Pattern Recognition in FP Programs

John S. Givler
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, New York 11794

March, 1983

OGC Technical Report No. CS/E83-003

Pattern Recognition in FP Programs

CHAPTER 1

Introduction

The primary purpose of a programming language is to provide a vehicle for human communication. It is judged on the ease with which algorithms are expressed, read, and reasoned about. Since many of these languages are also used to control computing machinery, the efficiency with which the (compiled) algorithms drive such machinery is of some interest.

Historically, the computing community has focussed most of its attention on imperative languages, (such as FORTRAN, Pascal, & Ada) which encourage obscure solutions in the interests of conserving machine resources. More recently, attention has shifted to applicative languages, (such as FP, 'pure' LISP, SASL, & graph-reduction languages) because of their superior mathematical basis.

It is an unfortunate fact of life that we must attack arbitrarily large problems with minds of distinctly limited ability. In this endeavor, the "divide-and-conquer" strategy has proven to be our sharpest weapon. Obviously, to effectively "divide" a problem, it is necessary to minimize the amount of interaction amongst the parts. Each part must be understood, ***in isolation***, in terms of its own input/output behavior relative to the input/output behavior of its primitive functions. Thus, a well-divided solution exhibits **referential transparency** in all of its parts.

Applicative languages embody referential transparency as a design principle.

The model of an imperative language operates by manipulating a global state, and so transparency must be compromised, at some level. Indeed, the 'efficiency' of an imperative program (vis-a-vis an applicative one) derives from the *history sensitivity* provided by the state. On the other hand, 'good programming style' dictates that such non-transparent interactions should take place at the lowest levels only; a 'good' imperative program is one which seems to be applicative at all but the lowest conceptual level of each module. (An equivalent statement would be that a 'good' program admits a *direct* (rather than *continuation*) style of denotational semantic description [Sto77].)

When an applicative language is naively implemented on a conventional machine, there are at least two major sources of inefficiency: (1) the slow-but-general memory management, and (2) the reliance upon recursion as the only iteration mechanism.

(1) Updates-in-place of aggregate objects are not done, as an object might have been shared. Therefore, to ensure consistency and freedom from side-effects, large data structures are copied and the *copy* is modified. In many such cases, of course, the original object is *not* shared, will no longer be used, and an update-in-place would indeed be appropriate.

(2) Most implementations of recursion are more expensive than those of a **while** statement. Typically, a stack is maintained, and the return addresses and actual parameters are copied onto it. The amount of space consumed at run-time is proportional to the depth of the recursion. A **while** loop, on the other hand, consumes a constant amount of space, irrespective of the repetition count. The difference is fundamental; recursion is a more powerful mechanism than iteration. Naturally, in those restricted cases where the **while** loop is appropriate, it is more efficient. Many compilers do detect instances of "tail recursion" [Fod80, Ste77], a commonly-occurring iterative pattern, in order to eliminate the stack operations in such instances. However, more general recursion removal is rarely attempted in practice, although the subject has attracted lots of academic attention [Bir77, Coo66, KiS81, PeB82, SMR75].

In this project [KiS81], we seek to perform recursion removal (and other changes) by means of program transformations. We wish to program in a high-level applicative language, and have the program mechanically massaged into a faster form. Since some applicative languages (notably, FP) were designed to be manipulated and reasoned about, it is not surprising that their programs may be conveniently transformed. (Contrast this with the difficulties of reliably transforming imperative programs, or even proving the validity of the transformation rules.)

Some researchers [Bal81, Fea82, LoF81, MaW81] start with a very-high-level "specification" (programming) language, and semi-manually transform "specifications" (programs) into procedures expressed in a lower-level language. In contrast, this paper is concerned with aspects of purely *mechanical* transformation processes, using no human intervention. Also, we employ only "source-to-source" transformations; no hierarchy of languages is involved. (I might also add that I dislike executable "specifications", and feel that researchers who start with them are sweeping the hardest specification problems under a rug and at the same time introducing implementation bias.)

