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 not be correct. 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 τ is not a variable in $\mu\alpha.\, \tau$. 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:

$$\vdash \mu\alpha.\, \tau = \tau\{\mu\alpha.\, \tau/\alpha\} \qquad\qquad \vdash \tau = \tau$$

$$\frac{\vdash \sigma = \tau}{\vdash \tau = \sigma} \qquad \frac{\vdash \sigma = \tau \quad \vdash \tau = \rho}{\vdash \sigma = \rho} \qquad \frac{\vdash \sigma_1 = \sigma_2 \quad \vdash \tau_1 = \tau_2}{\vdash \sigma_1 \to \tau_1 = \sigma_2 \to \tau_2}$$

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

$$\tau_0 \stackrel{\triangle}{=} \mu\alpha.\, 1 \to 1 \to \alpha \qquad\qquad \tau_{n+1} \stackrel{\triangle}{=} 1 \to \tau_n, \quad n \geq 0, \tag{1}$$

then $\vdash \tau_{2m} = \tau_{2n}$ and $\vdash \tau_{2m+1} = \tau_{2n+1}$ for any m and n, but not $\vdash \tau_n = \tau_{n+1}$.

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.

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

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

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

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

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

$$\dfrac{E \vdash \sigma = \tau \quad E \vdash \tau = \rho}{E \vdash \sigma = \rho}.$$

On the surface, the transitivity rule seems quite harmless, and it seems like couldn't 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 $\vdash 1 = 1 \to 1$:

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

It is also essential that we explicitly rule out $\mu\alpha.\, \alpha$; otherwise we would have

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

for any τ.

3 Types as Labeled Trees

A more revealing view of the proof system given above is the *coinductive* view, in which 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 look for an input string on which they differ. This will give an algorithm whose worst-case running time is proportional to $|\sigma| \cdot |\tau|$.

Let $\{L, R\}^*$ be the set of finite-length strings over $\{L, R\}$ (L="left", R="right"). We model (possibly infinite) types as partial functions $T : \{L, R\}^* \rightharpoonup \{1, \to\}$ such that

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

We restrict our attention to the constructors $\rightarrow, 1$; we could add more if we wanted to, but these suffice for the purpose of illustration.

A *path* in T is a maximal subset of $\operatorname{dom} T$ 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 $T(y) = \rightarrow$ for all proper prefixes y of x. An infinite path has $T(x) = \rightarrow$ for all elements x along the path.

Let T be a type and $x \in \{L, R\}^*$. Define the partial function $T_x : \{L, R\}^* \rightharpoonup \{1, \rightarrow\}$ by

$$T_x(y) \stackrel{\triangle}{=} T(xy).$$

If T_x has nonempty domain, then it is a type. Intuitively, it is the subexpression of T at position x.

A type T is *finite* if its domain $\operatorname{dom} T$ is a finite set. By König's Lemma, a type is finite iff it has no infinite paths. A type T is *regular* if $\{T_x \mid x \in \{L, R\}^*\}$ 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* Q;
- a *start state* $s \in Q$;
- a partial function $\delta : Q \times \{L, R\} \rightharpoonup Q$ called the *transition function*; and
- a (total) *labeling function* $\ell : Q \rightarrow \{\rightarrow, 1\}$,

such that for any state $q \in Q$,

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

The partial function δ extends naturally to a partial function $\widehat{\delta} : Q \times \{L, R\}^* \rightarrow Q$ inductively as follows:

$$\widehat{\delta}(q, \epsilon) \stackrel{\triangle}{=} q \qquad\qquad \widehat{\delta}(q, xa) \stackrel{\triangle}{=} \delta(\widehat{\delta}(q, x), a).$$

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

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

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 q, then $[\![M]\!](x) = \ell(q)$.

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 Q such that

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

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

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

(The restriction that $\mu\alpha.\,\sigma$ is not a variable 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 $[\![\tau]\!] = [\![M_\tau]\!]$.

For those with an interest in such things, 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 $x \in \{L, R\}^*$, $[\![\sigma]\!](x) = [\![\tau]\!](x)$; equivalently, $[\![\sigma]\!] \neq [\![\tau]\!]$ iff there exists $x \in \mathrm{dom}\,[\![\sigma]\!] \cap \mathrm{dom}\,[\![\tau]\!]$ such that $[\![\sigma]\!](x) \neq [\![\tau]\!](x)$. Form the two term automata $M_\sigma = (Q_\sigma, \delta_\sigma, \ell_\sigma, s_\sigma)$ and $M_\tau = (Q_\tau, \delta_\tau, \ell_\tau, s_\tau)$. Then form the product automaton $M_\sigma \times M_\tau$ with states $Q_\sigma \times Q_\tau$, transition function $\lambda((p, q), D).\,(\delta_\sigma(p, D), \delta_\tau(q, D))$, 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 $x \in \{L, R\}$ that causes the product automaton to move from its start state to a state (p, q) such that $\ell_\sigma(p) \neq \ell_\tau(q)$. 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.

6 Subtyping

In this section we indicate how to extend the algorithm to handle equirecursive subtyping. Here we take types to be finite and infinite terms over the ranked alphabet $\Sigma = \{\bot, \to, \top, 1\}$, where $\to$ is binary and $\bot$, $\top$, 1 are constants. The type $\bot$ is supposed to be a subtype of all types and the type $\top$ is supposed to be a supertype of all types.

