Generating Witness of Non-Bisimilarity for the pi-Calculus

Ki Yung Ahn
Nanyang Technological University
Singapore
yaki@ntu.edu.sg

Ross Horne
Nanyang Technological University
Singapore
rhorne@ntu.edu.sg

Alwen Tiu
Nanyang Technological University
Singapore
atiu@ntu.edu.sg

Abstract
In the logic programming paradigm, it is difficult to develop an elegant solution for generating distinguishing formulae that witness the failure of open-bisimilarity between two pi-calculus processes; this was unexpected because the semantics of the pi-calculus and open bisimulation have already been elegantly specified in higher-order logic programming systems. Our solution using Haskell defines the formulae generation as a tree transformation from the forest of all nondeterministic bisimulation steps to a pair of distinguishing formulae. Thanks to laziness in Haskell, only the necessary paths demanded by the tree transformation function are generated. Our work demonstrates that Haskell and its libraries provide an attractive platform for symbolically analyzing equivalence properties of labeled transition systems in an environment sensitive setting.

CCS Concepts → Theory of computation → Process calculi; Modal and temporal logics; Constraint and logic programming; Operational semantics; Program verification; Software and its engineering → Functional languages; Networks → Protocol testing and verification; Formal specifications;

Keywords process calculus, observational equivalence, labeled transition systems, open bisimulation, modal logic, dynamic logic, distinguishing formula, Haskell, lazy evaluation, name binding, constraint programming, nondeterministic programming

1 Introduction
The main idea of this paper is that Haskell and its libraries provide a great platform for analyzing behaviors of nondeterministic transition systems in a symbolic way. Our main contribution is identifying an interesting problem from process calculus and demonstrating its solution in Haskell that supports this idea. More specifically, we implement automatic generation of modal logic formulae for two non-open bisimilar processes in the π-calculus, which can be machine-checked to witness that the two processes are indeed distinct.

In this section, we give a brief background on the π-calculus, bisimulation, and its characterizing logic; discuss the motivating example; and summarize our contributions.

The π-calculus [22, 23] is a formal model of concurrency meant to capture a notion of mobile processes. The notion of names plays a central role in this formal model; communication channels are presented by names; mobility is represented by scoping of names and scope extraction of names. The latter is captured in the operational semantics via transitions that may send a restricted channel name, and thereby enlarging its scope. There are several bisimulation equivalences for the π-calculus, notably, early [23], late [23], and open [27] bisimilarities. Only the latter is a congruence and is of main interest in this paper.

Bisimulation equivalence can be alternatively characterized using modal logics. A modal logic is said to characterize a bisimilarity relation if whenever two processes are bisimilar then they satisfy the same set of assertions in that modal logic and vice versa. Such a characterization is useful for analyzing why bisimulation between two processes fails, since an explicit witness of non-bisimilarity, in the form of a modal logic formula (also called a distinguishing formula), can be constructed such that one process satisfies the formula while the other does not. Early and late bisimilarities can be characterized using fragments of Milner-Parrow-Walker (MPW) logic [24], and a characterization of open bisimilarity has been recently proposed by Ahn et al. [3] using a modal logic called OM. Our work can be seen as a companion of the latter, showing that the construction of the distinguishing formula described there can be effectively and naturally implemented in Haskell.

One main complication in implementing bisimulation checking for the π-calculus (and name passing calculi in general) is that the transition system that a process generates can have infinitely many states, so the traditional partition-refinement-based algorithm for computing bisimulation and distinguishing formulae [9] does not work. Instead, one needs to construct the state space ‘on-the-fly’, similar to that
satisfies it by taking the right branch. Therefore, they are not open bisimilar \((P \not\approx_o Q)\) due to the failure in (1).

A depth first search for bisimulation, scanning from left to right, only needs to traverse the first tree (1) to notice non-bisimilarity. Our existing bisimulation checker (prior to this work) is a higher-order logic program, which runs in this manner. However, the witness we want to generate contains extra information (wavy underlined), which are not found in (1) but in (3). Therefore, simply logging all the visited steps during a run of a bisimulation check is insufficient.

The extra information \(\sigma = [(x, y)]\) represents a substitution that unifies \(x\) and \(y\). The third tree (3) considers the leading step initiated by the subprocess \((x \leftrightarrow y) (\tau (\tau \emptyset))\), which can only make a step in a world (or environment) where \(x\) and \(y\) are equivalent. Our earlier implementation uses a logic programming language, relying on a representation of \(x\) and \(y\) as unifiable logic variables and on backtracking for non-determinism. However, it is difficult to access \(\sigma\) in this setting because \(\sigma\) resides inside the system state rather than being a first-class value. Access to logic variable substitutions since the definition of open bisimulation and the generation of distinguishing formulae require access to and manipulations of such substitutions. Moreover, the information is lost when backtracking to another branch, for instance, from (3) to (4).

On the other hand, it is very natural in Haskell to view all possible nondeterministic steps as tree structured data because of laziness. Once we are able to produce the trees in Figure 1 (Section 4), our problem reduces to a transformation from trees to formulae (Section 5). Thanks to laziness, only those nodes demanded by the tree transformation function will actually be computed. We also have constraints (i.e., substitutions) as first-class values with an overhead of being more explicit about substitutions compared to logic programming.

In order to produce the trees of bisimulation steps, we first need to define the syntax (Section 2.1) and semantics (Section 3) of the \(\pi\)-calculus in Haskell. We also need to define the syntax of our modal logic formulae (Section 2.2) for the return value of the tree transformation function. However, we do not need to implement the semantics of the logic because we can check the generated formulae with our existing formula satisfaction \((\models)\) checker.

Our contributions are summarized as follows:

- We identified a problem that generating certificates witnessing the failure of process equivalence checking is non-trivial in a logic programming setting (Figure 1), even though the equivalence property itself has been elegantly specified as a logic program.
- The crux of our solution is a tree transformation from the forest of all possible bisimulation steps to a pair of distinguishing formulae (Section 5). The definition of tree transformation (Figure 9) is clear and easy to understand because we are conceptually working on all
possible nondeterministic steps. Nevertheless, unnecessary computations are avoided by laziness.

