Higher-order Recursive Types

Working Notes, 21st March 2002

Gert Smolka

We are interested in a lambda calculus whose expressions describe types that are possibly higher-order and recursive. Types are modelled as trees, which can be seen as possibly infinite, normal lambda expressions in de Bruijn's notation.

In general, equivalence of type expressions is undecidable. We are interested in subclasses for which equivalence is efficiently decidable. We define a subclass whose equivalence problem can be solved in $O(n \log n)$ time.

1 Trees and Graphs

As usual, we define a *tree* as a function from a tree domain (possibly infinite) to a set of so-called labels. We consider nonempty, ordered trees whose domains are prefix-closed subsets of $\mathbb{N}^*$. A tree is called *regular* if it has only finitely many distinct subtrees.

Trees can be described by *graphs* whose edges are labeled with natural numbers. Each node of such a graph describes exactly one tree. The graph itself describes a set of trees that is closed under taking subtrees.

A graph together with one of its nodes can be seen as a kind of automaton, which accepts the elements of the tree the node (acting as initial state) describes (up to details concerning the final states).

The trees described by a finite graph are all regular. For every regular tree t there is a finite graph whose nodes describe exactly the subtrees of t.

A graph is *minimal* if different nodes of the graph describe different trees. Minimal graphs are unique up to renaming of nodes; that is, for a finite set of regular trees there is one and only one minimal graph representing it (up to renaming of nodes). A finite graph can be minimized in $O(n \log n)$ time by an algorithm known as graph partioning (originally devised by Hopcroft (1971) for minimizing automata).

2 Rho Calculus

Regular trees can be described by so-called *μ-expressions* (e.g., $\mu x.f(a, x)$). We define the *rho calculus* as the system that, given a set of labels $f \in F$ and a set of variables $x \in X$, consists of the expressions

$$M \ = \ fM_1 \dots M_n \mid x \mid \mu x.M \quad (M \text{ not a variable})$$

and the reduction rule

$$(\mu) \quad \mu x.M \ \rightarrow \ M[\mu x.M \,/\, x]$$

We need to define which tree is denoted by an expression (given an environment E mapping its free variables to trees). This can be done by first defining inductively an *acceptance relation*

$$(\pi, u), E \vdash M$$

and then defining the *denotation function* as

$$[\![M]\!]E = \{(\pi, u) \mid (\pi, u), E \vdash M\}$$

The definition of the acceptance relations involves μ-reductions that are applied when needed:

$$\frac{(\pi, u), E \vdash M[\mu x.M \,/\, x]}{(\pi, u), E \vdash \mu x.M}$$

The equation corresponding to the μ-rule is not a complete characterisation of the μ-operator. For instance, it fails to establish the equivalence of the expressions $\mu x.f(gx)$ and $f(\mu x.g(fx))$.

There is the minor complication that certain μ-expressions do not describe trees, for instance, $\mu x.x$ and $f(\mu x.x)$. This degenerate μ-expressions can be excluded by a straightforward syntactic condition, which requires that the M in $\mu x.M$ is not a variable.[1]

The natural computer representation for regular trees are graphs. Expressions are used as a convenient representation for input and output. In contrast to trees, graphs can

[1] Degenerate expressions could be taken as descriptions of the empty tree. However, excluding the empty tree and degenerate expressions seems to save a lot of complications.

express structure sharing. Minimal graphs are graphs with maximal structure sharing. Expressions with a let construct can also express structure sharing.

Chapter 21 of Benjamin Pierces's book [2002] studies a variant of the rho calculus.

Problem 1 *Define the acceptance relation for the rho calculus.*

Problem 2 *Show that the equivalence problem of the rho calculus can be decided in $O(n \log n)$ time by the graph partitioning algorithm mentioned in the previous section.*

Problem 3 *Define a denotational semantics for the rho calculus (i.e., a denotation function defined by structural induction on expressions).*

3 Lambda Mu Calculus

The canonical framework for the description of higher-order recursive objects is the simply typed lambda calculus with fixed point combinators:

$$\begin{array}{l} \mathit{fix}_\sigma \colon (\sigma \to \sigma) \to \sigma \\ (\mathit{fix}) \quad \mathit{fix}_\sigma = \lambda f \colon \sigma \to \sigma.\ f(\mathit{fix}_\sigma f) \end{array}$$

We assume the usual axioms (α), (β) and (η).

The μ-operator of the rho calculus can be expressed with fixed point combinators:

$$\mu x \colon \sigma.M \stackrel{\text{def}}{=} \mathit{fix}_\sigma(\lambda x \colon \sigma.M)$$

Moreover, the μ-rule from the rho calculus

$$(\mu) \quad \mu x \colon \sigma.M \ \to\ M[\mu x \colon \sigma.M \,/\, x]$$

can be simulated by one application of the fix-rule and two applications of the β-rule. On the other hand, the fixed point combinators can be expressed with the μ-operator:

$$\mathit{fix}_\sigma = \lambda f \colon \sigma \to \sigma.\ f(\mu x \colon \sigma.fx)$$

Moreover, from the equations corresponding to the μ-rule we can derive the axioms for the fixed point combinators. We conclude that the simply typed lambda fix calculus has the same expressivity and the same axiomatic equivalence as the simply typed *lambda mu calculus*, where the fixed point operators are replaced by the μ-operator. We prefer to work with the lambda mu calculus since it facilitates the formulation of constrained versions of recursion.

