A system for incorporating higher order functions in Equational Programs

Srinivas Padmanabhuni,
Department of Computing Science,
University Of Alberta, Canada T6G 2H1.
e-mail: srinivas@cs.ualberta.ca

Abstract

Higher order functions, with the ability to manipulate functions as first-class objects, increase the expressive power of a programming language. Equational programs provide a convenient paradigm to utilize the rewriting technique of functional languages in combination with the unification technique of logic programming. Hence the equational programming paradigm can be enhanced considerably by incorporating higher order functions.

In this paper we present a model with enhanced equation-solving capabilities, for incorporating higher order functions in an equational programming framework. We are able to achieve higher order features by extending first order procedures. We show that this model preserves the properties of an equational program in going from first order to higher order. We also show how this extension captures the dual notions of higher order rewriting in the functional sense and higher order logic programming in the sense of higher order equation solving capabilities.

Keywords: Equational Programming, Term Rewriting System, Automated Reasoning.

1 Motivation & Introduction

Equational programming [Gougen 86][O'Donnell 85] refers to the use of equations as a programming language. This paradigm involves a combination of *functional programming* and *logic programming* features. In functional programming, equations are used by repeatedly substituting equal terms in a given formula till the normal (simplest) form is obtained. In logic programming, we write a program as a set of implications between formulas in logic. Goals are executed by a theorem prover that derives the consequences of the given statements until the desired output is obtained by unification.

Equational programming combines both the aforementioned schemes into a single paradigm. Conventionally,for computation, equational programs are converted to term rewriting systems, which are sets of *oriented* equations. Knuth and Bendix[Knuth 70] characterized

certain properties of a term rewriting system to be able to exactly represent a equational program.

In this paper we present a model with enhanced equation-solving capabilities, for incorporating higher order functions in an equational programming framework. It has been observed that incorporation of higher order features in equational programming framework requires combining of λ-calculus [Barendregt 84] and a first order equational program[Dougherty 91] [Tannen et al 89]. We borrow the notion of *currying* from λ-calculus and propose a system , which is essentially a first order system with notion of *currying* embedded in it.

Our work can be seen as a first step towards the implementation of a fully higher order equational programming system.

A preliminary version of this paper appears in [Srinivas 94].

2 Rewrite systems for equational programs

2.1 Terminology

Let T(F,X) be the set of first order terms, where F is the set of function symbols and X is the set of variables. With every f $\in F$ we associate a unique number of arguments (*arity*). Any variable is a first order term and if f is an *n-ary* function symbol $\in F$, and $t_1, \ldots, t_n$ are first order terms, then $f(t_1, t_2, \ldots, t_n)$is a first order term. The subterm of s at position p is denoted as $s\mid_p$.

A *substitution* is a mapping from the set of variables to the set of terms which is equal to identity almost everywhere. The *composition* of two substitutions σ and θ,denoted by $\sigma \circ \theta$ or simply $\sigma\theta$, is a composition of the two functions. Thus if $x\sigma = s$ for some $x \in X$ then $x\sigma\theta = s\theta$. We say that a substitution σ is at least as general as a substitution ρ and write $\sigma \leq \rho$ if there is substitution τ such that $\sigma\tau = \rho$. A term s *matches* a term t if t=sσ for some substitution σ. We also say that t is an instance of s in this case. A term s *unifies* with a term t if $s\sigma = t\sigma$ for some substitution σ. A substitution σ is called the *most general unifier* (mgu) of two terms s and t if for any unifier θ of s and t, there exists a substitution τ such that $\theta = \tau \circ \sigma$.

In this paper, a *higher order function* refers to any function which takes functions as input and yields functions as output.

2.2 Equations and rewrite systems

An *equational theory* is a set of equations. An *equation* is an unordered pair of first order terms written as s = t. Given a set of equations E we define a *replacement* step ($\leftrightarrow_E$) between two terms s and t, denoted $s \leftrightarrow_E t$,if l=r $\in E, s\mid_p = l\sigma$ and t=s$[l\sigma \leftarrow r\sigma]_p$(i.e t= s with the subterm at p replaced by $r\sigma$), for some substitution σ. Thus for E we say $s = t$ iff $s \leftrightarrow_E^* t$,where $\leftrightarrow_E^*$ is the reflexive-transitive closure of $\leftrightarrow_E$.

The following two problems are fundamental in equational theories:

a). *Validity Problem*: Given E, and an equation s =t , is $s \leftrightarrow_E^* t$?

b).*Satisfiability Problem*: Given E, an equation s = t, find a substitution σ such that sσ $\leftrightarrow_E^*$ tσ.

