January 2005

A Bisimulation for Type Abstraction and Recursion

Eijiro Sumii  
*University of Pennsylvania*

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

Follow this and additional works at: [https://repository.upenn.edu/cis_papers](https://repository.upenn.edu/cis_papers)

**Recommended Citation**

Postprint version. Copyright ACM, 2005. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2005*, pages 63-74.  
Publisher URL: [http://doi.acm.org/10.1145/1040305.1040311](http://doi.acm.org/10.1145/1040305.1040311)

This paper is posted at ScholarlyCommons. [https://repository.upenn.edu/cis_papers/151](https://repository.upenn.edu/cis_papers/151)  
For more information, please contact repository@pobox.upenn.edu.
A Bisimulation for Type Abstraction and Recursion

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 TT-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.

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

Comments
Postprint version. Copyright ACM, 2005. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 2005, pages 63-74. Publisher URL: http://doi.acm.org/10.1145/1040305.1040311

This conference paper is available at ScholarlyCommons: https://repository.upenn.edu/cis_papers/151
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 ⊤-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 $\text{in}_i(M)$ and $\text{in}_j(N)$ 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 ⊤-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 con-
verge. 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 \( \pi \)-calculus [32]. Unlike logical relations, bisimulations have no difficulty with recursion (or even concurrency). However, existing bisimulation methods for typed \( \lambda \)-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:

\[
M = \text{pack int,}(1, \lambda x: \text{int}. \ x = 0) \text{ as } \tau \\
M' = \text{pack bool,}(\text{true}, \lambda x: \text{bool}. \neg x) \text{ as } \tau
\]

where \( \tau = \exists \alpha. \alpha \times (\alpha \to \text{bool}) \). 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 \( \alpha \) are 1 in the “left-hand world” and \( \text{true} \) in the right-hand world. The same observation applies to context bisimulation.

The only exceptions to the problem above are bisimulations for polymorphic \( \pi \)-calculi [28, 7]. However, \( \pi \)-calculus is name-based and low-level. As a result, it is rather difficult to encode polymorphic \( \lambda \)-calculus into polymorphic \( \pi \)-calculus while preserving equivalence (though there are some results [7] for the case without recursion), so it is at least as difficult to use \( \pi \)-calculus for reasoning about abstraction in \( \lambda \)-calculus or similar languages with (in particular higher-order) functions and recursion. In addition to the problem of encoding, existing bisimulations for polymorphic \( \pi \)-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 \( M' \):

\[
N = \lambda f: \sigma. f[\text{int}](1, \lambda x: \text{int}. x = 0) \\
N' = \lambda f: \sigma. f[\text{bool}](\text{true}, \lambda x: \text{bool}. \neg x)
\]

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

\[
f[\text{int}](1, \lambda x: \text{int}. x = 0) \Downarrow \iff f[\text{bool}](\text{true}, \lambda x: \text{bool}. \neg x) \Downarrow
\]

for any observer function \( f \) of type \( \sigma \), 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

\[
X = \{(\emptyset, R_0), (\Delta, R_1), (\Delta, R_2), (\Delta, R_3)\}
\]

where

\[
R_0 = \{(M, M'), \tau\} \\
R_1 = R_0 \cup \{(1, \lambda x: \text{int}.x \equiv 0), \text{true}, \lambda x: \text{bool}. \neg x, \alpha \times (\alpha \to \text{bool})\} \\
R_2 = R_1 \cup \{(\lambda x: \text{int}. x \equiv 0, \lambda x: \text{bool}. \neg x, \alpha \to \text{bool}\} \\
R_3 = R_2 \cup \{(\text{false}, \text{false}, \text{bool})\} \\
\Delta = \{((\alpha, \text{int}. \alpha)\}
\]

Because we are ultimately interested in the equivalence of \( M \) and \( M' \), we begin by including \( (\emptyset, R_0) \) 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 \( (\Delta, R_1) \) to \( X \), where \( \Delta \) is a concretion environment mapping the abstract type \( \alpha \) 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 \( (\Delta, R_2) \) to \( X \). Last, since the second elements of the pairs are functions of type \( \alpha \to \text{bool} \), a context can apply them to any arguments of type \( \alpha \); the only such arguments are, in fact, 1 in the left-hand side and \( \text{true} \) in the right-hand side, so we add \( (\Delta, R_3) \) 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 \( R \) occuring in a pair \( (\Delta, R) \in X \) 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 on \( X \) that is closed under this increase of contexts’ knowledge. (Thus, in fact, not only \( Y \) but also the singleton set \( \{(\Delta, R_3)\} \) is a bisimulation in our definition.)