It is well-known that reduction in the simply typed lambda fix calculus is confluent and that *lambda reduction* (i.e., reduction with the β- and η-rule) is terminating (see, e.g., [Mitchell]). Hence equivalence of expressions not containing a fixed point operator is decidable. In general, equivalence of expressions in the lambda fix calculus is undecidable.

It is easy to see that the termination result for lambda reduction carries over to the lambda mu calculus. We also expect that the lambda mu calculus is confluent, but the proof of this result is not obvious since reduction in the lambda fix calculus cannot be fully simulated in the lambda mu calculus. Ariola and Klop [LICS 1994, Section 8] consider an untyped lambda mu calculus *without the η-rule*, which is confluent since it is an orthogonal, combinatory reduction system.

Problem 4 *Show that the lambda mu calculus is confluent.*

A head normal form is an expression that has the property that no reduction rule applies at the top level, even after preparatory reduction steps at lower levels. The set of *head normal forms* is defined inductively as follows:

1. $M_0 \dots M_n$ is a head normal form if M_0 is a constant or a variable.
2. $\lambda x.M_0 \dots M_n$ is a head normal form if M_0 is a constant or a variable and $M_n \neq x$.
3. $\lambda x.\lambda y.M$ is a head normal form if $\lambda y.M$ is a head normal form.

We say that M' is a *head normal form of M* if M' is a head normal form and $M \rightarrow^* M'$.

Problem 5 *Suppose M and M' are head normal forms of the same expression. Then either M and M' are both lambda abstractions, or there exist $n \in \mathbb{N}$ and expressions F, $M_1, \dots, M_n$ and $M'_1, \dots, M'_n$ such that $M = FM_1 \dots M_n$, $M' = FM'_1 \dots M'_n$, and F is a constant or a variable.*

Problem 6 *Show that the existence of head normal forms is undecidable.*

4 Tau Calculus and Lambda Trees

The *tau calculus* is the specialized simply typed lambda mu calculus that has only one base type $*$. For the expressions of the tau calculus we define an equivalence that is weaker than the canonical equivalence (defined by the axioms).[2] The need for weak equivalence was already motivated in the context of the rho calculus; for instance, we would like that the expressions $\mu x.f(gx)$ and $f(\mu x.g(fx))$, which are not equivalent,

[2] Weaker means, that expression that are equivalent are also weakly equivalent, and that weakly equivalent expressions are not necessarily equivalent.

be weakly equivalent. For μ-free expressions, *weak equivalence* should coincide with canonical equivalence.

We will define weak equivalence for the tau calculus in a similar way as it was done for the rho calculus. This time we take so-called *lambda trees* as denotations of expressions.

Finite lambda trees correspond to closed and normal expressions, where variable bindings are represented with de Bruijn's notation.[3] For instance, the expression

$$\lambda x.\lambda y.\lambda z.(xz)(yz)$$

corresponds to the lambda tree

$$\lambda(\lambda(\lambda(2@0)(1@0)))$$

where the labels λ and @ (written in infix notation) represent lambda abstraction and application. Nodes labeled with de Bruijn indices (i.e., natural numbers) are called *alpha nodes*. Constants appear as leaves of lambda trees.

Infinite lambda trees can be seen as infinite, closed, and normal expressions.

If an expression is in head normal form, its top level corresponds exactly to the top level of the denoted lambda tree.

A *convergence property* is a set S of expressions as follows:

1. S is closed under reduction.
2. S is closed under taking subexpressions.
3. Every expression in S has a head normal form.

The union of convergence properties is again a convergence property. Hence there exists a largest convergence property. An expression is called *convergent* if and only if it is an element of the largest convergence property.

Problem 7 *Show that convergence is undecidable.*

Nonconvergent expressions in the tau calculus correspond to degenerate expressions in the rho calculus. We will define weak equivalence only for convergent expressions.

Intuitively, convergent expressions have possibly infinite normal forms. We express this intuition by defining an acceptance relation

$$(\pi, u), E \vdash M$$

that relates lambda trees and convergent expressions.

[3] Note that a normal expression cannot contain the μ-operator.

Problem 8 *Define the acceptance relation for convergent expressions.*

5 Higher-order and Recursive Types

We model higher-order recursive types as possibly infinite lambda trees, and we use the tau calculus for describing them.[4] To avoid confusion, the simple types of the tau calculus are called *kinds*. There is only one base kind $*$. Types that can be described by closed expressions of kind $*$ are called *proper types*. Types that can be described by closed expressions of a functional kind are called *higher-order types*.

An expression is called *lambda normal* if it cannot be reduced with the β- or η-rule. An expression is called *weakly lambda normal* if it is lambda normal and lambda normality is preserved by iterated μ-reduction.

