1 Testing Equirecursive Equality

In Lecture 35, we gave a quadratic-time automata-based algorithm for testing equality of equirecursive types. In this handout 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.

2 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

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

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.

3 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].

4 Coalgebras and Coterms

As mentioned in Lecture 35, 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

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

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

$$\ell_1(\widehat{\delta_1}(s,x)) = \ell_2(\widehat{\delta_2}(h(s),x)). \tag{1}$$

Recall from Lecture 35 that *coterms* are partial functions $t : \{0,1\}^* \rightharpoonup \{1, \to\}$ such that

- $\mathsf{dom}\, t$, the domain of t, is nonempty and prefix-closed;
- if $x \in \mathsf{dom}\, t$ and $t(x) = \to$, then $x0, x1 \in \mathsf{dom}\, t$; and
- if $x \in \mathsf{dom}\, t$ and $t(x) = 1$, then $x0, x1 \notin \mathsf{dom}\, t$.

Let C be the set of coterms. We can make C into a coalgebra by defining

- $\ell_C(t) = t(\varepsilon)$
- $\delta_C(t,a) = \lambda x.\, t(ax) = t \mathbin{@} a$ for $a \in \{0,1\}$.

[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.

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

$$t(x) = \ell_C(\widehat{\delta}_C(t,x)). \tag{2}$$

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

$$h_M(s) \stackrel{\triangle}{=} \lambda x.\, \ell_M(\widehat{\delta}_M(s,x)) : \{0,1\}^* \to \{1,\to\} \tag{3}$$

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

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

Moreover, the morphism h_M is unique, because by (1) and (2), 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 $h = h_M$.

5 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 (3),

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

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. □

6 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 $\ell_1(u) = \ell_2(v) = \rightarrow$, put $(\delta_1(u, 0), \delta_2(v, 0))$ and $(\delta_1(u, 1), \delta_2(v, 1))$ on the worklist.

6.1 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.

6.2 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.