Checking the equivalence of context-free session types

Bernardo Almeida
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
balmeida@lasige.di.fc.ul.pt

Andreia Mordido
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
afmordido@fc.ul.pt

Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
vmvasconcelos@fc.ul.pt

Abstract

We present an algorithm to decide the equivalence of context-free session types, practical to the point of being incorporated in a compiler. We prove its soundness and completeness. We also study different optimizations that improve the running time and memory allocated in more than 12,000,000%.

2012 ACM Subject Classification CCS → Theory of computation → Logic → Type theory; CCS → Theory of computation → Formal languages and automata theory → Grammars and context-free languages

Keywords and phrases Types, Type equivalence, Bisimulation, Algorithm

Digital Object Identifier 10.4230/LIPIcs...

Funding This work was supported by FCT through project Confident (PTDC/EEI-CTP/ 4503/2014) and the LASIGE Research Unit (UID/CEC/00408/2019).
Andreia Mordido: https://orcid.org/0000-0002-1547-0692
Vasco T. Vasconcelos: https://orcid.org/0000-0002-9539-8861

1 Introduction

Session types enhance the expressivity of traditional types for programming languages by enabling describing structured communication on heterogeneously typed channels [13, 14, 21]. Traditional session types are *regular* in the sense that the sequences of communication actions admitted by a type are in the union of a regular language (for finite executions) and an ω-regular language (for infinite executions). Introduced by Thiemann and Vasconcelos, context-free session types liberate traditional session types from the shackles of tail recursion, allowing for example, the safe serialization of arbitrary recursive datatypes [23].

If the algorithmic aspects of type equivalence for regular session types are well known (Gay and Hole authored an algorithm to decide subtyting [8], from which type equivalence can be derived), the same does not apply to context-free session types. In the aforementioned work, Thiemann and Vasconcelos showed that the equivalence of context-free session types is decidable, by reducing the problem to the verification of bisimulation for Basic Process Algebra (BPA) which, in turn, was proved decidable by Christensen, Hüttel, and Stirling [6]. Even if the equivalence problem for context-free session types is known to be decidable, the only implementation of context-free session types we are aware of is that of Padovani [17], included in a programming language that requires a structural alignment between code and types (enforced by an explicit resumption process operator that explicitly breaks a type

$S;T$), thus sidestepping checking type equivalence.

After the breakthrough by Christensen, Hüttel, and Stirling— a result that provides no immediate practical algorithm—the problem of deciding the equivalence of BPA terms has been addressed by several researchers [4, 6, 16], but again, no actual practical algorithm can be readily extracted from these papers. Furthermore, context-free session types are not necessarily normed, which precludes using the original result by Baeten, Bergstra, and Klop [3], as well as improvements by Hirshfeld, Jančăr, and Moller [11, 12].

In its turn, the decidability of deterministic pushdown automata has also been a subject of much study [15, 19, 20]. Several techniques have been proposed to solve the problem, however no immediate practical algorithm had been proposed until Henry and Sénizergues provide an implementation of a correct algorithm for this problem [9]. Its poor performance precludes its incorporation in a compiler.

Our algorithm to decide the equivalence of context-free session types can also be seen as an algorithm to decide the equivalence of simple grammars. It follows three distinct stages. The *first stage* builds a context-free grammar in Greibach Normal Formal (GNF)—in fact a simple grammar—from a context-free session type in a way that bisimulation is preserved. A basic result from Baeten, Bergstra, and Klop states that any guarded BPA system can be transformed in Greibach Normal Formal (GNF) while preserving bisimulation equivalence, but unfortunately no procedure is presented [3]. The *second stage* prunes the grammar by removing unreachable symbols in unnormed sequences of non-terminal symbols. This stage builds on the result of Christensen, Hüttel, and Stirling [6]. The *third stage* constructs an expansion tree, by alternating between expansion and simplification steps. This last stage uses ideas on the expansion operations proposed by Jančăr, Moller, and Hirshfeld [10, 16], and ideas on the simplification rules proposed by Caucal, Christensen, Hüttel, Stirling, Jančăr, and Moller [5, 6, 16]. The finite representation of bisimulations of BPA transition graphs [5, 6] is paramount for our results of soundness and completeness. The branching nature of the expansion tree confers an exponential complexity to the algorithm, however we propose heuristics that allow constructing the relation in a reasonable time.

We present an algorithm to decide the equivalence of context-free session types, practical to the point that it may be readily included in any compiler, an exercise that we conducted in parallel [2]. The main contributions of this work are:

- The proposal and implementation of an algorithm to decide type equivalence of context-free session types and that of simple grammars (in 300 lines of Haskell code),
- A proof of its soundness and completeness against the declarative definition,
- The exploration of several optimizations that cut the running time in 12,000,000%.

The rest of the paper is organized as follows: context-free session types in Section 2, the algorithm in Section 3, the main results in Section 4, optimizations in Section 5, evaluation in Section 6, and conclusions in Section 7.