CHAPTER 2

Our Dialect of FP

2.1. Expressions

We generally follow the notation and semantics of FP as given in Backus's Turing Award Lecture [Bac78]; most of the changes are explicitly noted. Familiarity with the Lecture (especially § 11 - 12.2) is useful but not essential.

2.1.1. Objects

The set of FP objects includes the domains of

> truth values (**true** and **false**);
> rational numbers;
> sequences (lists) of objects, e.g. <1, 5, 3> and the empty list ε;
> n-tuples of objects, e.g. [1, 5, 3];

and perhaps others.

The object $\perp$ is the undefined element. Sequences and tuples are both *strict*; if any element of a sequence (or n-tuple) is $\perp$, then the sequence (or n-tuple) is $\perp$.

An alternative meta-notation for n-tuples is helpful, when the length (and contents) of the object under discussion are not explicitly known. An n-tuple $[o_1, o_2, \ldots o_n]$ may be represented without ellipsis by $\prod_{i=1}^{n} o_i$. If for some subscript k, $1 \le k \le n$, we know o_k to be x, then we write $\prod_{i=1}^{n} o_i \, [o_k \leftarrow x]$, rather than $[o_1, \ldots, o_{k-1}, x, o_{k+1}, \ldots, o_n]$.

2.1.2. Primitive Functions

The set of FP *primitive functions* includes operators from various object-level algebras:

> Boolean logic (*and*, *or*, *not*, ...),
> rational arithmetic (+, −, ×, ÷, ...),
> sequences (*head*, *tail*, *append*, *concat*, ...),

and others. It also includes relational predicates: =, ≠, ≤, ≥, and so on. All functions take exactly one argument, which may be an n-tuple: $+:[1,3] = 4$. (We are using the colon as the function application operator.)

The *constant* functions are also primitive. For any object x, the expression $\bar{x}$ denotes the function which maps any argument to the value x. For instance, $\bar{\bot}$ is the everywhere-undefined function.

The *identity* function **id** maps any object to itself.

2.1.3. Functional Forms

The primary *functional forms* of FP are composition, construction, and condition. These are higher-level operators which map functions to functions, and so form the backbone of the functional algebra of FP programs. There are several other FP functional forms (mostly specialized forms of iteration), but the functional algebra based on

(i) the primitive functions,

(ii) the functionals of composition, construction, & condition, with

(iii) recursive definitions

is complete in the sense of being able to describe any partial recursive function.

2.1.3.1. Composition: $f \circ g$

Against our better judgement we conform to tradition, and define the evaluation rule as proceeding from right to left:

$$(f \circ g):x = f:(g:x)$$

2.1.3.2. Construction: $\prod_{i=1}^{n} f_i$

Constructions produce tuples. Their evaluation rule is:

$$(\prod_{i=1}^{n} f_i):x = \prod_{i=1}^{n} (f_i:x)$$

An alternative notation for construction is available, based on that of tuples. We let $[f_1, f_2, \ldots f_n]$ be synonymous with $\prod_{i=1}^{n} f_i$.

Unlike Backus's FP, our dialect distinguishes between sequences (lists) and tuples. The operators defined on sequences do not apply to tuples, and vice versa. Some related differences are found in the meanings of 0-tuples and 1-tuples. We believe that our treatment corresponds to standard mathematical usage, and in any event is more convenient for our purposes.

$$\prod_{i=1}^{0} o_i \equiv \perp \tag{1}$$

$$\prod_{i=1}^{1} o_i \equiv o_1 \tag{2}$$

Axiom 1 states that there is no "empty tuple". Furthermore, the 1-tuple $[o]$ is *identical* to o, and not merely isomorphic to it, as is the case in the sequence domain. These conventions may be surprising, until one remembers that n-tuples are not an inductively-defined type.

The selector functions have been changed, too. Aside from distinguishing between the sequence operators (such as *head*, *tail*) and tuple selectors, we have added a crude form of typing, based on the length of the tuple: the expression $\langle i\ n \rangle$ denotes the function which selects the i^{th} field of an n-tuple object, viz., $\langle 1\ 3\rangle$:[5,6,7] = 5, but $\langle 1\ 2\rangle$:[5,6,7] = $\perp$.