- We demonstrate that the overhead of re-implementing the syntax (Section 2), labeled transition semantics (Section 3), and open bisimulation checker (Section 4) in Haskell, which we already had as a logic program, and then augmenting it to produce trees is relatively small. In fact, most of the source code, omitting repetitive symmetric cases, is laid out as figures (Figures 2, 4, 5, 6, and 8).

- Our implementation of generating distinguishing formulae is a pragmatic evidence that reassures our recent theoretical development [3] of the modal logic \( \Omega M \) being a characterizing logic for open bisimilarity (i.e., distinguishing formulae exists iff open bisimilar). In this paper, we define the syntax of \( \Omega M \) formulae in Haskell and explain their intuitive meanings (Section 2.2), and provide pointers to related work (Section 7).

We used lhs2TeX to format the paper from literate haskell scripts (https://github.com/kyagrd/hs-picalc-unbound-example).

2 Syntax

In this section, we define the syntax for the \( \pi \)-calculus and the modal logic, which characterizes open bisimilarity. Haskell definitions of the syntax for both are provided in the module \texttt{PiCalc} as illustrated in Figure 2.

Since we consider only the original version of the \( \pi \)-calculus with name passing, terms (\( Tm \)) that can be sent through channel names consist only of names. Processes (\( Pr \)) may contain bound names due to value passing and name restriction. In the Haskell definition, we define these name binding constructs with the generic binding scheme (\( Bind \)) from the unbound [35] library. We can construct a bound process (\( Pr_b \), i.e., \( Bind \ Nm \ Pr \)) by applying the binding operator (\( \lambda \)) to a name (\( Nm \)) that may be used in a process (\( Pr \)), i.e., (\( x_1 \ldots p \)) : \( Pr_b \) given \( x \equiv Nm \) and \( p \equiv Pr \). Intuitively, our Haskell expression (\( x \equiv p \)) corresponds to a lambda-term (\( \lambda x.p \)). Similarly, we define name bindings in the logic formulae (\( Form \)) with \( Form_b \) defined as \( Bind \ Nm \ Form \). We get a-equivalence and capture-avoiding substitutions over processes and formulae almost for free, with a few lines of instance declarations, thanks to the unbound library.

In addition to the binding operator (\( \lambda \)), we define some utility functions: (\( \equiv \)), \( inp \), and \( out \) are wrappers to the data constructors of \( Pr \), for example, \( out x y \equiv Out \ (V \ x) \ (V \ y) \); \( \tau \) and \( \tau \tau \) are shorthand names of example processes; \( conj \) and \( disj \) are wrappers of \( \land \) and \( \lor \) with obvious simplifications, for example, \( f \equiv conj \ [ \tau , f ] \); and \( unbind2 \) is a wrapper to the library function \( unbind2 \), which unbinds two bound structures by a common name, for example, \( (x, out x x \ 0, out x x \ 1) \equiv unbind2 \ (x_1, out x x \ 0, out x x \ 1) \ (y_1, out y y \ 1) \). There is of course a more basic library function \( unbind \) for a single bound structure, which is formatted as \( (\lambda) \) in this paper because it acts like an inverse of \( (\lambda) \).

module PiCalc where
import Unbound.LocallyNameless
type Nm = Name Tm
newtype Tm = V Nm deriving (Eq, Ord, Show)
data Pr = \emptyset | \tau \cdot Pr | Out Tm Tm Pr | In Tm Pr_b | (Tm \parallel Tm) Pr
      | Pr :\: Pr | Pr parallel Pr | V Pr_b deriving (Eq, Ord, Show)
type Pr_b = Bind Nm Pr
instance Eq Pr_b where (\( \equiv \)) = aeqBinders
instance Ord Pr_b where compare = acompare
data Act = Tm Tm deriving (Eq, Ord, Show)
data Act_b = Tm deriving (Eq, Ord, Show)
data Form = \emptyset | T | \land [Form] | \lor [Form]
           | \Diamond Act Form | \Box Act Form
instance Eq Form where (\( \equiv \)) = aeqBinders
instance Ord Form where compare = acompare
instance Alpha Tm; instance Alpha Act; instance Alpha Act_b
instance Alpha Pr; instance Alpha Form
instance Subst Tm Tm where isvar (V x) = Just (SubsetName x)
instance Subst Tm Act; instance Subst Tm Act_b
instance Subst Tm Pr; instance Subst Tm Form
inf" x y = (V x \parallel V y) ; inst = In \ o V ; out x y = Out (V x) (V y)
\tau = \tau \emptyset ; \tau \tau = \tau (\tau \emptyset )
conj = cn \ o filter (\( \# \tau ) \) where cn [ ] = \tau ; cn [f] = f ; cn fs = \land fs
disj = ds \ o filter (\( \# \land ) \) where ds [ ] = \land ; ds [f] = f ; ds fs = \lor fs
unbind2 b_1 \_ b_2 = do Just (x_1, p_1 , \ldots , p_n) \leftarrow unbind2 b_1 b_2
    return (x, p_1, p_2)

Figure 2. Syntax of the \( \pi \)-calculus and the modal logic \( \Omega M \).

As a convention, we use Haskell names suffixed by \( n \) to emphasize that those definitions are related to bound structures. Naming conventions for the values of other data types in Figure 2 are: \( x, y, z, \) and \( w \) for both terms (\( Tm \)) and names (\( Nm \)); \( v \) for terms (\( Tm \)); \( p \) and \( q \) for processes (\( Pr \)); \( b \) for bound processes (\( Pr_b \)); \( a \) and \( l \) for both free and bound actions (\( Act \) and \( Act_b \)); and \( f \) for formulae (\( Form \)).

In the following subsections, we explain further details of the finite \( \pi \)-calculus (Section 2.1) and the modal logic (Section 2.2) including the intuitive meanings of their syntax.

2.1 Finite \( \pi \)-Calculus

A process (\( Pr \)) in the finite \( \pi \)-calculus is either the \( \emptyset \) process, a \( \tau \)-prefixed process (\( \tau \ p \)), an input-prefixied process (\( In \ x \ (y_1, p) \)), an output-prefixied process (\( Out \ x y p \)), a parallel composition of processes (\( p \parallel q \)), a nondeterministic choice between processes (\( p \cdot q \)), a name-restricted process (\( V (x \ p) \)), or a match-prefixied process (\( (x \equiv y) p \)).
The operational semantics of the finite $\pi$-calculus is given in Figure 3. Here we follow a style of specification [19] of the $\pi$-calculus where the continuation of an input or a bound output transition is represented as an abstraction over processes.