The finite types are ordered naturally by the binary relation $\leq_{\mathrm{FIN}}$ defined inductively by

(i) $\bot \leq_{\mathrm{FIN}} \tau \leq_{\mathrm{FIN}} \top$ for all finite τ;

(ii) if $\sigma' \leq_{\mathrm{FIN}} \sigma$ and $\tau \leq_{\mathrm{FIN}} \tau'$ then $\sigma \to \tau \leq_{\mathrm{FIN}} \sigma' \to \tau'$.

Note that the converse of (ii) holds as well. This relation captures the natural type inclusion order in that it is covariant in the range and contravariant in the domain of a function type.

In order to handle recursive types, we need to extend the ordering $\leq_{\mathrm{FIN}}$ to infinite types in a natural way. One natural definition involves infinite sequences of finite approximations. Here we use an equivalent and simpler definition that does not involve approximations.

The *parity* of a string $x \in \{L, R\}^*$, denoted πx, is the number mod 2 of L's in x. A string x is said to be *even* if $\pi x = 0$ and *odd* if $\pi x = 1$.

Let $\leq_0$ and $\leq_1$ be the following two partial orders on Σ:

$$\begin{array}{ccccc} \bot & \leq_0 & \to & \leq_0 & \top \\ \bot & \leq_0 & 1 & \leq_0 & \top \end{array} \qquad\qquad \begin{array}{ccccc} \top & \leq_1 & \to & \leq_1 & \bot \\ \top & \leq_1 & 1 & \leq_1 & \bot. \end{array}$$

Note that $\leq_0$ and $\leq_1$ are reverses of each other. For types σ, τ, define $\sigma \leq \tau$ if $\sigma(x) \leq_{\pi x} \tau(x)$ for all $x \in \operatorname{dom}\sigma \cap \operatorname{dom}\tau$.

One can show without much difficulty that the relation $\leq$ is a partial order on types and agrees with $\leq_{\mathrm{FIN}}$ on the finite types. In particular, for any σ, τ, σ', τ',

(i) $\bot \leq \tau \leq \top$

(ii) $\tau \leq \bot$ if and only if $\tau = \bot$

(iii) $\top \leq \tau$ if and only if $\tau = \top$

(iv) $\sigma \to \tau \leq \sigma' \to \tau'$ if and only if $\sigma' \leq \sigma$ and $\tau \leq \tau'$.

7 An Algorithm

To decide whether $[\![\sigma]\!] \leq [\![\tau]\!]$ for two given type expressions σ and τ, we proceed as in Section 4. We first construct the term automata M_σ and M_τ, then form their product; however, we also include one extra bit of information in the state to record the parity of the path scanned so far. This is to account for contravariance of function types in their domain.

Recall that $[\![\sigma]\!] \leq [\![\tau]\!]$ iff $[\![\sigma]\!](x) \leq_{\pi x} [\![\tau]\!](x)$ for all $x \in \operatorname{dom}[\![\sigma]\!] \cap \operatorname{dom}[\![\tau]\!]$. Equivalently, $[\![\sigma]\!] \not\leq [\![\tau]\!]$ iff the set

$$\{x \in \operatorname{dom}[\![\sigma]\!] \cap \operatorname{dom}[\![\tau]\!] \mid [\![\sigma]\!](x) \not\leq_{\pi x} [\![\tau]\!](x)\}$$

is nonempty. This is a regular subset of $\{L, R\}^*$, as it is the set accepted by the finite-state automaton

$$(Q, \{L, R\}, s, \delta, F)$$

where

- $Q \stackrel{\triangle}{=} Q_\sigma \times Q_\tau \times \{0, 1\}$ are the states;
- $s \stackrel{\triangle}{=} (s_\sigma, s_\tau, 0)$ is the start state;

- $\delta : Q \times \{L, R\} \to Q$ is the partial function which for $b \in \{0, 1\}$, $D \in \{L, R\}$, $p \in Q_\sigma$, and $q \in Q_\tau$ gives

$$\delta((p, q, b), D) \quad \stackrel{\triangle}{=} \quad (\delta_\sigma(p, D), \delta_\tau(q, D), b \oplus \pi D)$$

  where $\oplus$ denotes mod 2 sum;

- $F \stackrel{\triangle}{=} \{(p, q, b) \mid \ell_\sigma(p) \not\leq_b \ell_\tau(q)\}$ is the set of accept states.

Then $\delta((p, q, b), D)$ is defined if and only if $\ell_\sigma(p) = \ell_\tau(q) = \rightarrow$. The automaton is nondeterministic only in that the state (p, q, b) has no D-successors if either $\ell_\sigma(p)$ or $\ell_\tau(q) \in \{\bot, \top, 1\}$. If $\ell_\sigma(p) = \ell_\tau(q) = \rightarrow$, then the D-successor of (p, q, b) is defined and is unique.

Now to decide whether $[\![\sigma]\!] \leq [\![\tau]\!]$, we construct the automaton and ask whether it accepts a nonempty set, that is, whether there exists a path from the start state to some final state. This can be determined in linear time in the size of the automaton using depth first search.

The automaton has $2 \cdot |Q_\sigma| \cdot |Q_\tau|$ states and at most two transition edges from each state. Thus the entire algorithm takes no more than $O(|\sigma| \cdot |\tau|)$ time, where $|\sigma|$ and $|\tau|$ are the sizes of the original type expressions representing the regular terms $[\![\sigma]\!]$ and $[\![\tau]\!]$.