2.1.3.3. Condition: $(p \rightarrow f ; g)$

This is FP's **if-then-else** operator. Its evaluation rule is:

```
(p → f ; g) : x =
      case p:x of
            "true"      ⇒ f:x
            "false"     ⇒ g:x
            default     ⇒ ⊥
      esac
```

2.1.3.4. Iteration

We shall use the most elementary form of iteration, in some of our sample transformations. Its evaluation rule is:

$$(\mathbf{while}\, p\ \mathbf{do}\, f) : x = (p \rightarrow (\mathbf{while}\, p\ \mathbf{do}\, f) \circ f ; \mathrm{id}) : x$$

We shall also use a pair of more specialized iterators, known as the 'insert-left' (/) and 'insert-right' (\) functional forms, which encapsulate one common style of recursive list traversal. Our definitions are extensions of the standard FP 'insert' functional form, and in fact more closely resemble the APL operator 'reduce'.

As before, we use f and g to denote functions, while x and y represent objects.

$$\begin{aligned}(/\ f\ y) : <> &= y\\ (/\ f\ y) : <x_0, x_1, \ldots, x_n> &= f : [x_0, (/\ f\ y) : <x_1, \ldots, x_n>]\end{aligned}$$

$$\begin{aligned}(\backslash f\ y) : <> &= y\\ (\backslash f\ y) : <x_0, \ldots, x_{n-1}, x_n> &= f : [(\backslash f\ y) : <x_0, \ldots, x_{n-1}>, x_n]\end{aligned}$$

Thus, the functions $(/\ +\ 0)$ and $(\backslash\ +\ 0)$ both compute the sum of a sequence of integers. We shall not use this form of insertion; we mention it only because of its resemblance to forms occurring in the literature [Ive62, Bac78].

The next version, which we shall use in our later examples, generalizes the 'inductive basis' from a object y to a function g.

$$\begin{aligned}(/\ f\ g) : <x_0> &= g : x_0\\ (/\ f\ g) : <x_0, x_1, \ldots, x_n> &= f : [x_0, (/\ f\ g) : <x_1, \ldots, x_n>]\end{aligned}$$

$$\begin{aligned}(\backslash f\ g) : <x_0> &= g : x_0\\ (\backslash f\ g) : <x_0, \ldots, x_{n-1}, x_n> &= f : [(\backslash f\ g) : <x_0, \ldots, x_{n-1}>, x_n]\end{aligned}$$

The two types of insertion are of course related; the second kind is the more general.

$$(/\ f\ y) \equiv (/\ f\ \text{id}) \circ \text{appendL} \circ [\bar{y}, \text{id}]$$

$$(\backslash f\ y) \equiv (\backslash f\ \text{id}) \circ \text{appendR} \circ [\text{id}, \bar{y}]$$

2.2. Computation Rules

An important property of FP functions is that they are *strict*;

$$(\forall f)\ f : \bot = \bot .$$

This implies, for example, that $\bar{0} : \bot$ and $\times : [0, \bot]$ and $\triangleleft 1\ 2 \triangleright : [0, \bot]$ are all equal to $\bot$ rather than to 0.

Backus [Bac78] used the continuity of the functional forms, the strictness property, and the (applicative-order) evaluation rules to justify a least-fixed-point semantics for recursive definitions of FP programs.

CHAPTER 3

Program Transformations

3.1. A Very Brief Survey

There are many types of program transformations in the literature. Among them we mention symbolic evaluation, strength reduction, and recursion removal. There are also some lesser optimizations of a bookkeeping nature, such as dead-code removal and common-expression elimination.

Prominent among the systems which do symbolic evaluation (of expressions drawn from numeric algebras) are Moses's MACSYMA and Hearn's REDUCE. MACSYMA, for example, can calculate the derivatives and indefinite integrals of arbitrary expressions [FaM82].