The process $\theta$ is a terminated process so that it will never make any transitions. ($\tau \cdot p$) will make a (free) transition step evolving into $p$ labeled with (free) action $\tau :: Act$, that is, $\tau \cdot p \rightarrow p$. ($\tau \cdot p$) will make a step evolving into $p$ labeled with $\tau :: Act$ and produces a value $v$ on channel $x$, which can be consumed by another process expecting an input value on the same channel.

($\text{In } x \ (y \ \| p)$) can make a step evolving into $p$ once an input value is provided on channel $x$. When an input value $v :: Tm$ is provided on the channel, at some point in time, the process consumes the value and evolves to $((\gamma_x) \ p)$, which is a process where ($V \ x \ y$) inside $p$ are substituted by $v$. This concept of a conditional step described above can be understood as if it steps to a bound process $(y \ \| p) :: Pr_n$, waiting for an input value for $y$. It is called a bound step ($\Xi_q$) in contrast to the (free) step ($\Xi_{\Delta}$) for the $\tau$-prefix case. Bound steps are labeled by bound actions, which can viewed as partially applied actions.

($\text{In } x \ (y \ \| p)$) behaves as $p$ when $x$ is same as $y$. Otherwise, it cannot make any further steps.

($p \parallel q$) nondeterministically becomes either $p$ or $q$, and takes steps thereafter. Only the rules for choosing $p$ are illustrated in Figure 3 while the rules for choosing $q$ are omitted.

($p \parallel q$) has eight possible cases; modulo symmetry between $p$ and $q$, four. First, it may step to ($p' \parallel q$) with action $a$ when $p$ steps to $p'$ with the same action. Second, there is a bound step version of the first. Third, the two parallel processes can interact when $p$ steps to $p'$ with an output action $\uparrow x \ v$ and $q$ steps to $(y \ \| q')$ with an (bound) input action $\downarrow y \ x$ on the same channel. This interaction step is labeled with $\tau$ and the process evolves into $(p \parallel q')$. Forth (close scope-ext) is a bound interaction step similar to the third. The differences from the third is that there is a bounded output ($\downarrow y$) instead of a free output ($\uparrow$) and that the resulting process becomes restricted with the name $x$ from the output value ($V \ x$). The bound output (open scope-ext) is driven by name-restricted processes, as explained next.

The labeled transition rules of the finite $\pi$-calculus (symmetric cases for $\sigma$ and $\parallel$ are omitted).

![Figure 3](image-url)
We discuss implementations of the labeled transition rules (\(\rho\)) from Figure 3. There are two versions: the first implements free and bound steps \(p \rightarrow p'\) and \(p \rightarrow b\) respectively.

The type signatures of \(\rho\) and \(\rho_b\) indicates that freshness of names and nondeterminism are handled by a monadic computation that returns a pair of a (bound) action and a (bound) process. In this paper, you may simply consider \(\rho\) and \(\rho_b\) as returning a list of all possible pairs. For example, we can compute all the three possible next steps from the process \(\text{Out} (V x) (V y)\theta\|\text{Out} (V w) (V y)\theta\|\text{In} (V z) (y')\theta\) using ghci as follows:

\[
\begin{align*}
&\text{\texttt{Main> :type \ runFreshMT \ IdSubLTS\.one}} \\
&\text{\texttt{runFreshMT \ \ IdSubLTS\.one :: MonadPlus m \Rightarrow Pr \rightarrow \text{Act}, Pr}} \\
&\text{\texttt{Main> :type \ map \ id \ \ runFreshMT \ \ IdSubLTS\.one}} \\
&\text{\texttt{map id \ \ runFreshMT \ \ IdSubLTS\.one \ : \ \Pr \Rightarrow \text{Act}, Pr}} \\
&\text{\texttt{Main> \ let \ \ p = \ Out (V x) (V x)\theta \|\text{Out} (V y) (V y)\theta \| \text{In} (V z) (v')\theta}} \\
&\text{\texttt{Main> \ mapM_ \ pp \ \ runFreshMT \ \ IdSubLTS\.one \$ \ p}} \\
&\text{\texttt{\{\textit{V (V x) (V x)}, (\textit{V (V y) (V y)}) \| \text{Out} (V y) (V y)\theta \| \text{In} (V z) (v')\theta} \\
&\text{\texttt{\{\textit{V (V y) (V y)}, (\textit{Out} (V x) (V x)\theta \| \text{In} (V z) (v')\theta}\} \| (\textit{Out} (V z) (v')\theta)\} \| (\textit{Out} (V z) (v')\theta))}} \\
&\text{\texttt{Main> \ mapM_ \ pp \ \ runFreshMT \ \ IdSubLTS\.one \$ \ p}} \\
&\text{\texttt{\{\textit{V (V y) (V y)}, (\textit{Out} (V x) (V x)\theta \| \text{Out} (V y) (V y)\theta) \| \text{In} (V z) (v')\theta\}} \\
&\text{\texttt{\{\textit{V (V y) (V y)}, (\textit{Out} (V x) (V x)\theta \| \text{Out} (V y) (V y)\theta) \| \text{In} (V z) (v')\theta)\} \| (\textit{Out} (V z) (v')\theta)\} \| \text{Out} (V z) (v')\theta))}}
\end{align*}
\]

In principle, the possible worlds semantics could be implemented using \(\rho\) and \(\rho_b\) in this \(\text{IdSubLTS}\) module by brute force enumeration of all substitutions over the free names in the process. For instance, there are three free names \(x,y,z\) in the process \(p\) above. Enumerating all substitutions over 3 names amounts to considering all possible integer set partition of the 3 elements. Let us establish a 1-to-1 mapping of \(x\) to 0, \(y\) to 1, and \(z\) to 2. Then, a substitution that maps \(x\) and \(z\) to the same value but \(y\) to a different value corresponds to the partition \([[0,2],[1]]\) where 0 and 2 belong to the same equivalence class. In such a world, there is an additional possible step for \(p\) above, which is the interaction between \(\text{Out} (V x) (V y)\theta\) and \(\text{In} (V z) (y')\theta\) due to the unification of \(x\) and \(z\). More generally, we can generate all possible partitions, starting from the distinct partition \([[0],[1],[2]]\), by continually joining a pair of elements from different equivalence classes until all possible joining paths reaches \([[0,1,2]]\) where all elements are joined. Although this brute force approach is a terminating algorithm, the number of partition sets is exponential to the number of names [26].

