Efficient Type Matching

S. Jha (jha@cs.wisc.edu)
Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA.

J. Palsberg (palsberg@cs.purdue.edu)
Department of Computer Science, Purdue University, W. Lafayette, IN 47907, USA.

T. Zhao (tzhao@cs.uwm.edu)
Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, WI 53211, USA.

F. Henglein (henglein@diku.dk)
Department of Computer Science (DIKU), University of Copenhagen, DK-2100 Copenhagen, Denmark.

July 22, 2003

Abstract. Palsberg and Zhao (2001) presented an $O(n^2)$ time algorithm for matching two recursive types; that is, deciding type isomorphism under associative-commutative type constructors. In this paper, we present an $O(n \log n)$ algorithm for matching recursive types and an $O(n)$ algorithm for matching nonrecursive types. The linear-time algorithm for nonrecursive types works without hashing or pointer arithmetic, by employing multiset discrimination due to Paige *et al.*. The $O(n \log n)$ algorithm for recursive types works by reducing the type matching problem to the problem of finding a size-stable partition of a graph, which has $O(n \log n)$ algorithms due to Cardon/Crochemore and Paige/Tarjan. The key to these algorithms is the use of a "modify-the-smaller-half" approach pioneered by Hopcroft and Ullman for DFA minimization.

Our results may help improve systems, such as Polyspin and Mockingbird, that are designed to facilitate interoperability of software components. We also discuss possible applications of our algorithm to Java. Issues related to subtyping of recursive types are also discussed.

1. Introduction

Interoperability is a fundamental problem in software engineering. Interoperability issues arise in various contexts, such as software reuse, distributed programming, use of legacy components, and integration of software components developed by different organizations. Interoperability of software components has to address two fundamental problems: *matching* and *bridging*. Matching deals with determining whether two components A and B are compatible, and bridging allows one to use component B using the interface defined for component A.

Matching: A common technique for facilitating matching is to associate signatures with components. These signatures can then be used as keys to retrieve relevant components from an existing library of components. Use of finite types as signatures was first proposed by Rittri (1991). Zaremski and Wing (1995a; 1995b) used a similar approach for retrieving components from an ML-like functional library. Moreover, they also emphasized flexibility and support for user-defined types.

Bridging: In a multi-lingual context, *bridge code* for "gluing" components written in different languages (such as C, C++, and Java) has to be developed. CORBA (OMG, 1999), PolySpin (Barrett et al., 1996), and Mockingbird (Auerbach et al., 1998; Auerbach and Chu-Carroll, 1997) allow composing components implemented in different languages. Software components are considered to be of two kinds: *objects*, which provide public interfaces, and *clients*, which invoke the methods of the objects and thereby use the services provided by the objects.

The Problem: Assume that we use types as signatures for components. Thus, the type matching problem reduces to the problem of determining whether two types are equivalent. Much previous work on type matching focuses on non-recursive types (Bruce et al., 1992; Cosmo, 1995; Narendran et al., 1993; Rittri, 1990; Rittri, 1991; Rittri, 1993; Soloviev, 1983; Zaremski and Wing, 1995a; Aponte and Cosmo, 1996). In this paper, we consider equivalence of recursive types. Equality and subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli (1993); Kozen, Palsberg, and Schwartzbach (1995); Brandt and Henglein (1998); Jim and Palsberg (1997); and others. These papers concentrate on the case where two types are considered equal if their infinite unfoldings are identical. In this case, type equivalence can be decided in $O(n\alpha(n))$ time. If we allow a product-type constructor to be associative and commutative, then two recursive types may be considered equal *without* their infinite unfoldings being identical. Alternatively, think of a product type as a multiset, by which associativity and commutativity are obtained for free. Such flexibility has been advocated by Auerbach, Barton, and Raghavachari (1998). Palsberg and Zhao (2001) presented a definition of type equivalence that supports this idea. They also presented an $O(n^2)$ time algorithm for deciding their notion of type equivalence.

A notion of subtyping defined by Amadio and Cardelli (1993) can be decided in $O(n^2)$ time (Kozen et al., 1995). We also briefly discuss subtyping of recursive types in this paper.

Our results: We present $O(n \log n)$ and $O(n)$ time algorithms for deciding type equivalence with and without type recursion, respectively. The $O(n \log n)$ algorithm improves upon the $O(n^2)$ algorithm of Palsberg and Zhao (2001). It works by reducing the type matching problem to the well-understood problem of finding a size-stable partition of a graph (Paige and Tarjan, 1987). The $O(n)$ algorithm employs multiset discrimination due to Paige *et al.*

The organization of the paper: A small example is described in Section 2. This example will be used throughout the paper for illustrative purposes. In Section 3 we recall the notions of terms and term automata (Amadio and Cardelli, 1993; Cardelli and Wegner, 1985; Courcelle, 1983; Kozen et al., 1995), and we state the definitions of types and type equivalence from the paper by Palsberg and Zhao (2001). In Section 4 we present our $O(n)$ algorithm for finite types, and in Section 5 we present our $O(n \log n)$ algorithm for recursive types. An implementation of our $O(n \log n)$ algorithm is discussed Section 6. Subtyping of recursive types is discussed in Section 7. Concluding remarks appear in Section 8.

2. Example

In this section we provide a small example which will be used throughout the paper. It is straightforward to map a `Java` type to a recursive type of the form

$$
\begin{array}{ll}
\text{interface } I_1 \; \{ & \text{interface } I_2 \; \{ \\
\quad \mathit{float} \quad m_1(I_1\ a); & \quad I_1 \quad m_3(\mathit{float}\ a); \\
\quad \mathit{int} \quad m_2(I_2\ a); & \quad I_2 \quad m_4(\mathit{float}\ a); \\
\} & \}
\end{array}
$$

Figure 1. Interfaces I_1 and I_2

$$
\begin{array}{ll}
\text{interface } J_1 \; \{ & \text{interface } J_2 \; \{ \\
\quad J_1 \quad n_1(\mathit{float}\ a); & \quad \mathit{int} \quad n_3(J_1\ a); \\
\quad J_2 \quad n_2(\mathit{float}\ a); & \quad \mathit{float} \quad n_4(J_2\ a); \\
\} & \}
\end{array}
$$

Figure 2. Interfaces J_1 and J_2

considered in this paper. A collection of method signatures can be mapped to a product type, a single method signature can be mapped to a function type, and in case a method has more than one argument, the list of arguments can be mapped to a product type. Recursion, direct or indirect, is expressed with the μ operator. This section provides an example of of `Java` interfaces and provides an illustration of our algorithm.

Suppose we are given the two sets of Java interfaces shown in Figures 1 and 2. We would like to find out whether interface I_1 is "structurally equal" to or "matches" with interface J_2. We want a notion of equality for which interface names and method names do not matter, and for which the order of the methods in an interface and the order of the arguments of a method do not matter.

