Efficient and Flexible Matching of Recursive Types

Jens Palsberg Tian Zhao
Purdue University*

Abstract

Equality and subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli; Kozen, Palsberg, and Schwartzbach; Brandt and Henglein; and others. Potential applications include automatic generation of bridge code for multi-language systems and type-based retrieval of software modules from libraries. Auerbach, Barton, and Raghavachari advocate a highly flexible combination of matching rules for which there, until now, are no efficient algorithmic techniques. In this paper, we present an efficient decision procedure for a notion of type equality that includes unfolding of recursive types, and associativity and commutativity of product types, as advocated by Auerbach et al. For two types of size at most n, our algorithm decides equality in $O(n^2)$ time. The algorithm iteratively prunes a set of type pairs, and eventually it produces a set of pairs of equal types. In each iteration, the algorithm exploits a so-called coherence property of the set of type pairs produced in the preceding iteration. The algorithm takes $O(n)$ iterations each of which takes $O(n)$ time, for a total of $O(n^2)$ time.

1 Introduction

Much previous work on type equality focuses on non-recursive types [8, 11, 21, 24, 25, 26, 27, 29]. In this paper we consider equality of recursive types.

Background. Potential applications of flexible type equality include automatic generation of bridge code for multi-language systems [4, 6], and type-based retrieval of software modules from libraries [24, 25, 26, 29].

Software engineers often look into a software library to find reusable components for their applications. A large library can be hard to search, however. It may be organized in alphabetical order or coarsely sorted according to some structure. Beyond the structural information of the library, the only thing that we can rely on is the component name to retrieve the code we need. Component names are difficult to guess. So, it makes sense to search by the type of the components. A component that fits the specification of a programmer does not always have the exact same type as the one the user is using as search key. That is why we need a flexible notion of type equality.

Designing and maintaining a multi-language application often calls for bridge code for components written in various programming languages such as C, C++ and Java. The conversion of values of isomorphic (equivalent) types is essential. The foundation of deciding whether a conversion makes sense at all is a flexible notion of type equality. An alternative might be to start with just one type, and then translate it into a type in a different language [14]. Such a translation may be helpful when building a new software component that should be connected to an existing one. However, when faced with connecting two existing software components, type matching and automatic bridge code generation seems more helpful.

In object-oriented languages such as C++ and Java, many types are recursive. Thus, to be useful for such languages, a flexible notion of type equality should be able to handle recursive types.

The Problem. Equality and subtyping of recursive types have been studied in the 1990s by Amadio and Cardelli [1]; Kozen, Palsberg, and Schwartzbach [19]; Brandt and Henglein [7]; Jim and Palsberg [17]; and others. These papers concentrate on the case where two types are considered equal if their infinite unfoldings are identical. Type equality can be decided in $O(n\alpha(n))$ time, and a notion of subtyping defined by Amadio and Cardelli [1] can be decided in $O(n^2)$ time [19].

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 [4]. Until now,

*Purdue University, Dept of Computer Science, W Lafayette, IN 47907, {palsberg,tzhao}@cs.purdue.edu.

there are no efficient algorithmic techniques for deciding type equality in this case. One approach would be to guess an ordering and a bracketing of all products, and then use a standard polynomial-time method for checking that the infinite unfoldings of the resulting types are identical. For types without infinite products, such an algorithm runs in NP time. One of the inherent problems with allowing the product-type constructor to be associative and commutative is that

$$A \times A \times B \;=\; A \times B \times A,$$

while

$$A \times A \times B \;\neq\; A \times B \times B.$$

Notice the significance of the multiplicity of a type in a product. One could imagine that an algorithm for deciding type equality would begin by determining the multiplicities of all components of product types, or even order the components. However, it seems like this would have to rely on being able to decide type equality for the component types, and because the types may be recursive, this seems to lead to a chicken-and-egg problem.

Our Result. We have developed an efficient decision procedure for a notion of type equality that includes unfolding of recursive types, and associativity and commutativity of product types, as advocated by Auerbach et al. For two types of size at most n, our algorithm decides equality in $O(n^2)$ time. The algorithm iteratively prunes a set of type pairs, and eventually it produces a set of pairs of equal types. In each iteration, the algorithm exploits a so-called coherence property of the set of type pairs produced in the preceding iteration. The algorithm takes $O(n)$ iterations each of which takes $O(n)$ time, for a total of $O(n^2)$ time.

Implementation. We have implemented a type-matching tool based on our algorithm. The tool is for matching Java interfaces. It supports 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. When given two Java interfaces, our tool will determine whether they are equivalent, and if they are, it will present the user with a textual representation of all possible ways of matching them. In case there is more one way of matching the interfaces, the user can input some restrictions, and invoke the matching algorithm again. These restrictions may come from non-structural information known to the user such as the semantics of the methods. In this way, the user can interact with the tool until a unique matching has been found.

Open Problems. A natural next step is to investigate how to automatically generate bridge code for a multi-language system. We would also like to find out whether our notion of type equality is sound and complete for some class of models of recursive types. On the implementation side, we want to make connections to work on multiset discrimination and chaotic fixed-point iteration.

Rest of the Paper. In the following section we give an overview of our techniques by way of an example, and in Section 3 we present our algorithm in detail.

2 Example

The purpose of this section is to give a gentle introduction to the algorithm and some of the definitions in Section 3. We do that by walking through a run of our algorithm on a simple example. While the example does not require all of the sophistication of our algorithm, it may give the reader a taste of what follows in Section 3.

Suppose we are given the following two sets of Java interfaces.

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

and

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

We would like to find out whether interface I_1 is structurally equal to 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.

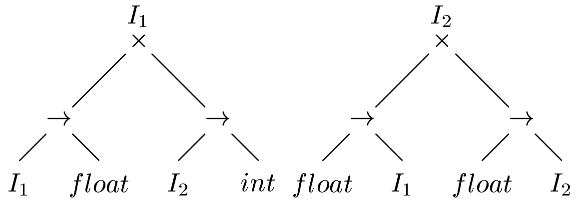
Notice that interface I_1 is recursively defined. The method m_1 takes an argument of type I_1 and returns a floating point number. In the following, we use names of interfaces and methods to stand for their type structures. The type of method m_1 can be expressed as $I_1 \to \mathit{float}$. The symbol $\to$ 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

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

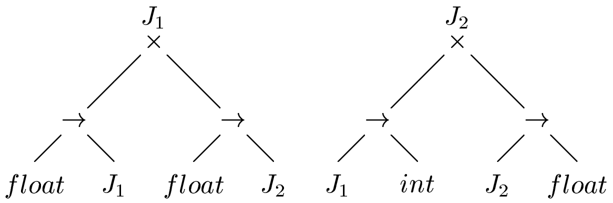
We think of such types as regular trees, and we represent them by automata.