Why do we consider a bisimulation \( X \) to be a set of Rs (with corresponding \( \Delta s \)) 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” Rs is not always a valid \( R \). For instance, consider the union of \( R_3 \) and its inverse \( R_3^{-1} = \{(V', V, \tau) \mid (V, V', \tau) \in R_3\} \). 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 \( \{(\Delta, R_3)\} \) and \( \{(\Delta^{-1}, R_3^{-1})\} \) are bisimulations (where \( \Delta^{-1} = \{(\alpha, \tau, \pi) \mid (\alpha, \tau, \pi) \in \Delta\} \) and so is their union \( \{(\Delta, R_3), (\Delta^{-1}, R_3^{-1})\} \), but neither \( \{(\Delta, R_3 \cup R_3^{-1})\} \) nor \( \{(\Delta^{-1}, R_3 \cup R_3^{-1})\} \) 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 pro-

cocols).

Contributions. This is the first sound, complete, and ele-
mentary proof method for contextual equivalence in a lan-
guage with higher-order functions, impredicative polymor-
phism (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
\(\lambda\)-calculus with dynamic sealing (also known as perfect en-
cryption). This form of information hiding is very differ-
ent 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 pre-
vious work), requiring many refinements to take type infor-
mation 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. Accord-
ingly, 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 equiva-

cence, generalized in a non-trivial way for open types as re-
quired by the technicalities which follow. Section 3 de-


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 \(\overline{f} / \overline{x}\), \(\overline{\pi, \pi', \pi''}\) and \(\overline{\pi} : \overline{\tau}\) in-
stead of \(x_1, \ldots, x_n, \{V_1, \ldots, V_n/x_1, \ldots, x_n\}, (\alpha_1, \sigma_1, \alpha'_1), \ldots\)
\((\alpha_n, \sigma_n, \alpha''_n)\) 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 \(\lambda\)-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

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

Figure 1: Syntax

\[
\text{fix } f(x:\tau) : \sigma = M \quad \text{as a primitive for the sake of exposition; }
\]

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
\(\alpha\)-conversion and write \(\lambda x:\tau. M\) for \(\text{fix } f(x:\tau) : \sigma = M\)
when \(f\) is not free in \(M\). We will write \(\text{let } x : \tau = M \text{ in } N\) for
\(\lambda x : \tau. N\). We sometimes omit type annotations—
as in \(\lambda x. M\) and \(\text{let } x = M \text{ in } N\)—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 Sec-


tion 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 \(\text{types}\). For instance, we will
to have to consider whether \(\lambda x : \text{int}. x\) is contextually equiva-
lent to \(\lambda x : \text{int}. x - 1\) at type \(\alpha = \text{int}\), where the implemen-
tation of abstract type \(\alpha\) is \(\text{int}\) in fact. But this clearly de-
pends on what values of type \(\alpha\) (or, more generally, what
values involving type \(\alpha\)) exist in the context: for instance, if
the only values of type \(\alpha\) are 2 in the left-hand world and 3 in
the right-hand world, then the equivalence does hold; howev-
er, if some integers \(i\) on the left and \(j\) on the right have type \(\alpha\)
where \(i \neq j - 1\), then it does not hold. In order to capture at
once all such values in the context involving type \(\alpha\), we con-
sider the equivalence of \(\text{multiple}\) pairs of values—annotated
with their types—such as \(\{(2, 3, \alpha), ((\lambda x : \text{int}. x), (\lambda x : \text{int}. x - 1), \alpha = \text{int})\}\) and
\(\{(i, j, \alpha), ((\lambda x : \text{int}. x), (\lambda x : \text{int}. x - 1),\alpha = \text{int})\}\).
Definition 2.1. A concretion environment $\Delta$ is a finite set of triples of the form $(\alpha, \sigma, \sigma')$ with $\sigma$ and $\sigma'$ closed and $(\alpha, \tau, \tau') \in \Delta$ leads to $\tau = \sigma \land \tau' = \sigma'$.

The intuition is that, under $\Delta$, abstract type $\alpha$ is implemented by concrete type $\sigma$ in the left-hand side and by another concrete type $\sigma'$ in the right-hand side (of an equivalence). For instance, in the example in Section 1, the concrete implementations of abstract type $\alpha$ were $\text{int}$ in the left-hand world and $\text{bool}$ in the right-hand world, so $\Delta = \{(\alpha, \text{int}, \text{bool})\}$. We write $\text{dom}(\Delta)$ for $\{\alpha_1, \ldots, \alpha_n\}$ when $\Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_n, \sigma_n, \sigma'_n)\}$ and write $\Delta_1 \cup \Delta_2$ for $\Delta_1 \cup \Delta_2$ when $\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset$.

Definition 2.2. A typed value relation $R$ is a (either finite or infinite) set of triples of the form $(V, V', \tau)$.