"Strength reduction" refers to the substitution of a weak-but-inexpensive operator for a powerful-but-expensive one, in those special cases where the full generality of the expensive operator is not needed. Phrased this way, the term covers a wide class of optimizations. Conventionally, however, it only refers to two types of reductions: simple, static optimizations, such as using a right-shift instead of a division by 2; and a technique called finite differencing [PaK82], which may be used to improve the efficiency of computations involving a loop induction variable.

Various types of recursion removal could perhaps be referred to as strength reduction, in the general sense mentioned above. A popular instance of this genre is ***tail-recursion elimination***. An algorithm exhibits tail recursion iff at least one of its recursive calls is followed immediately by a *RETURN*; that is, no further processing is done (by the algorithm) on the value returned by the call. Such recursions are equivalent to iterative loops; the stack is unnecessary.

The removal of tail-recursion is demonstrated by the following transformation. Given a (recursive) definition of a program f matching the template

$$\textbf{define}\, f \equiv (p \rightarrow g \; ; f \circ j),$$

an equivalent (non-recursive) definition of f is

define *f* ≡ *g* ∘ (**while** not∘*p* **do** *j*).

An instance of this transformation would be the recursive definition of the list operator *concat*:

```
define concat ≡
    (null?∘⟨1 2⟩ → ⟨2 2⟩ ;
                concat∘[tailR∘⟨1 2⟩, appendL∘[headR∘⟨1 2⟩, ⟨2 2⟩]])
```

which may be re-written non-recursively:

```
define concat ≡
        ⟨2 2⟩∘(while not∘null?∘⟨1 2⟩ do
                    [tailR∘⟨1 2⟩, appendL∘[headR∘⟨1 2⟩, ⟨2 2⟩]])
```

As a sample of a more complicated transformation schema, it has been observed [Coo66, DaB76, KiS81] that any program definition of *f* matching the template

define *f* ≡ (*p* → *g* ; *h*∘[*i*, *f*∘*j*])

in which the function instantiating *h* is associative and commutative (actually, a weaker constraint is possible here) and where *g* is a constant function (ditto), may be transformed into this opaque but non-recursive definition of *f*:

```
define f ≡
   ⟨1 2⟩ ∘ (while not∘p∘⟨2 2⟩ do [h∘[⟨1 2⟩, i∘⟨2 2⟩], j∘⟨2 2⟩]) ∘ [g, id]
```

The canonical example of this transformation is the factorial function:

define ! ≡ (eq0? → $\bar{1}$; ×∘[id, !∘sub1])

satisfies our constraints ($g \equiv \bar{1}$ is a constant function, and $h \equiv \times$ is both associative and commutative), and so may be rewritten as

```
define ! ≡
⟨1 2⟩∘(while not∘eq0?∘⟨2 2⟩ do [×∘[⟨1 2⟩, id∘⟨2 2⟩], sub1∘⟨2 2⟩])∘[1̄,id]
```

which may, in turn, be simplified somewhat:

```
define ! ≡
        ⟨1 2⟩∘(while not∘eq0?∘⟨2 2⟩ do [×, sub1∘⟨2 2⟩])∘[1̄,id]
```

3.2. The Rate-Determining Step

These transformations (and others) have been independently discovered by several authors [as shown above], and proved correct in [KiS81]. The derivation of such rules is a necessary and perhaps the most elegant step in the creation of a transformation system, but certainly the easiest. The crucial problems lie in determining the applicability of a given transformation to a given program. For imperative programs, these problems are virtually hopeless once the program reaches modest complexity. Much work has been done on theorem-provers, for both imperative- and applicative- language programs, but schema recognition (including template instantiation) is still quite difficult for non-toy languages, and probably beyond hope for degenerate languages like FORTRAN and **C**.

For applicative languages, there is optimism that these problems are solvable in a large number of cases of practical interest. (Of course, the general cases are often undecidable.) The property of referential transparency is of decisive importance in these applications, wherein the representation of a function is being manipulated. On the other hand, imperative languages (especially ones with global variables, call-by-reference parameter passage, *GOTO* statements, and arrays or pointer variables) easily befuddle compile-time transformation algorithms.