Notice that interface I_1 is recursively defined. The method m_1 takes an argument of type I_1 and returns a floating point number. In the following, we use names of interfaces and methods to stand for their type structures. The type of method m_1 can be expressed as $I_1 \to \mathit{float}$. The symbol $\to$ stands for the function type constructor. Similarly, the type of m_2 is $I_2 \to \mathit{int}$. We can then capture the structure of I_1 with conventional μ-notation for recursive types:

$$I_1 = \mu\alpha.(\alpha \to \mathit{float}) \times (I_2 \to \mathit{int})$$

The symbol α is the type variable bound to the type I_1 by the symbol μ. The interface type I_1 is a product type with the symbol $\times$ as the type constructor. Since we think of the methods of interface I_1 as unordered, we could also write the structure of I_1 as

$$
\begin{aligned}
I_1 &= \mu\alpha.(I_2 \to \mathit{int}) \times (\alpha \to \mathit{float})\ , \\
I_2 &= \mu\delta.(\mathit{float} \to I_1) \times (\mathit{float} \to \delta)\ .
\end{aligned}
$$

In the same way, the structures of the interfaces J_1, J_2 are:

$$
\begin{aligned}
J_1 &= \mu\beta.(\mathit{float} \to \beta) \times (\mathit{float} \to J_2) \\
J_2 &= \mu\eta.(J_1 \to \mathit{int}) \times (\eta \to \mathit{float}).
\end{aligned}
$$

Trees corresponding to the two types are shown in Figures 3 and 4. The interface types I_1, J_2 are equivalent iff there exists a one-to-one mapping or a bijection from the methods in I_1 to the methods in J_2 such that each pair of methods in the bijection relation have the same type. The types of two

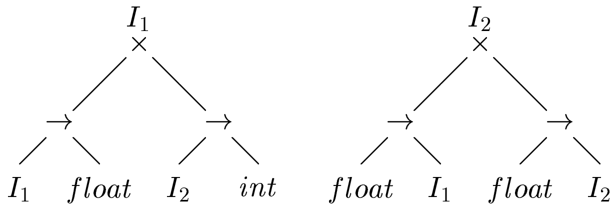

Figure 3. Trees for interfaces I_1 and I_2

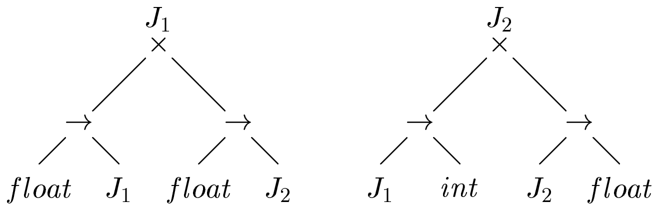

Figure 4. Trees for interfaces J_1 and J_2

methods are equal iff the types of the arguments and the return types are equal.

The equality of the interface types I_1 and J_2 can be determined by trying out all possible orderings of the methods in each interface and comparing the two types in the form of finite automata. In this case, there are only few possible orderings. However, if the number of methods is large and/or some methods take many arguments, the above approach becomes time consuming because the number of possible orderings grows exponentially. An efficient algorithm for determining equality of recursive types will be given later in the paper.

3. Definitions

A (uniform) recursive type is a type described by a set of equations involving the μ operator. An example of a recursive type was provided in Section 2. This section provides representation of recursive types as terms and term automata.

Term automata and representation of types are described in Subsection 3.1. A definition of type equivalence for recursive types in terms of bisimulation is give in Subsection 3.2. An efficient algorithm for determining whether two types are equivalent is given in Section 5.

3.1. Terms and Term Automata

Here we give a general definition of (possibly infinite) terms over an arbitrary finite ranked alphabet Σ. Such terms are essentially labeled trees, which we represent as partial functions labeling strings over ω (the natural numbers) with elements of Σ.

Let Σ_n denote the set of elements of Σ of arity n. Let ω denote the set of natural numbers and let ω^* denote the set of finite-length strings over the alphabet ω.

A *term* over Σ is a partial function

$$t : \omega^* \rightarrow \Sigma$$

satisfying the following properties:

- the domain of t is nonempty and prefix-closed;
- if $t(\alpha) \in \Sigma_n$, then $\{\ i \mid \alpha i \in$ the domain of $t\ \} = \{0, 1, \ldots, n-1\}$.

Let t be a term and $\alpha \in \omega^*$. Define the partial function $t\downarrow\alpha : \omega^* \to \Sigma$ by $t\downarrow\alpha(\beta) = t(\alpha\beta)$. If $t\downarrow\alpha$ has nonempty domain, then it is a term, and is called the *subterm of t at position* α.

A term t is said to be *regular* if it has only finitely many distinct subterms; that is, if $\{t\downarrow\alpha \mid \alpha \in \omega^*\}$ is a finite set.

Every regular term over a finite ranked alphabet Σ has a finite representation in terms of a special type of automaton called a *term automaton*. A term automaton over Σ is a tuple

$$A = (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

- Q is a finite set of *states*,
- $q_0 \in Q$ is the *start state*,
- $\delta : Q \times \omega \to Q$ is a partial function called the *transition function*, and
- $\ell : Q \to \Sigma$ is a (total) *labeling function*,

such that for any state $q \in Q$, if $\ell(q) \in \Sigma_n$ then

$$\{i \mid \delta(q,i) \text{ is defined}\} = \{0, 1, \ldots, n-1\}\ .$$

The partial function δ extends naturally to an inductively-defined partial function:

$$\begin{aligned} \hat{\delta} &: Q \times \omega^* \to Q \\ \hat{\delta}(q,\epsilon) &= q \\ \hat{\delta}(q,\alpha i) &= \delta(\hat{\delta}(q,\alpha), i). \end{aligned}$$

For any $q \in Q$, the domain of the partial function $\lambda\alpha.\hat{\delta}(q,\alpha)$ is nonempty (it always contains ϵ) and prefix-closed. Moreover, because of the condition on the existence of i-successors in the definition of term automata, the partial function

$$\lambda\alpha.\ell(\hat{\delta}(q,\alpha))$$

is a term.

Let A be a term automaton. The term *represented by* A is the term

$$t_A = \lambda\alpha.\ell(\hat{\delta}(q_0,\alpha))\ .$$

A term t is said to be *representable* if $t = t_A$ for some A.

Intuitively, $t_A(\alpha)$ is determined by starting in the start state q_0 and scanning the input α, following transitions of A as far as possible. If it is not possible to scan all of α because some i-transition along the way does not exist, then $t_A(\alpha)$ is undefined. If on the other hand A scans the entire input α and ends up in state q, then $t_A(\alpha) = \ell(q)$.

It is straightforward to show that a term t is regular if and only if it is representable. Moreover, a term t is regular if and only if it can be described by a finite set of equations involving the μ operator.

3.2. Equivalence of Recursive Types

A recursive type is a regular term over the ranked alphabet

$$\Sigma = \Gamma \cup \{\rightarrow\} \cup \{\prod^{n}, n \geq 2\},$$

where Γ is a set of base types, $\rightarrow$ is binary, and $\prod^n$ is of arity n. Given a type σ, if $\sigma(\epsilon) = \rightarrow$, $\sigma(0) = \sigma_1$, and $\sigma(1) = \sigma_2$, then we write the type as $\sigma_1 \rightarrow \sigma_2$. If $\sigma(\epsilon) = \prod^n$ and $\sigma(i) = \sigma_i$, $\forall i \in \{0, 1, \ldots, n-1\}$, then we write the type σ as $\prod_{i=0}^{n-1} \sigma_i$.

A syntactic presentation of a recursive type can be generated by the grammar:

$$t \ ::= \ \Gamma \ \mid \ t \rightarrow t \ \mid \ \prod_{i=1}^{n} t_i \ \mid \ \mu\alpha.t \ \mid \ \alpha$$

where α ranges over type variables. There are standard algorithms for translating syntactic presentations of types into term automata, and vice versa, see (Kozen et al., 1995) for a summary.

Palsberg and Zhao (2001) presented three equivalent definitions of type equivalence. Here we will work with the one which is based on the idea of bisimilarity. A relation R on types is called a *bisimulation* if it satisfies the following three conditions:

- if $(\sigma, \tau) \in R$, then $\sigma(\epsilon) = \tau(\epsilon)$
- if $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$
- if $(\prod_{i=0}^{n-1} \sigma_i, \prod_{i=0}^{n-1} \tau_i) \in R$, then there exists a bijection $b : \{0..n-1\} \rightarrow \{0..n-1\}$ such that $\forall i \in \{0..n-1\}$, $(\sigma_i, \tau_{b(i)}) \in R$.

