1 Equirecursive Equality

In the equirecursive view of recursive types, types are regular labeled trees, possibly infinite. However, we still represent them by finite type expressions involving the fixpoint operator μ. There can be many type expressions representing the same type; for example, $\mu\alpha.\, 1 \to \alpha$ and $\mu\alpha.\, 1 \to 1 \to \alpha$. This raises the question: given two finite type expressions σ and τ, how do we tell whether they represent the same type?

In the isorecursive view, the finite type expressions σ and τ themselves are the types, and there are no infinite types. In this case, the question does not arise.

One might conjecture that two type expressions are equivalent (that is, represent the same type) iff they are provably so using ordinary equational logic with the unfolding rule $\mu\alpha.\, \tau = \tau\{\mu\alpha.\, \tau/\alpha\}$ and the usual laws of equality (reflexivity, symmetry, transitivity, congruence). But this would be incorrect. To see why, let us formulate the problem more carefully.

Suppose we have type expressions $\sigma, \tau, \ldots$ over variables $\alpha, \beta, \ldots$ defined by the grammar

$$\tau \;::=\; 1 \;\mid\; \sigma \to \tau \;\mid\; \alpha \;\mid\; \mu\alpha.\, \tau,$$

where the τ in $\mu\alpha.\, \tau$ is not a variable. Let $[\![\sigma]\!]$ be the type denoted by σ. This is a possibly infinite regular labeled tree obtained from σ by "unfolding" all μ-subexpressions.

Write $\vdash \sigma = \tau$ if the equality of σ and τ can be proved from the following axioms and rules:

$$\mu\alpha.\, \tau = \tau\{\mu\alpha.\, \tau/\alpha\} \qquad \tau = \tau \qquad \frac{\sigma = \tau}{\tau = \sigma} \qquad \frac{\sigma = \tau \qquad \tau = \rho}{\sigma = \rho} \qquad \frac{\sigma_1 = \sigma_2 \qquad \tau_1 = \tau_2}{\sigma_1 \to \tau_1 = \sigma_2 \to \tau_2}$$

These rules generate the smallest congruence relation on type expressions satisfying the unfolding rule $\mu\alpha.\, \tau = \tau\{\mu\alpha.\, \tau/\alpha\}$. One can show inductively that if $\vdash \sigma = \tau$, then $[\![\sigma]\!] = [\![\tau]\!]$, so the rules are sound. However, they are not complete. If we define

$$\tau_0 \;\triangleq\; \mu\alpha.\, 1 \to 1 \to \alpha \qquad\qquad \tau_{n+1} \;\triangleq\; 1 \to \tau_n,\; n \geq 0, \tag{1}$$

then $\vdash \tau_{2m} = \tau_{2n}$ and $\vdash \tau_{2m+1} = \tau_{2n+1}$ for any m and n, but not $\vdash \tau_n = \tau_{n+1}$, whereas $[\![\tau_m]\!] = [\![\tau_n]\!]$ for all m and n.

2 A Dangerous Proof System

The following proof system is sound and complete for type equivalence, but great care must be taken, because the system is fragile in a sense to be explained. Judgements are sequents of the form $E \vdash \sigma = \tau$, where E is a set of type equations.

$$E,\, \sigma = \tau \vdash \sigma = \tau \qquad\qquad E \vdash 1 = 1$$

$$\frac{E,\, \mu\alpha.\, \sigma = \tau \vdash \sigma\{\mu\alpha.\, \sigma/\alpha\} = \tau}{E \vdash \mu\alpha.\, \sigma = \tau} \qquad \frac{E \vdash \sigma = \tau}{E \vdash \tau = \sigma} \qquad \frac{E \vdash \sigma_1 = \sigma_2 \qquad E \vdash \tau_1 = \tau_2}{E \vdash \sigma_1 \to \tau_1 = \sigma_2 \to \tau_2}$$