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

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

The tree forms of J_1, J_2 are the following.



The interface types I_1, J_2 are equal iff there exists 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 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 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.

Our approach is related to the pebbling concept used by Dowling and Gallier [13]. We propagate information about inequality from the type pairs known to be unequal towards the ones we are interested in.

We organize the types of interfaces, methods, and base types (such as int) into a bipartite graph (V, W, R), where V represents the types in interfaces I_1, I_2 and W represents the types in interfaces J_1, J_2. That is, $V = \{I_1, I_2, m_1, m_2, m_3, m_4, int, float\}$, and $W = \{J_1, J_2, n_1, n_2, n_3, n_4, int, float\}$. The set of edges R represents "hoped-for" equality of types.

We initialize R as $(V \times W)$, that is, we treat every pair of types as equivalent types at the start. The idea is that by iteration, we remove edges between types that are not equal. When no more edges can be removed, the algorithm stops. The types connected in the final graph are equal.

First, we remove the edges between types that are obviously not equal. For example, an interface type and a method type are not equal; and a base type and a method type are not equal. We remove edges that connect interface types and method types, and edges between method types and base types.

In the iterations that follow, we remove edges between types that are not equal based on the information known from previous iterations. For example, we can determine that the method types m_1 and n_1 are not equal because the argument type of m_1 is I_1 while the argument type of n_1 is $float$, and the edge between I_1 and $float$ is removed the preceding iteration. Therefore, we remove the edge between m_1 and n_1.

The interesting part is to determine whether the types of two interfaces with n methods each are not equal based on information from previous iterations. This subproblem is equivalent to the perfect matching problem of a bipartite graph (V', W', R'), where V' and W' are the sets of methods in each interface, and there is an edge between two methods iff the types of the two methods have not been determined unequal in the previous iterations. If the set of edges R' is arbitrary, then the complexity of the perfect matching problem is $O(n^{5/2})$ (see [16]).

However, the graph (V, W, R) has a coherence property: if a vertex in V can reach a vertex in W, then there is an edge between these two vertices. Coherence both enables us to perform each iteration efficiently, and guarantees that the whole algorithm will terminate within $|V| + |W|$ iterations.

The resulting bipartite graphs after the second, the third, and the fourth iterations are given in Figure 1. In the third iteration, we examine the edges between interface types and determine whether we should remove some of the edges. For the types of interfaces I_1 and J_1 to be equal, there must exist a bijection from $\{m_1, m_2\}$ to $\{n_1, n_2\}$ such that the pair of methods in the bijection relation are connected in the bipartite graph after the second iteration. It is clear that the types of interface I_1 and J_1 are not equal since there is no edge between m_1, m_2 and n_1, n_2 at all. Thus,

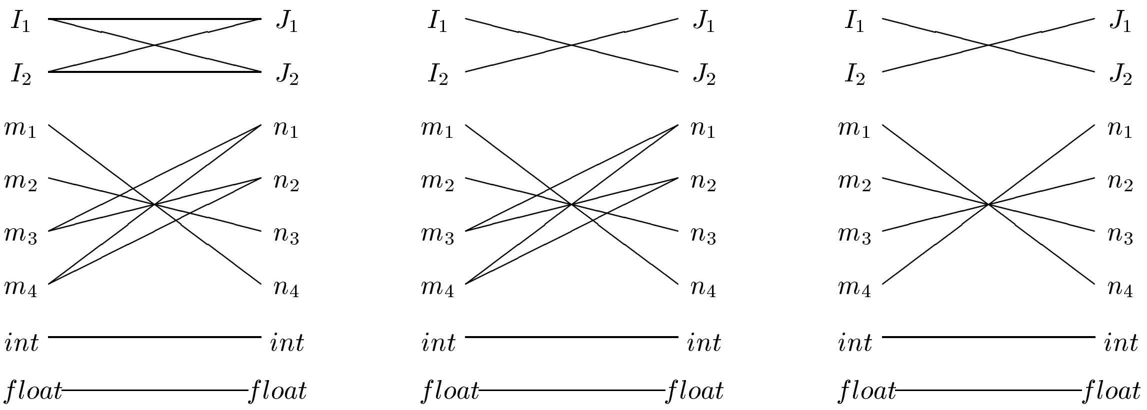


Figure 1: From the left to the right are the bipartite graphs after the second, the third and the fourth iterations.

the edge between I_1 and J_1 is removed. Similarly, we remove the edge between I_2 and J_2.

By the same steps, we are able to remove the edge between m_3 and n_1, and the edge between m_4 and n_2 in the fourth iteration. After that, we can not remove any more edges from the graph. Now the algorithm terminates and we can conclude that interface I_1 is equal to interface J_2. If we compare two types that can be represented with two automata each of size at most n, then the above algorithm will spend $O(n)$ time in each iteration and will terminate within $O(n)$ iterations, for a total of $O(n^2)$ time.

The simple example above does not reveal how the coherence property of an edge set can help speed up an iteration. This is because interfaces I_1, I_2, J_1, J_2 only have two methods each. In the Section 3 we present an efficient algorithm for the general case.

3 Type Equality

In this section, we define a notion of type equality where the product-type constructor is associative and commutative, and we present an efficient decision procedure.

We will use the notions of terms and term automata defined in [19]. For the convenience of the reader, we provide an excerpt of the relevant material from [19] in Appendix B. Our algorithm relies on that the types to be matched are represented as term automata.

In Section 3.1 we define our notion of type, and in Sections 3.2 and 3.3 we give some preliminaries about bipartite graphs and fixed points needed later. In Section 3.4 we present our notion of type equality, in Section 3.5 we show a convenient characterization of type equality, and in Section 3.6 we present an efficient decision procedure.

3.1 Recursive Types

A 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. With the notation of Appendix B, the root symbol of a type t is written $t(\epsilon)$.