2 Context-free session types

The types we consider build upon a denumerable set of *type variables* denoted by X, Y, Z, and a set *type labels* denoted by ℓ. We assume given a set of base types B that include the unit type. Further base types could include the integer and boolean types, functions, and pairs. The syntax of types is derived from the grammar below.

$$S, T \;::=\; \mathsf{skip} \;\mid\; \sharp B \;\mid\; \star\{\ell_i \colon S_i\}_{i\in I} \;\mid\; S;T \;\mid\; \mu X.S \;\mid\; X$$

$$\sharp \;::=\; ! \mid ? \qquad \star \;::=\; \oplus \mid \& \qquad a \;::=\; \sharp B \mid \star l$$

We assume that all occurrences of variables in a type are introduced by some μ-binder (thus precluding free variables in types). We further assume that types are renamed so that all variables introduced by μ are distinct. Finally, we require types to be contractive (thus forbidding subterms of the form $\mu X_1.\mu X_2 \dots \mu X_n.X_1$) [7, 23]. For simplicity we removed polymorphic type variables (not bound by μ) from the grammar; they can be treated as $\sharp B$.

The labelled transition system (LTS) for context-free session types is given by the set of types as *states*, the set of *actions* ranged over by a, and the *transition relation* $\xrightarrow{a}_{\mathcal{T}}$ defined by the rules below, taken from Thiemann and Vasconcelos [23]. The transition relation makes use of an auxiliary judgment $S\checkmark$ that characterizes terminated states: session types that exhibit no further action [1] .

$$\mathsf{skip}\checkmark \qquad \frac{S\checkmark \quad T\checkmark}{S;T\checkmark} \qquad \frac{[\mu X.S/X]S\checkmark}{\mu X.S\checkmark} \qquad\qquad \sharp B \xrightarrow{\sharp B}_{\mathcal{T}} \mathsf{skip} \qquad \star\{l_i\colon S_i\} \xrightarrow{\star l_j}_{\mathcal{T}} S_j$$

$$\frac{S \xrightarrow{a}_{\mathcal{T}} S'}{S;T \xrightarrow{a}_{\mathcal{T}} S';T} \qquad \frac{S\checkmark \quad T \xrightarrow{a}_{\mathcal{T}} T'}{S;T \xrightarrow{a}_{\mathcal{T}} T'} \qquad \frac{[\mu X.S/X]S \xrightarrow{a}_{\mathcal{T}} T}{\mu X.S \xrightarrow{a}_{\mathcal{T}} T}$$

Type bisimulation, $\sim_{\mathcal{T}}$, is defined in the usual way from the labelled transition system [18].

3 An algorithm to decide type bisimilarity

This section presents an algorithm to decide whether two types are in a bisimulation relation. In the process we also provide an algorithm to decide the equivalence of simple context-free languages. The algorithm comprises three stages. It starts by converting types into grammars and then streamlines the grammar by pruning unreachable symbols in productions. The last stage explores an expansion tree, alternating between simplification and expansion operations, until either finding an empty node—case in which it decides positively—or failing to expand a node—case in which it decides negatively.

3.1 Converting types to grammars

A context-free grammar in Greibach normal form is a pair (X, P) where X is the *start symbol* and P a *set of productions* of the form $Y \to a\vec{Z}$ (we do not allow productions of the form $X \to \varepsilon$). Type variables are the *non-terminal symbols* and LTS labels the *terminal symbols*. We call *words* to sequences of type variables $\vec{X}$, and denote by ε the empty word. The grammars we are interested in are *simple*: for each non-terminal symbol X and each terminal symbol a, there is at most one production of the form $X \to a\vec{Y}$.

Grammars in Greibach Normal Form naturally induce an LTS by taking sequences of non-terminal symbols $\vec{X}$ as states, terminal symbols a as the set of actions, and the transition relation $\xrightarrow{a}_{\mathcal{P}}$ defined as $X\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Z}\vec{Y}$ when $X \to a\vec{Z} \in \mathcal{P}$. The associated bisimulation is denoted by $\sim_{\mathcal{P}}$.

Given a context-free session type S, the algorithm starts by inserting an initial production of the form $X_S \to !\mathsf{unit}\,(\mathsf{toGrammar}\ S)$ in the set of productions. Function toGrammar (Listing 1) returns a sequence of non-terminal symbols, while computing the remaining productions. It uses a predicate isChecked S to determine whether S is terminated, that is whether S$\checkmark$. The algorithm keeps the set of productions and an integer (to generate fresh non-terminal symbols) in the monadic state TransState. It uses the following functions to manipulate state.

- freshVar returns a fresh non-terminal symbol (a type variable);
- addProduction $X\,a\,\vec{Y}$ updates the state by inserting the production $X \to a\vec{Y}$;

- getTransitions X retrives the transitions from X (a map from non-terminal symbols a to sequences $\vec{Y}$ of type variables).

```haskell
type Transitions = Map.Map LTSLabel [TypeVar]
type Productions = Map.Map TypeVar Transitions
type Visited = Set.Set TypeVar
type TransState = State (Productions, Int)

toGrammar :: Type → TransState [TypeVar]
toGrammar Skip =
  return []
toGrammar (Message p b) = do
  y ← freshVar
  addProduction y (MessageLabel p b) []
  return [y]
toGrammar (Choice c m) = do
  y ← freshVar
  mapM_ (assocsToGrm y c) (Map.assocs m)
  return [y]
toGrammar (Semi t u) = do
  xs ← toGrammar t
  ys ← toGrammar u
  return (xs ++ ys)
toGrammar (Rec x t) =
  | isChecked (Rec x t) = return []
  | otherwise = do
    zs ← toGrammar t
    m ← getTransitions (head zs)
    addProductions x (Map.map (++ tail zs) m)
    return [x]
toGrammar (Var x) =
  return [x]

assocsToGrm :: TypeVar → ChoiceView → (TypeLabel,Type) → TransState ()
assocsToGrm y c (l, t) = do
  xs ← toGrammar t
  addProduction y (ChoiceLabel c l) xs
```