Since the original development of open bisimulation, Sangiorgi [27] was well aware that enumerating all possible worlds is intractable and provided a more efficient set of transition rules, known as the symbolic transition semantics. We implement another version of \(\rho\) and \(\rho_b\) following the style of symbolic transition in the next subsection. Nevertheless, \(\rho\) and \(\rho_b\) in this subsection are still used in our implementation of open bisimulation, together with the symbolic version. We will explain why we use both versions to implement open bisimulation in Section 4.
module ldSubLTS where
import PCalc
import Control.Applicative
import Control.Monad
import Unbound.LocallyNameless hiding (empty)

one :: (Fresh m, Alternative m) ⇒ Pr → m (Act, Pr)
one (Out x y p) = return (↑ x y p)
one (σ p) = return (↑ (σ p))
one ((x ↷ y) p) | x ≡ y = one p
one (p ∩ q) = one p ∩ one q
one (p || q) = do (l, p') ← one p; return (l, p' || q)
   ∩ do (l', q') ← one q; return (l, p || q')
   ∩ do (l', b) ← ones b; (l', b) ← ones b
      case (l', b) of (b x, b x') | x ∋ x' → close
      ∩ do (y, q', p') ← unbind2 b p b q
          return (↑ (σ, V (y, p' || q')))
      ∩ do (y, q', p') ← unbind2 b p b q
          return (↑ (σ, V (y, p' || q')))
      ∩ do (↑ x, y) ← one p; (v, y) ← ones p
          guard $ x ≡ y → empty
          return (↑ (σ, V (y, p' || q')))
      ∩ do (↑ x, y) ← one p; (↑ x, y) ← ones p
          guard $ x ≡ y → empty
          return (↑ (σ, V (y, p' || q')))
      ∩ do (↑ x, y) ← one p; (↑ x, y) ← ones p
          guard $ x ≡ y → empty
          return (↑ (σ, V (y, p' || q')))

one _ = empty

ones :: (Fresh m, Alternative m) ⇒ Pr → m (Act, Pr)
one (In x p) = return (ψ x p)
one ((x ↷ y) p) | x ≡ y = ones p
one (p ∩ q) = do (l, p') ← ones p; return (l, p' || q')
   ∩ do (l, q') ← one q; return (l, p || q')
   ∩ do (l, b) ← ones b; (l, b) ← ones b
      case (l, b) of (b x, b x') | x ∋ x' → close
      ∩ do (y, q, p) ← unbind2 b p b q
          return (↑ (σ, V (y, p' || q')))
      ∩ do (y, q, p) ← unbind2 b p b q
          return (↑ (σ, V (y, p' || q')))
      ∩ do (↑ x, y) ← one p; (↑ x, y) ← ones p
          guard $ x ≡ y → empty
          return (↑ (σ, V (y, p' || q')))
3.2 Labeled Transition Steps over Possible Worlds

The key idea behind the symbolic transition is that it is not worth considering every single difference between worlds. For example, consider the process $p_1 \parallel \cdots \parallel p_n \parallel (y \leftrightarrow z)$ $\tau$ where $p_i = \text{Out}(V x_i)(V x_i)\emptyset$ for each $i \in \{1 \ldots n\}$. The only difference that matters is whether $y$ and $z$ are unified in another world so that it can make a $\tau$-step, which were not possible in the current world. Other details such as whether $x_i$ and $y$, $x_i$ and $z$, or $x_i$ and $x_j$ are unifiable.

A symbolic transition step collects necessary conditions, which are equality constraints over names in our case, for making further steps in possible worlds and keeps track of those constraints. Here is a run of a symbolic transition step for the same example we ran with the fixed world version:

```
import Data.Map.Strict (fromList, (!))

mkPartitionFromEqC :: EqC -> EqC
mkPartitionFromEqC = foldr union (\()\(())\)

part :: EqC -> EqC
part = foldr union (\()\(())\)

mkPartitionFromEqC :: EqC -> EqC
mkPartitionFromEqC = foldr union (\()\(())\)
```

Two more interactions steps are possible: one where $x$ and $z$ are unified and the other where $y$ and $z$ are unified.

The return types of `one` and `one_n` in Figure 5 reflect such characteristics of symbolic transition. For instance, one returns the equality constraint (EqC) along with the transition label (Act) and the process (Pr). Another difference from the fixed world version is that there is an additional context (Ctx) argument. The definitions of EqC and Ctx are provided in Figure 6 along with related helper functions. As a naming convention, we use $\sigma$ for equality constraints and $\Gamma$ for contexts. We follow through the definitions in Figure 6 explaining how they are used in the implementation symbolic transition steps in Figure 5 while pointing out the differences from the fixed world version in Figure 4 laid out side-by-side.

An equality constraint (EqC) is conceptually a set of name pairs represented as a list. Basic operations over EqC are single element insertion (\(\hat{\cup}\)) and union (\(\cup\)). These operations are used on the necessary constraints for the additional steps, which were not possible in the current world. Such additional steps may occur in match-prefixes, closing of scope extrusions, and interaction steps.

A context (Ctx) is a list of either universally ($\forall$) or nably ($\forall$) quantified names (Quan). We assume that names in a context must be distinct (i.e., no duplicates). When using the symbolic transition step (one $\Gamma p$), we assume that $p$ is closed by $\Gamma$, that is, (fv $p$) $\subseteq$ (quanzm $\emptyset$ $\Gamma$). Similarly, for (one $\Gamma b$), we assume that $b$ is closed by $\Gamma$.