We impose the restriction that given a type σ and a path α, if $\sigma(\alpha) = \prod^n$, then $\sigma(\alpha i) \in \Gamma \bigcup \{\rightarrow\}$, for all $i \in \{1..n\}$. The set of types is denoted $\mathcal{T}$. 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 $\sigma(i) = \sigma_{i+1}\ \forall i \in \{0, 1, \ldots, n-1\}$, then we write the type σ as $\prod_{i=1}^{n} \sigma_i$.

Intuitively, our restriction means that products cannot be immediately nested, that is, one cannot form a product one of whose immediate components is again a product. We impose this restriction for two reasons:

1. it effectively rules out infinite products such as $\mu\alpha.(int \times \alpha)$, and
2. it ensures that types are in a "normal form" with respect to associativity, that is, the issue of associativity is reduced to a matter of the order of the components in a $\prod_{i=1}^{n} \sigma_i$ type.

Currently, we are unable to extend our algorithm to handle infinite products. Types without infinite products can easily be "flattened" to conform to our restriction.

For Java interfaces, our restriction has no impact. We model interfaces using one kind of product-type constructor, we model argument-type lists using *another* kind of product-type constructor, and we model method types using the function-type constructor. The syntax of Java interfaces ensures that a straightforward translation of a Java interface to our representation of types will automatically satisfy our restriction.

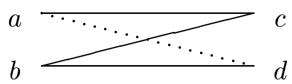
3.2 Bipartite Graphs

A bipartite graph (V, W, R) is given by two sets V, W of vertices, and a set $R \subseteq V \times W$ of undirected edges.

For our application, we will only be interested in bipartite graphs where the edge sets are *coherent*. A relation R is coherent iff

$$\text{if } (a, c), (b, c), (b, d) \in R, \text{ then } (a, d) \in R.$$

It can be illustrated by the following picture,



where the edges (a, c), (b, c), and (b, d) imply the existence of the edge (a, d).

Lemma 3.1. *Suppose $\mathcal{G} = (V, W, R)$ is a bipartite graph where R is coherent. If $a \in V$ can reach $d \in W$, then $(a, d) \in R$.*

Proof. Straightforward. □

Definition 3.2. *Suppose $\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i$ are two types and R is a relation on types. The matching function $match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is* true *iff there exists a bijection $t : \{1..n\} \to \{1..n\}$ such that $\forall i$, $(\sigma_i, \tau_{t(i)}) \in R$.*

Lemma 3.1 enables a simple algorithm for $match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ where R is coherent and finite. Choose two finite sets V, W such that $\sigma_i \in V$, for all $i \in \{1..n\}$, $\tau_i \in W$, for all $i \in \{1..n\}$, and $R \subseteq V \times W$. Let $N = |V| + |W|$. The bipartite graph (V, W, R) has at most N connected components, $C_1, C_2, \ldots$, and we label them with numbers starting at 1. Thus, all the numbers are in the set $\{1..N\}$.

Define a function $I : (V \cup W) \to \{1..N\}$, where $I(\sigma) = i$ iff $\sigma \in C_i$. Two types σ and τ are in the same connected component iff σ can reach τ in (V, W, R). Thus, by Lemma 3.1, we have $(\sigma, \tau) \in R$ iff $I(\sigma) = I(\tau)$.

Let $[.]$ denotes a multi-set of elements.

Lemma 3.3. *$match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is* true *iff $[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)]$.*

Proof. If $[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)]$, then there exists bijection $t : \{1..n\} \to \{1..n\}$, such that $\forall i$, $I(\sigma_i) = I(\tau_{t(i)})$. According to the numbering scheme, vertex σ_i can reach vertex $\tau_{t(i)}$; thus, by Lemma 3.1, $(\sigma_i, \tau_{t(i)}) \in R, \forall i$. Therefore, $match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is *true*.

Suppose $match(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is *true*. Then, there exists bijection t such that $(\sigma_i, \tau_{t(i)}) \in R, \forall i$. Thus, $I(\sigma_i) = I(\tau_{t(i)})$ since σ_i and $\tau_{t(i)}$ are connected. Since $[I(\tau_1), I(\tau_2), .., I(\tau_n)] = [I(\tau_{t(1)}), I(\tau_{t(2)}), .., I(\tau_{t(n)})]$, we have $[I(\sigma_1), I(\sigma_2), .., I(\sigma_n)] = [I(\tau_1), I(\tau_2), .., I(\tau_n)]$. □

3.3 Monotone Functions and Fixed Points

Let $\mathcal{P}$ denote the unary operator which maps a set to its power-set. Consider the lattice $(\mathcal{P}(S), \subseteq)$ and a function

$$F \quad : \quad \mathcal{P}(S) \to \mathcal{P}(S).$$

We say that F is monotone iff if $s_1 \subseteq s_2$, then $F(s_1) \subseteq F(s_2)$. If F is monotone, then Tarski's fixed point theorem [28] gives that F has a greatest fixed point νF given by:

$$\nu F \quad = \quad \bigcup \{ \, X \mid X \subseteq F(X) \, \}.$$

Suppose F is monotone, and $K \subseteq S$. Define

$$\begin{array}{rcl} H & \in & \mathcal{P}(K) \to \mathcal{P}(K) \\ H(X) & = & F(X) \cap K. \end{array}$$

Lemma 3.4. *$\nu H \subseteq \nu F \cap K$.*

Proof.

$$\begin{array}{rcl} \nu H & = & \bigcup \{ \, X \mid X \subseteq H(X) \, \} \\ & = & \bigcup \{ \, X \mid X \subseteq F(X) \cap K \, \} \\ & = & (\bigcup \{ \, X \mid X \subseteq F(X) \cap K \, \}) \cap K \\ & \subseteq & (\bigcup \{ \, X \mid X \subseteq F(X) \, \}) \cap K \\ & = & \nu F \cap K. \end{array}$$

□

The converse of Lemma 3.4 may be false. For example, consider

$$\begin{array}{rcl} S & = & \{1, 2\} \\ K & = & \{1\} \\ F(\{1, 2\}) & = & \{1, 2\} \\ F(\{1\}) & = & F(\{2\}) \; = \; F(\emptyset) \; = \; \emptyset. \end{array}$$

We have that F is monotone, $\nu F = \{1, 2\}$, and $\nu H = \emptyset$. We conclude that $\nu F \cap K = \{1, 2\} \cap \{1\} = \{1\} \not\subseteq \emptyset = \nu H$.