Consider the following higher-order type expressions:

$$\begin{aligned} A &= \lambda x : *.\ \mu y : *.\ 1 + (x \times y) \\ B &= \mu f : * \to *.\ \lambda x : *.\ 1 + (x \times f x) \end{aligned}$$

Both expressions describe the same higher-order type (a function mapping a type x to the type of lists over x). We consider the expressions $A1$ and $B1$, which describe the proper type of list over 1. A single β-reduction

$$\begin{aligned} A1 &\overset{\beta}{\to} \mu y : *.\ 1 + (1 \times y) \\ &\overset{\mu}{\to} 1 + (1 \times (\mu y : *.\ 1 + (1 \times y))) \\ &\overset{\beta}{\leftarrow} 1 + (1 \times A1) \end{aligned}$$

yields a weakly lambda normal expression that describes the proper type described by $A1$. The situation is more complicated for $B1$. We need a μ-reduction followed by a β-reduction

$$\begin{aligned} B1 &\overset{\mu}{\to} (\lambda x : *.\ 1 + (x \times Bx))\ 1 \\ &\overset{\beta}{\to} 1 + (1 \times B1) \end{aligned}$$

[4] We follow the so-called *equi-recursive approach* to recursive types, which treats recursive types exactly as one would expect from the statement that a recursive type is an infinite tree. The rho calculus can be used to describe regular recursive types. In contrast, the *iso-recursive approach* to recursive types is a formal trick that takes a recursive type and its unfolding as different (see Mitchell's book, for instance). In the iso-recursive approach the two type expressions $\mu x.1 + x$ and $1 + (\mu x.1 + x)$ describe different types; in the equi-recursive approach they describe the same type.

to unravel the top level of the described type. The lambda-abstraction is not removed, and we need infinitely many β-reductions to unravel the described type completely (which is a regular tree).

The tau calculus can describe nonregular types. For instance:

$$\begin{aligned} C &= \mu f : * \to *.\ \lambda x : *.\ 1 + (x \times f(x + x)) \\ C1 &= 1 + (1 \times C(1 + 1)) \\ &= 1 + (1 \times (1 + ((1 + 1) \times C((1 + 1) + (1 + 1))))) \end{aligned}$$

In languages with polymorphic recursion (not SML) there are useful applications of nonregular types (see Okasaki's book on functional data structures).

6 Decidability of Type Expression Equivalence

We are interested in formal systems where well-typedness is a decidable property. For this to be the case, equivalence of expressions describing types must be decidable.

Marvin Solomon [POPL 1978] has shown that equivalence is decidable for type expressions with first-order recursion (i.e., μ_σ occurs only with kinds $\sigma = * \to \cdots \to *$) and parameters of the base kind $*$. Solomon shows that the problem is equivalent to the deterministic push-down automata equivalence problem (which at the time was still open but has now been settled as decidable [Sénizergues, ICALP 1997]).

It seems that equivalence of expressions that use recursion only for the base kind (i.e., contain μ_σ only with kind $\sigma = *$) is decidable.

We are interested in a subclass of expressions for which equivalence is decidable. The first restriction, called *BR (basic recursion)*, requires that all μ-variables have kind $*$. BR yields that μ-reduction preserves β-normality (normality with respect to β-reduction).

It seems that equivalence of expressions restricted to BR is decidable.

We can now λ-normalize an expression as follows: First, we normalize with respect to β and η. Then we apply μ-reduction to a bottom-most subexpression

$$\lambda x.\ \mu y.\ M \qquad \text{where } x \text{ free in } M$$

and try to apply η-reduction. After finitely many steps we reach a λ-normal expression, for which μ-reduction preserves λ-normality.

7 Definition of Lambda Trees

Lambda trees are trees with variable bindings. Finite lambda trees correspond exactly to lambda terms in de Bruijn's Notation that are normal with respect to β- and η-reduction. We are interested in infinite lambda trees since they may serve as a semantics for recursive and higher-order types.

We assume a set X and two distinct objects @ and λ such that X, $\{@, \lambda\}$ and the set of positive integers are pairwise disjoint.

A *lambda tree* over X is a tree as follows:

1. Each node is labeled with one of the following: an element of X, a positive integer, @ or λ. Nodes labeled with @ are called *application nodes*, nodes labeled with λ are called λ-*nodes*, and nodes labeled with a positive integer are called α-*nodes*. Alpha nodes represent bound or unbound variables.
2. Nodes labeled with elements of X or positive integers must be leaves.
3. Application nodes must have exactly two and lambda nodes must have exactly one successor.
4. There are neither β- nor η-redexes.
5. There is a fixed $k \in \mathbb{N}$ such that every α-node labeled with n is below of at least $n - k$ different λ-nodes.

Regular trees always satisfy condition (5).

It is easy to see that subtrees of lambda trees are lambda trees.

An α-node labeled with n is bound by the n'th λ-node that is above it. This minor derivation from the de Bruijn Notation is convenient for the following definition.

The *degree* of a lambda tree is the least k such that condition (5) is satisfied. A lambda tree of degree 0 contains no unbound variable and is called *closed*. A lambda tree of positive degree contains unbound variables and is called *open*.