A *rewrite rule* over a set of terms T, is an ordered pair (l,r) of terms written l $\longrightarrow$ r. A set of rewrite rules is called a *term rewriting system*,denoted by $\mathcal{R}$. Here *replacement* is allowed only from left to right. A term s *rewrites* to another term t in one step, denoted s $\rightarrow$ t, if for some rule $l \rightarrow r$ in $\mathcal{R}$, position p in s and substitution σ such that $l\sigma = s\mid_p$ and $t = s[l\sigma \leftarrow r\sigma]_p$. The reflexive-transitive closure of $\rightarrow$ is denoted by $\rightarrow^*$,and in such a case t is said to be *derivable* from s. We write $s \downarrow t$ if s and t *join*,i.e. $s \rightarrow^* w$ and $t \rightarrow^* w$ for some term w. A term s is said to be *irreducible* or in *normal form* if there is no term t such that $s \rightarrow t$.

A rewrite relation ($\rightarrow$) is *terminating* if there exists no infinite chain of rewrites of the form $t_1 \rightarrow \ldots \rightarrow t_k \ldots$ A rewrite relation is *(ground) confluent* if whenever two (ground) terms s and t are derivable from a term u, then there exists a term v derivable from both s and t A rewrite system which is both *terminating* and *(ground) confluent* is said to be *(ground) convergent.*

A rewrite system R is *sound* with respect to a set of equations E , if the derivability relation $\rightarrow^*$ of R is a subset of the replacement relation $\leftrightarrow_E^*$ of E. A system R is *complete* for E,if for any two terms that are provably equal in E are *joinable* in R.

The validity problem in an equational theory E is equivalent to the checking of the equality of normal forms of the two terms,in the corresponding convergent rewrite system R provided R is finite, sound and complete for E.This is possible because in a convergent rewrite system T every term t has exactly one normal form. This means that the terms s and t are joinable by a convergent system R iff they have the same normal form. Knuth and Bendix [Knuth 70] provided a convenient procedure for obtaining a convergent rewrite system from an equational theory based on the idea of *critical pairs.*

Let $l \rightarrow r$ and $g \rightarrow d$ be two rules (or two versions of the same rule, with variables renamed),in R. The equation s=t is said to be a *critical pair* between these two rules, if g unifies with a non-variable subterm of l at position π using a substitution σ and $s = r\sigma$ and $t = l[d]_\pi\sigma$. We say that $l\sigma$ is a critical *overlap*, and we have $r\sigma \leftarrow l\sigma \rightarrow l[d]_\pi\sigma$. That is s and t are the two terms obtained by rewriting the *overlap* between the two rules. For example, 0 + (u + v) = u + v is a (joinable) critical pair between the rules $0 + x \rightarrow x$ and $(y + u) + v \rightarrow y + (u + v)$. The overlap between these rules is $(0 + u) + v$; it is obtained by unifying a left hand side 0 + x with a non-variable subterm y + u of the other left hand side. A critical pair $s = t$ in R is said to be *joinable*, if $s \downarrow t$ in R. Critical pairs are useful for checking if a terminating system is confluent and hence convergent.

Lemma 2.1 (Critical Pair Lemma) *[Knuth 70] A terminating rewrite system is convergent iff all its critical pairs are joinable.*

The Critical Pair Lemma gives a good method of generating a convergent rewrite system equivalent to a set of equations. This method is called *Completion.* The method proceeds

by orienting equations into terminating rules, using a well-founded ordering, and generating new equations by superposing left-hand sides until all critical pairs are joinable.

3 A new model for higher order functions in equational programs

We present here a simple model using only first order equational programs, which captures higher order functions in the functional sense and allows us to solve for functions. The idea is called *currying* in λ-calculus.

3.1 Term structure

In this model , a term belonging to T(F,X) is defined by the following inductive rules:

[1]. Any constant C, first order or higher order is a term.

[2]. Any variable Z, first order or higher order is a term.

[3]. An application app(t_1, t_2), where t_1 and t_2 are terms, is a term.

This is the same as the original definition of first order term except that only one binary function symbol (app) is used here. A rewrite system is a collection of *oriented* equations of terms in this model.

Our set of terms forms a superset of the set of first order terms and includes higher order terms. The fundamental process involved in the translation of a first order system to this system is called *currying* (It is frequently quoted in functional programming literature). The translation is carried out by representing any n-ary function in terms of the only function symbol in this system, i.e. **app**. Any first order term $f(t_1 \ldots . t_n)$ is represented as $app(app(....app(f, t_1), t_2)..., t_n)$ in the model .

This is illustrated with some examples below:

Example 1: Definition Of Apply_n(Applying a function n times $f^n(x)$).