Listing 1 Haskell code for stage 1: the conversion of types into grammars

Notice that function toGrammar terminates on all inputs and that the resulting set of productions is finite, because recursion is always on subterms. Furthermore, due to the deterministic nature of the LTS, toGrammar returns a simple grammar. One can obtain a unique set of productions for two types by ensuring that fresh variables do not overlap.

▶ **Example 1.** Consider the following context-free session types:

$$\begin{aligned} S &\triangleq (\mu x.\&\{n : x; x; ?\mathsf{int}, \ell : ?\mathsf{int}\}); (\mu z.!\mathsf{int}; z; z) \\ T &\triangleq (\mu y.\&\{n : y; y, \ell : ?\mathsf{skip}\}; ?\mathsf{int}); (\mu w.!\mathsf{int}; w) \end{aligned}$$

Function toGrammar, when applied to S and T, produces the following productions.

Productions for type S		Productions for type T	
$X_S \to !\mathsf{unit}\, X_1 X_4$	$X_2 \to ?\mathsf{int}$	$Y_T \to !\mathsf{unit}\, Y_1 Y_3$	$Y_2 \to ?\mathsf{int}$
$X_1 \to \&n\, X_1 X_1 X_2$	$X_3 \to ?\mathsf{int}$	$Y_1 \to \&n\, Y_1 Y_1 Y_2$	$Y_3 \to !\mathsf{int}\, Y_3$
$X_1 \to \&\ell\, X_3$	$X_4 \to !\mathsf{int}\, X_4 X_4$	$Y_1 \to \&\ell\, Y_2$	

3.2 Pruning unnormed productions

For $\vec{a}$ a sequence of non-terminal symbols $a_1, \dots, a_k$ ($k \geq 1$), write $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$ when $\vec{Y} \xrightarrow{a_1}_{\mathcal{P}} \cdots \xrightarrow{a_k}_{\mathcal{P}} \vec{Z}$. We say that $\vec{Y}$ is *normed* when $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$ for some $\vec{a}$, and that $\vec{Y}$ is *unnormed* otherwise. When $\vec{Y}$ is normed, the *minimal path* of $\vec{Y}$ is the shortest $\vec{a}$ such that $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$. In this case, the *norm* of $\vec{Y}$, denoted by $|\vec{Y}|$, is the length of $\vec{a}$.

As observed by Christensen Hüttel, and Stirling [6], any unnormed words $\vec{Y}$ is bisimilar to its concatenation with any other nonterminal symbols, that is, if $\vec{Y}$ is unnormed, then $\vec{Y} \sim_{\mathcal{P}} \vec{Y} X$. We use this fact to prune out unreachable symbols in unnormed sequences of symbols. The code is in Listing 2.

```
prune :: Productions -> Productions
prune p = Map.map (Map.map (pruneWord p)) p

pruneWord :: Productions -> [TypeVar] -> [TypeVar]
pruneWord p = foldr (\x ys -> if normed p x then x:ys else [x]) []

normed :: Productions -> TypeVar -> Bool
normed p x = normedWord p Set.empty [x]

normedWord :: Productions -> Visited -> [TypeVar] -> Bool
normedWord _ _ []     = True
normedWord p v (x:xs) =
  x `Set.notMember` v &&
  any (normedWord p v') (Map.elems (transitions p (x:xs)))
  where v' = if any (x `elem`) (Map.elems (transitions p [x]))
             then Set.insert x v else v
```

Listing 2 Haskell code for stage 2: pruning unnormed productions

▶ **Example 2.** Recall Example 1 and notice that both X_S and Y_T are unnormed. We can easily see that the last occurrence of X_4 in the last production for S is unreachable. Hence, by pruning the productions for S we get:

Pruned productions for type S

$X_S \to \text{!unit}\, X_1 X_4$	$X_1 \to \&\ell\, X_3$	$X_1 \to \&n\, X_1 X_1 X_2$
$X_2 \to \text{?int}$	$X_3 \to \text{?int}$	$X_4 \to \text{!int}\, X_4$

3.3 Building expansion trees

We base the third stage of the algorithm on the notion of *expansion tree* proposed by Jančar and Moller [16], an adaption of an idea by Hirshfeld [10]. We say a set N' of pairs of words is an *expansion* of N if N' is a minimal set such that: for every pair $(\vec{X}, \vec{Y}) \in N$,

- if $\vec{X} \to a\vec{X}'$ then $\vec{Y} \to a\vec{Y}'$ with $(\vec{X}', \vec{Y}') \in N'$;
- if $\vec{Y} \to a\vec{Y}'$ then $\vec{X} \to a\vec{X}'$ with $(\vec{X}', \vec{Y}') \in N'$.

