ as below.

$$\frac{\begin{array}{c}[\Lambda\alpha.\,M/g][\overline{\sigma}/\overline{\alpha}](g[\rho]) \Downarrow W \\ [W, V_{3_2}, \ldots, V_{3_n}/x_{3_1}, x_{3_2}, \ldots, x_{3_n}][\overline{\sigma}/\overline{\alpha}]C_3 \Downarrow\end{array}}{[\Lambda\alpha.\,M, V_{3_2}, \ldots, V_{3_n}/g, x_{3_2}, \ldots, x_{3_n}][\overline{\sigma}/\overline{\alpha}](\texttt{let}\ x_{3_1} = g[\rho]\ \texttt{in}\ C_3) \Downarrow}$$

$$\frac{\begin{array}{c}[\Lambda\alpha.\,M'/g][\overline{\sigma}'/\overline{\alpha}](g[\rho]) \Downarrow W' \\ [W', V_{3_2}', \ldots, V_{3_n}'/x_{3_1}, x_{3_2}, \ldots, x_{3_n}][\overline{\sigma}'/\overline{\alpha}]C_3 \Downarrow\end{array}}{[\Lambda\alpha.\,M', V_{3_2}', \ldots, V_{3_n}'/g, x_{3_2}, \ldots, x_{3_n}][\overline{\sigma}'/\overline{\alpha}](\texttt{let}\ x_{3_1} = g[\rho]\ \texttt{in}\ C_3) \Downarrow}$$

Condition 4: Suppose that

$$((\texttt{pack}\ \sigma, V\ \texttt{as}\ \exists\alpha.\,\tau), (\texttt{pack}\ \sigma', V'\ \texttt{as}\ \exists\alpha.\,\tau'), \exists\alpha.\,\tau'') \quad \in \quad \mathcal{R}.$$

Then,

$$(\Delta \uplus \{(\alpha, \sigma, \sigma')\}, \mathcal{R} \cup \{(V, V', \tau'')\}) \quad \in \quad \equiv$$

follows from the definition of $\equiv$ if we prove:

A$_4$. $\vdash V_4 : [\overline{\sigma}, \sigma/\overline{\alpha}, \alpha]\tau_4$ and $\vdash V_4' : [\overline{\sigma}', \sigma'/\overline{\alpha}, \alpha]\tau_4$ for any $(V_4, V_4', \tau_4) \in \mathcal{R} \cup \{(V, V', \tau'')\}$, and

B$_4$. $[\overline{V}_4/\overline{x}_4][\overline{\sigma}, \sigma/\overline{\alpha}, \alpha]C_4 \Downarrow \iff [\overline{V}_4'/\overline{x}_4][\overline{\sigma}', \sigma/\overline{\alpha}, \alpha]C_4 \Downarrow$ for any $(\overline{V}_4, \overline{V}_4', \overline{\tau}_4) \in \mathcal{R} \cup \{(V, V', \tau'')\}$ and for any $\overline{\alpha}, \alpha, \overline{x}_4 : \overline{\tau}_4 \vdash C_4 : \tau_4$.

But A$_4$ follows from A$_0$ in the case where (V_4, V_4', τ_4) is drawn from $\mathcal{R}$ and, in the case where $(V_4, V_4', \tau_4) = (V, V', \tau'')$, by inversion of (T-Pack) with

$$\vdash \quad \texttt{pack}\ \sigma, V\ \texttt{as}\ \exists\alpha.\,\tau \quad : \quad [\overline{\sigma}/\overline{\alpha}](\exists\alpha.\,\tau'')$$

and

$$\vdash \quad \texttt{pack}\ \sigma', V'\ \texttt{as}\ \exists\alpha.\,\tau' \quad : \quad [\overline{\sigma}'/\overline{\alpha}](\exists\alpha.\,\tau''),$$

which follow from A_0 with

$$((\mathtt{pack}\,\sigma, V \;\mathtt{as}\; \exists\alpha.\,\tau), (\mathtt{pack}\,\sigma', V' \;\mathtt{as}\; \exists\alpha.\,\tau'), \exists\alpha.\,\tau'') \quad \in \quad \mathcal{R}.$$

B_4 holds as follows: without loss of generality, let $(V_{4_1}, V'_{4_1}, \tau_{4_1}) = (V, V', \tau'')$ and $(V_{4_i}, V'_{4_i}, \tau_{4_i}) \in \mathcal{R}$ for $2 \le i \le n$; then, it suffices to take in B_0 (for fresh p)

$$\begin{array}{rcl}
C_0 & = & \mathtt{open}\,p\;\mathtt{as}\;\alpha, x_{4_1}\;\mathtt{in}\;C_4 \\
\overline{x}_0 & = & p, x_{4_2}, \ldots, x_{4_n} \\
\overline{\tau}_0 & = & \exists\alpha.\,\tau'', \tau_{4_2}, \ldots, \tau_{4_n} \\
\overline{V}_0 & = & \mathtt{pack}\,\sigma, V\;\mathtt{as}\;\exists\alpha.\,\tau, V_{4_2}, \ldots, V_{4_n} \\
\overline{V}'_0 & = & \mathtt{pack}\,\sigma', V'\;\mathtt{as}\;\exists\alpha.\,\tau', V'_{4_2}, \ldots, V'_{4_n}
\end{array}$$

so that the evaluations of $[\overline{V}_0/\overline{x}_0][\overline{\sigma}/\overline{\alpha}]C_0$ and $[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'/\overline{\alpha}]C_0$ amount to the evaluations of $[\overline{V}_4/\overline{x}_4]$ $[\overline{\sigma}, \sigma/\overline{\alpha}, \alpha]C_4$ and $[\overline{V}'_4/\overline{x}_4][\overline{\sigma}', \sigma'/\overline{\alpha}, \alpha]C_4$ as below.

$$\frac{[V, V_{4_2}, \ldots, V_{4_n}/x_{4_1}, x_{4_2}, \ldots, x_{4_n}][\overline{\sigma}, \sigma/\overline{\alpha}, \alpha]C_4 \Downarrow}{[\mathtt{pack}\,\sigma, V\;\mathtt{as}\;\exists\alpha.\,\tau, V_{4_2}, \ldots, V_{4_n}/p, x_{4_2}, \ldots, x_{4_n}][\overline{\sigma}/\overline{\alpha}](\mathtt{open}\,p\;\mathtt{as}\;\alpha, x_{4_1}\;\mathtt{in}\;C_4) \Downarrow}$$

$$\frac{[V', V'_{4_2}, \ldots, V'_{4_n}/x_{4_1}, x_{4_2}, \ldots, x_{4_n}][\overline{\sigma}', \sigma'/\overline{\alpha}, \alpha]C_4 \Downarrow}{[\mathtt{pack}\,\sigma', V'\;\mathtt{as}\;\exists\alpha.\,\tau, V'_{4_2}, \ldots, V'_{4_n}/p, x_{4_2}, \ldots, x_{4_n}][\overline{\sigma}'/\overline{\alpha}](\mathtt{open}\,p\;\mathtt{as}\;\alpha, x_{4_1}\;\mathtt{in}\;C_4) \Downarrow}$$

Proofs of the other conditions are similar. □

C.2 Proof of Lemma 4.14

By induction on the derivation of $N \Downarrow W$.