Rule 1: app(app(app(apply_n,0),f),x) $\longrightarrow$ x
It represents: apply_n(0,f,x) $\rightarrow$ x
Rule 2: app(app(app(apply_n,app(s,n)),f),x) $\longrightarrow$
app(f,app(app(app(apply_n,n),f),x)).
It represents: apply_n(s(n),f,x) $\rightarrow$ f(apply_n(n,f,x))

Example 2 : Definition Of MapCar:

Rule 1: app(app(mapcar,f),nil) $\longrightarrow$ nil.
It represents: mapcar(f,nil) $\rightarrow$ nil
Rule 2: app(app(mapcar,f),app(app(cons,x),tl)) $\longrightarrow$
app(app(cons,app(f,x)),app(app(mapcar,f),tl)).
It represents: mapcar(f,cons(x,tl)) $\rightarrow$ cons(f(x), mapcar(f,tl))

Both the aforementioned functions are higher order in the sense that they take functions as input. It is easy to prove the termination of both the above rewrite systems. Further since

there are no critical-pairs between the rules in each system, both the systems are confluent. The rewrite systems representing the two functions are thus both convergent.

It is clear from the above examples that because we have only one function symbol **app**, any function constant is treated as a first order constant. Thus the model allows for the use of *function variables*, which give us the additional representation power required for manipulating higher order functions.

3.2 Preservation of properties in the model

We see that both higher order and first order terms are representable in this structure as no distinction is being made between first order constants(variables) and higher order constants (variables). Any first order term $f(t_1 \ldots . t_n)$ can be represented as $app(app(....app(f,$ $t_1), t_2)..., t_n)$ in the model. The converse is also true for any first order function f. Thus it is very easy to prove the following two theorems relating the model to first order terms. Let the translation of a first order rewrite system R into the model be represented as TRANS(R).

Theorem 3.1 *R is confluent iff TRANS(R) is confluent.*

• $\Rightarrow$ This follows trivially from the previous lemma by induction on the number of rewrite steps.

$\Leftarrow$ To Prove this let us consider a non-confluent R. Here we can derive two normal-forms s and t from the same starting term u in R. Starting from TRANS(u), we can get normal-forms TRANS(s) and TRANS(t) in TRANS(R), by induction on the number of rewrite steps. Hence TRANS(R) is non-confluent too. Thus $\Leftarrow$ is also proved.

Theorem 3.2 *R is terminating iff TRANS(R) is terminating.*

• $\Leftarrow$ is trivial. Because if there is nonterminating derivation in R, a corresponding derivation in TRANS(R) can be found.

$\Rightarrow$ If TRANS(R) is nonterminating, then there is an infinite derivation here. By induction on the number of rewrite steps and using TRANS^{-1} , we get a nonterminating derivation in R.

3.3 Capturing higher order functional programming

The system provides a mechanism to do higher order rewriting in the functional sense. This is illustrated with the following example. The *Mapcar* function defined in the previous section is being used here.

Consider the starting expression: mapcar((+ s(0)),cons(0,cons(s(0),nil))). The rewriting proceeds by application of Rule 2 of Mapcar giving cons((+ s(0) 0),mapcar((+ s(0)),cons(s(0), nil))) followed by Rule2 of Mapcar again to finally yield cons(s(0),cons(s(s(0)),nil)) as the normal-form.

Thus we have been able to incorporate higher order functional features in a first order system by using *currying*.

3.4 Solving for functions

In the first order domain, one is limited to solving for the first order equational goals(e.g. on solving plus(x,0) =?= s(0), we would get x=s(0)). But in the model presented here, even for the first order subset, we are able to solve for higher order equational goals, e.g. if we are having plus and product in the pure first order subset we can still answer queries like app(x,0) =?= s(0) ,to get the answer x = (plus 1),as the answer. In addition the method in the model easily extends to the higher order case which involves solving for higher order goals in higher order rewrite systems.

The existing procedures of narrowing [Siva 89] and top-down-decomposition methods [Mitra 90] [Martelli 86] for equation solving can be extended to this model. We shall illustrate how narrowing can be used to solve for higher order functions in this model. The mechanism of narrowing[Siva 89] is given by the following transformation rules:

Reflect: $\sigma = mgu(s,t) : \{s =^? t\} \cup G \Rightarrow G\sigma$

Narrow :$\mu = mgu(l, s\mid_p) :$
$\{s =^? t\} \Rightarrow \{s[u \leftarrow r]_p\mu =^? t\mu\} \cup G\mu$

Let R = $\{app(app(+,0),x) \rightarrow x; app(app(+,app(s,x)),y) \rightarrow app(s,app(app(+,x),y))\}$. Here, if we ask the query app(f,0) =?= 0 by narrowing the left-hand-side converts to 0 by application of first rule with the unification $\{f \mapsto app(+,0)\}$. Thus by narrowing we get $\{f \mapsto app(+,0)\}$, as the answer for the query.

