A Theory of Second-Order Trees

Neal Glew

aglew@acm.org

Abstract. This paper describes a theory of second-order trees, that is, finite and infinite trees where nodes of the tree can bind variables that appear further down in the tree. Such trees can be used to provide a natural and intuitive interpretation for type systems with equirecursive types and binding constructs like universal and existential quantifiers. The paper defines the set of *binding trees*, and a subset of these called *regular binding trees*. These are similar to the usual notion of regular trees, but generalised to take into account the binding structure. Then the paper shows how to interpret a second-order type system with recursive quantifiers as binding trees, and gives a sound and complete axiomatisation of when two types map to the same tree. Finally the paper gives a finite representation for trees called *tree automata*, and gives a construction for deciding when two automata map to the same tree. To tie everything together, the paper defines a mapping from types to automata, thus giving a decision procedure for when two types map to the same tree.

1 Introduction

In the theory of type systems there are two approaches to recursive types, the *isorecursive* and *equirecursive* approach. In the *isorecursive approach*, the types $\mathsf{rec}\ \alpha.\tau$ and $\tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$[1] are considered different but isomorphic types. The expression language includes type coercions[2] $\mathsf{roll}(e)$ and $\mathsf{unroll}(e)$ for converting a value of one type to the other. The isorecursive approach is easier to construct decision procedures for, and is easier to prove sound; but it requires programs to contain type coercions. In the *equirecursive approach*, the types $\mathsf{rec}\ \alpha.\tau$ and $\tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$ are considered equal, and there are no expression-level constructs for dealing with recursive types. The presence of this equality makes it more difficult to develop decision procedures for equality and subtyping, and more difficult to prove the type system sound. However, there are no type coercions in programs, and more types are equivalent. For example, $\mathsf{rec}\ \alpha.\tau$ and $\mathsf{rec}\ \alpha.(\tau\{\alpha := \tau\})$ are equal in the equirecursive types approach but are not intercoercible in the isorecursive approach.

A more fundamental problem with the equirecursive approach is that previous work on formalising it has gaps (see below). This paper fills these gaps by providing solid foundations for second-order type systems with equirecursive types.

[1] The notation $x\{y := z\}$ denotes the capture avoiding substitution of z for y in x.

[2] A type coercion changes the type of an expression but has no runtime effect.

Amadio and Cardelli [AC93] were the first to investigate the equirecursive approach. They proposed the *tree interpretation* of recursive types, which is based on the idea that if the equality between $\mathsf{rec}\ \alpha.\tau$ and $\tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}$ is applied repeatedly then recursive types expand into infinite trees that have no recursive quantifiers. Then, types are equal exactly when their fully-expanded trees are the same. Furthermore, subtyping can first be defined on trees and then lifted to types. Amadio and Cardelli defined a suitable tree model for a first-order type system, and a definition of subtyping between trees. Then they gave a set of type equality and subtyping rules, and proved them sound and complete with respect to the tree interpretation. Finally they gave an algorithm for deciding type equality and subtyping. Their algorithm is exponential time in the worst case, which is much worse than the linear time algorithm in the isorecursive approach.

Kozen et al. [KPS95] reduced this exponential time complexity to polynomial time. First they defined term automata,[3] which, like types, are a finite representation for trees. Briefly, a term automaton is finite state machine. Each state represents a collection of nodes in a tree, the initial state is the root of the tree, and the transition function gives the children for each node and the labels on the edges to these children. Kozen et al. gave an intersection-like construction on term automata that can be used to decide equality and subtyping in quadratic time. In both Amadio and Cardelli and Kozen et al.'s work, only first-order systems were considered, and their results do not generalise in a straightforward manner to second-order types.

Colazzo and Ghelli [CG99] investigated a second-order type system with equirecursive quantifiers. They gave a coinductively defined set of type rules (type rules are normally defined inductively), and described and proved correct an algorithm for deciding subtyping. They did not show the relation between their rules and Amadio and Cardelli's rules. Nor did they analyse the complexity of their algorithm, although they conjectured it was exponential. Their algorithm is essentially a search algorithm with the curious feature that it cuts off search when it sees the same subtyping judgement for the third, not second, time. They were able to show that this criterion is necessary and sufficient, but never gave an intuitive explanation.

This paper extends the tree interpretation and idea of tree automata from first-order to second-order trees. The first contribution is a notion of second-order finite and infinite trees suitable as a semantic model for types. The second contribution is a proof that the usual equality rules for equirecursive types are sound and complete for this model. The third contribution is notion of tree automata suitable for second-order trees. The fourth contribution is a polynomial time decision procedure for type equality. This paper deals only with equality and subtyping is left to future work. Since equality is not something specific to types, I will call them terms in the sequel. The rest of the paper presents each contribution in turn. Full details including proofs are avaiable in a companion technical report [Gle02].

[3] This paper will call Kozen et al.'s term automata, tree automata.

2 Preliminaries

The theory is meant to be general and to abstract over everything else in the term system. Therefore, I will assume the term language consists of variables, recursive quantifiers, and terms build from node labels $nl \in NL$. Each node label will take a number of arguments, which will be identified using labels $\ell \in L$, and for each argument may bind a certain number of variables. The function $spec \in NL \to L \overset{\text{fin}}{\rightharpoonup} \mathbb{N}$ defines which arguments a node label takes and how many variables it binds for each. For example, the system $\mathcal{F}_{\leq\mu}$ of Colazzo and Ghelli has $NL = \{\top, \to, \forall\}$, $L = \{\mathsf{arg}, \mathsf{bnd}, \mathsf{bdy}, \mathsf{res}\}$, and $spec = \{(\to, \mathsf{arg}, 0), (\to, \mathsf{res}, 0), (\forall, \mathsf{bnd}, 0), (\forall, \mathsf{bdy}, 1)\}$. In this system, $\forall$ is a quantifier for bounded polymorphic types and binds one variable in its body (label bdy), so its specification is 1; the bound (label bnd) does not bind a variable, so its specification is 0. The rest of the paper will not refer to *spec*, instead it will use functions $\text{LABELS}(nl) = \text{dom}(spec(nl))$ and $\text{BIND}(nl, \ell) = spec(nl)(\ell)$.