By the definition of $\sim^\circ$, we have

$$N \quad = \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_0$$

and

$$N' \quad = \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_0$$

for some

$$\Delta_0 \;\vdash\; \overline{V}_0 \;\sim_{\mathcal{R}_0}\; \overline{V}'_0 \;:\; \overline{\tau}_0$$

and

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \;\vdash\; M_0 \;:\; \tau$$

with $\Delta_0 = \{(\overline{\alpha}_0, \overline{\sigma}_0, \overline{\sigma}'_0)\}$. If N is a value, then N' is also a value (easy case analysis on the syntax of M_0) and the result is immediate, because every value evaluates only to itself. We consider the remaining possibilities—where M_0 is neither a value nor a variable—in detail; there is one case for each of the non-value evaluation rules.

Case (E-Open). Then M_0 has the form

$$M_0 \quad = \quad \texttt{open}\ M_1\ \texttt{as}\ \alpha, y\ \texttt{in}\ M_2$$

and the given evaluation derivations have the forms:

$$\frac{\begin{array}{c}[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \Downarrow \texttt{pack}\ \rho_1, W_1\ \texttt{as}\ \exists\alpha.\,\rho_2 \\ [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \Downarrow W\end{array}}{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0](\texttt{open}\ M_1\ \texttt{as}\ \alpha, y\ \texttt{in}\ M_2) \Downarrow W}$$

$$\frac{\begin{array}{c}[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \Downarrow \texttt{pack}\ \rho'_1, W'_1\ \texttt{as}\ \exists\alpha.\,\rho'_2 \\ [W'_1/y][\rho'_1/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2 \Downarrow W'\end{array}}{[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0](\texttt{open}\ M_1\ \texttt{as}\ \alpha, y\ \texttt{in}\ M_2) \Downarrow W'}$$

By inversion of (T-Open), we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \quad \vdash \quad M_1 \quad : \quad \exists\alpha.\,\rho''_2$$

and

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0, \alpha, y : \rho''_2 \quad \vdash \quad M_2 \quad : \quad \tau$$

with $\alpha \notin FTV(\tau)$. Thus, by the definition of $\sim^\circ$, we have

$$\Delta_0 \quad \vdash \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}]M_1 \quad \sim^\circ_{\mathcal{R}_0} \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}]M_1 \quad : \quad \exists\alpha.\,\rho''_2.$$

Therefore, by the induction hypothesis, we have

$$\Delta_1 \quad \vdash \quad \texttt{pack}\ \rho_1, W_1\ \texttt{as}\ \exists\alpha.\,\rho_2 \quad \sim^\circ_{\mathcal{R}_1} \quad \texttt{pack}\ \rho'_1, W'_1\ \texttt{as}\ \exists\alpha.\,\rho'_2 \quad : \quad \exists\alpha.\,\rho''_2$$

for some $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Then, by the definition of $\sim^\circ$, we have

$$\texttt{pack}\ \rho_1, W_1\ \texttt{as}\ \exists\alpha.\,\rho_2 \quad = \quad [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_3$$

and

$$\texttt{pack}\ \rho'_1, W'_1\ \texttt{as}\ \exists\alpha.\,\rho'_2 \quad = \quad [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_3$$

for some

$$\Delta_1 \quad \vdash \quad \overline{V}_1 \quad \sim_{\mathcal{R}_1} \quad \overline{V}'_1 \quad : \quad \overline{\tau}_1$$

and

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_3 \quad : \quad \exists\alpha.\,\rho''_2$$

with $\Delta_1 = \{(\overline{\alpha}_1, \overline{\sigma}_1, \overline{\sigma}'_1)\}$.

Sub-case $M_3 = (\texttt{pack}\ \rho''_1, M_4\ \texttt{as}\ \exists\alpha.\,\rho'''_2)$**.** Then

$$\begin{array}{rcl} W_1 & = & [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_4 \\ W'_1 & = & [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\alpha_1]M_4 \end{array}$$

and

$$\begin{array}{rcl} \rho_1 & = & [\overline{\sigma}_1/\overline{\alpha}_1]\rho_1'' \\ \rho_1' & = & [\overline{\sigma}_1'/\overline{\alpha}_1]\rho_1''. \end{array}$$

Since we have

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_4 \quad : \quad [\rho_1''/\alpha]\rho_2''$$

by inversion of (T-Pack), we have

$$\overline{\alpha}_1, \overline{x}_0 : \overline{\tau}_0, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad [M_4/y][\rho_1''/\alpha]M_2 \quad : \quad \tau$$

by weakening and the substitution lemmas for types and terms. Therefore, by the definition of $\sim^\circ$, we have

$$\Delta_1 \quad \vdash \quad [\overline{V}_0, \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \quad \sim^\circ_{\mathcal{R}_1} \quad [\overline{V}_0', \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \quad : \quad \tau.$$

Since

$$\begin{array}{rl} & [\overline{V}_0, \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \\ = & [([\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_4)/y][([\overline{\sigma}_1/\overline{\alpha}_1]\rho_1'')/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \\ = & [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \end{array}$$

and

$$\begin{array}{rl} & [\overline{V}_0', \overline{V}_1'/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \\ = & [([\overline{V}_1'/\overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1]M_4)/y][([\overline{\sigma}_1'/\overline{\alpha}_1]\rho_1'')/\alpha][\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2 \\ = & [W_1'/y][\rho_1'/\alpha][\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2, \end{array}$$

we have

$$\Delta_2 \quad \vdash \quad W \quad \sim^\circ_{\mathcal{R}_2} \quad W' \quad : \quad \tau$$

for some $\Delta_2 \supseteq \Delta_1$ and $\mathcal{R}_2 \supseteq \mathcal{R}_1$ by the induction hypothesis.

Sub-case $M_3 = x_{1_i}$. Then

$$\begin{array}{rcl} V_{1_i} & = & \texttt{pack}\,\rho_1, W_1\,\texttt{as}\,\exists\alpha.\,\rho_2 \\ V_{1_i}' & = & \texttt{pack}\,\rho_1', W_1'\,\texttt{as}\,\exists\alpha.\,\rho_2'. \end{array}$$

Since

$$\Delta_1 \quad \vdash \quad V_{1_i} \quad \sim_{\mathcal{R}_1} \quad V_{1_i}' \quad : \quad \tau_{1_i},$$

we have the following two possibilities by Condition 4 of bisimulation.

Sub-sub-case $(\beta, \rho_1, \rho_1') \in \Delta_1$ and $(W_1, W_1', [\beta/\alpha]\rho_2'') \in \mathcal{R}_1$. Since we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0, \alpha, y : \rho_2'' \quad \vdash \quad M_2 \quad : \quad \tau$$

with $\alpha \notin FTV(\tau)$, we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0, \beta, y : [\beta/\alpha]\rho_2'' \quad \vdash \quad [\beta/\alpha]M_2 \quad : \quad \tau.$$

Then, we have

$$\Delta_1 \quad \vdash \quad [\overline{V}_0, W_1/\overline{x}_0, y][\overline{\sigma}_1/\overline{\alpha}_1][\beta/\alpha]M_2 \quad \sim^{\circ}_{\mathcal{R}_1} \quad [\overline{V}'_0, W'_1/\overline{x}_0, y][\overline{\sigma}'_1/\overline{\alpha}_1][\beta/\alpha]M_2 \quad : \quad \tau$$

by the definition of $\sim^{\circ}$. Since we have $\beta = \alpha_{1_i}$ with $\rho_1 = \sigma_{1_i}$ and $\rho'_1 = \sigma'_{1_i}$ for some i, we have

$$[\overline{V}_0, W_1/\overline{x}_0, y][\overline{\sigma}_1/\overline{\alpha}_1][\beta/\alpha]M_2 \quad = \quad [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2$$

and

$$[\overline{V}'_0, W'_1/\overline{x}_0, y][\overline{\sigma}'_1/\overline{\alpha}_1][\beta/\alpha]M_2 \quad = \quad [W'_1/y][\rho'_1/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2,$$

so we have

$$\Delta_2 \quad \vdash \quad W \quad \sim^{\circ}_{\mathcal{R}_2} W' \quad : \quad \tau$$

for some $\Delta_2 \supseteq \Delta_1$ and $\mathcal{R}_2 \supseteq \mathcal{R}_1$ by the induction hypothesis.