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

Definition 2.3. Let $\Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_n, \sigma_n, \sigma'_n)\}$. We write $\Delta \vdash R$ if, for any $(V, V', \tau) \in R$, we have $\vdash V : [\sigma_1 \rightarrow \tau] \land \vdash V' : [\sigma'_1 \rightarrow \tau]$. We write $(\Delta, R)^o$ for the relation

$$(\{[\overline{U} / \overline{y}] [\sigma_1 \rightarrow \tau] D, [\overline{U}' / \overline{y}] [\sigma'_1 \rightarrow \tau] D, \tau\} | \Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_n, \sigma_n, \sigma'_n)\}, (U_1, U'_1, \rho_1), \ldots, (U_n, U'_n, \rho_n) \in R, \alpha_1, \ldots, \alpha_n, y_1 : \rho_1, \ldots, y_n : \rho_n \vdash D : \tau).$$

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

Definition 2.5. Generalized contextual equivalence is the set $\equiv$ of all pairs $(\Delta, R)$ such that:

A. $\Delta \vdash R$.

B. For any $(M, M', \tau) \in (\Delta, R)^o$, 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 $\Delta$ is empty and each $R$ is a singleton. Conversely, the standard contextual equivalence is implied by the generalized one in the following sense: if $(V, V', \tau) \in R$ for some $(\Delta, R) \in \equiv$ where $V$, $V'$, and $\tau$ 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 $\tau$. See also Section 6 for discussions on non-values and open terms.

We write

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

for

$$(\Delta, \{V_1, V'_1, \tau_1\}, (V_2, V'_2, \tau_2), \ldots) \in \equiv.$$  

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

The following properties follow immediately from the definition above.

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

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

Corollary 2.7 (Symmetry). If

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

then

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

Corollary 2.8 (Transitivity). If

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

and

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

then

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

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 $\Delta = \{\alpha, \text{int}, \text{int}\}$. Then we have:

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

More generally,

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

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

Example 2.10. Let $\Delta = \{\alpha, \text{int}, \text{bool}\}$. We have

$\Delta \vdash 1, (\lambda x:\text{int}.x = 0) \equiv \text{true}, (\lambda x:\text{bool}.\neg x) : \alpha, (\alpha \rightarrow \text{bool})$

$\Delta \vdash 1, (\lambda x:\text{int}.x = 0) \equiv \text{false}, (\lambda x:\text{bool}.x) : \alpha, (\alpha \rightarrow \text{bool})$

but

$\Delta \vdash 1, (\lambda x:\text{int}.x \equiv 0), 1, (\lambda x:\text{int}.x \equiv 0) \not\equiv \text{true}, (\lambda x:\text{bool}.\neg x), \text{false}, (\lambda x:\text{bool}.x) : \alpha, (\alpha \rightarrow \text{bool}), \alpha, (\alpha \rightarrow \text{bool})$.

The last example shows that, even if $(\Delta, R_1) \in \equiv$ and $(\Delta, R_2) \in \equiv$, the union $(\Delta, R_1 \cup R_2)$ 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 \( (v, w) \) with another pair \( (v', w') \), 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 \((\Delta, R)\) such that:

1. \( \Delta \vdash R \).
2. For each
   
   \[
   (\text{fix } f(x: \tau); \rho = M, \text{fix } f(x: \tau); \rho' = M', \tau \rightarrow \sigma) \in R
   \]
   
   and for any \((V, V', \tau) \in (\Delta, R)^\circ\), we have
   
   \[
   (\text{fix } f(x: \tau); \rho = M) V \downarrow \iff (\text{fix } f(x: \tau); \rho' = M') V' \downarrow.
   \]
   
   Furthermore, if \((\text{fix } f(x: \tau); \rho = M) V \downarrow W\) and \((\text{fix } f(x: \tau); \rho' = M') V' \downarrow W'\), then
   
   \[
   (\Delta, R \cup \{(W, W', \sigma)\}) \in X.
   \]

3. Let \( \Delta = \{(\alpha_1, \sigma_1, \sigma'_1), \ldots, (\alpha_m, \sigma_m, \sigma'_m)\} \). For each
   
   \[
   (\text{pack } \sigma, V \equiv \exists \alpha. \tau, \text{pack } \sigma', V' \equiv \exists \alpha'. \tau', \exists \alpha. \tau'') \in R,
   \]
   
   and for any \( \rho \) with \( \text{FTV}(\rho) \subseteq \text{dom}(\Delta) \), we have
   
   \[
   (\text{pack } \sigma, \text{pack } \sigma', \text{pack } \sigma', \text{pack } \sigma''; \rho) \in R
   \]
   
   and for any \( \rho \) with \( \text{FTV}(\rho) \subseteq \text{dom}(\Delta) \), we have
   
   \[
   (\text{pack } \sigma, \text{pack } \sigma', V \equiv \exists \alpha. \tau, \exists \alpha. \tau', \exists \alpha. \tau'') \in R
   \]
   
   or else \((\beta, \sigma, \sigma') \in \Delta \) and \((V, V', [\beta/\alpha]\tau'') \in R \) for some \( \beta \).