We now give a sufficient condition under which the converse of Lemma 3.4 is true.

Lemma 3.5. *Suppose that if $X \subseteq F(X)$, then $F(X) \cap K \subseteq F(X \cap K)$. We have $\nu F \cap K \subseteq \nu H$.*

Proof. First we will deduce some consequences of $X \subseteq F(X)$. It is straightforward to see that $X \subseteq F(X)$ implies $X \cap K \subseteq F(X) \cap K$, and by assumption $X \subseteq F(X)$ also implies $F(X) \cap K \subseteq F(X \cap K)$. Combining the two observations, we get

$$X \cap K \subseteq F(X) \cap K \subseteq F(X \cap K).$$

Now we can calculate as follows:

$$\begin{aligned}
\nu F \cap K &= \bigcup \{ X \mid X \subseteq F(X) \} \cap K \\
&= \bigcup \{ Y \mid \exists X : (Y = X \cap K) \wedge \\
&\qquad\qquad (X \subseteq F(X)) \} \\
&\subseteq \bigcup \{ Y \mid \exists X : (Y = X \cap K) \wedge \\
&\qquad\qquad (X \cap K \subseteq F(X \cap K)) \} \\
&= \bigcup \{ Y \mid \exists X : (Y = X \cap K) \wedge \\
&\qquad\qquad (Y \subseteq F(Y)) \} \\
&= \bigcup \{ Y \mid (Y \subseteq K) \wedge (Y \subseteq F(Y)) \} \\
&= \bigcup \{ Y \mid Y \subseteq F(Y) \cap K \} \\
&= \bigcup \{ Y \mid Y \subseteq H(Y) \} \\
&= \nu H.
\end{aligned}$$

□

If S is finite, then a well-known characterization of νF is given by:

$$\nu F = \bigcap_{i=0}^{\infty} F^i(S).$$

Lemma 3.6. *If H is a monotone function from $(\mathcal{P}(V \times W), \subseteq)$ to itself, where V, W are finite and $N = |V| + |W|$, and if for all non-negative integers i, $H^i(V \times W)$ is coherent, then $\nu H = H^N(V \times W)$.*

Proof. Let $S = (V \times W)$. Since H is monotone, $H^{i+1}(S) \subseteq H^i(S)\ \forall i \geq 0$.

If $H^{i+1}(S) = H^i(S)$, then $H^i(S)$ is a fixed point of H and $H^j(S) = H^i(S), \forall j > i$. Otherwise, if $H^{i+1}(S) \subset H^i(S)$, then $H^{i+1}(S) \subset \ldots \subset H^1(S) \subset S$.

Suppose $H^{i+1}(S) \subset H^i(S)$ and $(v, w) \in (H^i(S) \cap \neg H^{i+1}(S))$. We construct the bipartite graph $\mathcal{G}^i = (V, W, H^i(S))$. Each connected component of $\mathcal{G}^i$ corresponds to one or more connected component in $\mathcal{G}^{i+1}$, because any set of vertices that are connected in $\mathcal{G}^{i+1}$ are connected in $\mathcal{G}^i$ as well.

Since $(v, w) \in H^i(S)$, v, w are in the same connected component of $\mathcal{G}^i$. From $(v, w) \in \neg H^{i+1}(S)$ and Lemma 3.1, v can not reach w in $\mathcal{G}^{i+1}$. Therefore, v and w are in separate connected components of $\mathcal{G}^{i+1}$. Consequently, $\mathcal{G}^{i+1}$ has at least one more connected component than $\mathcal{G}^i$.

Consider $\{H^i(S)\}_{i=0}^k$ such that $H^k(S) \subset \ldots \subset H^1(S) \subset S$. Then the bipartite graph $\mathcal{G}^k$ has at least k connected components. However, $\mathcal{G}^k$ can have at most N connected components, which is the case when there is no edge in the graph and each vertex forms a connected component. Thus, $k \leq N$ and $H^N(S) = H^{N+1}(S)$.

We conclude that $\nu H = \bigcap_{i=0}^{\infty} H^i(S) = \bigcap_{i=0}^{N} H^i(S) = H^N(S)$. □

$$A, \sigma = \tau, A' \vdash \sigma = \tau \quad (\textsc{Hyp})$$

$$A \vdash \gamma = \gamma \quad (\textsc{Ref})$$

$$\frac{A, \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2 \vdash \sigma_i = \tau_i,\ i \in \{1,2\}}{A \vdash \sigma_1 \to \sigma_2 = \tau_1 \to \tau_2} \quad (\to/\textsc{Fix})$$

$$\frac{A, \prod_{i=1}^n \sigma_i = \prod_{i=1}^n \tau_i \vdash \sigma_i = \tau_{t(i)},\ i \in \{1..n\}}{A \vdash \prod_{i=1}^n \sigma_i = \prod_{i=1}^n \tau_i} \quad (\textstyle\prod/\textsc{Fix})$$

where $t : \{1..n\} \to \{1..n\}$ is a bijection

Figure 2: $EQ_{\to\prod}$.

3.4 Type Equality

We now give three equivalent definitions of type equality. They will be denoted $\mathbf{EQ}, E, \nu F$.

The first definition is based on the rule set $EQ_{\to\prod}$ in Figure 2. The rule $(\prod/\textsc{Fix})$ entails that the product-type constructor is associative and commutative. Define

$$\mathbf{EQ} = \{ (\sigma, \tau) \mid \emptyset \vdash \sigma = \tau \}.$$

The second definition of type equality is based on the idea of bisimilarity. A relation R on types is called a *bisimulation* if it satisfies the following three conditions:

(C) If $(\sigma, \tau) \in R$, then $\sigma(\epsilon) = \tau(\epsilon)$.

$(P1)$ If $(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \in R$, then $(\sigma_1, \tau_1) \in R$ and $(\sigma_2, \tau_2) \in R$.

$(P2)$ If $(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i) \in R$, then *match*$(\prod_{i=1}^n \sigma_i, \prod_{i=1}^n \tau_i, R)$ is *true*.

A relation R is said to be *consistent* if it satisfies property C, and it is said to be *closed* if it satisfies $P1, P2$.