Sub-sub-case $(\Delta_1 \uplus \{(\alpha, \rho_1, \rho'_1)\}, \mathcal{R}_1 \cup \{(W_1, W'_1, \rho''_2)\}) \in \sim$**.** Then we have

$$\Delta_2 \quad \vdash \quad [\overline{V}_0, W_1/\overline{x}_0, y][\overline{\sigma}_0, \rho_1/\overline{\alpha}_0, \alpha]M_2 \quad \sim^{\circ}_{\mathcal{R}_2} \quad [\overline{V}'_0, W'_1/\overline{x}_0, y][\overline{\sigma}'_0, \rho'_1/\overline{\alpha}_0, \alpha]M_2 \quad : \quad \tau$$

for $\Delta_2 = \Delta_1 \cup \{(\alpha, \rho_1, \rho'_1)\}$ and $\mathcal{R}_2 = \mathcal{R}_1 \cup \{(W_1, W'_1, \rho''_2)\}$ by the definition of $\sim^{\circ}$. Since we have

$$[\overline{V}_0, W_1/\overline{x}_0, y][\overline{\sigma}_0, \rho_1/\overline{\alpha}_0, \alpha]M_2 \quad = \quad [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2$$

and

$$[\overline{V}'_0, W'_1/\overline{x}_0, y][\overline{\sigma}'_0, \rho'_1/\overline{\alpha}_0, \alpha]M_2 \quad = \quad [W'_1/y][\rho'_1/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2,$$

we have

$$\Delta_3 \quad \vdash \quad W \quad \sim^{\circ}_{\mathcal{R}_3} \quad W' \quad : \quad \tau$$

for some $\Delta_3 \supseteq \Delta_2$ and $\mathcal{R}_3 \supseteq \mathcal{R}_2$ by the induction hypothesis.

Case (E-Pack). Then M_0 has the form

$$M_0 \quad = \quad \texttt{pack}\ \rho_1, M_1\ \texttt{as}\ \exists\alpha.\,\rho_2$$

and the given evaluation derivations have the forms

$$\frac{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \Downarrow W_1}{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0](\texttt{pack}\ \rho_1, M_1\ \texttt{as}\ \exists\alpha.\,\rho_2) \Downarrow \texttt{pack}\ [\overline{\sigma}_0/\overline{\alpha}_0]\rho_1, W_1\ \texttt{as}\ [\overline{\sigma}_0/\overline{\alpha}_0](\exists\alpha.\,\rho_2)}$$

$$\frac{[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \Downarrow W'_1}{[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0](\texttt{pack}\ \rho_1, M_1\ \texttt{as}\ \exists\alpha.\,\rho_2) \Downarrow \texttt{pack}\ [\overline{\sigma}'_0/\overline{\alpha}_0]\rho_1, W'_1\ \texttt{as}\ [\overline{\sigma}'_0/\overline{\alpha}_0](\exists\alpha.\,\rho_2)}$$

where

$$W \quad = \quad \texttt{pack}\ [\overline{\sigma}_0/\overline{\alpha}_0]\rho_1, W_1\ \texttt{as}\ [\overline{\sigma}_0/\overline{\alpha}_0](\exists\alpha.\,\rho_2)$$

and

$$W' \quad = \quad \texttt{pack}\ [\overline{\sigma}'_0/\overline{\alpha}_0]\rho_1, W'_1\ \texttt{as}\ [\overline{\sigma}'_0/\overline{\alpha}_0](\exists\alpha.\,\rho_2).$$

By inversion of (T-Pack), we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \quad \vdash \quad M_1 \quad : \quad [\rho_1/\alpha]\rho_2$$

with $\tau = \exists\alpha.\, \rho_2$. Thus, by the definition of $\sim^\circ$, we have

$$\Delta_0 \quad \vdash \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \quad \sim^\circ_{\mathcal{R}_0} \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \quad : \quad [\rho_1/\alpha]\rho_2.$$

Then, by the induction hypothesis, we have

$$\Delta_1 \quad \vdash \quad W_1 \quad \sim^\circ_{\mathcal{R}_1} \quad W'_1 \quad : \quad [\rho_1/\alpha]\rho_2$$

for some $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Therefore, by the definition of $\sim^\circ$, we have

$$\begin{aligned} W_1 &= [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_2 \\ W'_1 &= [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_2 \end{aligned}$$

for some

$$\Delta_1 \quad \vdash \quad \overline{V}_1 \quad \sim_{\mathcal{R}_1} \quad \overline{V}'_1 \quad : \quad \overline{\tau}_1$$

and

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_2 \quad : \quad [\rho_1/\alpha]\rho_2$$

with $\Delta_1 = \{(\overline{\alpha}_1, \overline{\sigma}_1, \overline{\sigma}'_1)\}$. Thus, by (T-Pack), we have

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_3 \quad : \quad \exists\alpha.\, \rho_2$$

for

$$M_3 \quad = \quad \texttt{pack}\ \rho_1, M_2\ \texttt{as}\ \exists\alpha.\, \rho_2.$$

Then, by the definition of $\sim^\circ$, we have

$$\Delta_1 \quad \vdash \quad [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_3 \quad \sim^\circ_{\mathcal{R}_1} \quad [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_3 \quad : \quad \exists\alpha.\, \rho_2,$$

i.e.,

$$\Delta_1 \quad \vdash \quad W \quad \sim^\circ_{\mathcal{R}_1} \quad W' \quad : \quad \tau.$$

Case (E-TApp). Then M_0 has the form

$$M_0 \quad = \quad M_1[\rho_1]$$

and the given evaluation derivations have the forms:

$$\frac{\begin{array}{c}[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \Downarrow \Lambda\alpha.\, M_2 \\ [([\overline{\sigma}_0/\overline{\alpha}_0]\rho_1)/\alpha]M_2 \Downarrow W\end{array}}{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0](M_1[\rho_1]) \Downarrow W}$$

$$\frac{\begin{array}{c}[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \Downarrow \Lambda\alpha.\, M'_2 \\ [([\overline{\sigma}'_0/\overline{\alpha}_0]\rho_1)/\alpha]M'_2 \Downarrow W'\end{array}}{[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0](M_1[\rho_1]) \Downarrow W'}$$

By inversion of (T-TApp), we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \quad \vdash \quad M_1 \quad : \quad \forall \alpha.\, \rho_2$$

with $\tau = [\rho_1/\alpha]\rho_2$. Thus, by the definition of $\sim^\circ$, we have

$$\Delta_0 \quad \vdash \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \quad \sim^\circ_{\mathcal{R}_0} \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \quad : \quad \forall \alpha.\, \rho_2.$$

Then, by the induction hypothesis, we have

$$\Delta_1 \quad \vdash \quad \Lambda\alpha.\, M_2 \quad \sim^\circ_{\mathcal{R}_1} \quad \Lambda\alpha.\, M'_2 \quad : \quad \forall \alpha.\, \rho_2$$

for some $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Therefore, by the definition of $\sim^\circ$, we have

$$\begin{array}{rcl} \Lambda\alpha.\, M_2 & = & [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_3 \\ \Lambda\alpha.\, M'_2 & = & [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_3 \end{array}$$

for some

$$\Delta_1 \quad \vdash \quad \overline{V}_1 \quad \sim_{\mathcal{R}_1} \quad \overline{V}'_1 \quad : \quad \overline{\tau}_1$$

and

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_3 \quad : \quad \forall \alpha.\, \rho_2$$

with $\Delta_1 = \{(\overline{\alpha}_1, \overline{\sigma}_1, \overline{\sigma}'_1)\}$.