Quantified names in a context appear in reversed order from how we usually write on paper as a mathematical notation. That is, $\forall x, \forall y, \forall z, \ldots$ would correspond to $[\forall z, \forall y, \forall x]$. This reversal of layout is typical for list representation of contexts where the most recently introduced name is added to the head of the list. Nabla quantified names must be fresh from all previously known names. Hence, $y$ may be unified with $z$ but never with $x$. A substitution $\sigma$ ‘respects’ $\Gamma$ when it obeys such nabla restrictions imposed by $\Gamma$. Otherwise, i.e., $\nabla (\sigma \text{ ‘respects’ } \Gamma)$, it is an impossible world, therefore, discarded by the guards involving the respect predicate in Figure 5. These are additional guards that were not present in the fixed world setting.

We use the helper function `subs` to build a substitution function ($\delta :: \text{Subst Tm} a \Rightarrow a \rightarrow a$) from the context ($\Gamma$) and equality constraints ($\sigma$). The substitution function ($\delta$) is used for testing name equivalence under the possible world given by $\sigma$ in the transition steps for the restricted process.
(V(x)\ p)). The name (in)equality test for the restricted process in Figure 4 are now tested as (in)equality modulo substitution in Figure 5. For instance, the equality tests against the restricted name (x) such as x \equiv x’ and V x \neq y for the restricted process in Figure 4 are replaced by x \equiv \sigma x’ and V x \neq \sigma y in Figure 5. We need not apply \sigma to the restricted name x, although it would be harmless, because of our particular scheme for computing substitutions using the helper function mkPartitionFromEqC, which is also used in the definition of the respects predicate discussed earlier.

### 3.3 Substitution modeled as Set Partitions

In mkPartitionFromEqC, we map names in \Gamma to integers in decreasing order so that most recently introduced names maps to larger values. For example, consider \Gamma = [\forall z, \forall y, \forall x], which represents the context \forall x, \forall y, \forall z,..., where x is mapped to 0, y to 1, and z to 2. We model substitutions as integer set partitions using the data-partition library and unification by its join operation (join). This merges equivalence classes of the joining elements (a.k.a., union-find algorithm).

Consider the substitution described by [(y, z)], which respects \Gamma, modeled by the partition part_1 = [0, 1, 2]. Also, consider [(x, y)], which does not respect \Gamma, modeled by part_2 = [0, 1, 2]. The representative of an equivalence class defined to be the minimal value. Then, we can decide whether a partition models a respectful substitution by examining (rep part n) :: Int for every n that is mapped from a nabla name. For instance, from y in our example. In the first partition, rep part_1 1 \equiv 1 is the same as the nabla mapped value. In the second partition, on the other hand, rep part_2 1 \equiv 0 is different from the nabla mapped value. This exactly captures the idea that a nabla quantified name only unifies with the names introduced later (larger values) but not with names introduced earlier (smaller values).

## 4 Open Bisimulation

In this section, we discuss the definition of simulation in Haskell to provide an understanding for the definition of bisimulation, which shares a similar structure but twice in length. Figure 8 illustrates two versions of the simulation definition. The first version sim :: Ctx \rightarrow Pr \rightarrow Pr \rightarrow Bool is the usual simulation checker that returns a boolean value, defined as a conjunction of the results from sim_. The second version sim’ is almost identical to sim_ except that it returns a forest that contains information about each simulation step. Similarly, we have two versions for bisimulation, bisim defined in terms of sim_ and bisim’ that returns a forest.

A process p is (openly) simulated by another process q, that is (sim \Gamma p q) where \Gamma = [\forall x | x \leftarrow fv(p, q)], when for every step from p to p’ there exists a step from q to q’ labeled with the same action in the same word such that (sim \Gamma p’ q’). Also, similarly for every bound step lead by p

\footnote{The function (and :: [Bool] \rightarrow Bool) implements “for every step” and the function (or :: [Bool] \rightarrow Bool) implements “there exists a step.”}

### Figure 7. Equational properties between fixed-world and symbolic transition steps where \Gamma is a closing context of p.

\footnote{Having bound step children share the same fresh variable makes it more convenient to generate the distinguishing formulae in Section 5.}
Generating Witness of Non-Bisimilarity for the pi-Calculus

Haskell'17, September 07–08, 2017, Oxford, UK

module OpenBisim where
import PiCalc; import Control.Applicative; import Control.Monad
import OpenLTS; import qualified IdSubLT; import Data.Tree
import Unbound.LocallyNameless hiding (empty)
data StepLog = One Ctx EqC Act Pr |
  Oneu Ctx EqC Actu Pru deriving (Eq, Ord, Show)
returnu log = return o Node (Left log) -- for the step on p’s side
returnr log = return o Node (Right log) -- for the step on q’s side

sim Γ p q = and $ sim_ Γ p q

sim_ :: Ctx → Pr → Pr → [Bool]
sim_ Γ p q = do (σ, r) ← runFreshMT (one Γ p); let σ = subs Γ σ
  let (lp, p') = σ r
  return o (or :: [Bool] → Bool) o runFreshMT $ do
    (lq, q') ← runSubsMT.one (σ q)
    guard $ lp ≡ lq
    return o (and :: [Bool] → Bool) $ sim_ Γ p’ q’

  do (σ, r) ← runFreshMT (oneu Γ p); let σ = subs Γ σ
  let (lp, p') = σ r
  return o (or :: [Bool] → Bool) $ runFreshMT $ do
    (lq, q') ← runSubsMT.oneu (σ q)
    guard $ lp ≡ lq
    return o (and :: [Bool] → Bool) $ sim_ Γ p’ q’


Figure 8. An implementation of the open simulation (sim) and its variant (sim’) producing a forest.

forest2df :: [Tree (Either StepLog StepLog)] → [(Form, Form)]
forest2df rs
  = do Node (Left (One _ _ a _ _)) [] ← rs
    let σ_a = subsMatchingAct a (right1s rs)
    return (preBase σ_a a, postBase σ_a a)
  ⋯

Figure 9. Generating distinguishing formulae from the forest produced by bisim'.
5 Distinguishing Formulae Generation

The distinguishing formulae generation is no more than a tree transformation. (Figure 9), which generates a pair of distinguishing formulae from the forest of rose trees produced by \((\text{bisim'}}\ T \ p \ q\). The first formula is satisfied by the left process \((p)\) but fails to be satisfied by the other. Likewise, the second formula is satisfied by the right process \((q)\) but not by the other. The tree transformation function \(\text{forest2df}\) returns a list \(\{(\text{Form}, \text{Form})\}\) because there can be more than one pair of such formulae for the given non-bisimilar processes. For bisimilar processes, \(\text{forest2df}\) returns the empty list. The definition of \(\text{forest2df}\) consists of eight \textbf{do}-blocks where the first four are base cases and the latter four are inductive cases. We only illustrate the cases lead by the left side \((p)\) while the cases lead by the right side \((q)\) are omitted in Figure 9.