An *expansion tree* is built from nodes. Children nodes are obtained by expansion from its parent node. Jančar and Moller observed that expansion alone often leads to infinite trees. We then alternate between expansion and simplification operations, until either finding an empty node—case in which we decide equivalence positively—or failing to expand a node—case in which we decide equivalence negatively. We say that a branch is *successful* if it is infinite or finishes in an empty node, otherwise it is said to be *unsuccessful*.

In the *expansion step*, each node N derives a single children node, obtained as an expansion of N. As we are dealing with simple grammars, no branching is expected in the expansion tree at this step. The *simplification step* consists on the application of the following rules:

Reflexive rule: Omit from a node any reflexive pair;
Congruence rule: Omit from a node N any pair that belongs to the least congruence containing the ancestors of N;
BPA1 rule: If $(X_0\vec{X}, Y_0\vec{Y})$ is in N and $(X_0\vec{X'}, Y_0\vec{Y'})$ belongs to the ancestors of N, then create a sibling node for N replacing $(X_0\vec{X}, Y_0\vec{Y})$ by $(\vec{X}, \vec{X'})$ and $(\vec{Y}, \vec{Y'})$;
BPA2 rule: If $(X_0\vec{X}, Y_0\vec{Y})$ is in N and X_0 and Y_0 are normed, then:

Case $|X_0| \leq |Y_0|$: Let $\vec{a}$ be a minimal path for X_0 and $\vec{Z}$ a word such that $\vec{Y_0} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including the pairs $(X_0\vec{Z}, Y_0)$ and $(\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$;
Case $|X_0| > |Y_0|$: Let $\vec{a}$ be a minimal path for Y_0 and $\vec{Z}$ a word such that $\vec{X_0} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including the pairs $(X_0, Y_0\vec{Z})$ and $(\vec{Z}\vec{X}, \vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$.

Contrarily to expansion and to the reflexive and congruence simplifications, BPA rules promote branching in the expansion tree. The number of children nodes generated by these rules is finite [6]. Notice that the sibling nodes do not exclude the (often) infinite branch resulting from successive expansions.

3.4 Checking the bisimilarity of context-free session types

Given two context-free session types, function bisimilar (in Listing 3) starts by converting the two session types into a grammar, which is then pruned. Function convertToGrammar (not shown) builds the initial monadic state, and runs the algorithm of section 3.1 to convert the session types given as parameters. An expansion tree is computed afterwards, through an alternation of expansion of children nodes and their simplification, using the reflexive, congruence, and BPA rules. To avoid getting stuck in an infinite branch of the expansion tree, we use a breadth-first search on the expansion tree. Upcoming nodes are stored in a queue. The simplification stage distinguishes the case where all type variables are normed, in which case BPA1 is not required to decide equivalence [5, 6], from the case where some type variables might be unnormed. The recursive procedure terminates as soon as all nodes fail to expand and, thus, the queue is empty, case in which the algorithm returns **False**, or an empty node is reached, case is which the algorithm returns **True**.

```
type Node = Set.Set ([TypeVar], [TypeVar])
type Ancestors = Node
type NodeQueue = Queue.Queue (Node, Ancestors)
type NodeTransformation = Productions → Ancestors → Node → Set.Set

bisimilar :: Type → Type → Bool
bisimilar t u = expand (prune p) [x] [y]
  where Grammar [x, y] p = convertToGrammar [t, u]

expand :: Productions → NodeQueue → Bool
expand ps q
  | Queue.null q = False
  | Set.null n   = True
  | otherwise    = case expandNode ps n of
```

```
      Nothing → expand ps (Queue.dequeue q)
      Just n' → expand ps (simplify ps n' (Set.union a n) q)
  where (n,a) = Queue.front q

simplify :: Productions → Node → Ancestors → NodeQueue → NodeQueue
simplify ps n a q = foldr Queue.append (Queue.dequeue q) nas'
  where nas  = Set.singleton (n,a)
        nas' = if allNormed ps
               then foldr (apply ps) nas [reflex,congruence,bpa2]
               else foldr (apply ps) nas [reflex,congruence,bpa1,bpa2]
```

Listing 3 Haskell code for checking the bisimilarity of context-free session types

▶ **Example 3.** The expansion tree for our running example is in Figure 1. Once a successful branch is reached (the ✓ in the figure), the algorithm in Listing 3 decides that $S \sim_\mathcal{T} T$.

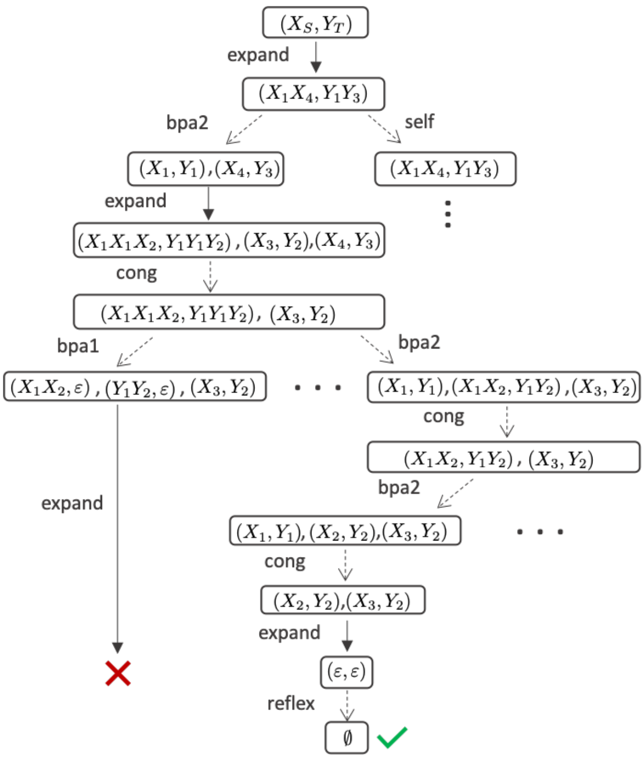

Figure 1 Expansion tree for the context-free session types S and T introduced in Example 1