Sub-case $M_3 = \Lambda\alpha.\, M''_2$. Then, by inversion of (T-TAbs), we have

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1, \alpha \quad \vdash \quad M''_2 \quad : \quad \rho_2.$$

By the substitution lemma for types, we have

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad [\rho_1/\alpha]M''_2 \quad : \quad [\rho_1/\alpha]\rho_2.$$

By the definition of $\sim^\circ$, we have

$$\Delta_1 \quad \vdash \quad [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1][\rho_1/\alpha]M''_2 \quad \sim^\circ_{\mathcal{R}_1} \quad [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1][\rho_1/\alpha]M''_2 \quad : \quad [\rho_1/\alpha]\rho_2,$$

i.e.,

$$\Delta_1 \quad \vdash \quad [(([\overline{\sigma}_0/\overline{\alpha}_0]\rho_1)/\alpha]M_2 \quad \sim^\circ_{\mathcal{R}_1} \quad [(([\overline{\sigma}'_0/\overline{\alpha}_0]\rho_1)/\alpha]M'_2 \quad : \quad \tau.$$

Again by the induction hypothesis, we have

$$\Delta_2 \quad \vdash \quad W \quad \sim^\circ_{\mathcal{R}_2} \quad W' \quad : \quad \tau$$

for some $\Delta_2 \supseteq \Delta_1$ and $\mathcal{R}_2 \supseteq \mathcal{R}_1$.

Sub-case $M_3 = x_{1_i}$. Then

$$\begin{array}{rcl} V_{1_i} & = & \Lambda\alpha.\, M_2 \\ V'_{1_i} & = & \Lambda\alpha.\, M'_2 \end{array}$$

and $\tau_{1_i} = \forall\alpha.\,\rho_2$. Since we have

$$(\Lambda\alpha.\,M_2)[[\overline{\sigma}_1/\overline{\alpha}_1]\rho_1] \quad \Downarrow \quad W$$

and

$$(\Lambda\alpha.\,M_2')[[\overline{\sigma}_1'/\overline{\alpha}_1]\rho_1] \quad \Downarrow \quad W'$$

with

$$\Delta_1 \quad \vdash \quad \Lambda\alpha.\,M_2 \quad \sim_{\mathcal{R}_1} \quad \Lambda\alpha.\,M_2' \quad : \quad \forall\alpha.\,\rho_2,$$

we have

$$(\Delta_1, \mathcal{R}_1 \cup \{(W, W', \tau)\}) \quad \in \quad \sim$$

by Condition 3 of bisimulation. (Recall $\tau = [\rho_1/\alpha]\rho_2$.) Thus, we have

$$\Delta_1 \quad \vdash \quad W \quad \sim^{\circ}_{\mathcal{R}_2} \quad W' \quad : \quad \tau$$

for $\mathcal{R}_2 = \mathcal{R}_1 \cup \{(W, W', \tau)\}$ by the definition of $\sim^{\circ}$.

Case (E-App). Then M_0 has the form

$$M_0 \quad = \quad M_1 M_2$$

and the given evaluation derivations have the forms:

$$\frac{\begin{array}{c}[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \Downarrow (\mathtt{fix}\,f(x\!:\!\pi)\!:\!\rho = M_3) \\ [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \Downarrow W_1 \\ [W_1/x][(\mathtt{fix}\,f(x\!:\!\pi)\!:\!\rho = M_3)/f]M_3 \Downarrow W\end{array}}{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0](M_1M_2) \Downarrow W}$$

$$\frac{\begin{array}{c}[\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_1 \Downarrow (\mathtt{fix}\,f(x\!:\!\pi')\!:\!\rho' = M_3') \\ [\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2 \Downarrow W_1' \\ [W_1'/x][(\mathtt{fix}\,f(x\!:\!\pi')\!:\!\rho' = M_3')/f]M_3' \Downarrow W\end{array}}{[\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0](M_1M_2) \Downarrow W'}$$

By inversion of (T-App), we have

$$\overline{\alpha}_0, \overline{x}_0 \!:\! \overline{\tau}_0 \quad \vdash \quad M_1 \quad : \quad \sigma \to \tau$$

and

$$\overline{\alpha}_0, \overline{x}_0 \!:\! \overline{\tau}_0 \quad \vdash \quad M_2 \quad : \quad \sigma$$

for some σ. Thus, by the definition of $\sim^{\circ}$, we have

$$\Delta_0 \quad \vdash \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \quad \sim^{\circ}_{\mathcal{R}_0} \quad [\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_1 \quad : \quad \sigma \to \tau.$$

Then, by the induction hypothesis, we have

$$\Delta_1 \quad \vdash \quad \mathtt{fix}\,f(x\!:\!\pi)\!:\!\rho = M_3 \quad \sim^{\circ}_{\mathcal{R}_1} \quad \mathtt{fix}\,f(x\!:\!\pi')\!:\!\rho' = M_3' \quad : \quad \sigma \to \tau$$

for some $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Therefore, by the definition of $\sim^\circ$, we have

$$\begin{array}{rcl} \texttt{fix}\, f(x\!:\!\pi)\!:\!\rho = M_3 & = & [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_4 \\ \texttt{fix}\, f(x\!:\!\pi')\!:\!\rho' = M_3' & = & [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_4 \end{array}$$

for some

$$\Delta_1 \ \vdash\ \overline{V}_1 \ \sim_{\mathcal{R}_1}\ \overline{V}'_1 \ :\ \overline{\tau}_1$$

and

$$\overline{\alpha}_1, \overline{x}_1 \!:\! \overline{\tau}_1 \ \vdash\ M_4 \ :\ \sigma \to \tau$$

with $\Delta_1 = \{(\overline{\alpha}_1, \overline{\sigma}_1, \overline{\sigma}'_1)\}$.