Bisimulations are closed under union, therefore, there exists a largest bisimulation

$$\mathcal{R} \ = \ \bigcup \{ \ R \ \mid \ R \text{ is a bisimulation } \}.$$

It is straightforward to show that $\mathcal{R}$ is an equivalence relation. Two types τ_1 and τ_2 are said to be *equivalent* (denoted by $\tau_1 \cong \tau_2$) iff $(\tau_1, \tau_2) \in \mathcal{R}$.

4. Linear-time type equivalence for nonrecursive types

Assume that we are given two nonrecursive types that are represented as two term automata. Corollary 2 uses an algorithm of Paige and Yang to decide type matching in $O(n)$ time, based on the notion of multiset discrimination.

Multiset discrimination is a simple, highly efficient technique for finding and eliminating duplicate values in a structured collection of values, even in the presence of associative (A), associative-commutative (AC), and associative-commutative-idempotent (ACI) operators; that is, list, multiset and set comprehensions. It was pioneered by Paige, Tarjan and Bonic (1985; 1987) for improved DFA minimization and lexicographic sorting; it has been applied to an array of language processing problems demonstrating that hashing and pointer (array index) arithmetic can be avoided, with improved worst-case complexity (Cai and Paige, 1991); it has been extended to provide complete dagification of forests and acyclic dags even in the presence of A, AC and ACI operators (Paige, 1994; Paige and Yang, 1997).

In the terminology of this paper, complete dagification is the transformation of a term automaton $A = (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$ to a term automaton $A^d = (Q^d,\ \Sigma,\ q_0^d,\ \delta^d,\ \ell^d)$ such that the following properties hold.

1. There is a surjective function $f : Q \longrightarrow Q^d$ such that q and $f(q)$ represent equivalent terms (are bisimilar) for all $q \in Q$.
2. For all $q, q' \in Q$, q and q' represent equivalent terms (are bisimilar) if and only if $f(q) = f(q')$.

Intuitively, complete dagification collapses all equivalent states in the original term automaton into a single node. We can associate the state $f(q)$ with q by representing q as a record with a field containing its dagified state $f(q)$. After complete dagification of A we can answer any equivalence query: "Given $q, q' \in A$, are q and q' bisimilar (do they represent equivalent types)?" in constant time: Look up $f(q)$ and $f(q')$ in the records for q and q', respectively, and check if they are equal. If so, q and q' are bisimilar; if not, q and q' are not bisimilar.

Definition 1 (Acyclic term automaton). *A term automaton is* acyclic *if there exists a total order $<$ on Q such that $\delta(q,i) > q$ for all q, i for which $\delta(q,i)$ is defined.*

In other words, a term automaton is acyclic if it contains no cycle of transitions. Acyclic term automata represent all and only nonrecursive types (types formed without the use of the fixpoint operator μ). A term automaton is a forest (set of trees), if every state has at most one predecessor; that is, for each $q \in Q$ there is at most one $q' \in Q$ such that $\delta(q', i) = q$ for some i.

Theorem 1 (Paige, Yang (Paige, 1994**), (**Paige and Yang, 1997**)).** *Let $A = (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$ be an acyclic term automaton and $q_1, q_2 \in Q$. We can decide in time $O(n+m)$ whether q_1 and q_2 represent equivalent terms; that is, whether $(q_1, q_2) \in \mathcal{R}$. Here, $n = |Q|$, the size of Q, and $m = |\delta|$, the number of transitions in δ.*

Proof. If A is a pair of trees rooted at q_1 and q_2, respectively, then this lemma is a direct consequence of Paige and Yang (1997, Theorem 2). The complete dagification of the term automaton corresponds to Stage 2 of Paige and Yang's transformation of an input string in their *external language* to an *abstract syntax dag (ASD)*. The term $\tau \rightarrow \tau'$ here corresponds to the the tuple (list) $[\rightarrow, \tau, \tau']$ in their external language, and $\tau_1 \times \ldots \times \tau_n$ corresponds to $[\times, \langle \tau_1, \ldots, \tau_n \rangle]$. Note that the angled brackets in $\langle \tau_1, \ldots, \tau_n \rangle$ signal a multiset expression, which captures the associativity and commutativity of the $\prod^n$-operator in our term language.

Paige and Yang's ASD construction also works in time $O(m+n)$ if A is already partially dagified. This follows from Lemmas 1 and 2 in their paper. □

Let us define the size of a term automaton $A = (Q, \Sigma, q_0, \delta, l)$ to be $|Q| + |\delta|$; i.e., the sum of the number of states and transitions in the automaton.

Corollary 2. *For nonrecursive types τ_1, τ_2 represented by acyclic term automata A_1, A_2, each of size at most n, we can decide $(\tau_1, \tau_2) \in \mathcal{R}$ in $O(n)$ time.*

Proof. The two automata A_1, A_2 can be turned into a single automaton of size at most $2n$, by taking the disjoint union of their states and transitions. The result then follows from Theorem 1. □

5. $O(n \log n)$ time type equivalence for recursive types

Assume that we are given two recursive types τ_1 and τ_2 that are represented as two term automata A_1 and A_2. Lemma 3 proves that $\tau_1 \cong \tau_2$ (or $(\tau_1, \tau_2) \in \mathcal{R}$) if and only if there is a reflexive bisimulation C between A_1 and A_2 such that the initial states of the term automata A_1 and A_2 are related by C. Lemma 5 essentially reduces the problem of finding a reflexive bisimulation C between A_1 and A_2 to finding a size-stable coarsest partition (Paige and Tarjan, 1987). Theorem 6 uses the algorithm of Paige and Tarjan to determine in $O(n \log n)$ time (n is the sum of the sizes of the two term automata) whether there exists a reflexive bisimulation C between A_1 and A_2.

Throughout this section, we will use A_1, A_2 to denote two term automata over the alphabet Σ:

$$\begin{aligned} A_1 &= (Q_1, \Sigma, q_{01}, \delta_1, \ell_1) \\ A_2 &= (Q_2, \Sigma, q_{02}, \delta_2, \ell_2). \end{aligned}$$

We assume that $Q_1 \cap Q_2 = \emptyset$. Define $Q = Q_1 \cup Q_2$, $\delta : Q \times \omega \to Q$ where $\delta = \delta_1 \oplus \delta_2$, and $\ell : Q \to \Sigma$, where $\ell = \ell_1 \oplus \ell_2$, where $\oplus$ denotes disjoint union of two functions. We say that A_1, A_2 are *bisimilar* if and only if there exists a relation $C \subseteq Q \times Q$, called a bisimulation between A_1 and A_2, such that:

- if $(q, q') \in C$, then $\ell(q) = \ell(q')$
- if $(q, q') \in C$ and $\ell(q) = \to$, then $(\delta(q, 0), \delta(q', 0)) \in C$ and $(\delta(q, 1), \delta(q', 1)) \in C$
- if $(q, q') \in C$ and $\ell(q) = \prod^n$, then there exists a bijection $b : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\}$: $(\delta(q, i), \delta(q', b(i))) \in C$.

Notice that the bisimulations between A_1 and A_2 are closed under union, therefore, there exists a largest bisimulation between A_1 and A_2. It is straightforward to show that the identity relation on Q is a bisimulation, and that any reflexive bisimulation is an equivalence relation. Hence, the largest bisimulation is an equivalence relation.

Lemma 3. *For types τ_1, τ_2 that are represented by the term automata A_1, A_2, respectively, we have $(\tau_1, \tau_2) \in \mathcal{R}$ if and only if there is a reflexive bisimulation C between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$.*

Proof. Suppose $(\tau_1, \tau_2) \in \mathcal{R}$. Define:

$$C \ = \ \{ \ (q, q') \in Q \times Q \ \mid \ (\lambda\alpha.\ell(\hat{\delta}(q, \alpha)), \lambda\alpha.\ell(\hat{\delta}(q', \alpha))) \in \mathcal{R} \ \}.$$

It is straightforward to show that C is a bisimulation between A_1 and A_2, and that $(q_{01}, q_{02}) \in C$, we omit the details.