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

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

The third definition of type equality is based on the notion of greatest fixed points. Define

$$\begin{array}{lcl} F & \in & \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T}) \\ F & = & \lambda R.\{ (\sigma, \tau) \mid \sigma, \tau \text{ are base types and } \sigma(\epsilon) = \tau(\epsilon) \} \\ & & \cup \{ (\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) \mid (\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R \} \\ & & \cup \{ (\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i) \mid \mathit{match}(\Pi_{i=1}^n \sigma_i, \Pi_{i=1}^n \tau_i, R) \} \end{array}$$

Notice that F is monotone so it has a greatest fixed point νF.

Lemma 3.7. *R is a bisimulation iff $R \subseteq F(R)$.*

Proof. Suppose first that R is a bisimulation. For every type pair $(\sigma, \tau) \in R$, if σ, τ are base types, then $\sigma(\epsilon) = \tau(\epsilon)$, so $(\sigma, \tau) \in F(R)$. If $\sigma = \sigma_1 \to \sigma_2$, $\tau = \tau_1 \to \tau_2$, then $(\sigma_1, \tau_1), (\sigma_2, \tau_2) \in R$, so $(\sigma, \tau) \in F(R)$. Similarly for $\sigma = \prod_{i=1}^n \sigma_i, \tau = \prod_{i=1}^n \tau_i$.

Conversely, suppose that $R \subseteq F(R)$. It is straightforward to prove that R is a bisimulation; we omit the details. □

Theorem 3.8. **EQ** $= E = \nu F$.

Proof. For a proof of **EQ** $= E$, see Appendix A. From Lemma 3.7 we have

$$\begin{array}{lcl} E & = & \bigcup \{ R \mid R \text{ is a bisimulation} \} \\ & = & \bigcup \{ R \mid R \subseteq F(R) \} = \nu F. \end{array}$$

□

We may apply the principle of *co-induction* to prove that two types are related in E. That is, to show $(\tau, \sigma) \in E$, it is sufficient to find a bisimulation R such that $(\sigma, \tau) \in R$.

Theorem 3.9. *E is a congruence relation.*

Proof. By co-induction, we omit the details. □

3.5 A Characterization of Type Equality

Lemma 3.10. *If $R \subseteq (\mathcal{T} \times \mathcal{T})$ is coherent, then $F(R)$ is coherent.*

Proof. First, notice that if $(\sigma, \tau) \in F(R)$, then $\sigma(\epsilon) = \tau(\epsilon)$ by the definition of F.

Suppose $(a, c), (b, c), (b, d) \in F(R)$, we want to show that $(a, d) \in F(R)$. There are three cases.

1. $a..d$ are base types. We have $a(\epsilon) = c(\epsilon) = b(\epsilon) = d(\epsilon)$, so $(a, d) \in F(R)$.

2. $a..d$ are $\to$ types. Suppose $a = a_1 \to a_2$, $b = b_1 \to b_2$, $c = c_1 \to c_2$, and $d = d_1 \to d_2$. We have (a_i, c_i), (b_i, c_i) and $(b_i, d_i) \in R$, $i = 1, 2$. Since R is coherent, $(a_i, d_i) \in R$, $i = 1, 2$, which means $(a, d) \in F(R)$.

3. $a..d$ are product types. Suppose $a = \prod_{i=1}^n a_i$, $b = \prod_{i=1}^n b_i$, $c = \prod_{i=1}^n c_i$, and $d = \prod_{i=1}^n d_i$. We have $(a, c) \in R$ and $\mathit{match}(\prod_{i=1}^n a_i, \prod_{i=1}^n c_i, R)$ is true. The same applies to (b, c) and (b, d). Therefore, $\exists$ bijections s, t, u from $\{1..n\}$ to $\{1..n\}$ such that $(a_i, c_{s(i)})$, $(b_i, c_{t(i)})$, $(b_i, d_{u(i)}) \in R, \forall i$. Let bijection $v = u \circ t^{-1} \circ s$, we have $(a_i, d_{v(i)}) \in R\ \forall i$, since R is coherent. Thus, $\mathit{match}(\prod_{i=1}^n a_i, \prod_{i=1}^n d_i, R)$ is true. and $(a, d) \in F(R)$.

□

For $\sigma \in \mathcal{T}$, define

$$V_\sigma = \{ \tau \mid \tau \text{ is a subterm of } \sigma \}.$$

Given σ, τ, define

$$\begin{array}{lcl} H & \in & \mathcal{P}(V_\sigma \times V_\tau) \to \mathcal{P}(V_\sigma \times V_\tau) \\ H & = & \lambda R.(F(R) \cap (V_\sigma \times V_\tau)). \end{array}$$

Lemma 3.11. *If $R \subseteq (V_\sigma \times V_\tau)$ is coherent, then $H(R)$ is coherent.*

Proof. By the definition of H, we have $H(R) = F(R) \cap (V_\sigma \times V_\tau)$.

Since $R \subseteq (V_\sigma \times V_\tau) \subset (\mathcal{T} \times \mathcal{T})$, by Lemma 3.10, $F(R)$ is coherent. Thus, if $(a, c), (b, c), (b, d) \in F(R) \cap (V_\sigma \times V_\tau)$, then $(a, d) \in F(R)$ and $(a, d) \in (V_\sigma \times V_\tau)$ because $a \in V_\sigma$ and $d \in V_\tau$. Therefore, $(a, d) \in F(R) \cap (V_\sigma \times V_\tau)$, and $H(R)$ is coherent. □

Lemma 3.12. *For all n, $H^n(V_\sigma \times V_\tau)$ is coherent.*

Proof. By induction on n, using Lemma 3.11. □

Lemma 3.13. $F(R) \cap (V_\sigma \times V_\tau) \subseteq F(R \cap (V_\sigma \times V_\tau))$.

Proof. Straightforward. □

Lemma 3.14. $\nu H = \nu F \cap (V_\sigma \times V_\tau)$.

Proof. By Lemma 3.4, we have $\nu H \subseteq \nu F \cap (V_\sigma \times V_\tau)$. By Lemma 3.13, $F(R) \cap (V_\sigma \times V_\tau) \subseteq F(R \cap (V_\sigma \times V_\tau))$. Therefore, by Lemma 3.5, we also have $\nu H \supseteq \nu F \cap (V_\sigma \times V_\tau)$.