Notations and Conventions $A \rightharpoonup B$ is the set of all partial functions from A to B; $A \overset{\text{fin}}{\rightharpoonup} B$ is the set of partial functions from A to B with finite domain. If e_1 and e_2 are possibly undefined expressions then $e_1 = e_2$ means that either both are defined and equal or both are undefined. A^* is the set of finite sequences of elements from A; ϵ is the empty sequence; prefix, concatenation, and append are written using juxtaposition; if $x, y \in A^*$ then $x \leq y$ means that x is a prefix of y. $|\cdot|$ is used both as the size of a set and the length of a sequence. $A + B$ is the disjoint union of A and B; tags in this font will be used for injections, which tags correspond to which arms should be clear. If f is a function then $f\{x \mapsto y\}$ is the same function except that it maps x to y. If R is an equivalence relation then $[R]$ is the set of its equivalence classes and $[x]_R$ is the equivalence class of x under R.

3 Binding Trees

Binding trees are just finite or infinite trees whose nodes are labeled by node labels or by variables in a De Bruijn representation [Bru72] and whose edges are labeled by L. A node labeled nl has edges labeled by $\text{LABELS}(nl)$ and variables are leafs. This can be formalised as follows.

Definition 1 *The set of binding trees is defined as:*

$$\begin{aligned} Tree = \{t : L^* \rightharpoonup NL + \mathbb{N} \mid \\ \epsilon \in \text{dom}(t) \wedge (p\ell \in \text{dom}(t) \Leftrightarrow \exists nl : t(p) = \mathsf{nl}(nl) \wedge \ell \in \text{LABELS}(nl))\} \end{aligned}$$

Distance between two trees is defined as $d(t_1, t_2) = 2^{-\min\{n \mid t_1(p) \neq t_2(p) \wedge n = |p|\}}$ *where* $2^{-\min \emptyset} = 0$.

For example, the term $\forall\alpha.\mathsf{rec}\ \beta.(\alpha \to \forall\gamma.\beta)$ maps to the tree $\{(\epsilon, \mathsf{nl}(\forall)), (\mathsf{bdy}, \mathsf{nl}(\to)), (\mathsf{bdyarg}, \mathsf{var}(0)), (\mathsf{bdyres}, \mathsf{nl}(\forall)), (\mathsf{bdyresbdy}, \mathsf{nl}(\to)), \ldots\}$ (tree `t1` in

Figure 1). De Bruijn indices are used to get a unique representation for the binding structure. The term above maps to the above tree rather than to an α-equivalence class of trees, as it would if explicit variables were used in trees.

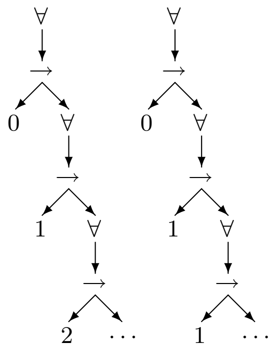


Fig. 1. Examples trees `t1`, on left, and `t2`, on right.

Recursive quantifiers will be interpreted as fixed points of maps on *Tree*, so it is important that such fixed points exist. In fact, (Tree, d) is a complete ultrametric space, which means that contractive maps on it have unique fixed points (c.f. [Kap77]). A map f is *contractive* on a metric space (X, d) if there exists a $c \in [0, 1)$ such that $d(f(x_1), f(x_2)) \leq cd(x_1, x_2)$ for all x_1 and x_2 in X—that is, f maps all pairs a certain fraction closer together.

The following two tree constructors will be needed latter. The first builds a tree whose root is the free variable n, and the second builds a tree with root nl and the given trees as the children of the root.

Definition 2 *Tree* $\textsc{var}(n)$ *is* $\{(\epsilon, \mathsf{var}(n))\}$ *and tree* $\textsc{nl}(nl, \ell = t_\ell)_{\ell \in \mathrm{LABELS}(nl)}$ *is* $\{(\epsilon, \mathsf{nl}(nl))\} \cup \{(\ell p, t_\ell(p)) \mid \ell \in \textsc{labels}(nl) \wedge p \in \mathrm{dom}(t_\ell)\}$ *if* $t_\ell \in \mathit{Tree}$*.*

3.1 Regular Trees

Not all trees are expressible using terms or automata, so it is important to define a subset of trees that are. In the theory of first-order trees, these are called regular trees, and are defined as those with a finite number of subtrees. This definition is inadequate for binding trees, as the above example tree (which should be regular) has a subtree of the form $\textsc{var}(n)$ for each $n \in \mathbb{N}$. The problem is that the De Bruijn indices represent conceptually the same variable as possibility different tree nodes. The definition of regular trees needs to take into account the binding structure and De Bruijn representation. Before giving the definition, two preliminary concepts are needed.

Definition 3 *The number of variables bound along path p in tree t is:*

$$\begin{aligned} \textsc{bind}(t,\epsilon) &= 0 \\ \textsc{bind}(t,p\ell) &= \textsc{bind}(t,p) + \textsc{bind}(nl,\ell) \textit{ where } t(p) = \mathsf{nl}(nl) \end{aligned}$$

The number of variables bound from path p_1 to p_2 where $p_1 \le p_2$ in tree t is $\textsc{bind}(t, p_1 \to p_2) = \textsc{bind}(t,p_2) - \textsc{bind}(t,p_1)$.

The key to regular trees is that variable nodes that conceptually refer to the same variable (e.g., the same free variable or are bound by the same node) should be considered equal. The next definition makes precise what a variable node in a tree refers to.