4. For each
   
   \[
   (\text{pack } \sigma, V \equiv \exists \alpha. \tau, \exists \alpha. \tau', \exists \alpha. \tau'') \in R,
   \]
   
   we have either
   
   \[
   (\Delta \cup \{(\alpha, \sigma, \sigma')\}, R \cup \{((V, V'), \tau'')\}) \in X,
   \]
   
   or else \((\beta, \sigma, \sigma') \in \Delta \) and \((V, V', [\beta/\alpha]\tau'') \in R \) for some \( \beta \).

5. For each \((V_1, \ldots, V_n), (V'_1, \ldots, V'_n), \tau_1 \times \ldots \times \tau_n \in R \)
   
   and for any \( 1 \leq i \leq n \), we have \((\Delta, R \cup \{(V_i, V'_i, \tau_i)\}) \in X \).

6. For each \((\text{in } (V), \text{in } (V'), \tau_1 + \ldots + \tau_n) \in R \)
   
   and for any \( i = j \) and \((\Delta, R \cup \{(V_i, V'_i, \tau_i)\}) \in X \).

7. For each \((\text{fold } (V), \text{fold } (V'), \mu_\alpha. \tau) \in R \)
   
   and for any \( \Delta, \mu_\alpha. \tau \in R \)
   
   we have \((\Delta, R \cup \{(V, V', [\mu_\alpha. \tau/\alpha]\tau)\}) \in X \).

As usual, \( \text{similarity, written } \sim \), is the largest bisimulation; it exists because the union of two bisimulations is again a bisimulation.

We write

\[
\Delta \vdash V_1, \ldots, V_n \ X \ V'_1, \ldots, V'_n : \tau_1, \ldots, \tau_n
\]

for

\[
(\Delta, \{V_1, V'_1, \tau_1), \ldots, (V_n, V'_n, \tau_n\}) \in X.
\]

We also write \( \Delta \vdash V X_R V' : \tau \) for \((V, V', \tau) \in R \) with \((\Delta, R) \in X \).

Intuitively, it can be read: values \( V \) and \( V' \) of type \( \tau \) with concretion environment \( \Delta \) 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 \( \Delta \). 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 \( (\text{fix } f(x: \tau); \rho = M, \text{fix } f(x: \tau'); \rho' = M') \) to some arguments \( V \) and \( V' \). To make up these arguments, the context can make use of any values it already knows \((U, U')\) 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 \((U/\overline{y})[\overline{\pi}/\overline{\alpha}]D \) and \((U'/\overline{y})[\overline{\pi}/\overline{\alpha}]D' \) 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 \( (U/\overline{y})[\overline{\pi}/\overline{\alpha}]D \) and \((U'/\overline{y})[\overline{\pi}/\overline{\alpha}]D' \) evaluate to \( \text{values} \) 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.\(^3\)

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

Condition 4 is for packages defining an abstract type \( \alpha \). 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 \( \beta \) whose implementations coincide with \( \alpha \)’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 \( \text{open} \), as mentioned in the introduction.

\(^3\)Another technical point may deserve mentioning here: instead of \((\Delta, R \cup \{(W, W', \sigma)\}) \in X \), we could require \((W, W', \sigma) \in R \) to reduce the number of \( \mathcal{R} \)s required to be in \( X \) by “predicting” the increase of contexts’ knowledge \( \text{a priori} \).

We rejected this alternative for the sake of uniformity with Condition 4, which anyway requires the concretion environment \( \Delta \) to be extended. This decision does not make it difficult to construct a bisimulation, as we will see soon in the examples.
Consider the following simple packages

4.1 Warm-Up

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 \( \lambda \)-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.

4.2 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 \( \exists \alpha. \tau \).

\[
\begin{align*}
U &= \text{pack (real \times real), (id, id, cmul) as } \exists \alpha. \tau \\
U' &= \text{pack (real \times real), (ctop, ptop, pmul) as } \exists \alpha. \tau \\
\tau &= (\text{real \times real \rightarrow } \alpha) \times (\alpha \rightarrow \text{real \times real}) \times (\alpha \rightarrow \alpha)
\end{align*}
\]

\[
\begin{align*}
id &= \lambda \alpha : \text{real} \times \text{real}.
\text{c}
\text{mul} &= \lambda \alpha_1 : \text{real} \times \text{real}. \lambda \alpha_2 : \text{real} \times \text{real}.
\langle \#^2(\alpha_1) \times (\#^2(\alpha_2)), \#^2(\alpha_1) \times \#^2(\alpha_2) \rangle
\end{align*}
\]