This paper is primarily concerned with the problem of template instantiation, and leaves the problems associated with the verification of constraints to (other people's) theorem-provers. We note (again) that even the theorem-proving routines must depend on an instantiator.

3.3. Transformation Systems

The behavior of program transformation *systems* has attracted comparatively little attention. Much of the literature has focussed on *ad hoc* collections of individual transformations, with no attempt made to predict or describe the properties of a collection as a whole. Transformations may be composed, several redexes may be available at any given step, systems may or may not have the Church-Rosser property. Normal forms may or may not exist, and if they exist, either they are unique or else their equivalence classes should be characterized. Transformations might or might not improve particular programs.

We do not have much to say about these things at this time, but wish to draw attention to an area of future research.

CHAPTER 4

Schema Recognition

4.1. The Problem

We need to recognize instances of transformable programs. For example, if we are given the template

$$\boldsymbol{f} \equiv (\boldsymbol{p} \rightarrow \boldsymbol{g}; \boldsymbol{h} \circ [\boldsymbol{i}, \boldsymbol{f} \circ \boldsymbol{j}])$$

and the definition of the factorial function

$$! \equiv (\mathrm{eq0?} \rightarrow \bar{1} \; ; \times \circ [\mathrm{id}, ! \circ \mathrm{sub1}]) \tag{4.1}$$

then it is obvious that the definition is an instance of the template. On the other hand, if we were given

$$\mathrm{eval} \equiv (\mathrm{not} \circ \mathrm{null?} \rightarrow + \circ [\times \circ [\overline{10}, \mathrm{eval} \circ \mathrm{tail}], \mathrm{head}] \; ; \bar{0}) \tag{4.2}$$

then it would be less clear. (What corresponds to $\boldsymbol{h}$? to $\boldsymbol{i}$?) However, the template can be instantiated by the substitution

$$\begin{array}{lcl} \boldsymbol{f} & \leftarrow & \mathrm{eval} \\ \boldsymbol{p} & \leftarrow & \mathrm{null?} \\ \boldsymbol{g} & \leftarrow & \bar{0} \\ \boldsymbol{h} & \leftarrow & + \circ [\times \circ [\overline{10}, \lhd 2\ 2 \rhd], \lhd 1\ 2 \rhd] \\ \boldsymbol{i} & \leftarrow & \mathrm{head} \\ \boldsymbol{j} & \leftarrow & \mathrm{tail} \end{array}$$

to yield an expression representing the same function as that denoted by 4.2. We want our prospective pattern-matching algorithm to recognize both examples as instances of the template.

4.2. Preliminaries

A *term* is an element of $T_\Sigma(V)$, the (freely-generated) initial Σ-algebra over a countable set of variables V and set of operators Σ. For all $n \geq 0$, $\Sigma_n \subset \Sigma$ is the set of n-ary operators; $\Sigma = \bigcup_{n=0}^{\infty} \Sigma_n$. In the subsequent chapters, we shall be especially interested in the Σ formed by the union of:

$$\Sigma_0 = \{ \text{ all FP primitive functions } \}$$
$$\Sigma_1 = \{ \text{ 1-tuple construction } \}$$

Σ_2 = { composition, 2-tuple construction }
Σ_3 = { condition, 3-tuple construction }
Σ_n = { n-tuple construction } for $n \geq 4$.

Occasionally, when we wish to write a term with (root) operator $\sigma \in \Sigma_n$ and children f_i for $1 \leq i \leq n$, we shall find it convenient to use the meta-notation

$$\sigma(\mathop{\mathrm{X}}_{i=1}^{n} f_i).$$

Expressions rooted by the nullary operators in Σ_0, which of course have no arguments, may be written as $\sigma(\mathrm{X}_{i=1}^{0} f_i)$, or as $\sigma()$, or just plain σ, as the whim takes us.

The expression $V(t)$ denotes the set of all variables $v \in V$ occurring in the term t.