Definition 4 *The variable identified by path p of tree t where $t(p) = \mathsf{var}(n)$ is:*

$$\textsc{varof}(t,p) = \begin{cases} \mathsf{free}(n - \textsc{bind}(t,p)) & \textsc{bind}(t,p) \le n \\ \mathsf{bound}(p_1,\ell,i) & p_1\ell \le p \wedge i = n - \textsc{bind}(t, p_1\ell \to p) \wedge \\ & \textsc{bind}(t,p_1\ell \to p) \le n < \textsc{bind}(t,p_1 \to p) \end{cases}$$

For example, if t is the tree above then $\textsc{varof}(t, \mathsf{bdyarg}) = \mathsf{bound}(\epsilon, \mathsf{bdy}, 0)$ and if t' is the subtree along the bdy edge then $\textsc{varof}(t', \mathsf{resbdyarg}) = \mathsf{free}(0)$.

Using this, when two subtrees of a tree might be equal is defined coinductively as follows.

Definition 5 *A relation R is an equivalence of t's subtrees exactly when R is an equivalence relation on* $\mathrm{dom}(t)$ *and $p_1 \mathrel{R} p_2$ implies either*

- $t(p_1) = \mathsf{nl}(nl)$, $t(p_2) = \mathsf{nl}(nl)$, *and $p_1\ell \mathrel{R} p_2\ell$ for all* $\ell \in \textsc{labels}(nl)$,
- $\textsc{varof}(t,p_1) = \mathsf{free}(n)$ *and* $\textsc{varof}(t,p_2) = \mathsf{free}(n)$, *or*
- $\textsc{varof}(t,p_1) = \mathsf{bound}(p'_1,\ell,n)$, $\textsc{varof}(t,p_2) = \mathsf{bound}(p'_2,\ell,n)$, *and* $p'_1 \mathrel{R} p'_2$.

However, more is needed. Consider this tree `t2` in Figure 1. There exists an equivalence of this trees subtrees that relates all the $\forall$ nodes and has a finite number of equivalence classes. However, no term or term automata generates this tree. The problem is that the variable on path $\mathsf{bdyresbdyarg}$ binds not to the nearest $\forall$ but to the previous one, and terms never have this kind of binding structure. Therefore I define nonoverlapping equivalences to rule out these kinds of binding structures.

Definition 6 *An equivalence R of t's subtrees is nonoverlapping exactly when* $\textsc{varof}(t,p_3) = \mathsf{bound}(p_1,\ell,n)$ *and $p_1 < p_2 < p_3$ implies $(p_1,p_2) \notin R$.*

A tree t is regular if there exists a nonoverlapping equivalence of t's subtrees with a finite number of equivalence classes. RegTree is the set of regular trees.

While the definition of regular binding trees is not nearly as elegant as that for regular trees without binding, it nevertheless does define exactly the set of trees that are expressible by terms or automata, as will be proven later.

It turns out that trees have a greatest nonoverlapping equivalence of their subtrees; later sections will use this to define a canonical term for each tree and to prove completeness of the equality rules.

Definition 7 *For $t \in Tree$,* $\textsc{eqst}(t)$ *is the union of the nonoverlapping equivalences of t's subtrees.*

Theorem 8 *If $t \in Tree$ then* $\textsc{eqst}(t)$ *is a nonoverlapping equivalence of t's subtrees. Hence, $t \in RegTree$ if and only if* $\textsc{eqst}(t)$ *has finite number of equivalence classes.*

Proof sketch: The proof shows that $\textsc{eqst}(t)$'s transitive closure is a nonoverlapping equivalence of t's subtrees. Then it is a subset and thus equal to $\textsc{eqst}(t)$, so the latter is a nonoverlapping equivalence of t's subtrees. Since $\textsc{eqst}(t)$ is a union of equivalence relations, its transitive closure is an equivalence relation. Since the other conditions for equivalence of subtrees are monotonic in the relation, it is easy to show that $\textsc{eqst}(t)$'s transitive closure is an equivalence of t's subtrees. The tricky part is to show that it is nonoverlapping. Space does not permit showing the details, these are in the companion technical report [Gle02]. $\square$

One final operator on trees is needed latter. It is the *shifting* operation common to De Bruijn representations, and returns an identical tree except that the tree's free variables are incremented by a constant.

Definition 9 *The shift of a tree t by n is:*

$$\textsc{shift}(t, n) = \lambda p. \begin{cases} \mathsf{var}(n+i) & t(p) = \mathsf{var}(i) \wedge i \geq \textsc{bind}(t, p) \\ t(p) & \textit{otherwise} \end{cases}$$

4 Terms

The second-order term language with recursive quantifiers is:

$$\tau ::= \alpha \mid nl(\ell = \overrightarrow{\alpha_\ell} . \tau_\ell)_{\ell \in \mathrm{LABELS}(nl)} \mid \mathsf{rec}\ \alpha . \tau$$

where α ranges over term variables and $\overrightarrow{\alpha}$ is a sequence of term variables. Terms are considered equal up to alpha conversion, where for $nl(\ell = \overrightarrow{\alpha_\ell} . \tau_\ell)$, $\overrightarrow{\alpha_\ell}$ binds in τ_ℓ, and for $\mathsf{rec}\ \alpha.\tau$, α binds in τ.

Not all phrases matching the above grammar are considered terms, but only those that satisfy two further constraints. In terms of the from $nl(\ell = \overrightarrow{\alpha_\ell} . \tau_\ell)$ it must be that $\mid \overrightarrow{\alpha_\ell} \mid = \textsc{bind}(nl, \ell)$. In terms of the form $\mathsf{rec}\ \alpha.\tau$ it must be that $\tau \downarrow \alpha$, where the latter means that τ is syntactically contractive in α and is defined as:

$$\begin{array}{ll} \beta \downarrow \alpha & \Leftrightarrow \alpha \neq \beta \\ nl(\ell = \overrightarrow{\alpha_\ell} . \tau_\ell) \downarrow \alpha & \\ \mathsf{rec}\ \beta.\tau \downarrow \alpha & \Leftrightarrow \alpha = \beta \vee \tau \downarrow \alpha \end{array}$$

Intuitively, $\tau \downarrow \alpha$ if mapping α to τ is not equivalent to the identity mapping, for which any term is a fixed point. Instead mapping α to τ produces a term whose outer most constructor is independent of α, and can have only one fixed point.