For example, here is a proof in this system of $\vdash \tau_0 = \tau_1$ as defined in (1):

$$\dfrac{\dfrac{\dfrac{\tau_0 = 1 \to \tau_0 \vdash 1 = 1 \qquad \tau_0 = 1 \to \tau_0 \vdash \tau_0 = 1 \to \tau_0}{\tau_0 = 1 \to \tau_0 \vdash 1 \to \tau_0 = 1 \to 1 \to \tau_0}}{\tau_0 = 1 \to \tau_0 \vdash 1 \to 1 \to \tau_0 = 1 \to \tau_0}}{\vdash \tau_0 = 1 \to \tau_0}$$

The rule for unfolding is quite unusual. Note that the very equation we are trying to prove in the conclusion appears as an assumption in the premise! This makes the system fragile. In fact, it breaks if we add a transitivity rule

$$\frac{E \vdash \sigma = \tau \qquad E \vdash \tau = \rho}{E \vdash \sigma = \rho}$$

On the surface, the transitivity rule seems quite harmless, and it seems like it could not hurt to add it to our system. However, with the addition of this rule, the system becomes unsound. Here is a proof of the false statement $\vdash 1 = 1 \to 1$:

$$\dfrac{\dfrac{\mu\alpha.\, 1 = 1 \to 1 \vdash \mu\alpha.\, 1 = 1}{\dfrac{\mu\alpha.\, 1 = 1 \vdash 1 = 1}{\vdash \mu\alpha.\, 1 = 1}} \qquad \dfrac{\dfrac{\dfrac{\mu\alpha.\, 1 = 1 \to 1 \vdash 1 = \mu\alpha.\, 1}{\mu\alpha.\, 1 = 1 \to 1 \vdash \mu\alpha.\, 1 = 1 \to 1} \qquad \dfrac{\mu\alpha.\, 1 = 1 \to 1 \vdash 1 = 1 \to 1}{\vdash \mu\alpha.\, 1 = 1 \to 1}}{\vdash 1 = \mu\alpha.\, 1}}{\vdash 1 = 1 \to 1}$$

It is also essential that we have ruled out $\mu\alpha.\, \beta$ where β is a variable. Otherwise, for any τ,

$$\frac{\mu\alpha.\, \alpha = \tau \vdash \mu\alpha.\, \alpha = \tau}{\vdash \mu\alpha.\, \alpha = \tau}$$

3 Types as Coterms

A more revealing view of the proof system given above is the *coalgebraic* view. We represent finite and infinite types as labeled trees, or *coterms*.

In this approach, we try to find witnesses to the *inequivalence* of two types. The idea is that if $[\![\sigma]\!] \neq [\![\tau]\!]$, then there is a witness to that fact in the form of a common finite path from the roots of $[\![\sigma]\!]$ and $[\![\tau]\!]$ down to some point where the labels differ. Moreover, one can calculate a bound b on the length of such a witness if it exists. The bound is quadratic in the sizes of σ and τ. This gives an algorithm for checking equivalence: unfold the trees down to depth b, and search for a witness; if none is found, then none exists.

This algorithm is still exponential in the worst case. One can do better using an automata-theoretic approach. We build deterministic automata out of σ and τ and check whether they accept all the same strings. This gives an algorithm whose worst-case running time is almost linear in the sizes of the automata.

We restrict our attention to the constructors $\to, 1$; we could add more if we wanted to, but these suffice for the purpose of illustration. Let $\{0,1\}^*$ be the set of finite-length strings over the alphabet $\{0,1\}$ (0 and 1 representing "left" and "right", respectively). A *coterm* over the signature $\{1, \to\}$ is a partial function $t : \{0,1\}^* \rightharpoonup \{1, \to\}$ such that

- $\mathsf{dom}\, t$, the domain of t, is nonempty and prefix-closed (thus the empty string $\varepsilon \in \mathsf{dom}\, t$ always; this is called the *root*);
- if $t(x) = \to$, then both $x0$ and $x1$ are in $\mathsf{dom}\, t$;
- if $t(x) = 1$, then neither $x0$ nor $x1$ is in $\mathsf{dom}\, t$; thus x is a *leaf*.