The top-down decomposition method[Martelli 86] of solving equations is given by the following transformation rules:

Transformation Rules for Top Down Decomposition
Decompose : $\{f(s_1,\ldots,s_n) =? f(t_1,\ldots,l_n)\} \cup D \Rightarrow \{s_1 =? t_1, ..., s_n =? t_n\} \cup G$
Restructure : $f(l_1,\ldots,l_n) \rightarrow r \in R :$ $\{f(s_1,\ldots s_n) =? t\} \cup G \Rightarrow \{s_1 =? l_1,\ldots, s_n =? l_n, r =? t\} \cup G$
Bind : $\sigma = mgu(X,t) : \{X =? t\} \cup G \Rightarrow G\sigma$
Expand: $occurs(X,t) \& t\mid_p = f(t_1,\ldots,t_n) \& f(l_1,\ldots,l_n) \rightarrow r \in R :$ $\{X =? t\} \cup G \Rightarrow \{X =? t[u \leftarrow r]_p, l_1 =? t_1, l_2 =? t_2, \ldots l_n =? t_n\} \cup G$

The top-down decomposition method can be applied in this model to solve for both first order terms and higher order terms. Consider the higher order query app(f, 0) =?= 0 , the top-down decomposition method yields the following steps for R = $\{app(app(+,0),x) \rightarrow x; app(app(+,app(s,x)),y) \rightarrow app(s,app(app(+,x),y))\}$:

app(f, 0) =?= 0 $\rightarrow_{Restr}$ $\{f =? = app(+,0), x =? = 0, 0 =? = x\}$

$\rightarrow_{Bind}$ $\{0 =? = 0\}, \sigma = \{f \mapsto app(s,0)\}$

$\rightarrow_{Decom}$ $\phi, \sigma = \{f \mapsto app(+,0)\}$

hence yielding $\{f \mapsto app(+,0)\}$ ad the result.

Thus we see that existing equation solving procedures of first order equations can be used to solve for higher order functions in this model. We have implemented a minor variation of the top-down decomposition method for this model, which is more efficient than narrowing. We can prove the correctness and completeness of the method borrowing results from [Mitra 90]. We find that strong typing of the terms (where any term in the language has a fixed type) increases the efficiency of the search because pruning of the unwarranted paths becomes easier. The method with strong typing of terms is sound. In the end, we relaxed the typing restrictions , by using type-inferencing on the terms of the equation and examined the search process. This was also found to be efficient.

4 Conclusions

In this paper we present a way to incorporate higher order functions in an equational program setting. The central concept involves extending the first order equational programs with the notion of *currying* - a mechanism to represent n-ary functions with a single unary function **app**, where functions are treated as first order terms. This phenomenon of abstraction of functions enables us to capture higher order rewriting in the sense of functional programming and higher order logic programming in the sense of equation solving.

The soundness and completeness of the method with embedded type-inferencing is currently under investigation.

References

[Barendregt 84] Barendregt H.P., "The Lambda-Calculus - Its syntax and semantics", 2nd ed., 1984, North Holland.

[Tannen et al 89] Breazu-Tannen V. and Gallier J., "Polymorphic rewriting conserves algebraic strong normalization and confluence", In Proc. ICALP 89, pp. 137-150. , LNCS 372.

[O'Donnell 85] O'Donnell M.J., "Equational Logic as a Programming Language", 1985, MIT Press.

[Dougherty 91] Dougherty D.J.,"Adding algebraic rewriting to the untyped λ-calculus, in R.V.Book, ed., 4th RTA 91, LNCS 488, pp. 37-48.

[Gougen 86] Gougen J.A. and Meseguer J., " EQLOG: Equality, types and generic modules for logic programming", In Logic Programming: Functions,relations and

equations,Prentice-Hall, Engelwood-cliffs,NJ, pp. 295,1986.

[Knuth 70] Knuth,D.E., and Bendix,P.B. "Simple word problems in universal algebras,", In: Computational Problems in Abstract Algebra, J.Leech, ed. Pergamon Press, Oxford, U.K., 1970, pp. 263-297.

[Martelli 86] Martelli,A., Moiso,C. and Rossi,G.F. "An algorithm for unification in Equational theories", Proceedings of the Third IEEE Symposium on Logic Programming, Salt Lake City, UT(September 86), pp. 180-186.

[Mitra 90] Mitra Subrata, " Top-down equation solving and extension to AC-theories", M.S. thesis, Univ Of Delaware, Dec 1990.

[Siva 89] Sivakumar G., "Proofs and computation in Conditional Equational theories", Ph.D. dissertation,UIUC, May 1989.

[Srinivas 94] Padmanabhuni S.,"Higher Order functions in Equational Programs", Proceedings of the Second ARD Workshop , Birbie Island, Queensland, Australia (September 1994).