A *substitution* is a mapping μ from V to $T_\Sigma(V)$, in which for most $v \in V$, $\mu(v) = v$. The *domain* of μ, written $Dom(\mu)$, is the set of all $v \in V$ such that $\mu(v) \neq v$.

We denote the assignment of a term t to variable v by $<v \leftarrow t>$, and a substitution is written as a set of assignments: $\{<v_1 \leftarrow t_1>, <v_2 \leftarrow t_2>, \ldots\}$. For convenience, all trivial (identity) assignments of the form $<v_i \leftarrow v_i>$ are omitted whenever possible, so that only the assignments to variables in the domain of the substitution are explicitly shown. The trivial (identity) substitution is denoted by the empty-set symbol, ϕ.

A *ground term* is a term which contains no variables, and thus is an element of T_Σ. A *ground substitution* is a mapping from V to T_Σ.

By a slight abuse of notation, we shall often refer to a "substitution μ" when we actually are referring to the natural extension (morphism) of μ to $T_\Sigma(V) \to T_\Sigma(V)$, wherein μ is used to effect replacements of all variables in the given term.

We define an infix operator $\oplus$ which effects the composition of two substitutions:

$$\mu_1 \oplus \mu_2 = \lambda x . \mu_2(\mu_1(x))$$

We note that $\oplus$ is associative but generally not commutative. Furthermore, it is clear that

$$\mu \oplus \phi = \mu = \phi \oplus \mu.$$

For pragmatic reasons, we shall only consider substitutions that are idempotent:

$$\mu \oplus \mu = \mu$$

4.3. Unification

Unification [Rob65] is a procedure for solving equations, using the instantiation of variables as the only permitted operation. Given a pair of expressions, containing (possibly shared) variables, the algorithm attempts to 'unify' the expressions by finding a substitution that makes them identical. The symbols in the expressions are not interpreted or evaluated. Thus, for any two terms t_1 and t_2, the unification procedure attempts to find the set U of substitutions μ that satisfy $\mu(t_1) = \mu(t_2)$.

For example, "$a \times b$" may be unified with "$13 \times (a+c)$" by the substitution $\mu = \{<a \leftarrow 13>, <b \leftarrow 13+c>\}$. All substitutions in U are instances of this *most general unifier* (m.g.u.); the less general ones are obtained by non-trivial assignments for c. We verify that

$$\mu(a \times b) = 13 \times (13+c) = \mu(13 \times (a+c)).$$

The unification algorithm, which is due to Robinson, runs as follows. There are three possible cases:

(1) both terms are variables;
(2) one term is a variable and the other is not;
(3) neither term is a variable.

In (1), it suffices to set either variable equal to the other.

For case (2), there are two possibilities. If the non-variable term contains the variable, then terminate with failure. Otherwise, assign the term to the variable.

When (3) applies, compare the operators at the root nodes. If they differ, then terminate with failure. Otherwise, iteratively unify the corresponding children, in sequence, from left to right.

In all cases, it is assumed that each component of the substitution is applied globally, to both terms, as soon as it is defined. This, and the serial (rather than parallel) unification of the children ensures consistency. The final substitution will be the most general unifier of the two terms. •

The unification algorithm is *complete* in the sense that all possible solutions (i.e. unifying substitutions) may be obtained by instantiating the free variables in the m.g.u. with arbitrary terms. Thus, the m.g.u. serves as a finite representation of a possibly infinite set of solutions.

4.3.1. Meta-Unification

Meta-unification is unification with respect to an equational theory E. Replacement of "equals by equals" is a permitted operation. The goal, for any pair of terms t_1 and t_2, is to find the set U of all substitutions μ that satisfy $\mu(t_1) =_E \mu(t_2)$. Ordinary unification is therefore equivalent to meta-unification w.r.t. an empty set of axioms ($=_\phi$ and $=$ are the same relation). With the empty theory, the terms

$$a \times b \quad \text{and} \quad a+c$$

cannot be unified. However, if we add the axioms (universally quantified on their free variables)

$$1 \times v = v$$
$$v \times 1 = v$$

we can meta-unify the terms via the substitution $\{<a \leftarrow 1>, <b \leftarrow 1+c>\}$. With the additional axioms

$$0+v = v$$
$$v+0 = v$$

we can derive the additional solutions $\{<b \leftarrow 1>, <c \leftarrow 0>\}$ and $\{<a \leftarrow 0>, <c \leftarrow 0 \times b>\}$. Thus, there may be several m.g.u.'s in the solution set for a meta-unification problem.