4.1 Terms to Trees

The interpretation of terms as trees depends upon the interpretation of its free variables. An environment η is a mapping from term variables to *Tree*. Terms under a binder are interpreted in a shifted environment.

Definition 10

$$\textsc{shift}(\eta, \alpha_0 \cdots \alpha_{n-1}) = \lambda\beta. \begin{cases} \textsc{var}(i) & \beta = \alpha_i \\ \textsc{shift}(\eta(\beta), n) & \beta \notin \{\alpha_0, \ldots, \alpha_{n-1}\} \end{cases}$$

With these preliminaries, the interpretation of a term as a tree is straightforward.

Definition 11

$$\begin{aligned} [\![\alpha]\!]_\eta &= \eta(\alpha) \\ [\![nl(\ell = \vec{\alpha_\ell}\ .\tau_\ell)]\!]_\eta &= \textsc{nl}(nl, \ell = [\![\tau_\ell]\!]_{\textsc{shift}(\eta, \vec{\alpha_\ell})}) \\ [\![\mathsf{rec}\ \alpha.\tau]\!]_\eta &= \mathrm{fix}([\![\alpha.\tau]\!]_\eta) \\ [\![\alpha.\tau]\!]_\eta &= \lambda t.[\![\tau]\!]_{\eta\{\alpha \mapsto t\}} \end{aligned}$$

where $\mathrm{fix}(\cdot)$ *maps a contractive function to its unique fixed point*

It is easy to show that a term's interpretation is a uniquely defined tree, and the proof also shows that syntactically contractive term and variable pairs define contractive maps.

Theorem 12 *If η maps term variables to Tree then $[\![\tau]\!]_\eta \in$ Tree. If $\tau \downarrow \alpha$ then $[\![\alpha.\tau]\!]_\eta$ is contractive.*

The interpretation of terms as trees produces a regular tree. Also, all regular trees are also expressible as terms; this is proven in the section on completeness.

Theorem 13 *If η maps term variables to RegTree then $[\![\tau]\!]_\eta \in$ RegTree.*

4.2 Term-Equality Rules

The term-equality rules derive judgements of the form $\vdash \tau_1 = \tau_2$ intended to mean that terms τ_1 and τ_2 are equal. The rules are essentially those of Amadio and Cardelli:

$$\text{(eqsym)}\ \frac{\vdash \tau_2 = \tau_1}{\vdash \tau_1 = \tau_2} \qquad \text{(eqtrans)}\ \frac{\vdash \tau_1 = \tau_2 \quad \vdash \tau_2 = \tau_3}{\vdash \tau_1 = \tau_3} \qquad \text{(eqvar)}\ \frac{}{\vdash \alpha = \alpha}$$

$$\text{(eqnl)}\ \frac{\vdash \tau_\ell = \sigma_\ell}{\vdash nl(\ell = \vec{\alpha_\ell}\ .\tau_\ell) = nl(\ell = \vec{\alpha_\ell}\ .\sigma_\ell)} \qquad \text{(eqrec)}\ \frac{\vdash \tau = \sigma}{\vdash \mathsf{rec}\ \alpha.\tau = \mathsf{rec}\ \alpha.\sigma}$$

$$\text{(eqroll)}\ \frac{}{\vdash \mathsf{rec}\ \alpha.\tau = \tau\{\alpha := \mathsf{rec}\ \alpha.\tau\}}$$

$$\text{(equnq)}\ \frac{\vdash \tau_1 = \sigma\{\alpha := \tau_1\} \quad \vdash \tau_2 = \sigma\{\alpha := \tau_2\}}{\vdash \tau_1 = \tau_2}\ (\sigma \downarrow \alpha)$$

The first two rules express that equality is symmetric and transitive. The next three rules express that equality is closed under all the term constructors and that equality is reflexive. Together these five rules make equality a congruence relation. The last two rules are the interesting ones. Rule (eqroll) expresses that $\mathsf{rec}\ \alpha.\tau$ is the fixed point of the mapping of α to τ, and is the rule from the introduction that defines the equirecursive approach. Rule (equnq) expresses the fact that contractive mappings have unique fixed points. The hypotheses expresses that τ_1 and τ_2 are a fixed points of the mapping α to σ; the side condition expresses that the mapping is contractive; and the conclusion expresses that the two fixed points are equal.

It is straightforward to prove the rules sound, that is, that provably equal terms map to the same tree.

Theorem 14 (Soundness) *If* $\vdash \tau_1 = \tau_2$ *then* $[\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta$ *for all environments* η.

4.3 Completeness

It is more difficult to show the converse, that terms that map to the same tree are provably equal. Amadio and Cardelli showed completeness by defining something called *systems of equations*. Unfortunately, it seems very difficult to define systems of equations for second-order terms. So I use a different approach to showing completeness, which also works in the first-order setting. The idea is to define a canonical term for every regular tree and show that a term is provably equal to the canonical term for its tree. Completeness then follows by transitivity.

I will define canonical terms for trees with respect to particular kinds of environments called distinguishing environments. These are environments of the form $\eta(\alpha) = \textsc{var}(g(\alpha))$ for some g that is a bijection from term variables to $\mathbb{N}$.