A *path* in t is a maximal subset of $\mathsf{dom}\, t$ linearly ordered by the prefix relation. Paths can be finite or infinite. A finite path ends in a leaf x, thus $t(x) = 1$ and $t(y) = \to$ for all proper prefixes y of x. An infinite path has $t(x) = \to$ for all elements x along the path.

Let t be a type and $x \in \mathsf{dom}\, t$. Define the partial function $t \mathbin{@} x : \{0,1\}^* \rightharpoonup \{1, \rightarrow\}$ by

$$(t \mathbin{@} x)(y) \triangleq t(xy).$$

Then $t \mathbin{@} x$ is a type. Intuitively, it is the subexpression of t at position x.

A type t is *finite* if its domain $\mathsf{dom}\, t$ is a finite set. By König's lemma, a type is finite iff it has no infinite paths. A type t is *regular* if $\{t \mathbin{@} x \mid x \in \{0,1\}^*\}$ is a finite set.

4 Term Automata

Types can be represented by a special class of automata called *term automata*. These can be defined over any signature, but for our application, we consider only term automata over $\{\rightarrow, 1\}$. A term automaton over this signature consists of

- a set of *states* S;
- a *start state* $s \in S$;
- a partial function $\delta : \{0,1\} \to S \rightharpoonup S$ called the *transition function*; and
- a (total) *labeling function* $\ell : S \to \{\rightarrow, 1\}$,

such that for any state $u \in S$,

- if $\ell(u) = \rightarrow$, then both $\delta(0)(u)$ and $\delta(1)(u)$ are defined; and
- if $\ell(u) = 1$, then both $\delta(0)(u)$ and $\delta(1)(u)$ are undefined.

The partial function δ extends naturally to a partial function $\widehat{\delta} : \{0,1\}^* \to S \rightharpoonup S$ inductively as follows:

$$\widehat{\delta}(\varepsilon)(u) \triangleq u \qquad\qquad \widehat{\delta}(xa)(u) \triangleq \delta(a)(\widehat{\delta}(x)(u)),$$

or more succinctly,

$$\widehat{\delta}(\varepsilon) \triangleq \mathsf{id}_S \qquad\qquad \widehat{\delta}(xa) \triangleq \delta(a) \circ \widehat{\delta}(x).$$

It follows by induction on the length of y that $\widehat{\delta}(xy) = \widehat{\delta}(y) \circ \widehat{\delta}(x)$. In other words, $\widehat{\delta}$ is the unique monoid homomorphism extending δ from the free monoid $\{0,1\}^*$ on generators $\{0,1\}$ to the monoid of partial functions $S \rightharpoonup S$ under composition.

For any $u \in S$, the domain of the partial function $\lambda x.\, \widehat{\delta}(x)(u)$ is nonempty (it always contains ε) and prefix-closed. Moreover, the partial function $\lambda x.\, \ell(\widehat{\delta}(x)(u))$ is a type. The type *represented by* M is the type

$$[\![M]\!] \triangleq \lambda x.\, \ell(\widehat{\delta}(x)(s)),$$

where s is the start state.

Intuitively, $[\![M]\!](x)$ is determined by starting in the start state s and scanning the input x, following transitions of M as far as possible. If it is not possible to scan all of x because some transition along the way does not exist, then $[\![M]\!](x)$ is undefined. If on the other hand M scans the entire input x and ends up in state u, then $[\![M]\!](x) = \ell(u)$.

One can show that a type t is regular iff $t = [\![M]\!]$ for some term automaton M with finitely many states. This is also equivalent to being $[\![\tau]\!]$ for some finite type expression τ. To construct a term automaton M_τ from a closed finite type expression τ, take the set of states of M_τ to be the smallest set S such that

- $\tau \in S$;
- if $\sigma \to \rho \in S$, then $\sigma \in S$ and $\rho \in S$; and
- if $\mu\alpha.\,\sigma \in S$, then $\sigma\{\mu\alpha.\,\sigma/\alpha\} \in S$.

