A Bisimulation for Type Abstraction and Recursion

Eijiro Sumii
University of Pennsylvania
sumii@saul.cis.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

ABSTRACT

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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms

Theory, Languages

Keywords

Lambda-Calculus, Contextual Equivalence, Bisimulations, Logical Relations, Existential Types, Recursive Types

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., [24, Chapter 8] for details) and can also be adapted [30, 29] to their term models; this adaptation is sometimes called syntactic logical relations [13]. 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" [30, 29, 9, 13]. 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 [9, 13]. 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 [21, 22, 23] and state transition systems in general. Abramsky [4] 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 [14, 17, 15, 16] extended applicative bisimulations to calculi with objects, subtyping, universal polymorphism, and recursive types. Sangiorgi [32] has defined *context bisimulation*, which is a variant of applicative bisimulation for higher-order π-calculus [32].

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

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

where $\tau = \exists\alpha.\, \alpha \times (\alpha \to \texttt{bool})$. 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 `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 [28, 7]. 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 [7] 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 [28] and complex [7].

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

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

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

$$\begin{array}{rl} & f[\texttt{int}]\langle 1, \lambda x\!:\!\texttt{int}.\, x \overset{\texttt{int}}{=} 0\rangle \Downarrow \\ \iff & f[\texttt{bool}]\langle \texttt{true}, \lambda x\!:\!\texttt{bool}.\, \neg x\rangle \Downarrow \end{array}$$

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 our previous work [34] on bisimulation for *sealing* [26, 27], 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

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

where

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

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 $\alpha \to \texttt{bool}$, 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 `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 $\mathcal{R}$ occuring 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 work (as well as our previous work [34]). 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 4.3 for generative functors or as we did in previous work [34, Examples 4.7 and 4.8] 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 our previous work [34] 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 paper is typed (unlike in our previous work), 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.

Overview. The rest of this paper is structured as follows. Section 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 3 defines our bisimulation. Section 4 gives examples to illustrate its uses and Section 5 proves soundness and completeness of the bisimulation with respect to the generalized contextual equivalence. Section 6 generalizes these results, which have been restricted to closed values for simplicity, to non-values and open terms. Section 7 discusses a limitation of our bisimulation concerning higher-order functions. Section 8 discusses related work, and Section 9 concludes with future work.

Throughout the paper, 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 $n \geq 0$.

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 is given in Figure 1. The (big-step) semantics $M \Downarrow V$ and typing rules $\Gamma \vdash M : \tau$ are standard; we omit them for brevity and refer readers to the full version [35] for details. We include recursive functions

$M, N, C, D ::=$	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 1: Syntax

$\texttt{fix}\ f(x : \tau) : \sigma = M$ 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 $\texttt{fix}\ f(x : \tau) : \sigma = M$ when f is not free in M. We will write $\texttt{let}\ x : \tau = M\ \texttt{in}\ N$ for $(\lambda x : \tau.\, N)M$. We sometimes omit type annotations—as in $\lambda x.\, M$ and $\texttt{let}\ x = M\ \texttt{in}\ N$—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 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 $\lambda x : \texttt{int}.\, x$ is contextually equivalent to $\lambda x : \texttt{int}.\, x - 1$ at type $\alpha \to \texttt{int}$, where the implementation of abstract type α is $\texttt{int}$ 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 $\{(2, 3, \alpha), ((\lambda x : \texttt{int}.\, x), (\lambda x : \texttt{int}.\, x - 1), \alpha \to \texttt{int})\}$ and $\{(i, j, \alpha), ((\lambda x : \texttt{int}.\, x), (\lambda x : \texttt{int}.\, x - 1),$

$\alpha \to \texttt{int})\}$; 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 2.1. *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 1, the concrete implementations of abstract type α were `int` in the left-hand world and `bool` in the right-hand world, so Δ was $\{(\alpha, \texttt{int}, \texttt{bool})\}$. 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 2.2. *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 2.3. *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 2.4 (Typed Value Relation in Context). *We write* $(\Delta, \mathcal{R})^\circ$ *for the relation*

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

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

Definition 2.5. 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 6 for discussions on non-values and open terms.