It is a base case when the leading step has no matching following step. That is, the children following the leading step specified by the root label of the tree is an empty list, as you can observe from the beginning lines of the first and third \textbf{do}-blocks in Figure 9. The formula satisfied by the leading side is \((\circlearrowleft \ \emptyset \ (\emptyset \ a \ T)\) or \((\circlearrowleft \ \emptyset \ \emptyset \ (\emptyset \ a \ (w \ \cdot \ T))\), generated by \texttt{prebase} or \texttt{preBbase}, whose intuitive meaning is that the process can make a step labeled with \(a\) in the world given by \(\emptyset\). This formula clearly fails to be satisfied by the other side because there is no following step (i.e., step labeled with \(a\) from \(q\) in the \(\emptyset\)-world) for the base case. If there were only one world to consider, the formula for the other side would be \((\circlearrowleft \ a \ \cdot\) or \((\circlearrowleft \ a \ (w \ \cdot \ \cdot))\), meaning that the process cannot make a step labeled with \(a\). However, we must consider the possible worlds where such step exists for the following side. Such worlds \((\emptyset \ a)\) are collected from the sibling nodes of the leading step using the helper functions \texttt{subMatchingAct} and \texttt{subMatchingAct\_}. The formula satisfied by the following side is \((\circlearrowleft \ (\emptyset \ (\emptyset \ a \ T))\) or \((\circlearrowleft \ (\emptyset \ a \ (w \ \cdot \ T))\), generated by \texttt{postbase} or \texttt{postBbase}.

In an inductive case where the leading step from \(p\) to \(p'\) is matched by a following step \(q\) to \(q'\), we find a pair of distinguishing formulae for each pair of \(p'\) and \(q'\) at next step by recursively applying \texttt{forest2df} to all the grandchildren following the steps lead by \(p\), that is, \((\text{sequence} (\text{forest2df} \ (\text{rs}'))):: \{(\text{Form}, \text{Form})\}\). The this list should not be empty; otherwise it had either been a base case or it had been a forest generated from bisimilar processes. The collected the left biased formulae \((\text{dfs}_L)\) are used for constructing the distinguishing formula satisfied by the leading side in the fifth and seventh \textbf{do}-blocks in Figure 9, which is \((\circlearrowleft \ \emptyset \ (\emptyset \ a \ (\land \text{dfs}_L)))\) or \((\circlearrowleft \ \emptyset \ (\emptyset \ a \ (w \ \cdot \ \land \text{dfs}_L)))\) where \(w\) is fresh in \text{dfs}_L. Similarly, the right biased formulae \((\text{dfs}_R)\) are used for constructing the formula satisfied by the other side, which is \((\circlearrowleft \ (\emptyset \ (\emptyset \ a \ T \land \text{dfs}_R)))\) or \((\circlearrowleft \ (\emptyset \ (\emptyset \ a \ (w \ \cdot \ T \land \text{dfs}_R)))\)) where \(x\) corresponds to the context \(x'\) in Figure 8, which is the fresh variable extending the context. Because we made sure that the same variable is used to extend the context across all the following bound steps from a leading step, we simply need to select the first one, using some number of selector functions to go inside the list, retrieve the context from the root, and grab the name in the first quantifier of the context.

6 Discussions

We point out three advantages of using Haskell for our problem of generating distinguishing formulae (Section 6.1) and discuss further optimizations and extensions to our current implementation presented in this paper (Section 6.2).

6.1 Advantages of using Haskell

First, having a well-tailored generic name binding library such as unbound [35] saves a great amount of effort on tedious boilerplate code for keeping track of freshness, collecting free variables, and capture-avoiding substitutions. Due to value passing and name restriction in the \(\pi\)-calculus, frequent management of name bindings is inevitable in implementations involving the \(\pi\)-calculus.

Second, lazy evaluation and monadic encoding of nondeterminism in Haskell makes it natural to view control flow as data. Distinguishing formula generation can be defined as a tree transformation (\texttt{forest2df}) over the forest of rose trees lazily produced from \texttt{bisim'}. Only a small amount of change was needed to abstract the control flow of computing a boolean by \texttt{bisim} into data production by \texttt{bisim'}.

The forest produced by \texttt{bisim'} is all possible traces of bisimulation steps. The control flow of \texttt{bisim} for non-bisimilar processes corresponds to a depth-first search traversal until the return value is determined to be \texttt{False}. For bisimilar processes, \texttt{bisim} returns \texttt{True} after the exhaustive traversal.

The traversal during the formulae generation does not exactly match the pattern of traversal by \texttt{bisim}. Alongside the depth-first search, there are traversals across the siblings of the leading step to collect \(\emptyset\) in Figure 9.

For process calculi with less sophisticated semantics, it is possible to log a run of bisimulation check and construct distinguishing formulae using the information from those visited nodes only. In contrast, we need additional information on other possible worlds, which come from the nodes not necessarily visited by \texttt{bisim}.

Third, constraints are first-class values in constraint programming using Haskell. We construct distinguishing formulae using substitutions (i.e., equality constraints) as values (e.g., \(\emptyset\) and \(\emptyset\) in Figure 9). This is not quite well supported in (constraint) logic programming. For example, consider a Prolog code fragment, \(\cdots \cdot \cdot \cdot \cdot X = Y, \zeta Z = W, \zeta \cdot \cdot \cdot \cdot\), and let \(\sigma_1, \sigma_2,\) and \(\sigma_3\) be the equality constraints at the points marked by \(\zeta, \zeta,\) and \(\zeta\). We understand that it should be \(\sigma_1 \cup \{X = Y\} \equiv \sigma_2\) and \(\sigma_2 \cup \{Z = W\} \equiv \sigma_3\). However, \(\sigma_1, \sigma_2,\) and \(\sigma_3\) are not values in a logic programming language.