The set S so defined is finite. The start state is τ. The transition function is given by the following rules:

- $\delta(0)(\sigma \to \rho) \stackrel{\triangle}{=} \sigma$;
- $\delta(1)(\sigma \to \rho) \stackrel{\triangle}{=} \rho$;
- $\delta(i)(1)$ is undefined, $i \in \{0, 1\}$;
- $\delta(i)(\mu\alpha.\,\sigma) \stackrel{\triangle}{=} \delta(i)(\sigma\{\mu\alpha.\,\sigma/\alpha\})$, $i \in \{0, 1\}$.

(The restriction that $\sigma \neq \alpha$ is crucial here.) The labeling function is given by:

- $\ell(\sigma \to \rho) \stackrel{\triangle}{=} \to$
- $\ell(1) \stackrel{\triangle}{=} 1$
- $\ell(\mu\alpha.\,\sigma) \stackrel{\triangle}{=} \ell(\sigma\{\mu\alpha.\,\sigma/\alpha\})$.

Then $[\![\tau]\!] = [\![M_\tau]\!]$.

Mathematically speaking, term automata are exactly the coalgebras of signature $\{\to, 1\}$ over the category of sets. The map $M \mapsto [\![M]\!]$ is the unique morphism from the coalgebra M to the final coalgebra, which consists of the finite and infinite types.

5 A Coinductive Algorithm for Type Equivalence

Now given pair σ, τ of finite type expressions, $[\![\sigma]\!] = [\![\tau]\!]$ iff for all $x \in \{0, 1\}^*$, $[\![\sigma]\!](x) = [\![\tau]\!](x)$; equivalently, $[\![\sigma]\!] \neq [\![\tau]\!]$ iff there exists $x \in \mathsf{dom}\,[\![\sigma]\!] \cap \mathsf{dom}\,[\![\tau]\!]$ such that $[\![\sigma]\!](x) \neq [\![\tau]\!](x)$. Form the two term automata $M_\sigma = (S_\sigma, \delta_\sigma, \ell_\sigma, s_\sigma)$ and $M_\tau = (S_\tau, \delta_\tau, \ell_\tau, s_\tau)$. Then form the product automaton $M_\sigma \times M_\tau$ with states $S_\sigma \times S_\tau$, transition function $\lambda d.\,\lambda(p, q).\,(\delta_\sigma(d)(p), \delta_\tau(d)(q))$, start state (s_σ, s_τ), and labeling function $\lambda(p, q).\,(\ell_\sigma(p), \ell_\tau(q))$. The product automaton runs the two automata M_σ and M_τ in parallel on the same input data. Then $[\![M_\sigma]\!] \neq [\![M_\tau]\!]$ iff there exists an input string $x \in \{0, 1\}^*$ that causes the product automaton to move from its start state to a state (u, v) such that $\ell_\sigma(u) \neq \ell_\tau(v)$. This can be determined by depth-first search in time linear in $|\,M_\sigma \times M_\tau\,|$, which is roughly $|\,M_\sigma\,| \cdot |\,M_\tau\,|$. This give a quadratic algorithm for testing type equivalence. One can improve this to almost linear time using a technique of Hopcroft and Karp [1].

6 Testing Equirecursive Equality

Now we introduce the notion of *bisimulation* and give a bisimulation-based implementation due to Hopcroft and Karp [1] using the *union-find* data structure for maintaining disjoint sets [2]. With these enhancements, the algorithm runs in linear space and almost linear time $O(n\,\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function.

7 Ackermann's Function

Ackermann's function $A : \mathbb{N}^2 \to \mathbb{N}$ is defined inductively as follows:

$$A(0,n) \triangleq n+1 \qquad A(m+1,0) \triangleq A(m,1) \qquad A(m+1,n+1) \triangleq A(m,A(m+1,n)).$$

Thus

$$A(0,n) = n+1 \quad A(1,n) = n+2 \quad A(2,n) = 2n+3 \quad A(3,n) = 2^{n+3}-3 \quad A(4,n) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3} - 3.$$

The function $\lambda m.\, A(m,2) : \mathbb{N} \to \mathbb{N}$ grows extremely fast, asymptotically faster than any primitive recursive function. The primitive recursive functions are those computable by IMP programs with nested for loops but no while loops (see Assignment 2). The inverse of this function is $\alpha(n) =$ the least k such that $A(k,2) \geq n$. This function grows without bound, but extremely slowly. Its value is 4 for all inputs less than the number of nanoseconds since the Big Bang.