We write

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

for

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

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

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

Corollary 2.7 (Symmetry). *If*

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

then

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

Corollary 2.8 (Transitivity). *If*

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

and

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

then

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

Example 2.9. *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* $\Delta = \{(\alpha, \texttt{int}, \texttt{int})\}$. *Then we have:*

$$\begin{array}{rl}\Delta \ \vdash & 2,\ (\lambda x{:}\texttt{int}.\,x) \\ \equiv & 3,\ (\lambda x{:}\texttt{int}.\,x-1) \\ : & \alpha,\ (\alpha \to \texttt{int})\end{array}$$

More generally,

$$\begin{array}{rl}\Delta \ \vdash & i,\ (\lambda x{:}\texttt{int}.\,x) \\ \equiv & j,\ (\lambda x{:}\texttt{int}.\,x-1) \\ : & \alpha,\ (\alpha \to \texttt{int})\end{array}$$

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

Example 2.10. *Let* $\Delta = \{(\alpha, \texttt{int}, \texttt{bool})\}$. *We have*

$$\begin{array}{rl}\Delta \ \vdash & 1,\ (\lambda x{:}\texttt{int}.\,x \overset{\texttt{int}}{=} 0) \\ \equiv & \texttt{true},\ (\lambda x{:}\texttt{bool}.\,\neg x) \\ : & \alpha,\ (\alpha \to \texttt{bool})\end{array}$$

$$\begin{array}{rl}\Delta \ \vdash & 1,\ (\lambda x{:}\texttt{int}.\,x \overset{\texttt{int}}{=} 0) \\ \equiv & \texttt{false},\ (\lambda x{:}\texttt{bool}.\,x) \\ : & \alpha,\ (\alpha \to \texttt{bool})\end{array}$$

but

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

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.

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 3.1 (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

   $$\begin{aligned} & (\mathtt{fix}\ f(x\!:\!\pi)\!:\!\rho = M)V \Downarrow \\ \iff\ & (\mathtt{fix}\ f(x\!:\!\pi')\!:\!\rho' = M')V' \Downarrow. \end{aligned}$$

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

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

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

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

   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*

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

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

7. *For each $(\mathtt{fold}(V), \mathtt{fold}(V'), \mu\alpha. \tau) \in \mathcal{R}$, 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

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

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 2.4) 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 5.3 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.

[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.

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. 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.

4.1 Warm-Up

Consider the following simple packages

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

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

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

where

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

Then, X is a bisimulation. To prove it, we must check each condition in Definition 3.1 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

  $$(\lambda x : \texttt{int}.\, x \overset{\texttt{int}}{=} 0,\ \lambda x : \texttt{bool}.\, \neg x,\ \alpha \rightarrow \texttt{bool}) \in \mathcal{R}_i$$

  for $i \geq 3$. Since V and V' are values, the D in Definition 2.4 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 5.5) and the generalized definition of contextual equivalence (Definition 2.5).

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 $\exists \alpha.\, \tau$.

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

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

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

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

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

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

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.)

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

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

$$\begin{array}{l} X = \{(\Delta,\mathcal{R}) \mid \\ \quad \Delta = \{(\beta_i,\texttt{int},\texttt{int}) \mid -n\le i\le n\}, \\ \quad \mathcal{R}\subseteq \cup_{-n\le i\le n}\mathcal{R}_i, \\ \quad n=0,1,2,\dots\} \\ \mathcal{R}_i = \{(U,U',\texttt{int}\to\tau), \\ \quad ([i/y]M,[i/y]M',\tau), \\ \quad (\langle i,\lambda x:\texttt{int}.\,x\rangle, \langle i+1,\lambda x:\texttt{int}.\,x-1\rangle, \beta_i\times(\beta_i\to\texttt{int})), \\ \quad (i,i+1,\beta_i), \\ \quad (\lambda x:\texttt{int}.\,x, \lambda x:\texttt{int}.\,x-1, \beta_i\to\texttt{int}), \\ \quad (i,i,\texttt{int})\} \end{array}$$

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 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.