Conversely, let C be a reflexive bisimulation between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$. Define:

$$R \ = \ \{ \ (\sigma_1, \sigma_2) \ \mid \ (q, q') \in C \ \wedge \ \sigma_1 = \lambda\alpha.\ell(\hat{\delta}(q, \alpha)) \ \wedge \ \sigma_2 = \lambda\alpha.\ell(\hat{\delta}(q', \alpha)) \ \}$$

From $(q_{01}, q_{02}) \in C$, we have $(\tau_1, \tau_2) \in R$. It is straightforward to prove that R is a bisimulation, we omit the details. From $(\tau_1, \tau_2) \in R$ and R being a bisimulation, we conclude that $(\tau_1, \tau_2) \in \mathcal{R}$. □

A *partitioned graph* is a 3-tuple (U, E, P), where U is a set of nodes, $E \subseteq U \times U$ is an edge relation, and P is a *partition* of U. A partition P of U is a set of pairwise disjoint subsets of U whose union is all of U. The elements of P are called its *blocks*. If P and S are partitions of U, then S is a *refinement* of P if and only if every block of S is contained in a block of P.

A partition S of a set U can be characterized by an equivalence relation K on U such that each block of S is an equivalence class of K. If U is a set and K is an equivalence relation on U, then we use U/K to denote the partition of U into equivalence classes for K.

A partition S is *size-stable* with respect to E if and only if for all blocks $B_1, B_2 \in S$, and for all $x, y \in B_1$, we have $|E(x) \cap B_2| = |E(y) \cap B_2|$, where $E(x)$ is the following set

$$\{y | (x, y) \in E\} \ .$$

If E is clear from the context, we will simply use size-stable. We will repeatedly use the following characterization of size-stable partitions.

Lemma 4. *For an equivalence relation K, we have that U/K is size-stable if and only if for all $(u, u') \in K$, there exists a bijection $\pi : E(u) \to E(u')$ such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$.*

Proof. Suppose that U/K is size-stable. Let $(u, u') \in K$. Let B_1 be the block of U/K which contains u and u'. For each block B_2 of U/K, we have that $|E(u) \cap B_2| = |E(u') \cap B_2|$. So, for each block B_2 of U/K, we can construct a bijection from $E(u) \cap B_2$ to $E(u') \cap B_2$, such that for all $u_1 \in E(u) \cap B_2$, we have $(u_1, \pi(u_1)) \in K$. These bijections can then be merged to single bijection $\pi : E(u) \to E(u')$ with the desired property.

Conversely, suppose that for all $(u, u') \in K$, there exists a bijection $\pi : E(u) \to E(u')$ such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$. Let $B_1, B_2 \in U/K$, and let $x, y \in B_1$. We have that $(x, y) \in K$, so there exists a bijection $\pi : E(x) \to E(y)$ such that for all $u_1 \in E(x)$, we have $(u_1, \pi(u_1)) \in K$. Each element of $E(x) \cap B_2$ is mapped by π to an element of $E(y) \cap B_2$. Moreover, each element of $E(y) \cap B_2$ must be the image under π of an element of $E(x) \cap B_2$. We conclude that π restricted to $E(x) \cap B_2$ is a bijection to $E(y) \cap B_2$, so $|E(x) \cap B_2| = |E(y) \cap B_2|$. □

Given two term automata A_1, A_2, we define a partitioned graph (U, E, P):

$$\begin{array}{rcl} U & = & Q \cup \{ \langle q,i \rangle \mid q \in Q \ \wedge\ \delta(q,i) \text{ is defined } \} \\ E & = & \{ (q, \langle q,i \rangle) \mid \delta(q,i) \text{ is defined } \} \\ & & \cup\ \{ (\langle q,i \rangle, \delta(q,i)) \mid \delta(q,i) \text{ is defined } \} \\ L & = & \{ (q,q') \in Q \times Q \mid \ell(q) = \ell(q') \} \\ & & \cup\ \{ (\langle q,i \rangle, \langle q',i' \rangle) \mid \ell(q) = \ell(q') \text{ and if } \ell(q) = \rightarrow, \text{ then } i = i' \} \\ P & = & U/L. \end{array}$$

The graph contains one node for each state and transition in A_1, A_2. Each transition in A_1, A_2 is mapped to two edges in the graph. This construction ensures that if a node in the graph corresponds to a state labeled $\prod^n$, then that node will have n distinct successors in the graph. This is convenient when establishing a bijection between the successors of two nodes labeled $\prod^n$.

The equivalence relation L creates a distinction between the two successors of a node that corresponds to a state labeled $\rightarrow$. This is done by ensuring that if $(\langle q,i \rangle, \langle q,i' \rangle) \in L$ and $\ell(q) = \rightarrow$, then $i = i'$. This is convenient when establishing a bijection between the successors of two nodes labeled $\rightarrow$.

Lemma 5. *There exists a reflexive bisimulation C between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$ if and only if there exists a size-stable refinement S of P such that q_{01} and q_{02} belong to the same block of S.*

Proof. Let $C \subseteq Q \times Q$ be a reflexive bisimulation between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$. Define an equivalence relation $K \subseteq U \times U$ such that:

$$\begin{array}{rcl} K & = & C \\ & & \cup\ \{ (\langle q,i \rangle, \langle q',i \rangle) \mid (q,q') \in C \ \wedge\ \ell(q) = \ell(q') = \rightarrow \} \\ & & \cup\ \{ (\langle q,i \rangle, \langle q',i' \rangle) \mid (q,q') \in C \ \wedge\ (\delta(q,i), \delta(q',i')) \in C \\ & & \qquad\qquad\qquad\qquad\qquad \wedge\ \ell(q) = \ell(q') \ \wedge\ \ell(q) \neq \rightarrow \} \\ S & = & U/K. \end{array}$$

From $(q_{01}, q_{02}) \in C$, we have $(q_{01}, q_{02}) \in K$, so q_{01} and q_{02} belong to the same block of S. We will now show that S is a size-stable refinement of P.

Let $(u, u') \in K$. From Lemma 4 we have that it is sufficient to show that there exists a bijection $\pi : E(u) \rightarrow E(u')$, such that for all $u_1 \in E(u)$, we have $(u_1, \pi(u_1)) \in K$. There are three cases.

First, suppose $(u, u') \in C$. We have

$$\begin{array}{rcl} E(u) & = & \{ \langle u,i \rangle \mid \delta(u,i) \text{ is defined } \} \\ E(u') & = & \{ \langle u',i' \rangle \mid \delta(u',i') \text{ is defined } \}. \end{array}$$

Let us consider each of the possible cases of u and u'. If $\ell(u) = \ell(u') \in \Gamma$, then $E(u) = E(u') = \emptyset$, and the desired bijection exists trivially. Next, if $\ell(u) = \ell(u') = \rightarrow$, then

$$\begin{array}{rcl} E(u) & = & \{ \langle u,0 \rangle, \langle u,1 \rangle \} \\ E(u') & = & \{ \langle u',0 \rangle, \langle u',1 \rangle \}, \end{array}$$

so the desired bijection is $\pi : E(u) \rightarrow E(u')$, where $\pi(\langle u,0 \rangle) = \langle u',0 \rangle$ and $\pi(\langle u,1 \rangle) = \langle u',1 \rangle$, because $(\langle u,0 \rangle, \langle u',0 \rangle) \in K$ and $(\langle u,1 \rangle, \langle u',1 \rangle) \in K$.