8 Union-Find

Let X be a set. The union-find data structure is for maintaining a partition of X into disjoint sets, called *partition elements*, on which we wish to perform the following operations:

1. $\mathsf{find}(u)$: Given $u \in X$, find the unique partition element containing u.
2. $\mathsf{union}(u,v)$: Given two elements $u, v \in X$ in different partition elements, destructively merge the partition elements containing u and v into one set.

We can test whether two elements u, v are in the same partition element by testing whether $\mathsf{find}(u) = \mathsf{find}(v)$.

We represent each partition element as a rooted tree with all nodes pointing to their parent. To do $\mathsf{find}(u)$, we start at u and follow parent pointers up to the root of the tree containing u. The root serves as the canonical representative of the partition element and is the value of $\mathsf{find}(u)$. To do $\mathsf{union}(u,v)$ where $\mathsf{find}(u) \neq \mathsf{find}(v)$, we make the root of one of the two trees point to the root of the other.

We also use two heuristics to improve efficiency. First, when merging sets, we always make the root of the smaller tree point to the root of the larger. This tends to make the trees more balanced so that paths are shorter. To do this in constant time, we maintain the sizes of the sets at the roots and update whenever a union is done. Second, when doing a $\mathsf{find}(u)$, we change the parent pointers of all the nodes along the path from u to the root to point directly to the root. This is called *path compression*. It makes subsequent finds on those nodes more efficient because they traverse shorter paths.

It can be shown that with these two heuristics, starting with the identity partition on a set X of size n (each element of $u \in X$ in a singleton set by itself), any sequence of m union and find operations takes at most $O((m+n)\,\alpha(n))$ time [2].

9 Coalgebras and Coterms

As mentioned previously, the set of coterms over the signature $\{1, \to\}$ forms the final coalgebra for that signature. A *coalgebra* for the signature $\{1, \to\}$ is a structure $M = (S, \delta, \ell)$, where S is a set of *states*, $\delta : S \times \{0,1\} \rightharpoonup S$ is a partial function called the *transition function*, and $\ell : S \to \{1, \to\}$ is a *labeling function*, such that

- if $\ell(s) = \to$, then both $\delta(s, 0)$ and $\delta(s, 1)$ are defined; and
- if $\ell(s) = 1$, then neither $\delta(s, 0)$ nor $\delta(s, 1)$ is defined.

The transition function δ can be extended inductively to a multistep version $\widehat{\delta} : S \times \{0,1\}^* \rightharpoonup S$ by

$$\widehat{\delta}(s, \varepsilon) \stackrel{\triangle}{=} s \qquad\qquad \widehat{\delta}(s, ax) \stackrel{\triangle}{=} \widehat{\delta}(\delta(s, a), x) \text{ for } x \in \{0,1\}^* \text{ and } a \in \{0,1\},$$

with the convention that the value is undefined if one of its arguments is undefined.[1]

If $M_1 = (S_1, \delta_1, \ell_1)$ and $M_2 = (S_2, \delta_2, \ell_2)$ are coalgebras, a function $h : S_1 \to S_2$ is a *coalgebra morphism* $h : M_1 \to M_2$ if h preserves the coalgebraic operations; formally, if

- $\ell_1(s) = \ell_2(h(s))$ for all $s \in S_1$; and
- $h(\delta_1(s, a)) = \delta_2(h(s), a)$ for all $s \in S_1$ and $a \in \{0,1\}$.