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

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

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

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:

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

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.

4.5 Higher-Order Functions

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

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

[2]This implementation can be viewed as a variant of the so-called recursive existential encoding of objects (see [12] 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.

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

$$\begin{array}{rl} & [1, (\lambda x : \mathtt{int}.\, x \overset{\mathtt{int}}{=} 0)/y, z][\mathtt{int}/\beta]D \Downarrow \\ \iff & [\mathtt{true}, (\lambda x : \mathtt{bool}.\, \neg x)/y, z][\mathtt{bool}/\beta]D \Downarrow \end{array}$$

for every $\beta, y : \beta, z : \beta \to \mathtt{bool} \vdash D : \tau$. (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

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

along with the soundness of bisimilarity in the next section.

5. SOUNDNESS AND COMPLETENESS

We prove that bisimilarity coincides with contextual equivalence (in the generalized form presented in Section 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 5.1 (COMPLETENESS OF BISIMULATION). $\equiv\ \subseteq\ \sim$.

PROOF. By checking that $\equiv$ satisfies each condition of bisimulation. □

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 5.2 (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 5.3 (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 $N \Downarrow W$. □

LEMMA 5.4 (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 5.3. □

COROLLARY 5.5 (SOUNDNESS OF BISIMILARITY). $\sim\ \subseteq\ \equiv$.

PROOF. By the definitions of $\equiv$ and $\sim^\circ$ together with Lemma 5.4. □

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

THEOREM 5.6. $\sim\ =\ \equiv$.

PROOF. By Corollary 5.5 and Lemma 5.1. □

Details of the proofs above are found in the full version [35]. Note that these proofs are much simpler than soundness proofs of applicative bisimulations in previous work [19, 14, 17, 15, 16, 4] thanks to the generalized condition on functions (Condition 2), which is anyway required in the presence of existential polymorphism as discussed in the introduction.

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 [30, 29] 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

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

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 $\vdash K[M] : \mathtt{unit}$ and $\vdash K[M'] : \mathtt{unit}$, where $\mathtt{unit}$ is the nullary tuple type. (In fact, any closed type works in place of $\mathtt{unit}$.)

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 2—and from the completeness of bisimulation. To prove the "if" direction, suppose $\emptyset \vdash V \sim V' : \forall\overline{\alpha}.\, \overline{\tau} \to \tau$. By the soundness of bisimulation, we have $\emptyset \vdash V \equiv V' : \forall\overline{\alpha}.\, \overline{\tau} \to \tau$. Given any K with $\vdash K[M] : \mathtt{unit}$ and $\vdash K[M'] : \mathtt{unit}$, take $C = K[z[\alpha_1] \ldots [\alpha_m] x_1 \ldots x_n]$ 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

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

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

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

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

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

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

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

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

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

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

Figure 2: β-Expansion

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 6.1 (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 2.*

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

Then, we can prove:

Lemma 6.2. *For any*

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

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

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

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 $\overline{\alpha}, \overline{x}\!:\!\overline{\tau} \vdash M \preceq M' : \tau$. $\square$

Theorem 6.3. *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 $K[M] \Downarrow \iff K[M'] \Downarrow$. $\square$

Example 6.4. *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* `int` *provided that x has type* `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 6.5. *The packages*

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

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

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

provided that y has type `int`. *This follows from the bisimilarity of $\lambda y\!:\!\texttt{int}.\,M$ and $\lambda y\!:\!\texttt{int}.\,M'$, which was shown in Section 4.3.*

7. LIMITATIONS (OR: THE RETURN OF HIGHER-ORDER FUNCTIONS)

Although the proof of contextual equivalence in Section 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.)