Meanwhile, by weakening, we have

$$\overline{\alpha}_1, \overline{x}_0 \!:\! \overline{\tau}_0 \ \vdash\ M_2 \ :\ \sigma.$$

Thus, by the definition of $\sim^\circ$, we have

$$\Delta_1 \ \vdash\ [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \ \sim^\circ_{\mathcal{R}_1}\ [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2 \ :\ \sigma$$

since $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Then, by the induction hypothesis, we have

$$\Delta_2 \ \vdash\ W_1 \ \sim^\circ_{\mathcal{R}_2} W'_1 \ :\ \sigma$$

for some $\Delta_2 \supseteq \Delta_1$ and $\mathcal{R}_2 \supseteq \mathcal{R}_1$. Therefore, by the definition of $\sim^\circ$, we have

$$\begin{array}{rcl} W_1 & = & [\overline{V}_2/\overline{x}_2][\overline{\sigma}_2/\overline{\alpha}_2]M_5 \\ W'_1 & = & [\overline{V}'_2/\overline{x}_2][\overline{\sigma}'_2/\overline{\alpha}_2]M_5 \end{array}$$

for some

$$\Delta_2 \ \vdash\ \overline{V}_2 \ \sim_{\mathcal{R}_2}\ \overline{V}'_2 \ :\ \overline{\tau}_2$$

and

$$\overline{\alpha}_2, \overline{x}_2 \!:\! \overline{\tau}_2 \ \vdash\ M_5 \ :\ \sigma$$

with $\Delta_2 = \{(\overline{\alpha}_2, \overline{\sigma}_2, \overline{\sigma}'_2)\}$.

Sub-case $M_4 = (\texttt{fix}\, f(x\!:\!\sigma)\!:\!\tau = M''_3)$. By inversion of (T-Fix), we have

$$\overline{\alpha}_1, \overline{x}_1 \!:\! \overline{\tau}_1, f \!:\! \sigma \to \tau, x \!:\! \sigma \ \vdash\ M''_3 \ :\ \tau.$$

Thus, by weakening and the substitution lemma for values, we have

$$\overline{\alpha}_2, \overline{x}_1 \!:\! \overline{\tau}_1, \overline{x}_2 \!:\! \overline{\tau}_2 \ \vdash\ [(\texttt{fix}\, f(x\!:\!\sigma)\!:\!\tau = M''_3)/f][M_5/x]M''_3 \ :\ \tau.$$

Then, by the definition of $\sim^\circ$, we have

$$\begin{array}{rl} \Delta_2 \ \vdash & [\overline{V}_1, \overline{V}_2/\overline{x}_1, \overline{x}_2][\overline{\sigma}_2/\overline{\alpha}_2][(\texttt{fix}\, f(x\!:\!\sigma)\!:\!\tau = M''_3)/f][M_5/x]M''_3 \\ \sim^\circ_{\mathcal{R}_2} & [\overline{V}'_1, \overline{V}'_2/\overline{x}_1, \overline{x}_2][\overline{\sigma}'_2/\overline{\alpha}_2][(\texttt{fix}\, f(x\!:\!\sigma)\!:\!\tau = M''_3)/f][M_5/x]M''_3 \ :\ \tau, \end{array}$$

i.e.,

$$\Delta_2 \ \vdash\ [W_1/x][(\texttt{fix}\, f(x\!:\!\pi)\!:\!\rho = M_3)/f]M_3 \ \sim^\circ_{\mathcal{R}_2}\ [W'_1/x][(\texttt{fix}\, f(x\!:\!\pi')\!:\!\rho' = M'_3)/f]M'_3 \ :\ \tau.$$

Again by the induction hypothesis, we obtain

$$\Delta_3 \ \vdash\ W \ \sim^\circ_{\mathcal{R}_3}\ W' \ :\ \tau$$

for some $\Delta_3 \supseteq \Delta_2$ and $\mathcal{R}_3 \supseteq \mathcal{R}_2$.

Sub-case $M_4 = x_{1_i}$**.** Then

$$\begin{array}{rcl} V_{1_i} & = & \mathtt{fix}\, f(x:\pi):\rho = M_3 \\ V'_{1_i} & = & \mathtt{fix}\, f(x:\pi'):\rho' = M'_3 \end{array}$$

and $\tau_{1_i} = \sigma \to \tau$. Since we have

$$(\mathtt{fix}\, f(x:\pi):\rho = M_3)([\overline{V}_2/\overline{x}_2][\overline{\sigma}_2/\overline{\alpha}_2]M_5) \quad \Downarrow \quad W$$

and

$$(\mathtt{fix}\, f(x:\pi'):\rho' = M'_3)([\overline{V}'_2/\overline{x}_2][\overline{\sigma}'_2/\overline{\alpha}_2]M_5) \quad \Downarrow \quad W'$$

with

$$\Delta_2 \quad \vdash \quad \mathtt{fix}\, f(x:\pi):\rho = M_3 \quad \sim_{\mathcal{R}_2} \quad \mathtt{fix}\, f(x:\pi'):\rho' = M'_3 \quad : \quad \sigma \to \tau,$$

we have

$$(\Delta_2, \mathcal{R}_2 \cup \{(W, W', \tau)\}) \quad \in \quad \sim$$

by Condition 2 of bisimulation. Thus, we have

$$\Delta_2 \quad \vdash \quad W \quad \sim^{\circ}_{\mathcal{R}_3} \quad W' \quad : \quad \tau$$

for $\mathcal{R}_3 = \mathcal{R}_2 \cup \{(W, W', \tau)\}$ by the definition of $\sim^{\circ}$.

Proofs of the other cases are similar. $\square$

C.3 Proof of Lemma 4.15

We assume $N \Downarrow W$ and prove $N' \Downarrow$ by induction on the derivation of $N \Downarrow W$. (The other direction follows by symmetry.) The argument is similar to the proof of Lemma 4.14, except that we are proving the *existence* of an evaluation derivation for N' by using the given evaluation derivation for N, instead of proving a property of given evaluation derivations for N and N'. We show just the most interesting case: the one for (E-Open).

By the definition of $\sim^{\circ}$, we have

$$N \quad = \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_0$$

and

$$N' \quad = \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_0$$

for some

$$\Delta_0 \quad \vdash \quad \overline{V}_0 \quad \sim_{\mathcal{R}_0} \quad \overline{V}'_0 \quad : \quad \overline{\tau}_0$$

and

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \quad \vdash \quad M_0 \quad : \quad \tau$$

with $\Delta_0 = \{(\overline{\alpha}_0, \overline{\sigma}_0, \overline{\sigma}'_0)\}$. In the case of (E-Open), M_0 has the form

$$M_0 \quad = \quad \mathtt{open}\, M_1\, \mathtt{as}\, \alpha, y\, \mathtt{in}\, M_2$$

and the given evaluation derivation for N has the following form:

$$\frac{\begin{array}{c}[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_1 \Downarrow \texttt{pack}\,\rho_1, W_1\,\texttt{as}\,\exists\alpha.\,\rho_2 \\ [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \Downarrow\end{array}}{[\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0](\texttt{open}\,M_1\,\texttt{as}\,\alpha, y\,\texttt{in}\,M_2) \Downarrow}$$

We aim to derive an evaluation of N' of a similar form:

$$\frac{\begin{array}{c}[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \Downarrow \texttt{pack}\,\rho'_1, W'_1\,\texttt{as}\,\exists\alpha.\,\rho'_2 \\ [W'_1/y][\rho'_1/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2 \Downarrow\end{array}}{[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0](\texttt{open}\,M_1\,\texttt{as}\,\alpha, y\,\texttt{in}\,M_2) \Downarrow}$$

By inversion of (T-Open), we have

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0 \quad \vdash \quad M_1 \quad : \quad \exists\alpha.\,\rho''_2$$

and

$$\overline{\alpha}_0, \overline{x}_0 : \overline{\tau}_0, \alpha, y : \rho''_2 \quad \vdash \quad M_2 \quad : \quad \tau$$

with $\alpha \notin FTV(\tau)$. Thus, by the definition of $\sim^\circ$, we have

$$\Delta_0 \quad \vdash \quad [\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}]M_1 \quad \sim^\circ_{\mathcal{R}_0} \quad [\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}]M_1 \quad : \quad \exists\alpha.\,\rho''_2.$$