It follows by induction on the length of $x \in \{0,1\}^*$ that

$$\ell_1(\widehat{\delta}_1(s, x)) = \ell_2(\widehat{\delta}_2(h(s), x)). \tag{2}$$

It can be shown by induction on the length of $x \in \{0,1\}^*$ that for any coterm t,

$$t(x) = \ell_C(\widehat{\delta}_C(t, x)). \tag{3}$$

The coalgebra of coterms is the *final coalgebra* for the signature $\{1, \to\}$, which means for any coalgebra $M = (S_M, \delta_M, \ell_M)$, there is a unique coalgebra morphism $h_M : M \to C$ given by

$$h_M(s) \stackrel{\triangle}{=} \lambda x.\, \ell_M(\widehat{\delta}_M(s, x)) : \{0,1\}^* \to \{1, \to\} \tag{4}$$

for $s \in S_M$. The map h_M is a coalgebra morphism because

$$\begin{aligned} \ell_C(h_M(s)) &= \ell_C(\lambda x.\, \ell_M(\widehat{\delta}_M(s, x))) \\ &= (\lambda x.\, \ell_M(\widehat{\delta}_M(s, x)))\ (\varepsilon) \\ &= \ell_M(\widehat{\delta}_M(s, \varepsilon)) \\ &= \ell_M(s) \end{aligned} \qquad\qquad \begin{aligned} \delta_C(h_M(s), a) &= \delta_C(\lambda x.\, \ell_M(\widehat{\delta}_M(s, x)), a) \\ &= \lambda x.\, \ell_M(\widehat{\delta}_M(s, x))\ @\ a \\ &= \lambda y.\, (\lambda x.\, \ell_M(\widehat{\delta}_M(s, x)))(ay) \\ &= \lambda y.\, \ell_M(\widehat{\delta}_M(s, ay)) \\ &= \lambda y.\, \ell_M(\widehat{\delta}_M(\delta_M(s, a), y)) \\ &= h_M(\delta_M(s, a)). \end{aligned}$$

Moreover, the morphism h_M is unique, because by (2) and (3), any morphism $h : M \to C$ satisfies

$$h(s)(x) = \ell_C(\widehat{\delta}_C(h(s), x)) = \ell_M(\widehat{\delta}_M(s, x)) = h_M(s)(x).$$

thus $h = h_M$.

[1] In general, we interpret equations involving partial functions as asserting that either both sides are defined or both are undefined, and if both are defined then they have the same value.

10 Bisimulation

Let $M_1 = (S_1, \delta_1, \ell_1)$ and $M_2 = (S_2, \delta_2, \ell_2)$ be coalgebras. A binary relation $R \subseteq S_1 \times S_2$ is called a *bisimulation* if it satisfies the following property:

For all $u \in S_1$ and $v \in S_2$, if $(u, v) \in R$, then

(i) $\ell_1(u) = \ell_2(v)$; and

(ii) if $\ell_1(u) = \ell_2(v) = \to$, then $(\delta_1(u, a), \delta_2(v, a)) \in R$ for $a \in \{0, 1\}$.

A pair of states are said to be *bisimilar* if there exists a bisimulation relating them.

Theorem 1. *States $u \in S_1$ and $v \in S_2$ are bisimilar if and only if $h_1(u) = h_2(v)$, where $h_1 : M_1 \to C$ and $h_2 : M_2 \to C$ are the unique coalgebra morphisms to the final coalgebra C from M_1 and M_2, respectively.*

Proof. If u and v are bisimilar, then there exists a bisimulation R such that $(u, v) \in R$. It follows by induction on length that for all $x \in \{0, 1\}^*$, either $\widehat{\delta}_1(u, x)$ and $\widehat{\delta}_2(v, x)$ are both undefined, or both are defined and $(\widehat{\delta}_1(u, x), \widehat{\delta}_2(v, x)) \in R$. Thus by (4),

$$h_1(u)(x) = \ell_1(\widehat{\delta}_1(u, x)) = \ell_2(\widehat{\delta}_2(v, x)) = h_2(v)(x).$$

As x was arbitrary, $h_1(u) = h_2(v)$.