□

Theorem 3.15. *$(\sigma, \tau) \in E$ iff $(\sigma, \tau) \in H^N(V_\sigma \times V_\tau)$, where $N = |V_\sigma| + |V_\tau|$.*

Proof. From $(\sigma,\tau) \in (V_\sigma \times V_\tau)$ we have that $(\sigma,\tau) \in E$ iff $(\sigma,\tau) \in E \cap (V_\sigma \times V_\tau)$. Moreover, from Theorem 3.8 and Lemma 3.14 we have

$$E \cap (V_\sigma \times V_\tau) = \nu F \cap (V_\sigma \times V_\tau) = \nu H.$$

Finally, Lemma 3.12 shows that $H^i(V_\sigma \times V_\tau)$ is coherent for all i, so by Lemma 3.6, $\nu H = H^N(V_\sigma \times V_\tau)$. □

3.6 Algorithm and Complexity

We can use Theorem 3.15 to give an algorithm for deciding type equality. Given a type pair (σ,τ), we can decide $(\sigma,\tau) \in E$ by deciding $(\sigma,\tau) \in H^N(V_\sigma \times V_\tau)$, where $N = |V_\sigma| + |V_\tau|$. To do this, we need to apply H at most N times. In each round, according to Lemma 3.12, H will be applied to a coherent relation R, where $H(R)$ is also coherent. Thus, we only need to represent coherent relations. We will now present such a representation scheme, and we will show that given a representation of R, we can efficiently compute a representation of $H(R)$.

Given a coherent relation R, we represent R by a function

$$I : (V_\sigma \cup V_\tau) \to \{1..N\},$$

where $(\sigma',\tau') \in R$ iff $I(\sigma') = I(\tau')$. The existence of such a representation was established in Section 3.2. The abstraction function *abs* maps a function I to the relation represented by I:

$$abs(I) = \{\ (\sigma',\tau') \in (V_\sigma \times V_\tau)\ \mid\ I(\sigma') = I(\tau')\ \}.$$

If I represents R, then we want to define $\mathcal{H}(I)$ as a representation of $H(R)$. The function $\mathcal{H}$ should have the following properties:

$$\begin{array}{rcl}
\mathcal{H}(I)(\sigma') & = & \mathcal{H}(I)(\tau') \\
& \Leftrightarrow & \sigma'(\epsilon) = \tau'(\epsilon) \\
\mathcal{H}(I)(\sigma_1 \to \sigma_2) & = & \mathcal{H}(I)(\tau_1 \to \tau_2) \\
& \Leftrightarrow & I(\sigma_1) = I(\tau_1) \wedge I(\sigma_2) = I(\tau_2) \\
\mathcal{H}(I)(\Pi_{i=1}^n \sigma_i) & = & \mathcal{H}(I)(\Pi_{i=1}^n \tau_i) \\
& \Leftrightarrow & [I(\sigma_1),\ldots,I(\sigma_n)] = [I(\tau_1),\ldots,I(\tau_n)]
\end{array}$$

where σ',τ' are base types.

Any such function $\mathcal{H}$ satisfies the following lemma.

Lemma 3.16. $H(abs(I)) = abs(\mathcal{H}(I))$.

Proof. Suppose $(\sigma',\tau') \in H(abs(I))$. We have $\sigma'(\epsilon) = \tau'(\epsilon)$ by definition of H and F.

There are three cases.

1. σ',τ' are base types. Since $\mathcal{H}(I)(\sigma') = \mathcal{H}(I)(\tau') \Leftrightarrow \sigma'(\epsilon) = \tau'(\epsilon)$, we have $(\sigma',\tau') \in abs(\mathcal{H}(I))$.

2. σ',τ' are $\to$ types. Suppose $\sigma' = \sigma_1 \to \sigma_2$ and $\tau' = \tau_1 \to \tau_2$. We have $(\sigma_1,\tau_1),(\sigma_2,\tau_2) \in abs(I)$. By the definition of $abs(I)$, $I(\sigma_1) = I(\tau_1)$ and $I(\sigma_2) = I(\tau_2)$. Hence, $\mathcal{H}(I)(\sigma_1 \to \sigma_2) = \mathcal{H}(I)(\tau_1 \to \tau_2)$ and $(\sigma',\tau') \in abs(\mathcal{H}(I))$.

3. σ',τ' are product types. Suppose $\sigma' = \prod_{i=1}^n \sigma_i$ and $\tau' = \prod_{i=1}^n \tau_i$. We have $match(\sigma',\tau',abs(I))$ true. By Lemma 3.3 and the definition of $abs(I)$, we have $[I(\sigma_1),\ldots,I(\sigma_n)] = [I(\tau_1),\ldots,I(\tau_n)]$ and consequently, $\mathcal{H}(I)(\Pi_{i=1}^n \sigma_i) = \mathcal{H}(I)(\Pi_{i=1}^n \tau)$, and $(\sigma',\tau') \in abs(\mathcal{H}(I))$.

Conversely, if $(\sigma',\tau') \in abs(\mathcal{H}(I))$, we have $\mathcal{H}(I)(\sigma') = \mathcal{H}(I)(\tau')$. It is straightforward to show that $(\sigma',\tau') \in H(abs(I))$ by a case analysis as above. We omit the details. □

Lemma 3.16 states that we can compute a representation of the result of applying H to the relation represented by I, by computing $\mathcal{H}(I)$.

Here is a particular definition of an $\mathcal{H}$ which satisfies the three properties. Given I, we define $\mathcal{H}(I)$ in three steps:

1. use I to define a preorder $\sqsubseteq$ on $(V_\sigma \cup V_\tau)$, and notice that $\sqsubseteq$ can be viewed as a directed graph;

2. number the strongly connected components of $\sqsubseteq$ in ascending order, starting from 1; and

3. define $\mathcal{H}(I)(\eta)$ to be the number of the strongly connected component to which η belongs.

Step (1) is done as follows. The relation $\sqsubseteq$ is the smallest preorder which includes the following definitions. First,

$$A \sqsubseteq \sigma_1 \to \sigma_2 \sqsubseteq \prod_{i=1}^n \tau_i$$

for all base types A, all function types $\sigma_1 \to \sigma_2$, and all product types $\prod_{i=1}^n \tau_i$. Next, we choose some arbitrary linear ordering of the base types. Finally, we use I to further sort the function types, and to further sort the product types. The idea of the further sorting is to define a lexicographical order based on I. Given a string of k numbers $m_1 \ldots m_k$, the notation $sort(m_1 \ldots m_k)$ denotes a string of the same k numbers but now in increasing order.