Definition 15 *If* $t \in \mathit{RegTree}$, η *is distinguishing, and* R *is a nonoverlapping equality of* t*'s subtrees with a finite number of equivalence classes, then* T, S, $\textsc{termof}_\eta(t, R)$, *and* $\textsc{termof}_\eta(t)$ *are defined as follows:*

– *If* f *maps* $[R]$ *to term variables,* f *maps the pair* $([p]_R, \ell)$ *to a sequence of term variables of length* $\textsc{bind}(nl, \ell)$ *when* $t(p) = \mathsf{nl}(nl)$ *and* $\ell \in \textsc{labels}(nl)$, S *is a subset of* $[R]$, *and* $p \in \mathrm{dom}(t)$ *then:*

$$\mathsf{T}^{t,\eta,R,f}_{S,p} = \begin{cases} f([p]_R) & [p]_R \in S \\ \mathsf{rec}\ f([p]_R).\mathsf{S}^{t,\eta,R,f}_{S\cup\{[p]_R\},p} & [p]_R \notin S \end{cases}$$

$$\mathsf{S}^{t,\eta,R,f}_{S,p} = \begin{cases} \beta & \textsc{varof}(t,p) = \mathsf{free}(n) \wedge \\ & \eta(\beta) = \textsc{var}(n) \\ f([p']_R, \ell)_n & \textsc{varof}(t,p) = \mathsf{bound}(p', \ell, n) \\ nl(\ell = f([p]_R, \ell).\mathsf{T}^{t,\eta,R,f}_{S,p\ell}) & t(p) = \mathsf{nl}(nl) \end{cases}$$

– $\textsc{termof}_\eta(t, R) = \mathsf{T}^{t,\eta,R,f}_{\emptyset,\epsilon}$ *for some appropriate* f *that maps to fresh variables*

– $\textsc{termof}_\eta(t) = \textsc{termof}_\eta(t, \textsc{eqst}(t))$

These terms do map to the tree they are based on, and this shows that every regular tree is expressible using a term.

Theorem 16 *If $t \in RegTree$ and η is distinguishing then $[\![\textsc{termof}_\eta(t)]\!]_\eta = t$.*

The key technical result used to show completeness is that every term is provably equal to the canonical term for its tree.

Lemma 17 *If η is distinguishing then $\vdash \textsc{termof}_\eta([\![\tau]\!]_\eta) = \tau$.*

Proof sketch: The proof is ultimately by induction on the structure of τ. If τ has the form $nl(\ell = \overrightarrow{\alpha_\ell} .\tau_\ell)$ then $\textsc{termof}_\eta([\![\tau]\!]_\eta)$ has the form $\mathsf{rec}\ \alpha.nl(\ell = \overrightarrow{\alpha_\ell} .\sigma_\ell)$. If the induction hypothesis is $\vdash \sigma_\ell\{\alpha := \textsc{termof}_\eta([\![\tau]\!]_\eta)\} = \tau_\ell$ (1), then the result follows by rules (eqnl), (eqroll), and (eqtrans). If τ has the form $\mathsf{rec}\ \alpha.\sigma$ then the key is to have the induction hypothesis $\vdash \textsc{termof}_\eta([\![\tau]\!]_\eta) = \sigma\{\alpha := \textsc{termof}_\eta([\![\tau]\!]_\eta)\}$ (2). Then by (eqroll), $\vdash \tau = \sigma\{\alpha := \tau\}$ and the result follows by rule (equnq). The key then, is to get the induction hypothesis to satisfy properties (1) and (2). The proof first defines for each subterm σ of τ a pair of terms (σ_1, σ_2) satisfying properties (1) and (2). Next it shows that the trees for these two terms and a corresponding subtree of $[\![\tau]\!]_\eta$ are all the same. Finally it shows by induction on σ that $\vdash \sigma_1 = \sigma_2$. Details are in the companion technical report [Gle02]. □

Completeness follows easily from this last lemma.

Theorem 18 (Completeness) *If η is distinguishing and $[\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta$ then $\vdash \tau_1 = \tau_2$.*

Proof: By Lemma 17, $\vdash \textsc{termof}_\eta([\![\tau_1]\!]_\eta) = \tau_1$ and $\vdash \textsc{termof}_\eta([\![\tau_2]\!]_\eta) = \tau_2$. Since $[\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta$, $\textsc{termof}_\eta([\![\tau_1]\!]_\eta) = \textsc{termof}_\eta([\![\tau_2]\!]_\eta)$. The result follows by (eqsym) and (eqtrans). □

5 Binding Tree Automata

The final step of my programme is to give a decision algorithm for term equality. The algorithm is in terms of a finite representation for trees, similar to the term automata of Kozen et al. The basic idea is that an automaton is given as input a path and gives as output the node at the end of that path. Automata are state machines, that is, each label in the path causes the automaton to transition from one state to another, starting with an initial state, and the state obtained at the end of the path determines the output. Rather than output the node in the form a tree does, that is, as an element of $NL + \mathbb{N}$, the automaton outputs either a node label, a free variable index, or a bound variable. A bound variable is specified as a state, label, and index; the idea being that the variable is bound by the most recent occurrence of the binder generated by the state, along the edge given by the label.

A few preliminaries are needed for the definition. If $\delta \in Q \times L \rightharpoonup Q$ for some Q then δ^* is its extension to L^*, specifically $\delta^*(q, \epsilon) = q$ and $\delta^*(q, p\ell) = \delta(\delta^*(q, p), \ell)$. If $\delta^*(q_1, p) = q_2$ then p is a path from q_1 to q_2; if in addition $\delta^*(q_1, p') \neq q$ for all $p' \leq p$ then p is a q-less path from q_1 to q_2.

Definition 19 *A binding tree automaton is a quadruple (Q, i, δ, sl) satisfying:*

- *Q is a finite set called the states of the automaton.*
- *$i \in Q$ is the initial state.*
- *$\delta \in Q \times L \rightharpoonup Q$ is the transition function.*
- *$sl \in Q \to NL + \mathbb{N} + Q \times L \times \mathbb{N}$ is the labeling or output function.*
- *$(q, \ell) \in \mathrm{dom}(\delta) \Leftrightarrow sl(q) = \mathsf{nl}(nl) \wedge \ell \in \textsc{labels}(nl)$.*
- *$sl(q) = \mathsf{fvar}(q', \ell, n)$ implies that:*
  - *$sl(q') = \mathsf{nl}(nl)$,*
  - *$\ell \in \textsc{labels}(nl)$,*
  - *$0 \leq n < \textsc{bind}(nl, \ell)$, and*
  - *if p is a path from i to q then there are paths p_1 and p_2 such that $p = p_1 \ell p_2$, p_1 is a path from i to q', and p_2 is q'-less (from $\delta(q', \ell)$ to q).*

5.1 Automata to Trees

This section explains how an automaton generates a tree. Letting t represent this tree then intuitively: if $sl(\delta^*(i, p)) = \mathsf{nl}(nl)$ then $t(p) = \mathsf{nl}(p)$; if $sl(\delta^*(i, p)) = \mathsf{bvar}(n)$ then $\textsc{varof}(t, p) = \mathsf{free}(n)$; finally, if $sl(\delta^*(i, p)) = \mathsf{fvar}(q', \ell, n)$ then $\textsc{varof}(t, p) = \mathsf{bound}(p', \ell, n)$ where p' is the longest path from i to q' that is a prefix of p.