Conversely, it is easily shown that the relation $\{(u, v) \mid h_1(u) = h_2(v)\}$ is a bisimulation. In fact, it is the unique maximal bisimulation between S_1 and S_2. □

11 An Algorithm

Consider the following algorithm for determining whether a given pair $s \in S_1$ and $t \in S_2$ are bisimilar. We will employ a *worklist algorithm* to try to construct a bisimulation containing (s, t). The worklist will contain pairs that are forced to be related by any such bisimulation. We will also maintain a partition of $S_1 \cup S_2$ (assuming $S_1 \cap S_2 = \varnothing$) using the union-find data structure, initially the identity partition with all states in their own singleton sets.

We initially put the pair (s, t) on the worklist. We then repeat the following as long as the worklist is nonempty:

1. Take the next pair (u, v) off the worklist.
2. If $\ell_1(u) \neq \ell_2(v)$, immediately halt and reject; there is no bisimulation containing (s, t).
3. Otherwise, if $\mathsf{find}(u) \neq \mathsf{find}(v)$,
   (a) perform $\mathsf{union}(u, v)$;
   (b) if $\ell_1(u) = \ell_2(v) = \to$, put $(\delta_1(u, 0), \delta_2(v, 0))$ and $(\delta_1(u, 1), \delta_2(v, 1))$ on the worklist.

11.1 Correctness

No pair (u, v) ever goes on the worklist unless u and v are forced to be bisimilar under any bisimulation R containing (s, t). This is because pairs are only put on the worklist in step 3(b) in order to fulfill clause (ii)

of the definition of bisimulation. Thus the rejection in step 2 of the algorithm is correct, because otherwise clause (i) of the definition of bisimulation would be violated.

If the algorithm does not reject in step 2, then it eventually halts by emptying the worklist (we argue this in the complexity analysis below). When this occurs, let $\mathsf{find}(u)$ and $\mathsf{find}(v)$ refer to their final values after the algorithm has halted. We claim that the relation

$$R = \{(u, v) \mid u \in S_1,\ v \in S_2,\ \mathsf{find}(u) = \mathsf{find}(v)\}$$

is a bisimulation containing (s, t). Surely R contains (s, t), since this pair was merged in the first iteration. To show that R satisfies clause (ii) of the definition, suppose $(u, v) \in R$. Then $\mathsf{find}(u) = \mathsf{find}(v)$. Let $E \subseteq S_1 \times S_2$ be the set of pairs (p, q) for which $\mathsf{union}(p, q)$ was performed in step 3(a). The set E forms a spanning forest for the partition elements, as each execution of step 3(a) connects two previously disconnected components. Thus there exists a zigzag path of E edges between u and v; that is, there exist $u_0, u_1, \ldots, u_{2k+1}$, $k \geq 0$, such that $u = u_0$, $v = u_{2k+1}$, and (u_i, u_{i+1}) or $(u_{i+1}, u_i) \in E$ for $0 \leq i \leq k-1$. Then the corresponding δ-successors of all these pairs went on the worklist in step 3(b). It follows by transitivity that $\mathsf{find}(\delta_1(u, a)) = \mathsf{find}(\delta_2(v, a))$ for $a \in \{0, 1\}$, thus $(\delta_1(u, a), \delta_2(v, a)) \in R$ for $a \in \{0, 1\}$. Moreover, clause (i) in the definition of bisimulation holds for R, otherwise the algorithm would have reported failure in step 2.

11.2 Complexity

Aside from the initial pair (s, t), pairs are only added to the worklist in step 3(b). But step 3(b) is executed only when step 3(a) is executed, and step 3(a) can be executed at most $n - 1$ times, where n is the size of $S_1 \cup S_2$, because each time two disjoint sets are merged. Thus at most $2n - 1$ pairs are ever added to the worklist. The loop body executes at most $2n - 1$ times, and apart from the union-find operations, each iteration takes constant time. The union-find operations take time $O(n\,\alpha(n))$ time amortized over the entire computation, thus the total time is $O(n\,\alpha(n))$.

References

[1] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[2] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.