$\sigma_1 \to \sigma_2 \sqsubseteq \tau_1 \to \tau_2$ iff $I(\sigma_1)I(\sigma_2)$ is lexicographically less than $I(\tau_1)I(\tau_2)$

$\prod_{i=1}^n \sigma_i \sqsubseteq \prod_{i=1}^n \tau_i$ iff $sort(I(\sigma_1)\ldots I(\sigma_n))$ is lexicographically less than $sort(I(\tau_1)\ldots I(\tau_n))$.

Since a lexicographical ordering is a linear order, $\sqsubseteq$ is a preorder. Moreover, with this definition of $\sqsubseteq$, it is straightforward to show that the resulting $\mathcal{H}(I)$ satisfies the three properties listed earlier.

The computation of $\mathcal{H}(I)$ mainly involves:

- sorting numbers, and
- deciding the lexicographical ordering of strings of numbers.

We will now consider the complexity computing $\mathcal{H}(I)$.

Let σ be represented by term automaton

$$\mathcal{M}_\sigma = (V_\sigma,\ \Sigma,\ q_0,\ \delta,\ \ell).$$

Notice that we can construct a directed graph (V_σ, E_σ), where $(q, q') \in E_\sigma$ iff $\delta(q, i) = q'$, for some $i \in \{0, 1, .., n-1\}$ and $\ell(q) \in \Sigma_n$. Similarly, for type τ, we can construct a directed graph (V_τ, E_τ). Let $M = |E_\sigma| + |E_\tau|$.

We now show that we can compute $\mathcal{H}(I)$ in $O(M)$ time.

The size of I and $\mathcal{H}(I)$ is N.

For each product type $\prod_{i=1}^{n_k} \sigma_i \in (V_\sigma \cup V_\tau)$, we compute $sort[I(\sigma_1), I(\sigma_2), .., I(\sigma_{n_k})]$ in $O(n_k)$ time using COUNTING SORT [10].

In graph (V_σ, E_σ), the vertex $\prod_{i=1}^{n_k} \sigma_i$ has n_k outgoing edges. Suppose there are K such vertices in the graph, then $\Sigma_{k=1}^{K} n_k \leq |E_\sigma|$. Similarly, for the product types $\prod_{i=1}^{m_k} \tau_i$ in graph (V_τ, E_τ), we have $\Sigma_{k=1}^{K'} m_k \leq |E_\tau|$, where K' is the total number of product types in V_τ. Since $M = |E_\sigma| + |E_\tau|$, the total amount of time for computing $sort(.)$ for all product types is $O(M)$.

To order all the $\rightarrow$ types and products types, we need to lexicographically order strings of numbers. Using RADIX SORT [10], the ordering of all strings can be computed in time linear in the total size of the strings. The size of the string corresponding to type $\prod_{i=1}^{n_k} \sigma_i \in V_\sigma$ is n_k, which is equal to the number of outgoing edges of $\prod_{i=1}^{n_k} \sigma_i$ in (V_σ, E_σ). The size of the string corresponding to $\sigma_1 \rightarrow \sigma_2 \in V_\sigma$ is 2, which is equal to the number of outgoing edges of $\sigma_1 \rightarrow \sigma_2$ in (V_σ, E_σ). Therefore, the total size of strings corresponding to $\rightarrow$ types and product types in V_σ is equal to $|E_\sigma|$. Similarly, the total size of strings corresponding to $\rightarrow$ types and product types in V_τ is equal to $|E_\tau|$. Thus, the lexicographical ordering of all strings costs $O(M)$ time.

In conclusion, our decision procedure for membership in E is given by $O(N)$ iterations which each take $O(M)$ time. Thus, we have shown the following result.

Theorem 3.17. *Type equality as defined by E can be decided in $O(N \times M)$ time.*

Acknowledgments. We thank Mikhail Atallah for many insights. We also thank Mikael Rittri for pointing out [2] to us. Special thanks to Wanjun Wang for implementing our algorithm. The LICS reviewers provided a wealth of helpful comments on a draft of the paper. Our work is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265, and by IBM.

A Proof of the first half of Theorem 3.8

Theorem A.1. EQ $= E$.

Proof. First we prove $\mathbf{EQ} \subseteq E$ (*soundness*). Suppose Δ is a derivation tree for $\emptyset \vdash \sigma = \tau$. Let R be the set of type pairs that are found in Δ on the right-hand side of $\vdash$, except for applications of the rule (HYP). It is straightforward to see that all other type pairs in Δ are elements of R. Notice that $(\sigma, \tau) \in R$. It is straightforward to show that $R \subseteq F(R)$. From that and Lemma 3.7 we have that R is a bisimulation, so, by co-induction, $(\sigma, \tau) \in E$.

Next we prove $E \subseteq \mathbf{EQ}$ (*completeness*). Suppose $(\sigma, \tau) \in E$. Choose a bisimulation R' such that $(\sigma, \tau) \in R'$. Define $R = R' \cap (V_\sigma \times V_\tau)$. Notice that R is a finite set, and $(\sigma, \tau) \in R$. Let us show that R is a bisimulation. First, from R' being a bisimulation and Lemma 3.7, $R' \subseteq F(R')$. It follows that $R' \cap (V_\sigma \times V_\tau) \subseteq F(R') \cap (V_\sigma \times V_\tau)$. From Lemma 3.13 we have $F(R') \cap (V_\sigma \times V_\tau) \subseteq F(R' \cap (V_\sigma \times V_\tau))$, so $R' \cap (V_\sigma \times V_\tau) \subseteq F(R' \cap (V_\sigma \times V_\tau))$, that is, $R \subseteq F(R)$. Thus, by Lemma 3.7, R is a bisimulation.

From R, we can now construct a derivation tree for $\emptyset \vdash \sigma = \tau$. The function $\mathcal{S}$, see below, is a recursive function that takes as inputs (1) an environment A, and (2) a type pair (σ, τ). The call $\mathcal{S}(A, (\sigma, \tau))$ returns a suggestion for a derivation tree for $A \vdash \sigma = \tau$.