Finally, if $\ell(u) = \ell(u') = \prod^n$, then

$$\begin{aligned} E(u) &= \{ \langle u, i \rangle \mid \delta(u,i) \text{ is defined } \} \\ E(u') &= \{ \langle u', i' \rangle \mid \delta(u',i') \text{ is defined } \}. \end{aligned}$$

From $(u, u') \in C$, we have a bijection $b : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\} : (\delta(u,i), \delta(u', b(i))) \in C$. From that, the desired bijection can be constructed.

Second, suppose $u = \langle q, i \rangle$ and $u' = \langle q', i \rangle$, where $(q, q') \in C$, and $\ell(q) = \ell(q') = \rightarrow$. We have

$$\begin{aligned} E(u) &= \{ \delta(q,i) \} \\ E(u') &= \{ \delta(q',i) \}, \end{aligned}$$

and from $(q, q') \in C$ we have $(\delta(q,i), \delta(q',i)) \in C \subseteq K$, so the desired bijection exists.

Third, suppose $u = \langle q, i \rangle$ and $u' = \langle q', i' \rangle$, where $(q, q') \in C$, $(\delta(q,i), \delta(q',i')) \in C$, $\ell(q) = \ell(q')$, and $\ell(q) \neq \rightarrow$. We have

$$\begin{aligned} E(u) &= \{ \delta(q,i) \} \\ E(u') &= \{ \delta(q',i') \}, \end{aligned}$$

and $(\delta(q,i), \delta(q',i')) \in C \subseteq K$, so the desired bijection exists.

Conversely, let S be a size-stable refinement of P such that q_{01} and q_{02} belong to the same block of S. Define:

$$\begin{aligned} K &= \{ (u, u') \in U \times U \mid u, u' \text{ belong to the same block of } S \} \\ C &= K \cap (Q \times Q). \end{aligned}$$

Notice that $(q_{01}, q_{02}) \in C$ and that C is reflexive. We will now show that C is a bisimulation between A and A'.

First, suppose $(q, q') \in C$. From S being a refinement of P we have $(q, q') \in L$, so $\ell(q) = \ell(q')$.

Second, suppose $(q, q') \in C$ and $\ell(q) = \rightarrow$. From the definition of E we have

$$\begin{aligned} E(q) &= \{ \langle q, 0 \rangle, \langle q, 1 \rangle \} \\ E(q') &= \{ \langle q', 0 \rangle, \langle q', 1 \rangle \}. \end{aligned}$$

From S being size-stable, $(q, q') \in C \subseteq K$, and Lemma 4 we have that there exists a bijection $\pi : E(q) \to E(q')$ such that for all $u \in E(q)$ we have that $(u, \pi(u)) \in K$. From $K \subseteq L$ and $\ell(q) = \rightarrow$ we have that there is only one possible bijection π:

$$\begin{aligned} \pi(\langle q, 0 \rangle) &= \langle q', 0 \rangle \\ \pi(\langle q, 1 \rangle) &= \langle q', 1 \rangle, \end{aligned}$$

so $(\langle q, 0 \rangle, \langle q', 0 \rangle) \in K$ and $(\langle q, 1 \rangle, \langle q', 1 \rangle) \in K$. From the definition of E we have, for $i \in \{0, 1\}$,

$$\begin{aligned} E(\langle q, i \rangle) &= \delta(q,i) \\ E(\langle q', i \rangle) &= \delta(q',i), \end{aligned}$$

and since S is size-stable, we have, for $i \in \{0, 1\}$, $(\delta(q,i), \delta(q',i)) \in K$. Moreover, for $i \in \{0, 1\}$, $(\delta(q,i), \delta(q',i)) \in Q \times Q$, so we conclude, $(\delta(q,i), \delta(q',i)) \in C$.

Third, suppose $(q, q') \in C$ and $\ell(q) = \prod^n$. From the definition of E we have

$$\begin{aligned} E(q) &= \{ \langle q, i \rangle \mid \delta(q,i) \text{ is defined } \} \\ E(q') &= \{ \langle q', i \rangle \mid \delta(q',i) \text{ is defined } \}. \end{aligned}$$

Notice that $|E(q)| = |E(q')| = n$. From S being size-stable, $(q, q') \in C \subseteq K$, and Lemma 4, we have that there exists a bijection $\pi : E(q) \to E(q')$ such that for all $u \in E(q)$ we have that $(u, \pi(u)) \in K$. From π we can derive a bijection $b : \{0..n-1\} \to \{0..n-1\}$ such that $\forall i \in \{0..n-1\}$: $(\langle q, i \rangle, \langle q', b(i) \rangle) \in K$. From the definitions of E and E' we have that for $i \in \{0..n-1\}$,

$$\begin{aligned} E(\langle q, i \rangle) &= \{ \delta(q,i) \} \\ E(\langle q', i \rangle) &= \{ \delta(q',i) \}, \end{aligned}$$

and since S is size-stable, and, for all $i \in \{0..n-1\}$, $(\langle q, i \rangle, \langle q', b(i) \rangle) \in K$, we have $(\delta(q,i), \delta(q', b(i))) \in K$. Moreover, $(\delta(q,i), \delta(q', b(i))) \in Q \times Q$, so we conclude $(\delta(q,i), \delta(q', b(i))) \in C$. □

Recall that the size of a term automaton $A = (Q, \Sigma, q_0, \delta, l)$ is $|Q| + |\delta|$; i.e., the sum of the number of states and transitions in the automaton.

Theorem 6. *For types τ_1, τ_2 represented by term automata A_1, A_2 of size at most n, we can decide $(\tau_1, \tau_2) \in \mathcal{R}$ in $O(n \log n)$ time.*

Proof. From Lemma 3 we have that $(\tau_1, \tau_2) \in \mathcal{R}$ if and only if there is a reflexive bisimulation C between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$. From Lemma 5 we have that there exists a reflexive bisimulation C between A_1 and A_2 such that $(q_{01}, q_{02}) \in C$ if and only if there exists a size-stable refinement S of P such that q_{01} and q_{02} belong to the same block of S.

Paige and Tarjan (1987) give an $O(m \log p)$ algorithm to find the coarsest size-stable refinement of P, where m is the size of E and p is the size of the universe U.

Our algorithm first constructs (U, E, P) from A_1 and A_2, then runs the Paige-Tarjan algorithm to find the coarsest size-stable refinement S of P, and finally checks whether q_{01} and q_{02} belong to the same block of S.

If A_1 and A_2 are of size at most n, then the size of E is at most $2n$, and the size of U is at most $2n$, so the total running time of our algorithm is $O(2n \log(2n)) = O(n \log n)$. □

Next, we illustrate how our algorithm determines that equivalence between the types. Details of the algorithm can be found in (Paige and Tarjan, 1987). Consider two types I_1 and J_1 defined in Section 2. The set of types corresponding to the two interfaces are:

$$\{I_1, I_2, m_1, m_2, m_3, m_4, \mathit{int}, \mathit{float}\}$$
$$\{J_1, J_2, n_1, n_2, n_3, n_4, \mathit{int}, \mathit{float}\}$$

Note that we abuse notation and use m_1, m_2, etc, to denote the *types* of the methods with those names. Figure 5 shows various steps of our algorithm.