The labeled transition semantics and open bisimulation can be elegantly specified in higher-order logic programming systems [30]; for those purposes, it fits better than functional programming. However, generating certificates regarding
open bisimulation requires the ability that amounts to accessing meta-level properties of logic programs (e.g., substitutions) across nondeterministic execution paths, where it is preferable to have constraints as first-class values.

6.2 Further Optimizations and Extensions

One obvious optimization to our current implementation is to represent the equality constraints as partitions instead of computing partitions from the list of name pairs on the fly every time we need a substitution function.

We can enrich the term structure to model applied variants of $\pi$-calculi by supporting unification in a more general setting [20] and constraints other than the equalities solvable by unification. When the constraints become more complex, we can no longer model them as integer set partitions. In addition, it would be better to abstract constraint handling with another layer of monad (e.g., state monad). In this work, we did not bother to abstract the constraints in a monad because they were very simple equalities over names only.

To handle infinite processes (or finite but quite large ones) effectively, we should consider using more sophisticated search strategies. For this, we would need to replace the list monad with a custom monad equipped with better control over traversing the paths of nondeterministic computation. Thanks to the monadic abstraction, the definitions could remain mostly the same and only their type signatures would be modified to use the custom monad.

Memoization or tabling is a well-known optimization technique to avoid repetitive computation by storing results of computations associated with their input arguments. When we have infinite processes, this is no longer an optional optimization but a means to implement the coinductive definition of bisimulation over possibly infinite transition paths. Parallel computing may also help to improve scalability of traversing over large space of possible transitions but memoization could raise additional concurrency issues [5, 36].

7 Related Work

In this section, we discuss nondeterministic programming using monads (Section 7.1), bisimulation and its characterizing logic (Section 7.2), and related tools (Section 7.3).

7.1 Monadic encodings of Nondeterminism

Wadler [34] modeled nondeterminism with a list monad. Monadic encodings of more sophisticated features involving nondeterminism (e.g., [12, 15, 17]) have been developed and applied to various domains (e.g., [8, 28]) afterwards. Fischer et al. [12] developed a custom monadic datatype for lazy nondeterministic programming. Their motivation was to find a way combine three desirable features found in functional logic programming [13, 18, 32] and probabilistic programming [11, 16] - laziness, sharing (memoization), and nondeterminism, which are known to be tricky to combine in functional programming. Having two versions of transitions (Figures 4 and 5) in our implementation was to avoid an instance of undesirable side effects from this trickiness – naive combination of laziness and nondeterminism causing needless traversals. We expect our code duplication can be lifted by adopting such a custom nondeterministic monad.

7.2 Bisimulation and its Characterizing Logic

Hennessy–Milner Logic (HML) [14] is a classical characterizing logic for the Calculus of Communicating Systems (CCS) [21]. The duality between diamond and box modalities related by negation (i.e., $[a]f \equiv \neg\langle a \rangle \neg f$ and $\langle a \rangle f \equiv \neg [a](\neg f)$) holds in HML. This duality continues to hold in the characterizing logics for early and late bisimulation for the $\pi$-calculus [24]. Presence of this duality makes it easy to obtain the distinguishing formula for the opposite side by negation. There have been attempts [25, 30] on developing a characterizing logic for open bisimulation, but it has not been correctly established until our recent development of OM [3]. Our logic OM captures the intuitionistic nature of the open semantics, which has a natural possible worlds interpretation typically found in Kripke-style model of intuitionistic logic. The classical duality between diamond and box modalities no longer hold in OM. This is why we needed to keep track of pairs of formulae for both sides during our distinguishing formulae generation in Section 5.

7.3 Tools for Checking Process Equivalence

There are various existing tools that implement bisimulation or other equivalence checking for variants and extensions of the $\pi$-calculus. None of these tools generate distinguishing formulae for open bisimulation. The Mobility Workbench [33] is a tool for the $\tau$-calculus with features including open bisimulation checking. It is developed using an old version of SML/NJ. SPEC [31] is security protocol verifier based on open bisimulation checking [29] for the spi-calculus [2]. The core of SPEC including open bisimulation checking is specified by higher-order logic predicates in Bedwyr [4] and the user interface is implemented in OCaml. ProVerif [6] is another security protocol verifier based on the applied $\pi$-calculus [1]. It implements a sound approximation of observational equivalence, but not bisimulation.

There are few tools using Haskell for process equivalence. Most relevant work to our knowledge is the symbolic (early) bisimulation for LOTOS [7], which is a message passing process algebra similar to value-passing variant of CCS but with distinct features including multi-way synchronization. Although not for equivalence checking, de Renzy-Martin [10] implemented an interpreter that can be used as a playground for executing applied $\pi$-calculus processes to communicate with actual HTTP servers and clients over the internet.
8 Conclusion

We implemented automatic generation of modal logic formula that witness non-open bisimilarity of processes in the $\pi$-calculus. These formulae can serve as certificates of process inequivalence, which can be validated with an existing satisfaction checker for the modal logic $OM$. Our implementation enjoys the benefits of laziness, nondeterministic monad, and first-class constraints; which are well known benefits of constraint programming in Haskell. Laziness and monadic abstraction allows us to view all possible control flow of nondeterminism as lazy generated trees, so that we can define formula generation as a tree transformation. First-class constraints allows us to manage information of possible worlds. Our problem setting particularly well highlights these benefits because we needed additional information outside the control flow of a usual bisimulation check. Our application of Haskell to distinguishing formula generation demonstrates that Haskell and its ecosystem are equipped with attractive features for analyzing equivalence properties of labeled transition systems in an environment sensitive (or knowledge aware) setting.

Acknowledgments

This material is based upon work supported by the Ministry of Education, Singapore under Grant No. MOE2014-T2-2-076.