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 \( X = \{ (\Delta, R) \} \) where

\[
\Delta = \{ (\alpha, \text{real} \times \text{real} \times \text{real} \times \text{real}) \}
\]

\[
R = \{ (U, U', \tau), \}
\]

\[
\langle (id, id, \text{cmul}), (\text{ctop}, \text{ptoc}, \text{pmul}), \tau \rangle,
\langle id, \text{ctop, real} \times \text{real} \rightarrow \alpha \rangle,
\langle id, \text{ptoc, } \alpha \rightarrow \text{real} \times \text{real} \rangle,
\langle \text{cmul, pmul, } \alpha \rightarrow \alpha \rightarrow \alpha \rangle
\]

\[
\cup \{ (v, w, \alpha) | w = \{ r, t \}, \langle r \times \cos(t), r \times \sin(t) \rangle \Downarrow v, \}
\]

\[
\{ r \geq 0 \}
\]

\[
\{ (c, c, \text{real} \times \text{real}) | \vdash c : \text{real} \times \text{real} \}
\]

\[
\cup \{ (r, r, \text{real}) | \vdash r : \text{real} \}
\]

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.)

\[
\begin{align*}
U &= \lambda y: \text{int}. M \\
u' &= \lambda y: \text{int}. M' \\
M &= \text{pack int, } (y, \lambda x: \text{int}. x) \text{ as } \tau \\
M' &= \text{pack int, } (y + 1, \lambda x: \text{int}. x - 1) \text{ as } \tau \\
\tau &= \exists a. \alpha \times (\alpha \rightarrow \text{int})
\end{align*}
\]

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

\[
X = \{ (\Delta, R) \mid \Delta = \{ (\beta_i, \text{int} \text{. int}) \mid -n \leq i \leq n \}, \\
R \subseteq \bigcup_{-n \leq i \leq n} R_i, \\
n = 0, 1, 2, \ldots \}
\]

\[
R_i = \{ (U, U', \text{int} \rightarrow \tau), \\
(i/y)M, (i/y)M', \tau), \\
(i, \lambda x: \text{int}. x, (i + 1, \lambda x: \text{int}. x - 1, \beta_i \rightarrow \text{int}), \\
(i, i + 1, \beta), \\
(\lambda x: \text{int}. x, \lambda x: \text{int}. x - 1, \beta_i \rightarrow \text{int}), \\
(i, i, \text{int}) \}
\]

The generativity of \( U \) and \( U' \) is given a simple account by having a different abstract type \( \alpha \) for each instantation of \( U \) and \( U' \) with \( y = i \).

The inclusion of all \( R \subseteq \bigcup_{-n \leq i \leq n} R_i \) in the definition of \( X \) simplifies the definition of this bisimulation; although it admits some \( R_i \)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 \( \text{st} \)) 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.

\[
\begin{align*}
\tau &= \exists \text{st}. \sigma \\
\sigma &= \mu \text{self}. \text{st} \times (\text{st} \rightarrow \rho) \\
\rho &= \text{self} \times (\text{self} \rightarrow \text{bool})
\end{align*}
\]

\[
C = \text{pack int, fold}((0, M)) \text{ as } \tau
\]

\[\text{C'} = \text{pack int, fold}((0, M')) \text{ as } \tau\]

\[
M = \text{fix } f(s: \text{int}): [\text{int} / \text{st}][\sigma / \text{self}] = \\
(\text{fold}(s + 1, f), \\
\lambda c: [\text{int} / \text{st}]. \sigma. (s ^{\text{int}} \#_1(\text{unfold}(c)))), \\
M' = \text{fix } f(s: \text{int}): [\text{int} / \text{st}][\sigma / \text{self}] = \\
(\text{fold}(s - 1, f), \\
\lambda c: [\text{int} / \text{st}]. \sigma. (s ^{\text{int}} \#_1(\text{unfold}(c))))
\]

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

\[
\Delta = \{ (\text{st}, \text{int} \text{. int}) \} \\
R = \{ (C, C', \tau), \\
(\text{fold}((n, M)), \text{fold}((-n, M')), \sigma), \\
((n, M), (-n, M'), \text{st} \times (\text{st} \rightarrow [\sigma / \text{self}])), \\
(n, -n, \text{st}), \\
(M, M', \text{st} \rightarrow [\sigma / \text{self}]), \\
(\text{fold}((n + 1, M)), \\
\lambda c: [\text{int} / \text{st}]. \sigma. (n ^{\text{int}} \#_1(\text{unfold}(c)))), \\
(\text{fold}((-n - 1, M')), \\
\lambda c: [\text{int} / \text{st}]. \sigma. (-n ^{\text{int}} \#_1(\text{unfold}(c))), \\
\sigma \rightarrow \text{bool}), \\
(\lambda c: [\text{int} / \text{st}]. \sigma. (n ^{\text{int}} \#_1(\text{unfold}(c))), \\
\lambda c: [\text{int} / \text{st}]. \sigma. (-n ^{\text{int}} \#_1(\text{unfold}(c))), \\
\sigma \rightarrow \text{bool}), \\
(\text{true, true, bool}), \\
(\text{false, false, bool}) \mid n = 0, 1, 2, \ldots \}
\]