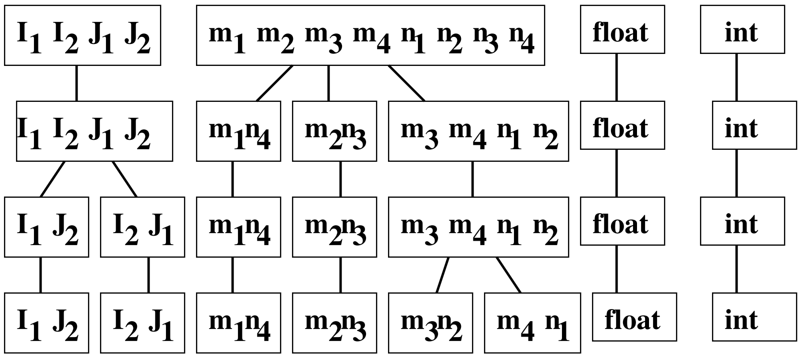

Figure 5. Blocks of types

For simplicity, the figure only shows the blocks of actual types, but not the blocks of the extra nodes of the form $\langle q, i \rangle$. The blocks in the first row are based on labels, e.g., states labeled with $\times$ are in the same block. In the next step, the block containing the methods are split based on the type of the result of the method, e.g.. methods m_1 and n_4 both return *float*, so they are in the same block. In the next step (corresponding to the third row) the block $\{I_1, I_2, J_1, J_2\}$ are split. The final partition, where block $\{m_3, m_4, n_1, n_2\}$ is split, is shown in the fourth row.

Our algorithm can be tuned to take a specific user needs into account. This is done simply by modifying the definition of the equivalence relation L. For example, suppose a user cares about the order of the arguments to a method. This means that the components of the product type that models the argument list should not be allowed to be shuffled during type matching. We can prevent shuffling by employing the same technique that the current definition of L uses for function types. The idea is to insist that two component types may only be matched when they have the same component index.

Another example of the tunability of our algorithm involves the modifiers in `Java`. Suppose a programmer is developing a product that is multi-threaded. In this case the programmer may only want to match `synchronized` methods with other `synchronized` methods. This can be handled easily in our framework by changing L such that two method types may only be matched when they are both synchronized. On the other hand if the user is working on a single-threaded product, the keyword `synchronized` can be ignored. The same observation applies to other modifiers such as `static`.

6. Our Implementation

We have implemented our algorithm in `Java` and the current version is based on the code written by Wanjun Wang. The implementation and documentation are freely available at

`http://www.cs.purdue.edu/homes/tzhao/matching/matching.htm.`

The current version has a graphical user interface so that users may input type definitions written in a file and also may specify restrictions on type isomorphism.

Suppose we are given the following file with four `Java` interfaces.

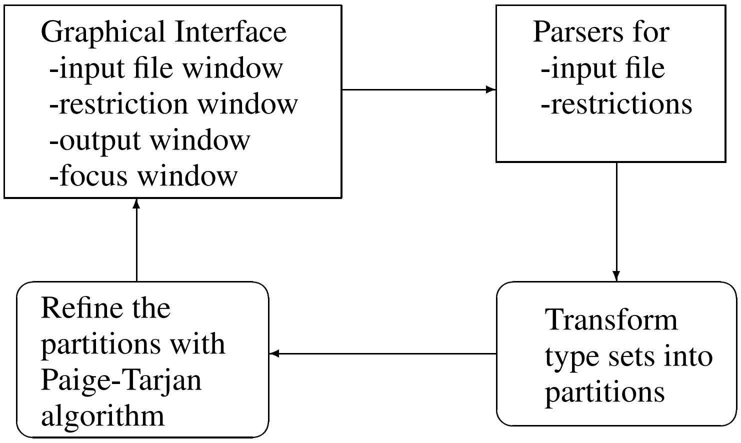

Figure 6. Schematic diagram for the implementation

interface I_1 {
  float m_1 $(I_1\ a, \textit{int}\ b)$;
  int m_2 $(I_2\ a)$;
}

interface I_2 {
  J_2 m_3 $(\textit{float}\ a)$;
  I_1 m_4 $(\textit{float}\ a)$;
}

interface J_1 {
  I_1 n_1 $(\textit{float}\ a)$;
  J_2 n_2 $(\textit{float}\ a)$;
}

interface J_2 {
  int n_3 $(J_1\ a)$;
  float n_4 $(\textit{int}\ a, J_2\ b)$;
}

The implementation, as illustrated in the Figure 6, will read and parse the input file and then transform the type definitions into partitions of numbers with each type definition and dummy type assigned a unique number. The partitions will be refined by Paige-Tarjan algorithm until it is *size-stable* as defined in this paper. Finally, we will be able to read the results from the final partitions. Two types are isomorphic if the numbers assigned to them are in the same partition.

The implementation will give the following output:

$$\begin{aligned} I_1 &= J_2 \\ I_2 &= J_1 \end{aligned}$$

$$\begin{aligned} I_1.m_1 &= J_2.n_4 \\ I_2.m_3 &= I_2.m_4 = J_1.n_1 = J_1.n_2 \\ I_1.m_2 &= J_2.n_3\ . \end{aligned}$$

We can see that the types of interfaces I_2 and J_1 are isomorphic and moreover, all method types of I_2, J_1 match. Suppose that we have additional information about the method types such that only method m_3 and n_1 should have isomorphic types. We can restrict the type matching by adding $I_2.m_3 = J_1.n_1$ to the *restrictions* window of the user interface. The new matching result is illustrated by the screen shot in figure 7.

Note that we are able to focus on the matching of two interface types such as I_2, J_1 as in the *focus* windows of Figure 7, where I_2, J_1 are matched and their methods are matched one to one.

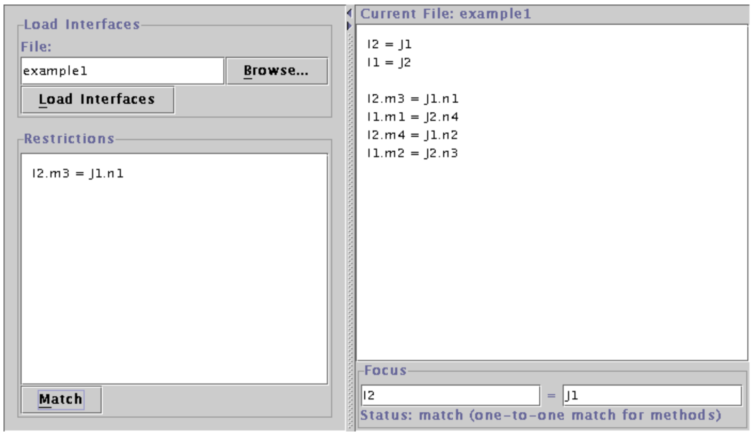

Figure 7. Screen shot

interface I_1 {
    $I_1\ \ m\ \ (\mathit{float}\ a, \mathit{boolean}\ b);$
    $\mathit{boolean}\ \ p\ \ (I_1\ j);$
}

interface I_2 {
    $I_2\ \ m\ \ (\mathit{int}\ i, \mathit{boolean}\ b);$
}

Figure 8. Interfaces I_1 and I_2

7. Subtyping of Recursive Types

In this section we discuss subtyping and formalize it using a simulation relation. We also discuss reasons why the algorithm given in Section 5 is not applicable to subtyping of of recursive types. Consider the interfaces I_1 and I_2 shown in Figure 8, and suppose a user is looking for I_2. The interfaces I_1 and I_2 can be mapped to the following recursive types:

$$\begin{aligned} \tau_1 &= \mu\alpha.((\mathit{float} \times \mathit{boolean}) \to \alpha) \times (\alpha \to \mathit{boolean}) \\ \tau_2 &= \mu\beta.(\mathit{int} \times \mathit{boolean}) \to \beta) \end{aligned}$$

Assuming that int is a subtype of float (we can always coerce integers into floats) we have that τ_1 is a subtype of τ_2. Therefore, the user can use the interface I_1. There are several points to notice from this example. In the context of subtyping, we need two kinds of products: one that models a collection of methods and another that models sequence of parameters. In our example, the user only specified a type corresponding to method m. Therefore, during the subtyping algorithm method p should be ignored. However, the parameters of method m are also modeled using products and none of these can be ignored. Therefore, we consider two types of product type constructors in our type systems and the subtyping rule for these two types of products are different.