Therefore, by the induction hypothesis and Lemma 4.14, we have

$$[\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_1 \quad \Downarrow \quad W'_2$$

for some

$$\Delta_1 \quad \vdash \quad \texttt{pack}\,\rho_1, W_1\,\texttt{as}\,\exists\alpha.\,\rho_2 \quad \sim^\circ_{\mathcal{R}_1} \quad W'_2 \quad : \quad \exists\alpha.\,\rho''_2$$

with $\Delta_1 \supseteq \Delta_0$ and $\mathcal{R}_1 \supseteq \mathcal{R}_0$. Then, by the definition of $\sim^\circ$, we have

$$\texttt{pack}\,\rho_1, W_1\,\texttt{as}\,\exists\alpha.\,\rho_2 \quad = \quad [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_3$$

and

$$W'_2 \quad = \quad [\overline{V}'_1/\overline{x}_1][\overline{\sigma}'_1/\overline{\alpha}_1]M_3$$

for some

$$\Delta_1 \quad \vdash \quad \overline{V}_1 \quad \sim_{\mathcal{R}_1} \quad \overline{V}'_1 \quad : \quad \overline{\tau}_1$$

and

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_3 \quad : \quad \exists\alpha.\,\rho''_2$$

with $\Delta_1 = \{(\overline{\alpha}_1, \overline{\sigma}_1, \overline{\sigma}'_1)\}$.

Sub-case $M_3 = (\texttt{pack}\, \rho_1'', M_4\ \texttt{as}\ \exists\alpha.\, \rho_2''')$. Then W_2' has the form

$$W_2' \quad = \quad \texttt{pack}\, \rho_1', W_1'\ \texttt{as}\ \exists\alpha.\, \rho_2'$$

where

$$\begin{array}{rcl} W_1 & = & [\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_4 \\ W_1' & = & [\overline{V}_1'/\overline{x}_1][\overline{\sigma}_1'/\alpha_1]M_4 \end{array}$$

and

$$\begin{array}{rcl} \rho_1 & = & [\overline{\sigma}_1/\overline{\alpha}_1]\rho_1'' \\ \rho_1' & = & [\overline{\sigma}_1'/\overline{\alpha}_1]\rho_1''. \end{array}$$

Since we have

$$\overline{\alpha}_1, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad M_4 \quad : \quad [\rho_1''/\alpha]\rho_2''$$

by inversion of (T-Pack), we have

$$\overline{\alpha}_1, \overline{x}_0 : \overline{\tau}_0, \overline{x}_1 : \overline{\tau}_1 \quad \vdash \quad [M_4/y][\rho_1''/\alpha]M_2 \quad : \quad \tau$$

by weakening and the substitution lemmas for types and terms. Therefore, by the definition of $\sim^\circ$, we have

$$\Delta_1 \quad \vdash \quad [\overline{V}_0, \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \quad \sim^\circ_{\mathcal{R}_1} \quad [\overline{V}_0', \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \quad : \quad \tau.$$

Since

$$\begin{array}{rl} & [\overline{V}_0, \overline{V}_1/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \\ = & [(([\overline{V}_1/\overline{x}_1][\overline{\sigma}_1/\overline{\alpha}_1]M_4)/y][([\overline{\sigma}_1/\overline{\alpha}_1]\rho_1'')/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \\ = & [W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2 \end{array}$$

and

$$\begin{array}{rl} & [\overline{V}_0', \overline{V}_1'/\overline{x}_0, \overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1][M_4/y][\rho_1''/\alpha]M_2 \\ = & [(([\overline{V}_1'/\overline{x}_1][\overline{\sigma}_1'/\overline{\alpha}_1]M_4)/y][([\overline{\sigma}_1'/\overline{\alpha}_1]\rho_1'')/\alpha][\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2 \\ = & [W_1'/y][\rho_1'/\alpha][\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2, \end{array}$$

we have

$$[W_1'/y][\rho_1'/\alpha][\overline{V}_0'/\overline{x}_0][\overline{\sigma}_0'/\overline{\alpha}_0]M_2 \quad \Downarrow$$

by the induction hypothesis, i.e., $N' \Downarrow$.

Sub-case $M_3 = x_{1_i}$. Then $W_2' = V_{1_i}'$ where

$$V_{1_i} \quad = \quad \texttt{pack}\, \rho_1, W_1\ \texttt{as}\ \exists\alpha.\, \rho_2$$

and $\tau_{1_i} = \exists\alpha.\, \rho_2''$. By Condition 1 of bisimulation, V_{1_i}' is a value of the existential type $[\overline{\sigma}_1'/\overline{\alpha}_1]\tau_{1_i} = \exists\alpha.\, ([\overline{\sigma}_1'/\overline{\alpha}_1]\rho_2'')$, so it has the form

$$V_{1_i}' \quad = \quad \texttt{pack}\, \rho_1', W_1'\ \texttt{as}\ \exists\alpha.\, \rho_2'.$$

Since

$$\Delta_1 \quad \vdash \quad V_{1_i} \quad \sim_{\mathcal{R}_1} \quad V_{1_i}' \quad : \quad \tau_{1_i},$$

we have the following two possibilities by Condition 4 of bisimulation.

Sub-sub-case $(\beta,\rho_1,\rho_1')\in\Delta_1$ **and** $(W_1,W_1',[\beta/\alpha]\rho_2'')\in\mathcal{R}_1$**.** Since we have

$$\overline{\alpha}_0,\overline{x}_0:\overline{\tau}_0,\alpha,y:\rho_2''\quad\vdash\quad M_2\quad:\quad\tau$$

with $\alpha\notin FTV(\tau)$, we have

$$\overline{\alpha}_0,\overline{x}_0:\overline{\tau}_0,\beta,y:[\beta/\alpha]\rho_2''\quad\vdash\quad[\beta/\alpha]M_2\quad:\quad\tau.$$

Then, we have

$$\Delta_1\quad\vdash\quad[\overline{V}_0,W_1/\overline{x}_0,y][\overline{\sigma}_1/\overline{\alpha}_1][\beta/\alpha]M_2\quad\sim^\circ_{\mathcal{R}_1}\quad[\overline{V}'_0,W_1'/\overline{x}_0,y][\overline{\sigma}'_1/\overline{\alpha}_1][\beta/\alpha]M_2\quad:\quad\tau$$

by the definition of $\sim^\circ$. Since we have $\beta=\alpha_{1_i}$ with $\rho_1=\sigma_{1_i}$ and $\rho_1'=\sigma'_{1_i}$ for some i, we have

$$[\overline{V}_0,W_1/\overline{x}_0,y][\overline{\sigma}_1/\overline{\alpha}_1][\beta/\alpha]M_2\quad=\quad[W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2$$

and

$$[\overline{V}'_0,W_1'/\overline{x}_0,y][\overline{\sigma}'_1/\overline{\alpha}_1][\beta/\alpha]M_2\quad=\quad[W_1'/y][\rho_1'/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2,$$

so we have

$$[W_1'/y][\rho_1'/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2\quad\Downarrow$$

by the induction hypothesis, i.e., $N'\Downarrow$.