4 Correctness of the algorithm

In this section we prove that our algorithm is sound and complete with respect to the meta-theory of context-free session types proposed by Thiemann and Vasconcelos [23].

We start by showing that the bisimulation relation on context-free session types, $\sim_\mathcal{T}$, is equivalent to the bisimulation relation obtained from the productions, $\sim_\mathcal{P}$. Then, based on results from Caucal [5], Christensen, Hüttel, and Stirling [6], Jančar and Moller [16], we conclude that our algorithm is sound and complete.

4.1 The two bisimulations coincide

For the purpose of providing a bisimulation between context-free session types and their corresponding symbols in the grammar, we start with a lemma that relates terminated types S to the result of the toGrammar S.

▶ **Lemma 4.** *$S\checkmark$ if and only if $\varepsilon, P =$ toGrammar S.*

Proof. Given that all variables in a type are under some μ-binder, there is a simple inductive characterization of $S\checkmark$, namely, skip$\checkmark$, $(S;T)\checkmark$ if $S\checkmark$ and $T\checkmark$, $(\mu X.S)\checkmark$ if $S\checkmark$, $X\checkmark$, and false in all other cases. The proof then follows by a simple induction on this characterization for the "if" direction, and induction on toGrammar for the "only if" direction. ◀

To conclude that the bisimulations on context-free session types and those on productions coincide, we present a bisimulation between context-free session types and the result of toGrammar. As the implementation of isChecked is not shown, we assume that isChecked S returns **True** if and only if $S\checkmark$.

▶ **Theorem 5.** *S is bisimilar to* toGrammar *S.*

Proof. Let $\mathcal{R}$ be the binary relation on types $\times$ (words $\times$ productions) that contains the following sets of pairs $(S, (\vec{X}, P))$ built in such a way that $\vec{X}, P =$ toGrammar S.

$(\mathsf{skip}, (\varepsilon, \emptyset))$

$(\sharp B, (X, \{X \to \sharp B\}))$

$(\star\{l_i\colon S_i\}_{i\in I}, (X, \{X \to \star l_i \vec{Y_i}\}_{i\in I} \cup (\cup P_i)_{i\in I}))$ when $\vec{Y_i}, P_i =$ toGrammar $S_i, i \in I$

$(S_1; S_2, (\vec{X}_1\vec{X}_2, P_1 \cup P_2))$ when $\vec{X}_i, P_i =$ toGrammar $S_i, i = 1, 2$

$(\mu X.S, (\varepsilon, \emptyset))$ when isChecked $\mu X.S$

$(\mu X.S, (X, \{X \to a\vec{Z}\vec{Y}\} \cup P))$ when not isChecked $\mu X.S$ and

$Y\vec{Y}, P =$ toGrammar S and $Y \to a\vec{Z} \in P$

That $\mathcal{R}$ is a bisimulation follows by co-induction, using Lemma 4. ◀

4.2 Correctness of the algorithm

We now focus on the correctness of the algorithm in Listing 3. Before proceeding to soundness, we recall the *safeness property* presented by Jančar and Moller.

▶ **Proposition 6** (Safeness Property [16]). *$\vec{X} \sim_{\mathcal{P}} \vec{Y}$ if and only if the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

Notice that function bisimilar (Listing 3) builds an expansion tree by alternating between expansion and simplification operations (reflexive, congruence and BPA rules), as proposed by Jančar and Moller. These simplification rules are *safe* [16], in the sense that the application of any rule preserves the bisimulation from a parent node to at least one child node and, reciprocally, that bisimulation on a child node implies the bisimulation of its parent node, thus proving the safeness property.

▶ **Lemma 7.** *If* bisimilar $S\ T$ *returns* **True**, then $X_S \sim_{\mathcal{P}} X_T$.

Proof. Function bisimilar returns **True** for S and T whenever it reaches a (finite) successful branch in the expansion tree rooted at $\{(X_S, X_T)\}$. Conclude with the safeness property, Proposition 6. ◀

From the previous results, the soundness of our algorithm is now immediate: the algorithm to check the bisimulation of context-free session types (Listing 3) is sound with respect to the meta-theory of context-free session types.

▶ **Theorem 8.** *If* bisimilar $S\,T$ *returns* **True** then $S \sim_{\mathcal{T}} T$.