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.

\[
\begin{align*}
U &= \lambda f: \sigma. f[\text{int}](1, \lambda x: \text{int}. x \equiv 0) \\
u' &= \lambda f: \sigma. f[\text{bool}](\text{true}, \lambda x: \text{bool}. \neg x) \\
\sigma &= \forall \alpha. \alpha \times (\alpha \rightarrow \text{bool}) \rightarrow \text{unit}
\end{align*}
\]

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

\[
[U/x]C \Downarrow \iff [U'/x]C \Downarrow
\]

for any \( x: \sigma \rightarrow \text{unit} \vdash C : \tau \). Since

\[
\begin{align*}
[U/x]C &= [1, (\lambda x: \text{int}. x \equiv 0)/y, z][\text{int} / \beta]D_0 \\
[U'/x]C &= [\text{true}, (\lambda x: \text{bool}. \neg x)/y, z][\text{bool} / \beta]D_0
\end{align*}
\]
for $D_0 = [(\lambda f : \sigma. f[\beta(y,z)]/x)C$, it suffices to prove

$$[1, (\lambda x : \text{int}. x \equiv 0, y, z)[\text{int}/\beta]D \Downarrow \iff [\text{true}, (\lambda x : \text{bool}. x)/y, z][\text{bool}/\beta]D \Downarrow$$

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

$$\Delta = \{ (\beta, \text{int}. \text{bool}) \}$$

$$R = \{ (1, 1, \beta),$$

$$(\lambda x : \text{int}. x \equiv 0, \lambda x : \text{bool}. x, \beta \rightarrow \text{bool})$$

along with the soundness of bisimilarity in the next section.

5. SOUNDCNESS AND COMPLETENESS

We prove that bisimilarity coincides with contextual equivalence (in the generalized form presented in Section 2). That is, two values can be proven 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 $\Delta \vdash N \sim_R N' : \tau$ if $(N, N', \tau) \in (\Delta, R) \circ \emptyset$ and $(\Delta, R) \subseteq \sim$.

The intuition is that $\sim^R$ relates bisimilar values put in contexts.

**LEMMA 5.3 (Fundamental Property, Part I).** Suppose $\Delta_0 \vdash N \sim_R N' : \tau$. If $N \Downarrow W$ and $N' \Downarrow W'$, then $\Delta \vdash W \sim_R W' : \tau$ for some $\Delta \supseteq \Delta_0$ and $R \supseteq R_0$.

**Proof.** By induction on the derivation of $N \Downarrow W$. □

**LEMMA 5.4 (Fundamental Property, Part II).** If $\Delta_0 \vdash N \sim_R N' : \tau$ 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 $\sim^o$ 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 $\alpha$-conversion and may capture free variables). Then, the standard contextual equivalence

$$\alpha_1, \ldots, \alpha_m, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M \stackrel{\text{std}}{=} M' : \tau$$

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

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

$$\emptyset \vdash \lambda \alpha_1, \ldots, \lambda \alpha_m, \lambda x_1 : \tau_1, \ldots, \lambda x_n : \tau_n, M \sim \lambda \alpha_1, \ldots, \lambda \alpha_m, \lambda x_1 : \tau_1, \ldots, \lambda x_n : \tau_n, M'$$

for fresh free type/term variables at all, it suffices just to take $V = \lambda \pi. M$, $M' = \lambda \pi. M'$ for any type variable $\alpha$. 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 $\emptyset \vdash V \sim V'$ for any $\pi, \pi' : \tau$. By the soundness of bisimulation, we have $\emptyset \vdash V \equiv V' : \forall \pi. \forall \pi'. V \rightarrow \tau$. Given any $K$ with $K \vdash K[M] : \text{unit}$ and $K \vdash K[M'] : \text{unit}$, take $C = K[z] \alpha_1, \ldots, \alpha_m, x_1 \ldots m$ 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 $\beta$-expanded version $(\lambda \pi. \lambda \pi' : \tau, M)[\pi] \pi'$ should behave equivalently under any context. Since the free type/term
variables \( \pi \) and \( \sigma \) are to be substituted by some types/values during evaluation under a context, this “\( \beta \)-expansion” relation needs to be generalized to allow nesting. Thus, we define:

**Definition 6.1 (\( \beta \)-Expansion).** \( \Gamma \vdash M \leq M' : \pi \) is the smallest relation on pairs of \( \lambda \)-terms \( M \) and \( M' \) (annotated with a type environment \( \Gamma \) and a type \( \pi \)) satisfying all the rules in Figure 2.