Sub-sub-case $(\Delta_1\uplus\{(\alpha,\rho_1,\rho_1')\},\mathcal{R}_1\cup\{(W_1,W_1',\rho_2'')\})\in\sim$**.** Then we have

$$\Delta_2\quad\vdash\quad[\overline{V}_0,W_1/\overline{x}_0,y][\overline{\sigma}_0,\rho_1/\overline{\alpha}_0,\alpha]M_2\quad\sim^\circ_{\mathcal{R}_2}\quad[\overline{V}'_0,W_1'/\overline{x}_0,y][\overline{\sigma}'_0,\rho_1'/\overline{\alpha}_0,\alpha]M_2\quad:\quad\tau$$

for $\Delta_2=\Delta_1\cup\{(\alpha,\rho_1,\rho_1')\}$ and $\mathcal{R}_2=\mathcal{R}_1\cup\{(W_1,W_1',\rho_2'')\}$ by the definition of $\sim^\circ$. Since we have

$$[\overline{V}_0,W_1/\overline{x}_0,y][\overline{\sigma}_0,\rho_1/\overline{\alpha}_0,\alpha]M_2\quad=\quad[W_1/y][\rho_1/\alpha][\overline{V}_0/\overline{x}_0][\overline{\sigma}_0/\overline{\alpha}_0]M_2$$

and

$$[\overline{V}'_0,W_1'/\overline{x}_0,y][\overline{\sigma}'_0,\rho_1'/\overline{\alpha}_0,\alpha]M_2\quad=\quad[W_1'/y][\rho_1'/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2,$$

we have

$$[W_1'/y][\rho_1'/\alpha][\overline{V}'_0/\overline{x}_0][\overline{\sigma}'_0/\overline{\alpha}_0]M_2\quad\Downarrow$$

by the induction hypothesis, i.e., $N'\Downarrow$. □

Bibliography

Abadi, Martín (1999). Secrecy by typing in security protocols. *Journal of the ACM 46*(5), 749–786. Preliminary version appeared in *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 611–638, 1997.

Abadi, Martín (2000). Tt-closed relations and admissibility. *Mathematical Structures in Computer Science 10*(3), 313–320.

Abadi, Martín and Cédric Fournet (2001). Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 104–115.

Abadi, Martín and Andrew D. Gordon (1998). A bisimulation method for cryptographic protocols. *Nordic Journal of Computing 5*, 267–303. Preliminary version appeared in *7th European Symposium on Programming*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1381, pp. 12–26, 1998.

Abadi, Martín and Andrew D. Gordon (1999). A calculus for cryptographic protocols: The spi calculus. *Information and Computation 148*(1), 1–70. Preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997.

Abramsky, Samson (1990). The lazy lambda calculus. In David A. Turner (Ed.), *Research Topics in Functional Programming*, pp. 65–117. Addison-Wesley.

Ahmed, Amal, Andrew W. Appel, and Roberto Virga (2003). An indexed model of impredicative polymorphism and mutable references. `http://www.cs.princeton.edu/~amal/papers/impred.pdf`.

Appel, Andrew W. and David McAllester (2001). An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems 23*(5), 657–683.

Berger, Martin, Kohei Honda, and Nobuko Yoshida (2003). Genericity and the pi-calculus. In *Foundations of Software Science and Computation Structures*, Volume 2620 of *Lecture Notes in Computer Science*, pp. 103–119. Springer-Verlag.

Bierman, Gavin M., Andrew M. Pitts, and Claudio V. Russo (2000). Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Higher Order Operational Techniques in Semantics*, Volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science.

Birkedal, Lars and Robert Harper (1999). Relational interpretations of recursive types in an operational setting. *Information and Computation 155*(1–2), 3–63. Summary appeared in *Theoreti-*

cal Aspects of Computer Software, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 458–490, 1997.

Boreale, Michele, Rocco De Nicola, and Rosario Pugliese (2002). Proof techniques for cryptographic processes. *SIAM Journal on Computing 31*(3), 947–986. Preliminary version appeared in *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 157–166, 1999.

Borgström, Johannes and Uwe Nestmann (2002). On bisimulations for the spi calculus. In *9th International Conference on Algebraic Methodology and Software Technology*, Volume 2422 of *Lecture Notes in Computer Science*, pp. 287–303. Springer-Verlag.

Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce (1999). Comparing object encodings. *Information and Computation 155*(1–2), 108–133. Extended abstract appeared in *Theoretical Aspects of Computer Software*, Springer-Verlag, vol. 1281, pp. 415–338, 1997.

Crary, Karl and Robert Harper (2000). Syntactic logical relations over polymorphic and recursive types. Draft.

Dolev, Danny and Andrew C. Yao (1983). On the security of public key protocols. *IEEE Transactions on Information Theory 29*(2), 198–208.

Dreyer, Derek, Karl Crary, and Robert Harper (2003). A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 236–249.

Durante, Antonio, Riccardo Focardi, and Roberto Gorrieri (1999). CVS: A compiler for the analysis of cryptographic protocols. In *12th IEEE Computer Security Foundations Workshop*, pp. 203–212.

Durante, Antonio, Riccardo Focardi, and Roberto Gorrieri (2000). A compiler for analysing cryptographic protocols using non-interference. *ACM Transactions on Software Engineering and Methodology 9*(4), 488–528.

Girard, Jean-Yves (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. thesis, Université Paris VII.

Gordon, Andrew D. (1995a). Bisimilarity as a theory of functional programming. mini-course. `http://research.microsoft.com/~adg/Publications/BRICS-NS-95-3.dvi.gz`.

Gordon, Andrew D. (1995b). Operational equivalences for untyped and polymorphic object calculi. In *Higher Order Operational Techniques in Semantics*, pp. 9–54.

Gordon, Andrew D. and Alan Jeffrey (2001). Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pp. 145–159.

Gordon, Andrew D. and Gareth D. Rees (1995). Bisimilarity for $F_{<:}$. Draft.

Gordon, Andrew D. and Gareth D. Rees (1996). Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 386–395.

Goubault-Larrecq, Jean, Slawomir Lasota, David Nowak, and Yu Zhang (2004). Complete lax logical relations for cryptographic lambda-calculi. In *Computer Science Logic*, Lecture Notes in Computer Science. Springer-Verlag. To appear.

Grossman, Dan, Greg Morrisett, and Steve Zdancewic (2000). Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems 22*(6), 1037–1080. Extended abstract appeared as *Principals in Programming Languages: A Syntactic Proof Technique* in *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pp. 197–207, 1999.

Heather, James, Gavin Lowe, and Steve Schneider (2000). How to prevent type flaw attacks on security protocols. In *13th IEEE Computer Security Foundations Workshop*, pp. 255–268.

Heintze, Nevin and Edmund Clarke (Eds.) (1999). *Workshop on Formal Methods and Security Protocols*. `http://cm.bell-labs.com/cm/cs/who/nch/fmsp99/`.

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

Hennessy, Matthew and James Riely (2000). Information flow vs. resource access in the asynchronous pi-calculus. In *Automata, Languages and Programming*, Volume 1853 of *Lecture Notes in Computer Science*, pp. 415–427. Springer-Verlag.

Honda, Kohei and Nobuko Yoshida (2002). A uniform framework for secure information flow. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 81–92.

Howe, Douglas J. (1996). Proving congruence of bisimulation in functional programming languages. *Information and Computation 124*(2), 103–112.

Hughes, Dominic J.D. (1997). Games and definability for System F. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pp. 76–86.

Jeffrey, Alan and Julian Rathke (1999). Towards a theory of bisimulation for local names. In *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 56–66.