Proof. From Theorem 5 and Lemma 7. ◀

Having observed that the safeness property was paramount for soundness, we now notice that the *finite witness property* is of utmost importance to prove completeness. This result follows immediately from the analysis by Jančar and Moller [16], which capitalizes on results by Caucal [5], and Christensen, Hüttel, and Stirling [6]:

▶ **Proposition 9** (Finite Witness Property). *If $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, then the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a finite successful branch.*

We refer to Caucal, Christensen, Hüttel, and Stirling for details on the proof of existence of a finite witness, as stated in Proposition 9. This proof is particularly interesting in that it highlights the importance of BPA rules and of pruning productions on reaching such (finite) witness. The results in these two papers also allow us to unravel the reason for the distinction of the simplification phase in the case where all the symbols in the grammar are normed from the case where they are not, as presented in Listing 3.

Proposition 9 paved the way to obtain the completeness result. We now prove that the algorithm to check the bisimulation of context-free session types is complete with respect to the meta-theory of context-free session types.

▶ **Theorem 10.** *If $S \sim_{\mathcal{T}} T$ then* bisimilar $S\,T$ *returns* **True**.

Proof. Assuming that $S \sim_{\mathcal{T}} T$, by Theorem 5 we have $X_S \sim_{\mathcal{P}} X_T$. Hence, the Proposition 9 ensures the existence of a finite successful branch on the expansion tree rooted at $\{(X_S, X_T)\}$, i.e., a branch terminating in an empty node. Since our algorithm traverses the expansion tree using breadth-first search it will, eventually, reach the empty node and conclude the bisimulation positively. ◀

5 Optimizations

Armed with the results in Section 3, we decided to benchmark the algorithm on a test suite of carefully crafted pair of types (more on this in Section 6). During this process we came across a pair of types,

$$\begin{aligned} S &\triangleq \mu x.\&\{\mathsf{Add}\colon x;x;!\,\mathsf{int}, \mathsf{Const}\colon ?\,\mathsf{int};!\mathsf{int}, \mathsf{Mult}\colon x;x;!\,\mathsf{int}\} \\ T &\triangleq \mu x.\&\{\mathsf{Add}\colon x;x, \mathsf{Const}\colon ?\,\mathsf{int}, \mathsf{Mult}\colon x;x\};!\,\mathsf{int} \end{aligned} \tag{1}$$

on which function bisimilar took 4379.98 seconds (that is one hour and forty minutes) to terminate. This is certainly not a reasonable running time for an algorithm to be included in a compiler. Hence we looked into ways to improve the running time. Among the different optimisations that we tried, two stand out:

1. Iterate the simplification stage until a fixed point is reached;
2. Use a double-ended queue where promising children are prepended rather than appended.

If, on the one hand, we believed that the computation of the expansion tree could be speeded up by extending the simplification phase, on the other hand we suspected that a double-ended queue would allow prioritizing nodes with potential to reach an empty node faster. Iterating the simplification procedure on a given node N, the algorithm computes the simplest possible children nodes derived from N. Of course, we need to make sure that a fixed-point exists, which we do with Theorem 11. Using a double-ended queue, the algorithm prepends (rather than appends) nodes that are already empty or whose pairs $(\vec{X}, \vec{Y})$ are such that $|\vec{X}| \leq 1$ and $|\vec{Y}| \leq 1$. The revised $\mathsf{simplify}$ function is in Listing 4.

The next theorem shows that the simplification function that consists in applying the reflexive, congruence and BPA rules has a fixed point. The result applies regardless of whether all nonterminal symbols symbols are normed or not.

▶ **Theorem 11.** *The simplification function that results from applying the reflexive, congruence, and BPA rules, has a fixed point in the complete partial ordered set* $\mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors})$*, where the set of ancestors is supposed to be fixed.*

Proof. Throughout the proof we abuse notation and denote the application of simplification rules to nodes and to elements of $\mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors})$ similarly, when no ambiguity arises. Consider the order $\sqsubseteq$, defined on $\mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors}) \times \mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors})$, as $S_1 \sqsubseteq S_2$ if $|S_1| \leq |S_2|$ and there exists an injective map $\sigma : S_1 \to S_2$ s.t. $\sigma(N_1, A) = (N_2, A)$ with $N_2 \subseteq N_1$.