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

Then, we can prove:

**Lemma 6.2.** For any

\[
\alpha_1, \ldots, \alpha_n, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M \leq M' : \pi,
\]

for any closed \( \alpha_i \)s and \( \pi \), and for any \( \vdash V \leq V' : [\pi/\tau] \)

for any closed \( x \)s, we have

\[
[\overline{\pi/\tau}] \overline{[\pi/\tau]} M \Downarrow \iff [\overline{\pi/\tau}] [\pi/\tau] M' \Downarrow.
\]

Furthermore, if \( [\overline{\pi/\tau}] [\pi/\tau] M \Downarrow W \) and \( [\overline{\pi/\tau}] [\pi/\tau] M' \Downarrow W' \), then \( \vdash W \leq W' : [\pi/\tau] \).

**Proof.** Straightforward induction on the derivation of \( \bar{\pi}, \bar{\tau} : \pi \vdash M \leq M' : \tau \).

**Theorem 6.3.** For any \( \bar{\pi}, \bar{\tau} : \pi \vdash M : \tau \) and \( \bar{\pi}, \bar{\tau} : \pi \vdash M' : \tau \), if \( \vdash \sum M : \pi \rightarrow {\tau} \), then \( \bar{\pi}, \bar{\tau} : \pi \vdash M' \equiv \sum M' : \tau \).

**Proof.** By the soundness of bisimulation, we have \( \vdash \sum \bar{\pi} \lambda \bar{\tau} : \tau \).

**Figure 2: \( \beta \)-Expansion**

\[
\begin{align*}
\frac{\Gamma \vdash M \leq M' : \rho \quad \{\pi\} \subseteq \text{dom}(\Gamma)}{\Gamma \vdash M \leq (\bar{\Lambda} \bar{\pi} \bar{\tau} : \pi, M') \bar{\pi} / \bar{\tau} : \rho} & \quad \text{(B-Exp)} \\
\frac{\Gamma \vdash f : \pi \rightarrow \sigma, x : \tau \vdash M \leq M' : \sigma}{\Gamma \vdash (\text{fix } f(x : \tau)) : \pi = \pi : \sigma \rightarrow \sigma} & \quad \text{(B-Fix)} \\
\frac{\Gamma \vdash M \leq M' : \tau \rightarrow \sigma}{\Gamma \vdash M \leq M' : \tau \rightarrow \sigma} & \quad \text{(B-App)} \\
\frac{\Gamma \vdash \alpha, \pi \leq \alpha, M' : \forall \alpha, \tau}{\Gamma \vdash \Lambda \alpha, \pi \leq \alpha, M' : \forall \alpha, \tau} & \quad \text{(B-TAbs)} \\
\frac{\Gamma \vdash M \leq M' : \exists \alpha, \tau}{\Gamma \vdash M \leq M' : \forall \alpha, \tau} & \quad \text{(B-TApp)} \\
\frac{\Gamma \vdash M \leq M' : [\tau/\alpha] \sigma \quad \text{FTV}(\tau) \subseteq \Gamma}{\Gamma \vdash M \leq M' : [\tau/\alpha] \sigma} & \quad \text{(B-Pack)} \\
\frac{\Gamma \vdash \text{pack } \tau, M \leq \alpha, \tau \quad \text{as } \alpha, \sigma \cdot \alpha}{\Gamma \vdash \text{pack } \tau, M \leq \alpha, \tau \cdot \alpha} & \quad \text{(B-Open)} \\
\frac{\Gamma \vdash M \leq M' : \exists \alpha, \tau}{\Gamma \vdash M \leq M' : \forall \alpha, \tau} & \quad \text{(B-Proj)} \\
\frac{\Gamma \vdash \text{in}_{\alpha} (M) \leq \text{in}_{\alpha} (M') : \tau}{\Gamma \vdash \text{in}_{\alpha} (M) \leq \text{in}_{\alpha} (M') : \tau} & \quad \text{(B-Inj)} \\
\frac{\Gamma \vdash \text{case } M \leq M' : \tau \quad \text{of in } \alpha, \tau}{\Gamma \vdash \text{case } M \leq M' : \tau} & \quad \text{(B-Case)} \\
\frac{\Gamma \vdash \text{fold}(M) \leq \text{fold}(M') : \mu \cdot \tau}{\Gamma \vdash \text{fold}(M) \leq \text{fold}(M') : \mu \cdot \tau} & \quad \text{(B-Unfold)}
\end{align*}
\]
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.)