$\mathcal{S}\ (A, (\sigma, \tau)) =$

- If σ, τ are base types, then return $A \vdash \sigma = \tau$
- If $(\sigma, \tau) \in A$, then return $A \vdash \sigma = \tau$
- If $\sigma = \sigma_1 \rightarrow \sigma_2, \tau = \tau_1 \rightarrow \tau_2$, then return
$$\frac{\mathcal{S}((A, \sigma = \tau), (\sigma_i, \tau_i)) \quad \forall i \in \{1, 2\}}{A \vdash \sigma = \tau}$$
- If $\sigma = \Pi_{i=1}^{n} \sigma_i, \tau = \Pi_{i=1}^{n} \tau_i$, then return
$$\frac{\mathcal{S}((A, \sigma = \tau), (\sigma_i, \tau_{t(i)})) \quad \forall i \in \{1..n\}}{A \vdash \sigma = \tau}$$
where $(\sigma_i, \tau_{t(i)}) \in R$ and
t is a bijection from $\{1..n\}$ to $\{1..n\}$.

Consider the call $\mathcal{S}(\emptyset, (\sigma, \tau))$. It is straightforward to see that in every recursive call to $\mathcal{S}$, all type pairs in the arguments are elements of R. Since R is a bisimulation, this ensures that the rules in $EQ_{\rightarrow\prod}$ apply. Moreover, every time $\mathcal{S}$ is called, the size of A will increase by one, since otherwise we could use the second case in the definition of $\mathcal{S}$ to avoid further recursive calls. This limits the depth of the recursion to the number of elements of R. Since R is finite, we conclude that $\mathcal{S}(\emptyset, (\sigma, \tau))$ has a finite depth of recursion and that the size of the resulting derivation tree for $\emptyset \vdash \sigma = \tau$ is finite. $\square$

B Excerpt from [19]

B.1 Terms

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 ω.

Definition B.1. *A* term *over Σ is a partial function*

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

with domain $\mathcal{D}(t)$ satisfying the following properties:

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

The set of all terms is denoted T_Σ.

Definition B.2. *Let t be a term and $\alpha \in \omega^*$. Define the partial function $t{\downarrow}\alpha : \omega^* \rightarrow \Sigma$ by*

$$t{\downarrow}\alpha(\beta) \quad = \quad t(\alpha\beta) \ .$$

If $t{\downarrow}\alpha$ has nonempty domain, then it is a term, and is called the subterm of t at position α.

Definition B.3. *A term t is said to be* regular *if it has only finitely many distinct subterms; i.e., if $\{t{\downarrow}\alpha \mid \alpha \in \omega^*\}$ is a finite set. The set of regular terms is denoted R_Σ.*

B.2 Term Automata

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.*

Definition B.4. *Let Σ be a finite ranked alphabet. A* term automaton *over Σ is a tuple*

$$\mathcal{M} \quad = \quad (Q,\ \Sigma,\ q_0,\ \delta,\ \ell)$$

where:

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

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

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

We decorate Q, δ, etc. with the superscript $\mathcal{M}$ where necessary.

Let $\mathcal{M}$ be a term automaton as in Definition B.4. The partial function δ extends naturally to a partial function

$$\hat{\delta} : Q \times \omega^* \rightarrow Q$$

inductively as follows:

$$\begin{aligned} \hat{\delta}(q, \epsilon) \quad &= \quad q \\ \hat{\delta}(q, \alpha i) \quad &= \quad \delta(\hat{\delta}(q, \alpha), i) \ . \end{aligned}$$

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 Definition B.4, the partial function

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

is a term.

Definition B.5. *Let $\mathcal{M}$ be a term automaton. The term* represented by $\mathcal{M}$ *is the term*

$$t_\mathcal{M} \quad = \quad \lambda\alpha.\ell(\hat{\delta}(q_0, \alpha)) \ .$$

A term t is said to be representable *if $t = t_\mathcal{M}$ for some $\mathcal{M}$.*

Intuitively, $t_\mathcal{M}(\alpha)$ is determined by starting in the start state q_0 and scanning the input α, following transitions of $\mathcal{M}$ 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 $t_\mathcal{M}(\alpha)$ is undefined. If on the other hand $\mathcal{M}$ scans the entire input α and ends up in state q, then $t_\mathcal{M}(\alpha) = \ell(q)$.

Lemma B.6. *Let $t \in T_\Sigma$. The following are equivalent:*

(i) t is regular;

(ii) t is representable;

(iii) t is described by a finite set of equations involving the μ operator.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. Prog. Lang. Syst.*, 15(4):575–631, 1993. Also in Proceedings of POPL'91.

[2] A. Andreev and S. Soloviev. A deciding algorithm for linear isomorphism of types with complexity $O(n \log^2(n))$. *Lecture Notes in Computer Science*, 1290:197ff, 1997.

[3] J. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 393–402, June 1999.

[4] J. Auerbach, C. Barton, and M. Raghavachari. Type isomorphisms with recursive types. Research report RC 21247, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, Aug. 1998.

[5] J. Auerbach and M. C. Chu-Carroll. 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, Feb. 1997.

[6] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *ACM SIGSOFT'96, Fourth Symposium on the Foundations of Software Engineering*, San Francisco, California, October 1996.

[7] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proceedings of TLCA'97, 3rd International Conference on Typed Lambda Calculus and Applications*, 1997.

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

[9] J. Considine. Deciding isomorphisms of simple types in polynomial time. Manuscript, 2000.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, Cambridge, Mass., 1990.

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

[12] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Comput. Sci.*, 25(1):95–169, 1983.

[13] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Progamming*, 1(3):267–84, Oct. 1984.

[14] D. E. Gay. Interface definition language conversions: Recursive types. *ACM SIGPLAN Notices*, 29(8):101–110, Aug. 1994.

[15] R. Goldblatt. *Topoi: The Categorical Analysis of Logic*, volume 98 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, Amsterdam, 1979.

[16] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[17] T. Jim and J. Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1997.

[18] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1 May 1994.

[19] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995. 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.

[20] R. Milner. A complete inference system for a class of regular behaviors. *Journal of Computer and System Sciences*, 28(3):439–466, June 1984.

[21] P. Narendran, F. Pfenning, and R. Statman. On the unification problem for Cartesian closed categories. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 57–63. IEEE Computer Society Press, 1993.

[22] OMG. The common object request broker: Architecture and specification. Technical report, Object Management Group, 1999. Version 2.3.1.

[23] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, pages 15–32. Springer-Verlag (*LNCS* 104), 1981.

[24] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 603–617, Kaiserslautern, FRG, July 1990. Springer Verlag.

[25] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[26] M. Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

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

[28] A. Tarski. A lattice-theoretical fixed point theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

[29] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering Methodology*, 4(2):146–170, Apr. 1995.