- $\sqsubseteq$ is a partial order. The proof that $\sqsubseteq$ is reflexive and transitive is straightforward. To prove that it is antisymmetric, assume that $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_1$. This means that $|S_1| = |S_2|$ and, furthermore, the maps $\sigma_1 : S_1 \to S_2$ and $\sigma_2 : S_2 \to S_1$ are bijective. Notice that $\sigma_1 \circ \sigma_2$ is the identity map, otherwise we could consider $(N, A) \in S_2$ where N is minimal w.r.t. inclusion and s.t. $(\sigma_1 \circ \sigma_2)(N, A) \neq (N, A)$, i.e., $(\sigma_1 \circ \sigma_2)(N, A) = (N', A)$ with $N' \subseteq N$ for some $(N', A) \in S_2$; due to the minimality of N, we would have $(\sigma_1 \circ \sigma_2)(N', A) = (N', A)$, which would contradict the injectivity of $\sigma_1 \circ \sigma_2$. Since $\sigma_1(N, A) = (N', A)$ is such that $N' \subseteq N$, we shall have $\sigma_1(N, A) = (N, A)$. Hence, $S_1 = S_2$.
- The simplification function is order-preserving. To prove that the reflexive rule preserves the order, let S_1 and S_2 be s.t. $S_1 \sqsubseteq S_2$ and let us prove that $\mathsf{reflex}\, S_1 \sqsubseteq \mathsf{reflex}\, S_2$. Let $(N, A) \in \mathsf{reflex}\, S_1$ and notice that there exists $(N_1, A) \in S_1$, such that $\mathsf{reflex}\, N_1 = N$, and so, in S_2 there is $(N_2, A) = \sigma(N_1, A)$ s.t. $N_2 \subseteq N_1$. Since $N_2 \subseteq N_1$, we have $\mathsf{reflex}\, N_2 \subseteq \mathsf{reflex}\, N_1 = N$. The same reasoning applies to prove that if $S_1 \sqsubseteq S_2$ then $\mathsf{congruence}\ S_1 \sqsubseteq \mathsf{congruence}\ S_2$. To prove that $\mathsf{bpa1}$ preserves the order, note that $S \subseteq \mathsf{bpa1}\ S$. Assume that $S_1 \sqsubseteq S_2$, let $(N, A) \in \mathsf{bpa1}\ S_1$, and denote by $(N_1, A) \in S_1$ and $(\vec{X}, \vec{Y}) \in N_1$ the node and the pair whose simplification led to (N, A). We know that exists $(N_2, A) \in S_2$ s.t. $N_2 \subseteq N_1$. If $(\vec{X}, \vec{Y}) \in N_2$, then the $\mathsf{bpa1}$ simplification of N_2 with the pair $(\vec{X}, \vec{Y})$ generates $(N', A) \in \mathsf{bpa1}\ S_2$ such that $N' \subseteq N$. On the other hand, if $(\vec{X}, \vec{Y}) \notin N_2$, then $N_2 \subseteq N$ and, since $S_2 \subseteq \mathsf{bpa1}\ S_2$, $(N_2, A) \in \mathsf{bpa1}\ S_2$ is such that $N_2 \subseteq N$. The same reasoning applies to $\mathsf{bpa2}$.

Having proved that each simplification function preserves the order, and since the simplification procedure results from the successive application of these rules, we have proved that the simplification function also preserves the order.

- $(\mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors}), \sqsubseteq)$ is a lattice. Given $S_1, S_2 \in \mathsf{Set}\ (\mathsf{Node}, \mathsf{Ancestors})$, $S_1 \cup S_2$ is an upper bound and $S_1 \cap S_2$ is a lower bound for S_1 and S_2.

- ($\mathsf{Set}\ (\mathsf{Node},\ \mathsf{Ancestors}), \sqsubseteq$) is a complete lattice. Given $\mathcal{B} \subseteq \mathsf{Set}\ (\mathsf{Node},\ \mathsf{Ancestors})$: $\bigcup_{S\in\mathcal{B}} S$ is an upper bound and $\bigcap_{S\in\mathcal{B}} S$ is a lower bound for the sets in $\mathcal{B}$.

Using Tarski's fixed point theorem [22], we conclude that the simplification function has a fixed point in Set (Node, Ancestors). ◀

Having proved that the fixed point exists, we can now adapt the simplification phase to, on the one hand, iterate the simplification rules until reaching a fixed point and, on the other hand, identify and prepend promising nodes. An improved version of the simplify function (Listing 3) is in Listing 4.

```haskell
simplify :: Productions → Node → Ancestors → NodeQueue → NodeQueue
simplify ps n a q = foldr enqueueNode (Queue.dequeue q) nas
  where nas = findFixedPoint ps (Set.singleton (n,a))

enqueueNode :: (Node, Ancestors) → NodeQueue → NodeQueue
enqueueNode (n,a) q
 | maxLength n <= 1 = Queue.prepend (n,a) q
 | otherwise        = Queue.append (n,a) q

findFixedPoint :: Productions → Set.Set (Node, Ancestors) →
                    Set.Set (Node, Ancestors)
findFixedPoint ps nas
  | nas == nas' = nas
  | otherwise   = findFixedPoint ps nas'
  where nas' = if allNormed ps
               then foldr (apply ps) nas [reflex, congruence, bpa2]
               else foldr (apply ps) nas [reflex, congruence, bpa1, bpa2]
```

Listing 4 Haskell code for the improved simplification step (replaces function simplify in Listing 3)

The optimisations we propose aim at improving the performance of the algorithm, however the branching nature of the expansion tree promotes an exponential complexity: each simplification step (potentially) generates a polynomial number of nodes, each of which with linear size on the size of the input. In turn, the same simplification phase may, in the worst case, be iterated a linear number of times on the size of the input. For these reasons the complexity turns out to be exponential. Nevertheless, these heuristics seems to work quite well in practice, as we show in the next section.

6 Evaluation

We implemented the algorithm skteched in Listings 1 to 4 in 300 lines of Haskell and used the Glasgow Haskell Compiler, GHC version 8.6.3, from which we have obtained the results we present in this section. Evaluation was conducted on a Mac mini equipped with a 3.6 GHz Intel Core i3, 8 GB of memory, running MacOS 10.14.3.

Once the improvement proposals were established, we benchmarked the algorithm on a test suite of carefully crafted pair of types. These tests comprise valid and invalid equivalences, for a total of 138 tests. We have profiled our program for the time and memory allocated during the tests. For this purpose, we have used GHC's profiling feature, that maintains a cost-centre stack to keep track of the incurred costs. We ran the tests 10 times, kept a record of the run time and memory allocated for each run, discarded the best and worst values obtained and, then, we have measured the average of the remaining values. The results are depicted in Figure 2.