As stated before, a type is a regular term, in this case over the ranked alphabet

$$\Sigma \ = \ \Gamma \cup \{\to\} \cup \{\prod^{n}, n \geq 2\} \cup \{\times^{n}, n \geq 2\}.$$

Roughly speaking, $\prod^n$ and $\times^n$ will model collection of parameters and methods respectively. Also assume that we are given a subtyping relation on the base types Γ. If τ_1 is a subtype of τ_2, we will write it as $\tau_1 \preceq \tau_2$. A relation S is called a *simulation* on types if it satisfies the following conditions:

- if $(\sigma, \tau) \in S$ and $\sigma(\epsilon) \in \Gamma$, then $\tau(\epsilon) \in \Gamma$ and $\sigma(\epsilon) \preceq \tau(\epsilon)$.
- if $(\sigma, \tau) \in S$ and $\sigma(\epsilon) \in (\{\rightarrow\} \cup \{\prod^n, n \geq 2\})$, then $\sigma(\epsilon) = \tau(\epsilon)$.
- if $(\sigma_1 \rightarrow \sigma_2, \tau_1 \rightarrow \tau_2) \in S$, then $(\tau_1, \sigma_1) \in S$ and $(\sigma_2, \tau_2) \in S$.
- if $(\prod_{i=0}^{n-1} \sigma_i, \prod_{i=0}^{n-1} \tau_i) \in S$, then there exists a bijection $b : \{0 \cdots n-1\} \rightarrow \{0 \cdots n-1\}$ such that for all $i \in \{0 \cdots n-1\}$, we have $(\sigma_i, \tau_{b(i)}) \in S$.
- Suppose $(\sigma, \tau) \in S$, $\sigma(\epsilon) = \times^n$, and $\sigma = \times_{i=0}^{n-1} \sigma_i$. If $\tau(\epsilon) \notin \{\times^m, m \geq 2\}$, then there exists a $j \in \{0 \cdots n-1\}$ such that $(\sigma_j, \tau) \in S$. Otherwise, assume that $\tau(\epsilon) = \times^m$, where $m \leq n$ and $\tau = \times_{i=0}^{m-1} \tau_i$. In this case, then there exists an injective function $c : \{0 \cdots m-1\} \rightarrow \{0 \cdots n-1\}$ such that for all $i \in \{0 \cdots m-1\}$, we have $(\sigma_{c(i)}, \tau_i) \in S$. Notice that this rule allows ignoring certain components of σ.

As is the case with bisimulations, simulations are closed under union, therefore there exists a largest simulation (denoted by $\mathcal{S}$).

Let A_1, A_2 denote two term automata over Σ:

$$\begin{aligned} A_1 &= (Q_1, \Sigma, q_{01}, \delta_1, \ell_1) \\ A_2 &= (Q_2, \Sigma, q_{02}, \delta_2, \ell_2). \end{aligned}$$

We assume that $Q_1 \cap Q_2 = \emptyset$. Define $Q = Q_1 \cup Q_2$, $\delta : Q \times \omega \rightarrow Q$ where $\delta = \delta_1 \oplus \delta_2$, and $\ell : Q \rightarrow \Sigma$, where $\ell = \ell_1 \oplus \ell_2$, where $\oplus$ denotes disjoint union of two functions. We say that A_2 *simulates* A_1 (denoted by $A_1 \preceq A_2$) if and only if there exists a relation $D \subseteq Q \times Q$, called a *simulation relation* between A_1 and A_2, such that:

- if $(q, q') \in D$ and $\ell(q) \in \Gamma$, then $\ell(q') \in \Gamma$ and $\ell(q) \preceq \ell(q'))$.
- if $(q, q') \in D$ and $\ell(q) \in (\{\rightarrow\} \cup \{\prod^n, n \geq 2\})$, then $\ell(q) = \ell(q')$.
- if $(q, q'), \in D$ and $\ell(q) = \rightarrow$, then $(\delta(q, 0), \delta(q', 0)) \in D$ and $(\delta(q, 1), \delta(q', 1) \in D$.
- if $(q, q'), \in D$ and $\ell(q) = \prod^n$, then there exists a bijection $b : \{0 \cdots n-1\} \rightarrow \{0 \cdots n-1\}$ such that for all $i \in \{0 \cdots n-1\}$, we have $(\delta(q, i), \delta(q' i)) \in D$.
- Suppose $(q, q') \in D$ and $\ell(q) = \times^n$. If $\ell(q') \notin \{\times^m, m \geq 2\}$, then there exists a $j \in \{0 \cdots n-1\}$ such that $(\delta(q, j), q') \in D$. Otherwise, assume that $\ell(q') = \times^m$, where $m \leq n$ and in this case, there exists an injective function $c : \{0 \cdots m-1\} \rightarrow \{0 \cdots n-1\}$ such that for all $i \in \{0 \cdots m-1\}$, we have $(\delta(q, c(i)), \delta(q', i)) \in D$.

Notice that the simulations between A_1 and A_2 are closed under union, therefore, there exists a largest simulation between A_1 and A_2. The proof of Lemma 7 is similar to the proof of Lemma 3 and is omitted.

Lemma 7. *For types τ_1, τ_2 that are represented by the term automata A_1, A_2, respectively, we have $(\tau_1, \tau_2) \in \mathcal{S}$ if and if only there is a reflexive simulation D between A_1 and A_2 such that $(q_{01}, q_{02}) \in D$.*

The largest simulation between the term automata A_1 and A_2 is given by the following greatest fixed point

$$\nu D.sim(q, q', D).$$

where $D \subseteq Q_1 \times Q_2$ and the predicate $sim(q, q', D)$ is the conjunction of the five conditions which appear in the definition of the simulation relation between two automata. Let n and m be the size of the term automata A_1 and A_2, respectively. Since nm is a bound on the size of D, the number of iterations in computing the greatest fixed point is bounded by nm. In general, the relation D (or for that matter the simulation relation) is not symmetric. On the other hand, the bisimulation relation was an equivalence relation, and so could be represented as a partition on the set $Q_1 \cup Q_2$, or in other words, partitions give us a representation of an equivalence relation that is linear in the sum of the sizes of the set of states Q_1 and Q_2. The Paige-Tarjan algorithm uses the partition representation of the equivalence relation. Since D is not symmetric (and thus not an equivalence relation), it cannot be represented by a partition. This is the crucial reason why our previous algorithm cannot be used for subtyping.

8. Conclusion

In this paper we addressed the problem of matching recursive and nonrecursive types. We presented algorithms with $O(n \log n)$ and $O(n)$ time complexities that decide whether two types are equivalent. To our knowledge, these are the most efficient algorithms for type matching with and without type recursion, respectively. Our results are applicable to the problem of matching signatures of software components. Applications to `Java` were also discussed. Issues related to subtyping of recursive types were also addressed. Next we discuss potential applications of our algorithms.

CORBA: The CORBA approach utilizes a separate definition language called IDL. Objects are associated with language-independent interfaces defined in IDL. These interfaces are then translated into the language being used by the client. The translated interface then enables the clients to call the objects. Since the IDL interfaces have to be translated into several languages, their type system is very restrictive. Therefore, IDL interfaces lack expressive power because, intuitively speaking, the type system used in IDL has to be the intersection of the type system for each language it supports. The drawback of for CORBA-style approaches to interoperability are well articulated in (Auerbach and Chu-Carroll, 1997; Barrett et al., 1996).

Polyspin and Mockingbird: The Polyspin and Mockingbird approaches do not require a common interface language, such as IDL. In both these approaches, clients and objects are written in languages with separate type systems, and an operation that crosses the language boundary is supported by bridge code that is automatically generated. Therefore, systems such as Polyspin and Mockingbird support seamless interoperability since the programmer is not burdened with writing interfaces in a second language, such as IDL in CORBA. Polyspin supports only finite types. Mockingbird on the other hand supports recursive types, including records, linked lists, and arrays. The type system used in Mockingbird is called the *Mockingbird Signature Language* or *MockSL*. The problem of deciding type equivalence for MockSL remains open (Auerbach et al., 1998). In this paper we considered a type system which is related to the one used in Mockingbird. However, we are investigating a translation from *MockSL* to recursive types.