$$\begin{aligned}
\mathtt{IntSet} &= \mathtt{pack}\ \mathtt{intList}, \mathtt{Nil}, \mathtt{add}, \mathtt{weigh}\ \mathtt{as}\ \exists\alpha.\,\tau \\
\mathtt{IntSet}' &= \mathtt{pack}\ \mathtt{intTree}, \mathtt{Lf}, \mathtt{add}', \mathtt{weigh}'\ \mathtt{as}\ \exists\alpha.\,\tau \\
\tau &= \alpha \times (\mathtt{int} \to \alpha \to \alpha) \times ((\mathtt{int} \to \mathtt{real}) \to \alpha \to \mathtt{real}) \\
\mathtt{add} &= \lambda i : \mathtt{int}.\, \lambda s : \mathtt{intList}.\, \mathtt{Cons}(i, s) \\
\mathtt{add}' &= \lambda i : \mathtt{int}.\, \mathtt{fix}\ f(s : \mathtt{intTree}) : \mathtt{intTree} = \\
&\qquad \mathtt{case}\ s\ \mathtt{of}\ \mathtt{Lf} \Rightarrow \mathtt{Nd}(i, \mathtt{Lf}, \mathtt{Lf}) \\
&\qquad \| \ \mathtt{Nd}(j, s_1, s_2) \Rightarrow \mathtt{if}\ i < j\ \mathtt{then}\ \mathtt{Nd}(j, fs_1, s_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ \mathtt{Nd}(j, s_1, fs_2) \\
\mathtt{weigh} &= \lambda g : \mathtt{int} \to \mathtt{real}.\, \mathtt{fix}\ f(s : \mathtt{intList}) : \mathtt{real} = \\
&\qquad \mathtt{case}\ s\ \mathtt{of}\ \mathtt{Nil} \Rightarrow 0 \ \|\ \mathtt{Cons}(j, s_0) \Rightarrow gj + fs_0 \\
\mathtt{weigh}' &= \lambda g : \mathtt{int} \to \mathtt{real}.\, \mathtt{fix}\ f(s : \mathtt{intTree}) : \mathtt{real} = \\
&\qquad \mathtt{case}\ s\ \mathtt{of}\ \mathtt{Lf} \Rightarrow 0 \ \|\ \mathtt{Nd}(j, s_1, s_2) \Rightarrow gj + fs_1 + fs_2
\end{aligned}$$

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 $\mathtt{weigh}\ V\ W \Downarrow \iff \mathtt{weigh}'\ V'\ W' \Downarrow$ for a certain class of V, W, V', and W'. In particular, V and V' can be of the forms $\lambda z : \mathtt{int}.\, [\mathtt{IntSet}/y]D$ and $\lambda z : \mathtt{int}.\, [\mathtt{IntSet}'/y]D$ for any D of appropriate type. Thus, because of the function application gj in $\mathtt{weigh}$ and $\mathtt{weigh}'$, we must prove

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

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

Resolving this problem requires weakening Condition 2. By a close examination of the soundness proof [35, Appendix], we find the following candidate.

2′. Take any

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

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

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

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

$$\begin{aligned} &(\mathtt{fix}\ f(x : \pi) : \rho = M)V \Downarrow \\ \iff\ &(\mathtt{fix}\ f(x : \pi') : \rho' = M')V' \Downarrow. \end{aligned}$$

Furthermore, if $(\mathtt{fix}\ f(x : \pi) : \rho = M)V \Downarrow W$ and $(\mathtt{fix}\ f(x : \pi') : \rho' = M')V \Downarrow W'$, 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* [6, 5], 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 $\mathtt{weigh}$ and $\mathtt{weigh}'$ 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.

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 [20]—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 [36], 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 [30] 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 [29] 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.

Birkedel and Harper [9] and Crary and Harper [13] 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 [4] proposed *applicative bisimulations* for proving contextual equivalence of untyped λ-terms. Gordon and Rees [14, 17, 15, 16] adapted applicative bisimulations to calculi with objects, subtyping, universal polymorphism, and recursive types. As discussed in Section 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 3.1), 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 [28] 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 [7] defined two equivalence proof methods for linear π-calculi, one based on the syntactic logical relations of Pitts [29, 30] and the other based on the bisimulations of Pierce and Sangiorgi [28]. 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. Completness 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 [2, 1, 10, 11] have been proposed for extensions of π-calculus with cryptographic primitives [3, 1]. Their main idea is similar to Pierce and Sangiorgi's: using a separate environment to represent attackers' knowledge. In previous work [34], 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.