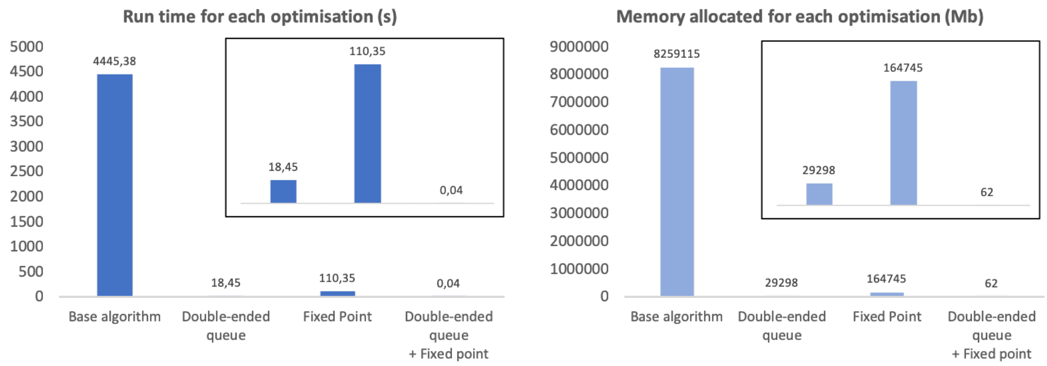

Figure 2 Test results: running times (on the left) and memory allocated (on the right) checking the equivalence of context-free session types in 138 tests.

For the base algorithm, proposed in Listing 3, we obtained an average running time of about 4445.38 seconds and 8,259,115 Mb memory allocated. From the moment we introduced the optimizations the results improved remarkably: iterating the simplification phase in the search for a fixed point allowed to reduce the running time to 110.35 seconds and the memory allocated to 164,745 Mb, whereas the implementation of the double-ended queue allowed to reduce the running time to 18.45 seconds and the allocated memory to 29,298. The combination of both exhibit an improvement on more than 12,000,000% from the base case, achieving an average of 0.04 seconds for the running time and 62 Mb of allocated memory.

We should also highlight that, we run example (1) with the improved algorithm, in a battery os 100 runs, and obtained an average running time of 0.008 seconds.

The heuristic we proposed actually circumvents the exponential complexity inherent to the expansion tree, thus allowing to obtain running times that are manifestly small, thus allowing the use of this algorithm as an integral part of a compiler, as we had intended from the beginning.

7 Conclusion

Context-free session types are a promising tool to describe protocols in concurrent programs. In order to be incorporated in programming languages and effectively used in compilers, a practical algorithm to decide bisimulation is called for. Taking advantage of a process algebra graph representation of types to decide bisimulation [11, 12], we have developed one such algorithm and proved it correct. The algorithm is incorporated in a compiler for a concurrent functional language equipped with context-free session types [2].

References

1 Luca Aceto and Matthew Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39(1):147–187, 1992.

2 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. FreeST: context-free session types in a functional language. To appear, 2019.

3 Jos CM Baeten, Jan A Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for process generating context-free languages. *Journal of the ACM (JACM)*, 40(3):653–682, 1993.

4 Olaf Burkart, Didier Caucal, and Bernhard Steffen. An elementary bisimulation decision procedure for arbitrary context-free processes. In Jirí Wiedermann and Petr Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, Proceedings*, volume 969 of *Lecture Notes in Computer Science*, pages 423–433. Springer, 1995.

5 Didier Caucal. Décidabilité de l'égalité des langages algébriques infinitaires simples. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 37–48. Springer, 1986.

6 Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.*, 121(2):143–148, 1995.

7 Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

8 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

9 Patrick Henry and Géraud Sénizergues. Lalblc a program testing the equivalence of dpda's. In *International Conference on Implementation and Application of Automata*, pages 169–180. Springer, 2013.

10 Yoram Hirshfeld. Bisimulation trees and the decidability of weak bisimulations. *Electr. Notes Theor. Comput. Sci.*, 5:2–13, 1996.

11 Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1&2):143–159, 1996.

12 Yoram Hirshfeld and Faron Moller. A fast algorithm for deciding bisimilarity of normed context-free processes. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, volume 836 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 1994.

13 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

14 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

15 Petr Jančar. Selected ideas used for decidability and undecidability of bisimilarity. In *International Conference on Developments in Language Theory*, pages 56–71. Springer, 2008.

16 Petr Jančar and Faron Moller. Techniques for decidability and undecidability of bisimilarity. In *International Conference on Concurrency Theory*, pages 30–45. Springer, 1999.

17 Luca Padovani. Context-free session type inference. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 804–830. Springer, 2017.

18 Davide Sangiorgi. *An Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2014.

19 Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *International Colloquium on Automata, Languages, and Programming*, pages 671–681. Springer, 1997.

20 Colin Stirling. Decidability of dpda equivalence. *Theoretical Computer Science*, 255(1-2):1–31, 2001.

21 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

22 Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

23 Peter Thiemann and Vasco Thudichum Vasconcelos. Context-free session types. In *ACM SIGPLAN Notices*, volume 51, pages 462–475. ACM, 2016.