Jeffrey, Alan and Julian Rathke (2004). A theory of bisimulation for a fragment of Concurrent ML with local names. *Theoretical Computer Science*. To appear. Extended abstract appeared in *15th Annual IEEE Symposium on Logic in Computer Science*, pp. 311–321, 2000.

Leifer, James J., Gilles Peskine, Peter Sewell, and Keith Wansbrough (2003). Global abstraction-safe marshalling with hash types. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pp. 87–98.

Leroy, Xavier (1995). Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Liskov, Barbara (Ed.) (1981). *CLU Reference Manual*, Volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag.

Liskov, Barbara (1993). A history of CLU. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pp. 133–147.

Lowe, Gavin (1995). An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters 56*(3), 131–133.

Meadows, Catherine (1995). Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology – Asiacrypt '94*, Volume 917 of *Lecture Notes in Computer Science*, pp. 133–150. Springer-Verlag.

Meadows, Catherine (2000). Open issues in formal methods for cryptographic protocol analysis. In *DARPA Information Survivability Conference and Exposition*, pp. 237–250. IEEE Computer Society.

Menezes, Alfred J., Paul C. van Oorshot, and Scott A. Vanstone (1996). *Handbook of Applied Cryptography*. CRC Press.

Millen, Jonathan K. (1999). A necessarily parallel attack. In *Workshop on Formal Methods and Security Protocols*. `http://www.cs.bell-labs.com/who/nch/fmsp99/program.html`.

Millen, Jonathan K. (2004). CSFW home page. `http://www2.csl.sri.com/csfw/`.

Milner, Robin (1980). *A Calculus of Communicating Systems*. Springer-Verlag.

Milner, Robin (1995). *Communication and Concurrency*. Springer-Verlag.

Milner, Robin (1999). *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press.

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.

Mitchell, John C. (1991). On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 305–330. Academic Press.

Mitchell, John C. (1996). *Foundations for Programming Languages*. MIT Press.

Mitchell, John C. and Gordon D. Plotkin (1988). Abstract types have existential types. *ACM Transactions on Programming Languages and Systems 10*(3), 470–502. Preliminary version appeared in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 37–51, 1985.

Moggi, Eugenio (1991). Notions of computation and monads. *Information and Computation 93*(1), 55–92.

Morris, Jr., James H. (1968). *Lambda-Calculus Models of Programming Languages*. Ph. D. thesis, Massachusetts Institute of Technology.

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

Morris, Jr., James H. (1973b). Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 120–124.

Needham, Roger and Michael Schroeder (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM 21*(12), 993–999.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press.

Pierce, Benjamin C. (Ed.) (2005). *Advanced Topics in Types and Programming Languages*. MIT Press.

Pierce, Benjamin C. and Davide Sangiorgi (2000). Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM 47*(3), 531–586. Extended abstract appeared in *Proceedings of* *the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 531–584.

Pierce, Benjamin C. and Eijiro Sumii (2000). Relating cryptography and polymorphism. Draft. `http://www.cis.upenn.edu/~sumii/pub/`.

Pierce, Benjamin C. and David N. Turner (1993). Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming* 4(2), 207–247. Preliminary version appeared as *Object-Oriented Programming without Recursive Types* in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993, pp. 299–312.

Pitts, Andrew M. (1998). Existential types: Logical relations and operational equivalence. In *Automata, Languages and Programming*, Volume 1443 of *Lecture Notes in Computer Science*, pp. 309–326. Springer-Verlag.

Pitts, Andrew M. (2000). Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10*, 321–359. Preliminary version appeared in *HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics*, *Electronic Notes in Theoretical Computer Science*, vol. 10, 1998.

Pitts, Andrew M. and Joshua R. X. Ross (1998). Process calculus based upon evaluation to committed form. *Theoretical Computer Science 195*, 155–182. Preliminary version appeared in *CONCUR'96: Concurrency Theory*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1119, pp. 18–33, 1996.

Pitts, Andrew M. and Ian Stark (1998). Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pp. 227–273. Cambridge University Press.

Plotkin, Gordon D., John Power, Donald Sannella, and Robert D. Tennent (2000). Lax logical relations. In *Automata, Languages and Programming*, Volume 1853 of *Lecture Notes in Computer Science*, pp. 85–102. Springer-Verlag.

Pottier, François (2002). A simple view of type-secure information flow in the π-calculus. In *15th IEEE Computer Security Foundations Workshop*, pp. 320–330.

Pottier, François and Vincent Simonet (2002). Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 319–330.

Reynolds, John C. (1974). Towards a theory of type structure. In *Colloque sur la Programmation*, Volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag.

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

Rossberg, Andreas (2003). Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 241–252.

Russo, Claudio V. (1998). *Types For Modules*. Ph. D. thesis, University of Edinburgh. `http://www.dcs.ed.ac.uk/home/cvr/ECS-LFCS-98-389.html`.

Ryan, Peter Y. A. and Steve A. Schneider (1999). Process algebra and non-interference. In *12th IEEE Computer Security Foundations Workshop*, pp. 214–227.

Sangiorgi, Davide (1992). *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigm*. Ph. D. thesis, University of Edinburgh.

Sangiorgi, Davide and Robin Milner (1992). The problem of "weak bisimulation up to". In *CONCUR '92, Third International Conference on Concurrency Theory*, Volume 630 of *Lecture Notes in Computer Science*, pp. 32–46. Springer-Verlag.

Schneier, Bruce (1996). *Applied Cryptography*. John Wiley & Sons, Inc.

Sewell, Peter (2001). Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 236–247.

Shaw, Mary (Ed.) (1981). *Alphard: Form and Content*. Springer-Verlag.

Smith, Geoffrey and Dennis Volpano (1998). Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 355–364.

Stark, Ian (1994). *Names and Higher-Order Functions*. Ph. D. thesis, University of Cambridge. `http://www.dcs.ed.ac.uk/home/stark/publications/thesis.html`.

Stark, Ian (1996). Categorical models for local names. *LISP and Symbolic Computation 9*(1), 77–107.

Stinson, Douglas R. (1995). *Cryptography – Theory and Practice*. CRC Press.

Sumii, Eijiro and Benjamin C. Pierce (2003). Logical relations for encryption. *Journal of Computer Security 11*(4), 521–554. Extended abstract appeared in *14th IEEE Computer Security Foundations Workshop*, pp. 256–269, 2001.

Sumii, Eijiro and Benjamin C. Pierce (2004a). A bisimulation for dynamic sealing. *Theoretical Computer Science*. To appear. Extended abstract appeared in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 161–172, 2004.

Sumii, Eijiro and Benjamin C. Pierce (2004b). A bisimulation for type abstraction and recursion. Technical Report MS-CIS-04-27, Department of Computer and Information Science, University of Pennsylvania. `http://www.cis.upenn.edu/~sumii/pub/`. Extended abstraced is to appear in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

Taft, S. Tucker and Robert A. Duff (Eds.) (1995). *Ada 95 Referential Manual: Language and Standard Libraries*, Volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag.

Turner, David N. (1995). *The Polymorphic Pi-calculus: Theory and Implementation*. Ph. D. thesis, University of Edinburgh.

Volpano, Dennis (1999). Formalization and proof of secrecy properties. In *12th IEEE Computer Security Foundations Workshop*, pp. 92–95.

Wadler, Philip (1989). Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pp. 347–359. ACM.

Wirth, Niklaus (1989). *Programming in Modula-2* (4th ed.). Texts and Monographs in Computer Science. Springer-Verlag.