9. CONCLUSION

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" [33] 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. [31, 8]). 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 [25].

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 [32], where restriction hides names and complicates equivalence—and compare the outcome with context bisimulation.

Finally, as suggested in the previous section, the idea of considering sets of relations as bisimulations may be useful for other forms of information hiding such as secrecy typing [18]. 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.

Acknowledgements

We would like to thank Karl Crary, Andy Gordon, Bob Harper, and Andrew Pitts for information and discussions on their work and its relationship to ours. Comments from anonymous reviewers, the members of the PL Club at the University of Pennsylvania, and Naoki Kobayashi helped us to sharpen the presentation.

10. REFERENCES

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115, 2001.

[2] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998. Preliminary version appeared in *7th European Symposium on Programming*, *Lecture Notes in*

Computer Science, Springer-Verlag, vol. 1381, pp. 12–26, 1998.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. Preliminary version appeared in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pp. 36–47, 1997.

[4] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[5] A. Ahmed, A. W. Appel, and R. Virga. An indexed model of impredicative polymorphism and mutable references. `http://www.cs.princeton.edu/~amal/papers/impred.pdf`, 2003.

[6] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[7] M. Berger, K. Honda, and N. Yoshida. Genericity and the pi-calculus. In *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2003.

[8] G. M. Bierman, A. M. Pitts, and C. V. Russo. 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, 2000.

[9] L. Birkedal and R. Harper. Relational interpretations of recursive types in an operational setting. *Information and Computation*, 155(1–2):3–63, 1999. Summary appeared in *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 1281, pp. 458–490, 1997.

[10] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. *SIAM Journal on Computing*, 31(3):947–986, 2002. Preliminary version appeared in *14th Annual IEEE Symposium on Logic in Computer Science*, pp. 157–166, 1999.

[11] J. Borgström and U. Nestmann. On bisimulations for the spi calculus. In *9th International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2002.

[12] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2):108–133, 1999. Extended abstract appeared in *Theoretical Aspects of Computer Software*, Springer-Verlag, vol. 1281, pp. 415–338, 1997.

[13] K. Crary and R. Harper. Syntactic logical relations over polymorphic and recursive types. Draft, 2000.

[14] A. D. Gordon. Bisimilarity as a theory of functional programming. mini-course. `http://research.microsoft.com/~adg/Publications/BRICS-NS-95-3.dvi.gz`, 1995.

[15] A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In *Higher Order Operational Techniques in Semantics*, pages 9–54, 1995.

[16] A. D. Gordon and G. D. Rees. Bisimilarity for $F_{<:}$. Draft, 1995.

[17] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 386–395, 1996.

[18] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[19] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[20] D. J. Hughes. Games and definability for System F. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 76–86, 1997.

[21] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[22] R. Milner. *Communication and Concurrency*. Springer-Verlag, 1995.

[23] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[24] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[25] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

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

[27] J. H. Morris, Jr. Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 120–124, 1973.

[28] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–586, 2000. Extended abstract appeared in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 531–584.

[29] A. M. Pitts. Existential types: Logical relations and operational equivalence. In *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 1998.

[30] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. Preliminary version appeared in *HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics*, *Electronic Notes in Theoretical Computer Science*, vol. 10, 1998.

[31] A. M. Pitts and I. Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[32] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigm*. PhD thesis, University of Edinburgh, 1992.

[33] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In *CONCUR '92, Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[34] E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–172, 2004.

[35] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. Technical Report MS-CIS-04-27, Department of Computer and Information Science, University of Pennsylvania, 2004. `http://www.cis.upenn.edu/~sumii/pub/`.

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