The formal definition extends the state space to include enough information to compute the De Bruijn indices for free and bound variables. An extended state is a triple consisting of a state of the automaton, the number of variables bound along the path so far (needed to determine free variables), and a function $f \in Q \to \mathbb{N}$, which gives the number of variables bound since the last occurrence of each state (need to determine bound variables). The transition function is lifted to extended states in such a way as to track the binding information, and the labeling function is lifted to extended states to use the binding information to generate nodes for trees. Then the tree of an automaton at a path is just the lifted labeling function of the lifted transition function on the path.

Definition 20 *The tree associated with an automaton is defined as follows:*

$$
\begin{aligned}
\textsc{shift}(f, q := n) &= \lambda q'. \begin{cases} 0 & q' = q \\ f(q') + n & q' \neq q \end{cases} \\
\hat{\delta}((q, n, f), \ell) &= (\delta(q, \ell), n + i, \textsc{shift}(f, q := i)) \\
&\quad\ \textit{where } sl(q) = \mathsf{nl}(nl) \wedge i = \textsc{bind}(nl, \ell) \\
\hat{sl}(q, n, f) &= \begin{cases} \mathsf{nl}(nl) & sl(q) = \mathsf{nl}(nl) \\ \mathsf{var}(n + i) & sl(q) = \mathsf{bvar}(i) \\ \mathsf{var}(f(q') + i) & sl(q) = \mathsf{fvar}(q', \ell, i) \end{cases} \\
\textsc{tree}((Q, i, \delta, sl), qnf) &= \lambda p. \hat{sl}(\hat{\delta}^*(qnf, p)) \\
\hat{\imath} &= (i, 0, \lambda q. 0) \\
\textsc{tree}(Q, i, \delta, sl) &= \textsc{tree}((Q, i, \delta, sl), \hat{\imath})
\end{aligned}
$$

It is not hard to see that automata generate trees, in fact, they generate regular trees.

Theorem 21 *If ta is an automaton then* $\textsc{tree}(ta) \in \mathit{RegTree}$.

Proof sketch: Let $R = \{(p_1, p_2) \mid \delta^*(i, p_1) = \delta^*(i, P)\}$ where $ta = (Q, i, \delta, sl)$. Then R is a nonoverlapping equivalence of $\textsc{tree}(ta)$'s subtrees. It is clearly an equivalence relation. The other conditions for being an equivalence essentially follow from the fact that $\textsc{tree}(ta)(p)$ is determined by $\delta^*(i, p)$'s label. It is nonoverlapping because of the last condition in the definition of tree automata. $\square$

The converse is also true—regular trees are expressible as tree automata.

Theorem 22 *If t is a regular tree then there exists an automaton ta such that* $\textsc{tree}(ta) = t$.

Proof: The proof uses the equivalence classes of $\textsc{eqst}(t)$ as the states Q. The initial state i is $[\epsilon]$. If $t(p) = \mathsf{nl}(nl)$ then $sl([p]) = \mathsf{nl}(nl)$; if $\textsc{varof}(t, p) = \mathsf{free}(n)$ then $sl([p]) = \mathsf{bvar}(n)$; if $\textsc{varof}(t, p) = \mathsf{bound}(p', \ell, n)$ then $sl([p]) = \mathsf{fvar}([p'], \ell, n)$. The transition function δ is $\lambda([p], \ell).[p\ell]$. The conditions for equivalence of subtrees ensure that sl and δ are consistently defined. It is easy to check that this defines an automaton, the last part of the last condition follows from $\textsc{eqst}(t)$ being nonoverlapping. An easy induction shows that $\hat{\delta}^*(\hat{\imath}, p) = ([p], \textsc{bind}(t, p), f)$ where f is such that if p' is such that $p = p'\ell p''$ and p' is the largest strict prefix of p in $[p']$ then $f([p']) = \textsc{bind}(t, p'\ell \to p)$. Then it is easy to show that $\hat{sl}([p], \textsc{bind}(t, p), f) = t(p)$. $\square$

5.2 Equality Algorithm

Two trees are different if they differ at some path, but more specifically if they differ at some minimal path. This minimal path will be in the domain of both trees. Therefore to determine if two automata generate the different trees, it suffices to search for paths in their common domain that have different outputs. If the outputs along some path are the same up to but not including the last state then the number of variables bound up to the last state is the same. Thus, two free variable states will differ exactly when their indices differ, free variable states will differ from bound variable states, and bound variable states will differ if the most recent occurrence of the binding state occurred at different prefixes of the path. Thus determining if the outputs are different requires only keeping track of the states and the correspondence between binding states. As in Kozen et al., this can be expressed as a deterministic finite state automaton. The states of this equality automaton are triples, one state from each automata, and the correspondence between binding states. The transition function updates the states according to the input automata's transition functions and updates the correspondence. The accepting states are those where the output of the two states differs according to the binding state correspondence. The two trees differ if the language of the equality automaton is nonempty. The correspondence between binding states can be expressed as *partial bijections*, introduced next.

Definition 23 *The partial bijections between A and B are* $A \leftrightharpoons B = \{R \in \mathcal{P}(A \times B) \mid (a_1, b_1) \in R \wedge (a_2, b_2) \in R \Rightarrow (a_1 = a_2 \Leftrightarrow b_1 = b_2)\}$. *Bijection update of R by a maps to b is* $R\{a \leftrightharpoons b\} = \{(a', b') \in R \mid a' \neq a \wedge b' \neq b\} \cup \{(a, b)\}$.