\[
\text{IntSet} = \text{pack intList, Nil, add, weigh as } \exists \alpha. \tau \\
\text{IntSet'} = \text{pack intTree, Lf, add', weigh' as } \exists \alpha. \tau \\
\tau = \alpha \times (\text{int} \rightarrow \alpha) \times ((\text{int} \rightarrow \text{real}) \rightarrow \alpha) \\
\text{add} = \lambda i: \text{IntSet}. \text{Cons}(i, s) \\
\text{add'} = \lambda i: \text{int}. \text{fix } \lambda x: \text{intTree}: \text{IntTree} = \\
\quad \text{case } s \text{ of Lf } \Rightarrow \text{Nd}(i, Lf, Lf) \\
\quad \quad \| \text{Nd}(j, s_1, s_2) \Rightarrow \text{if } i < j \text{ then } \text{Nd}(j, f s_1, s_2) \\
\quad \quad \quad \text{else } \text{Nd}(j, s_1, f s_2) \\
\text{weigh} = \lambda g: \text{int} \rightarrow \text{real}. \text{fix } \lambda x: \text{intList}: \text{real} = \\
\quad \text{case } s \text{ of Nil } \Rightarrow 0 \| \text{Cons}(j, s_0) \Rightarrow g j + f s_0 \\
\text{weigh'} = \lambda g: \text{int} \rightarrow \text{real}. \text{fix } \lambda x: \text{intTree}: \text{real} = \\
\quad \text{case } s \text{ of Lf } \Rightarrow 0 \| \text{Nd}(j, s_1, s_2) \Rightarrow g j + f s_1 + f s_2
\]

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 \( \text{weigh} \ V \ W \downarrow \iff \text{weigh'} \ V' W' \downarrow \) for a certain class of \( V, W, V', \) and \( W' \). In particular, \( V \) and \( V' \) can be of the forms \( \lambda \alpha: \text{int}. \ [\text{IntSet}/y] D \) and \( \lambda \alpha: \text{int}. \ [\text{IntSet'}/y] D \) for any \( D \) of appropriate type. Thus, because of the function application \( g j \) in \( \text{weigh} \) and \( \text{weigh'} \), we must prove

\[
[\text{IntSet}, j, y, z] D \downarrow \iff [\text{IntSet'}, j, y, z] D \downarrow
\]

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

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

\[2. \text{ Take any} \]

\[
(f \text{fix } f(x: \pi): \rho = M, f \text{fix } f(x: \pi') : \rho' = M', \tau \rightarrow \sigma) \in \mathcal{R} \\
\text{and any } (V, V', \tau) \in (\Delta, \mathcal{R})^o. \text{ Assume that, for any } \]

\[
N \sqsubseteq (f \text{fix } f(x: \pi) : \rho = M) V \\
N' \sqsubseteq (f \text{fix } f(x: \pi') : \rho' = M') V'
\]

with \( (N, N', \sigma) \in (\Delta_0, \mathcal{R}_0)^o \) and \( (\Delta_1, \mathcal{R}_1) \in X \), we have \( N \downarrow \iff N' \downarrow \). Assume furthermore that, if \( N \downarrow U \) and \( N' \downarrow U' \), then \( (U, U', \sigma) \in (\Delta_1, \mathcal{R}_1)^o \) 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

\[
(f \text{fix } f(x: \pi) : \rho = M) V \downarrow \iff (f \text{fix } f(x: \pi') : \rho' = M') V' \downarrow.
\]

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

Here, \( N_1 \sqsubseteq N_2 \) 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 \( \text{weigh} \) and \( \text{weigh'} \) 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 \( (\equiv \text{ is still a bisimulation because Condition 2'} \) 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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-calculus are also useful for proving parametricity properties \[36\], e.g., that all functions of type \( \forall \alpha. \alpha \rightarrow \alpha \) 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 \( \lambda \)-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 \( \lambda \)-calculus to prove contextual equivalence. He introduced the notion of \( \upharpoonright \upharpoonright \) 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 \( \lambda \)-calculus with recursive functions and universal types (and lists).

Pitts \[29\] also proposed syntactic relations for a variant of call-by-value polymorphic \( \lambda \)-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 re-
stricted 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 $\lambda$-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 typed bisimulations. 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 $\pi$-calculus. Pierce and Sangiorgi [28] developed a bisimulation proof technique for polymorphic $\pi$-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 $\pi$-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 $\pi$-calculus, 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 $\pi$-calculus. They proved soundness and completeness of their logical relations for one of the linear $\pi$-calculus, which directly corresponds to polymorphic $\lambda$-calculus (without recursion). They also proved full abstraction of the call-by-value and call-by-name encodings of the polymorphic $\lambda$-calculus to this version of linear $\pi$-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 $\lambda$-terms.

Bisimulations for cryptographic calculi. Various bisimulations [2, 1, 10, 11] have been proposed for extensions of $\pi$-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 $\lambda$-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 $\lambda$-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 $\lambda$-calculus with full universal, existential, and recursive types.

Although full automation is impossible because equivalence of $\lambda$-terms (with recursion) is undecidable, some mechanical support would be useful. The technique of “bisimulation up to $\top$” [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 $\pi$-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