Megaprogramming: Techniques suitable for very large software systems have been a major goal of software engineering. The term *megaprogramming* was introduced by DARPA to motivate this goal (Boehm and Scherlis, 1992). Roughly speaking, in megaprogramming, *megamodules* provide a higher level of abstraction than modules or components. For example, a megamodule can encapsulate the entire logistics of ground transportation in a major city. Megaprogramming is explained in detail in (Wiederhold et al., 1992). Interoperability issues arise when megaprograms are constructed using megamodules, see (Wiederhold et al., 1992, Section 4.2). We believe that the framework presented in this paper can be used to address mismatch between interfaces of megamodules.

Independently of our work, Zibin, Gil, and Considine (2003) presented a linear-time type equivalence algorithm for nonrecursive types. Their algorithm is more general than ours in that it can handle a larger equivalence, namely one that also includes the axioms

$$\begin{aligned} 1 \to A &= A \\ A \times 1 &= A \\ (A \times B) \to C &= A \to (B \to C) \end{aligned}$$

where 1 is a unit type and A, B, C range over types.

In a continuation of our work, Di Cosmo, Pottier, and Rémy (2003) presented an algorithm for deciding the subtyping problem discussed in Section 7 .

Future work includes investigating type inference. the recent paper of Coppo (2001) on type inference with recursive type equations may contain techniques that are applicable.

Acknowledgments A preliminary version of the paper, authored by Jha, Palsberg, and Zhao, was presented at FOSSACS 2002. Palsberg was supported by an NSF CAREER award, CCR–9734265, and by IBM.

References

Amadio, R. M. and L. Cardelli: 1993, 'Subtyping Recursive Types'. *ACM Transactions on Progamming Languages and Systems* **15**(4), 575–631. Also in Proceedings of POPL'91.

Aponte, M.-V. and R. D. Cosmo: 1996, 'Type Isomorphisms for Module Signatures'. In: *Proceedings of PLILP '96*. pp. 334–346, Springer-Verlag (*LNCS* 1140).

Auerbach, J., C. Barton, and M. Raghavachari: 1998, 'Type Isomorphisms with Recursive Types'. Research report RC 21247, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY.

Auerbach, J. and M. C. Chu-Carroll: 1997, 'The Mockingbird System: A Compiler-based Approach to Maximally Interoperable Distributed Programming'. Research report RC 20718, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY.

Barrett, D. J., A. Kaplan, and J. C. Wileden: 1996, 'Automated Support for Seamless Interoperability in Polylingual Software Systems'. In: *ACM FSE'96, Fourth Symposium on the Foundations of Software Engineering*.

Boehm, B. and B. Scherlis: 1992, 'Megaprogramming'. In: *Proceedings of DARPA Software Technology Conference*.

Brandt, M. and F. Henglein: 1998, 'Coinductive axiomatization of recursive type equality and subtyping'. *Fundamenta Informaticae* **33**, 309–338. Invited submission to special issue featuring a selection of contributions to the 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), 1997.

Bruce, K. B., R. D. Cosmo, and G. Longo: 1992, 'Provable isomorphisms of types'. *Mathematical Structures in Computer Science* **2**(2), 231–247.

Cai, J. and R. Paige: 1991, 'Look Ma, No Hashing, and No Arrays Neither'. In: January (ed.): *Proc. 18th Annual ACM Symp. on Principles of Programming Languages (POPL), Orlando, Florida*. pp. 143–154.

Cardelli, L. and P. Wegner: 1985, 'On Understanding Types, Data Abstraction, and Polymorphism'. *ACM Computing Surveys* **17**(4), 471–522.

Coppo, M.: 2001, 'Type Inference with Recursive Type Equations'. In: *Proceedings of FOSSACS'01, Foundations of Software Science and Computation Structures*. pp. 184–198, Springer-Verlag (*LNCS* 2030).

Cosmo, R. D.: 1995, *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhäuser.

Cosmo, R. D., F. Pottier, and D. Rémy: 2003, 'Subtyping Recursive Types Modulo Associative Commutative Products'. Manuscript.

Courcelle, B.: 1983, 'Fundamental Properties of Infinite Trees'. *Theoretical Computer Science* **25**(1), 95–169.

Jim, T. and J. Palsberg: 1997, 'Type Inference in Systems of Recursive Types with Subtyping'. Manuscript.

Kozen, D., J. Palsberg, and M. I. Schwartzbach: 1995, 'Efficient Recursive Subtyping'. *Mathematical Structures in Computer Science* **5**(1), 113–125. Preliminary version in Proceedings of POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 419–428, Charleston, South Carolina, January 1993.

Narendran, P., F. Pfenning, and R. Statman: 1993, 'On the Unification Problem for Cartesian Closed Categories'. In: *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*. pp. 57–63.

OMG: 1999, 'The Common Object Request Broker: Architecture and Specification'. Technical report, Object Management Group. Version 2.3.1.

Paige, R.: 1994, 'Efficient Translation of External Input in a Dynamically Typed Language'. In: B. Pehrson and I. Simon (eds.): *Proc. 13th World Computer Congress, Vol. 1*. Elsevier Science B.V. (North Holland).

Paige, R. and R. Tarjan: 1987, 'Three Partition Refinement Algorithms'. *SIAM Journal on Computing* **16**(6), 973–989.

Paige, R., R. Tarjan, and R. Bonic: 1985, 'A Linear Time Solution to the Single Function Coarsest Partition Problem'. *Theoretical Computer Science* **40**, 67–84.

Paige, R. and Z. Yang: 1997, 'High Level Reading and Data Structure Compilation'. In: *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), Paris, France*. http://www.acm.org, pp. 456–469, ACM Press.

Palsberg, J. and T. Zhao: 2001, 'Efficient and Flexible Matching of Recursive Types'. *Information and Computation* **171**, 364–387. Preliminary version in Proceedings of LICS'00, Fifteenth Annual IEEE Symposium on Logic in Computer Science, pages 388–398, Santa Barbara, California, June 2000.

Rittri, M.: 1990, 'Retrieving Library Identifiers via Equational Matching of Types'. In: M. E. Stickel (ed.): *Proceedings of the 10th International Conference on Automated Deduction*, Vol. 449 of *LNAI*. Kaiserslautern, FRG, pp. 603–617, Springer Verlag.

Rittri, M.: 1991, 'Using Types as Search Keys in Function Libraries'. *Journal of Functional Programming* **1**(1), 71–89.

Rittri, M.: 1993, 'Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism'. *RAIRO Theoretical Informatics and Applications* **27**(6), 523–540.

Soloviev, S. V.: 1983, 'The category of finite sets and cartesian closed categories'. *Journal of Soviet Mathematics* **22**, 1387–1400.

Wiederhold, G., P. Wegner, and S. Ceri: 1992, 'Towards Megaprogramming: A Paradigm for Component-Based Programming'. *Communications of the ACM* **35**(11), 89–99.

Zaremski, A. M. and J. M. Wing: 1995a, 'Signature Matching: a Tool for Using Software Libraries'. *ACM Transactions on Software Engineering Methodology* **4**(2), 146–170.

Zaremski, A. M. and J. M. Wing: 1995b, 'Specification Matching of Software Components'. In: *Proceedings of 3rd ACM SIGSOFT Symposium on the Foundation of Software Engineering*. pp. 6–17.

Zibin, Y., Y. Gil, and J. Considine: 2003, 'Efficient algorithms for isomorphisms of simple types'. In: *Proceedings of POPL'03, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. pp. 160–171.