Definition 24 *The equality deterministic finite state automaton (over alphabet L) of two automata $ta_1 = (Q_1, i_1, \delta_1, sl_1)$ and $ta_2 = (Q_2, i_2, \delta_2, sl_2)$ is:*

$$
\begin{array}{l}
\textsc{correspond}(sl_1, sl_2, q_1, q_2, R) = \\
\quad \vee (sl_1(q_1) = \mathsf{nl}(nl) \wedge sl_2(q_2) = \mathsf{nl}(nl)) \\
\quad \vee (sl_1(q_1) = \mathsf{bvar}(n) \wedge sl_2(q_2) = \mathsf{bvar}(n)) \\
\quad \vee (sl_1(q_1) = \mathsf{fvar}(q_1', \ell, n) \wedge sl_2(q_2) = \mathsf{fvar}(q_2', \ell, n) \wedge q_1' \; R \; q_2') \\
\textsc{equal}(ta_1, ta_2) = \\
\quad (Q_1 \times Q_2 \times (Q_1 \leftrightharpoons Q_2), \\
\quad\; (i_1, i_2, \emptyset), \\
\quad\; \lambda(q_1, q_2, R, \ell).(\delta_1(q_1, \ell), \delta_2(q_2, \ell), R\{q_1 \leftrightharpoons q_2\}), \\
\quad\; \{(q_1, q_2, R) \mid \neg\textsc{correspond}(sl_1, sl_2, q_1, q_2, R)\})
\end{array}
$$

The next theorem proves the correctness of this construction.

Theorem 25 $\textsc{tree}(ta_1) = \textsc{tree}(ta_2) \Leftrightarrow L(\textsc{equal}(ta_1, ta_2)) = \emptyset$

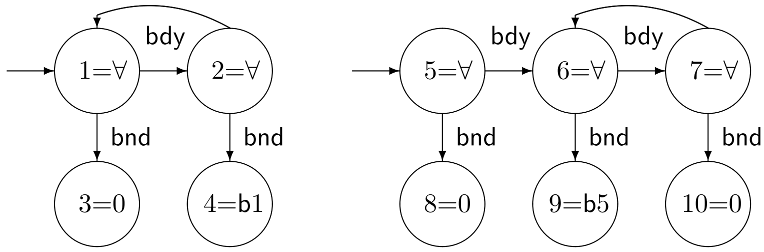
Proof sketch: The main part of the proof shows that if $\textsc{tree}(ta_1)(p') = \textsc{tree}(ta_2)(p')$ for $p' < p$ then: $\textsc{tree}(ta_1)(p) = \textsc{tree}(ta_2)(p)$ if and only if $p \in L(\textsc{equal}(ta_1, ta_2))$; from which the result easily follows. This property holds because of the correspondence between an automaton's tree and its state labels, and because $q_1 \; R \; q_2$, where $\delta^*(i, p) = (\_, \_, R)$, if and only if the most recent occurrence of q_1 and q_2 are at the same prefix of p. □

The previous theorem gives an algorithm for deciding the equality of the trees of two automata. Since emptiness of a deterministic finite state automaton's language is linear time, and the equality automaton is exponential in the size of the tree automata, it is an exponential algorithm. However, an optimisation yields a polynomial time algorithm. This optimisation is based on the following lemma.

Lemma 26 *If $ta_1 = (Q_1, i_1, \delta_1, sl_1)$ and $ta_2 = (Q_2, i_2, \delta_2, sl_2)$ are automata, $p_2 \neq \epsilon$, $p_3 \neq \epsilon$, $\delta_1^*(i_1, p_1) = \delta_1^*(i_1, p_1 p_2) = \delta_1^*(i_1, p_1 p_2 p_3) = q_1$, $\delta_2^*(i_2, p_1) = \delta_2^*(i_2, p_1 p_2) = \delta_2^*(i_2, p_1 p_2 p_3) = q_2$, and $p \notin L(\textsc{equal}(ta_1, ta_2))$ for $p \ngeq p_1 p_2 p_3$ then $L(\textsc{equal}(ta_1, ta_2)) = \emptyset$.*

This lemma says that the search for a word in $L(\textsc{equal}(ta_1, ta_2))$ does not need to search the entire space $Q_1 \times Q_2 \times (Q_1 \leftrightharpoons Q_2)$, but only needs to search until it sees the same $Q_1 \times Q_2$ pair three times. It is insufficient to stop after seeing the same pair twice, as in Colazzo and Ghelli's algorithm. For example, consider the following two automata:[4]

[4] The notation for automata depicts states as circles with $x = y$ inside; x is the state's identifier, and y is the states label, either a node label, a natural number for a free variable, or a $\mathsf{b}n$ for a variable bound by state n. An arrow not from another circle points to the initial state. The transition function is depicted with arrows between the circles labeled by L.



The path bdybdybdy produces the pair $(2, 6)$ for the second time, and while bdybnd is not accepted by the equality automaton, bdybdybdybnd is accepted. The problem is that states 4 and 9 are part of repetitive structures and the path bdybnd checks that the first repetition of 4 matches the first repetition of 9, but not that the other repetitions match. The path bdybdybdybnd checks whether the second repetitions match. Only when both the first and second repetitions match will all repetitions necessarily match.

With an appropriate choice of representation and implemented carefully, the above algorithm is quadratic in the sizes of the input automata.

6 Putting it Together

The previous section gave an algorithm for deciding equality of two automata, so an algorithm to convert terms into automata gives an algorithm for deciding term equality. This section gives that algorithm and its correctness.

First an algorithm to convert terms into automata. The states of this automaton are the parts of the term not involved with recursive quantifiers.