References

[1] Martín Abadi and Cédric Fournet. 2001. Mobile Values, New Names, and Secure Communication. SIGPLAN Not. 36, 3 (Jan. 2001), 104–115.
[2] Martín Abadi and Andrew D. Gordon. 1997. A Calculus for Cryptographic Protocols: The Spi Calculus. In CCS ’97. ACM, New York, NY, USA, 36–47.
[3] Ki Yung Ahn, Ross Horne, and Alwen Tiu. 2017. A Characterisation of Open Bisimulation using an Intuitionistic Modal Logic. CoRR abs/1701.05324 (2017). http://arxiv.org/abs/1701.05324
[4] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. 2007. The Bedwyr System for Model Checking over Syntactic Expressions. In CADE-21. Springer-Verlag, 391–397.
[5] Lars Bergstrom. 2013. Parallel Functional Programming with Mutable State. Ph.D. Dissertation. University of Chicago.
[6] Bruno Blanchet and Cédric Fournet. 2005. Automated Verification of Selected Equivalences for Security Protocols. In LICS ’05. IEEE Computer Society, Washington, DC, USA, 331–340.
[7] Muffy Calder and Carron Shankland. 2001. A Symbolic Semantics and Bisimulation for Full LOTOS. In FORTE ’91. Klwer, B.V., 185–200.
[8] Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C.R. Lock. 1998. GOFFIN: Higher-order functions meet concurrent constraints. Science of Computer Programming 30, 1 (1998), 157 – 199.
[9] Rance Cleaveland. 1991. On Automatically Distinguishing Inequivalent Processes. In Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990. Vol. 3. DIMACS/AMS, 463–476.
[10] William de Renzy-Martin. 2014. The Applied $\pi$-calculus Interpreter. 3rd year computing individual project. Imperial College London, London, UK. http://hackage.haskell.org/package/pi-calculus
[11] Martin Erwig and Steve Kollmansberger. 2006. FUNCTIONAL PEARLS: Probabilistic Functional Programming in Haskell. J. Funct. Program. 16, 1 (Jan. 2006), 21–34.
[12] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2009. Purely Functional Lazy Non-deterministic Programming. In ICFP ’09. ACM, New York, NY, USA, 11–22.
[13] Michael Hanus. 2011. Lazy and Enforceable Assertions for Functional Logic Programs. In WLPF ’10. Springer-Verlag, 84–100.
[14] Matthew Hennessy and Robin Milner. 1980. On Observing Nondeterminism and Concurrency. In Proceedings of the 7th Colloquium on Automata, Languages and Programming. Springer-Verlag.
[15] Ralf Hinze. 2000. Deriving Backtracking Monad Transformers. In ICFP ’00. ACM, New York, NY, USA, 186–197.
[16] Oleg Kiselyov. 2016. Probabilistic Programming Language and its Incremental Evaluation. In APLAS ’16 (LNCS). Vol. 10017. 357–376.
[17] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In ICFP ’05. ACM, New York, NY, USA, 192–203.
[18] Francisco Javier López-Fraguas and Jaime Sánchez Hernández. 1999. TOY: A Multiparadigm Declarative System. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA ’99). Springer-Verlag, 244–247.
[19] Raymond McDowell, Dale Miller, and Catuscia Palamidessi. 1996. Encoding Transition Systems in Sequent Calculus. Electr. Notes Theor. Comput. Sci. 3 (1996), 138–152.
[20] Dale Miller. 1992. Unification Under a Mixed Prefix. J. Symb. Comput. 14, 4 (Oct. 1992), 321–355.
[21] Robin Milner. 1982. A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
[22] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes. Inf. Comput. 100, 1 (Sept. 1992), 1–40.
[23] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, II. Inf. Comput. 100, 2 (Sept. 1992), 41–77.
[24] Robin Milner, Joachim Parrow, and David Walker. 1993. Modal logos for mobile processes. Theoretical Computer Science 114, 1 (1993), 149 – 171. DOI: http://dx.doi.org/10.1016/0304-3975(93)90156-N
[25] Joachim Parrow, Johannes Borgström, Lars-Hennessynrik Eriksson, Ramunas Gutkovas, and Tjark Weber. 2015. Modal Logics for Nominal Transition Systems. In CONCUR ’15 (LIPIcs). Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 198–211.
[26] Gian-Carlo Rota. 1964. The Number of Partitions of a Set. The American Mathematical Monthly 71, 5 (May 1964), 498–504.
[27] Davide Sangiorgi. 1996. A theory of bisimulation for the $\pi$-calculus. Acta Informatica 33, 1 (1996), 69–97.
[28] Tom Schrijvers, Peter Stuckey, and Philip Wadler. 2009. Monadic Constraint Programming. J. Funct. Program. 19, 6 (2009), 663–697.
[29] Alwen Tiu and Jermy Dawson. 2010. Automating Open Bisimulation Checking for the Spi Calculus. In 23rd IEEE Computer Security Foundations Symposium (CSF ’10). IEEE Computer Society, 307–321.
[30] Alwen Tiu and Dale Miller. 2010. Proof Search Specifications of Bisimulation and Modal Logics for the $\pi$-calculus. ACM Trans. Comput. Logic 11, 2, Article 13 (Jan. 2010), 35 pages.
[31] Alwen Tiu, Nam Nguyen, and Ross Horne. 2016. SPEC: An Equivalence Checker for Security Protocols. In APLAS ’16 (LNCS). Vol. 10017. 87–95. http://dx.doi.org/10.1007/978-3-319-47958-3
[32] Andrew Tolmach, Sergio Antoy, and Marius Nita. 2004. Implementing Constraint Programming. J. Symb. Comput. 34, 1 (July 2004), 69–92.
[33] Andrew Tolmach, Sergio Antoy, and Marius Nita. 2004. Implementing Functional Logic Languages Using Multiple Threads and Stores. In ICFP ’04. ACM, New York, NY, USA, 90–102.
[34] Bjorn Victor and Faron Moller. 1994. The Mobility Workbench - A Tool for the pi-Calculus. In CAV ’94. Springer-Verlag, 428–440.
[35] Philip Wadler. 1985. How to Replace Failure by a List of Successes. In Proc. of Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag New York, Inc., 113–128.
[36] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. 2011. Binders Unbound. In ICFP ’11. ACM, New York, NY, USA, 333–345.
[37] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. 2009. Partial Memoization of Concurrency and Communication. In ICFP ’09. ACM, New York, NY, USA, 161–172.