Definition 27 *The set* $\textsc{proper}(\tau)$ *is the subterms of* τ *that are not recursive quantifiers or variables bound by recursive quantifiers of* τ*. Every subterm of* τ *can be mapped to* $\textsc{proper}(\tau)$ *as follows: if* $\sigma \in \textsc{proper}(\tau)$ *then* $\textsc{proper}(\tau, \sigma) = \sigma$*;* $\textsc{proper}(\tau, \mathsf{rec}\ \alpha.\sigma) = \textsc{proper}(\tau, \sigma)$*; if* α *is a subterm of* τ *bound by the subterm* $\mathsf{rec}\ \alpha.\sigma$ *then* $\textsc{proper}(\tau, \alpha) = \textsc{proper}(\tau, \mathsf{rec}\ \alpha.\sigma)$*.*

$\textsc{proper}(\tau, \sigma)$ is uniquely defined for all subterms σ of τ, because recursive quantifiers are required to be syntactically contractive.

Armed with these constructs, the automaton of a term is easily defined.

Definition 28 *If* η *is distinguishing then*

$$\textsc{automatonof}_{\eta}(\tau) = (\textsc{proper}(\tau), \textsc{proper}(\tau, \tau), \delta, sl)$$

where δ *and* sl *are as follows. If* $\sigma = nl(\ell = \overrightarrow{\alpha_\ell}\ .\sigma_\ell)$ *then* $\delta(\sigma, \ell) = \textsc{proper}(\tau, \sigma_\ell)$ *and* $sl(\sigma) = \mathsf{nl}(nl)$*. If* σ *is variable subterm of* τ *that is free in* τ *then* $sl(\sigma) = \mathsf{bvar}(n)$ *where* $\eta(\sigma) = \textsc{var}(n)$*. If* σ *is a variable subterm of* τ *that is bound by the nth binder of edge* ℓ *of* σ' *subterm of* τ *then* $sl(\sigma) = \mathsf{fvar}(\sigma', \ell, n)$*.*

The automaton of a term is defined so that its tree is the same as the term's tree.

Theorem 29 *If η is distinguishing then* $\textsc{tree}(\textsc{automatonof}_\eta(\tau)) = [\![\tau]\!]_\eta$.

Finally, all the previous results can be combined into a decision procedure for equality of terms. Since the algorithm for equality of automata is quadratic and the conversion from terms to automata produces a linear output in linear time, the decision procedure below is quadratic.

Theorem 30 *If η is distinguishing then:*

$$\vdash \tau_1 = \tau_2 \quad \Leftrightarrow \quad L(\textsc{equal}(\textsc{automatonof}_\eta(\tau_1), \textsc{automatonof}_\eta(\tau_2)) = \emptyset$$

Proof:

$$\begin{array}{ll} & \vdash \tau_1 = \tau_2 \\ \Leftrightarrow & \langle\text{Theorem 14 and Theorem 18}\rangle \\ & [\![\tau_1]\!]_\eta = [\![\tau_2]\!]_\eta \\ \Leftrightarrow & \langle\text{Theorem 29}\rangle \\ & \textsc{tree}(\textsc{automatonof}_\eta(\tau_1)) = \textsc{tree}(\textsc{automatonof}_\eta(\tau_2)) \\ \Leftrightarrow & \langle\text{Theorem 25}\rangle \\ & L(\textsc{equal}(\textsc{automatonof}_\eta(\tau_1), \textsc{automatonof}_\eta(\tau_2))) = \emptyset \end{array}$$

□

7 Summary and Future Work

This paper has shown how to give a natural interpretation to a second-order term system with recursive quantifiers. In particular it extends the well known tree interpretation, introduced by Amadio and Cardelli, to second-order constructs by defining a theory of trees with binding. It gives an appropriate generalisation of regularity to binding trees, and shows that regular trees characterise both those generated by terms and by automata. It shows the usual set of equality rules are sound and complete in the second-order case. It generalises Kozen et al.'s term automata to the second-order case, providing a polynomial time decision procedure for equality of terms. The result is a natural and intuitive theory of second-order type systems with equirecursive types.

The obvious next step is to add subtyping to the theory. The main idea is to define subtyping on trees coinductively. Then it should be possible to show that a certain set of rules is sound and complete with respect to this definition. Interestingly, the rules I believe are sound and complete are a nonconservative extension of rules for $\mathcal{F}_\leq$ with the Kernel-Fun rule for bounded quantification (c.f. [Pie94] and [CG99]). Finally, it should be possible to extend the construction for equality of automata's trees to subtyping in a way that combines my ideas for second-order constructs with Kozen et al.'s ideas for subtyping and a simple idea for dealing with bounded variables. The result should be a polynomial time algorithm for deciding subtyping in a system with Kernel-Fun recursively-bounded quantifiers and equirecursive types.

Another extension of the ideas is to higher-order kinds. Languages like ML and Haskell allow the definition of type constructors, which could be thought of as type variables with second-order kinds. Thus at a minimum, it would be good to include this in the theory if not a larger system with a fuller set of kinds. Full $\mathcal{F}_\omega$ (c.f. [Gir71] and [Gir72]) with equirecursive types is likely to be undecidable since it contains the simply-typed lambda calculus with recursive functions at the type level, for which equality is at least as hard as the halting problem. But, it might be possible to restrict $\mathcal{F}_\omega$ with equirecursive types to a decidable system, perhaps by allowing only syntactically-contractive recursive types.

References

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Progamming Languages and Systems*, 15(4):575–631, September 1993.

[Bru72] N. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.

[CG99] Dario Colazzo and Giorgio Ghelli. Subtyping recursive types in kernel fun. In *1999 Symposium on Logic in Computer Science*, pages 137–146, Trento, Italy, July 1999.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Gle02] Neal Glew. A theory of second-order trees. Technical Report TR2001-1859, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, January 2002.

[Kap77] Irving Kaplansky. *Set Theory and Metric Spaces*. Chelsea Pub Co, 2nd edition, June 1977.

[KPS95] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, March 1995.

[Pie94] Benjamin Pierce. Bounded quantification is undecidable. *Information and Computation*, 112:131–165, 1994.