Papers on meta-unification include [HuO80, LaB77, LiS76, Mak77, PeS81, Sti81]. These authors devise unification algorithms which handle equivalences induced by sets of axioms that describe operators which are:

commutative;
associative & commutative;
associative, commutative, & idempotent;
associative & commutative, with an identity-element.

The associative axioms in particular cause difficulties; algorithms handling them often suffer from incompleteness or non-termination. For example, given just the associative axiom

$$(x+y)+z = x+(y+z)$$

and the terms

$$a+6 \text{ and } 6+a$$

then there is an infinite set of distinct solutions

$$\{<a \leftarrow 6>\} \quad \{<a \leftarrow 6+6>\} \quad \{<a \leftarrow 6+6+6>\} \quad \{<a \leftarrow 6+6+6+6>\} \quad \ldots$$

which cannot be generated by any finite set of m.g.u.'s with free variables. However, if we add the commutative axiom

$$x+y = y+x$$

then the unique m.g.u. is the identity substitution ϕ.

4.4. Pattern Matching

Pattern matching is a special case of unification, in which one term (the *target*) contains no variables. The term which might contain variables is called the *pattern*.

Meta-pattern matching is pattern matching with respect to an equational theory. Many pathological cases which occur in meta-unification do not occur in meta-pattern matching; pattern matching is a strictly simpler problem. As a consequence, *it is possible to use more complicated axiom sets*, ones beyond the scope of today's meta-unifiers. We shall use such an axiom set. In passing, we note that meta-pattern matching seems to be ignored in the literature (as a problem in its own right); this author has not found any references to previous work.

Actually, it appears that (meta-)unification and (meta-)pattern-matching form two points of a spectrum, based on the distribution and sharing of variables. In particular, an intermediate problem is of some interest: (meta-)unification in which the sets of variables occurring in the two terms are disjoint. Clearly, (meta-)pattern matching is an extreme example of this type, since the target has an empty set of variables. We conjecture that this intermediate problem, too, is simpler than general (meta-)unification. We shall raise the point again, later.

4.5. The Schema-Recognition Problem, Revisited

It is clear that our problem is one of meta-pattern matching, with respect to some set E of FP axioms. We desire a function

$$\Omega : [\mathit{Patterns} \times \mathit{Targets}] \rightarrow \mathit{Sets\_of\_Substitutions}$$

which returns a set of m.g.u.'s such that

$$[\forall \mu \in \Omega(p,t)] \quad \mu(p) =_E t\,.$$

CHAPTER 5

An Equational Theory

5.1. Our Functional Algebra

We introduce those algebraic axioms of FP which we shall use to recognize instances of templates. We are concerned only with the algebra of FP functional forms, and ***not*** with the object-level algebras, such as arithmetic or lists. However, we do provide the standard interpretation for Boolean algebra, when it occurs in the predicate of a conditional form.

We note that our set does *not* form a complete axiomization of FP.

5.1.1. Objects Considered Harmful

Programs are functions, not objects. We are concerned with reasoning about, and transformations of, *programs*. Denotations that are based upon objects introduce needless and unhelpful clutter.

5.1.2. The Axioms

In the axioms involving n-tuples, the n quantifies over the positive integers; therefore, such equations are actually axiom *schemes*, each one representing an infinite set of axioms. At this time, we shall be purposely ambiguous as to whether we are dealing with a finite set of 2^{nd}-order axioms or an infinite set of 1^{st}-order axioms.

We remind the reader that the meta-notation $\prod_{i=1}^{n} e_i \,[\![e_k \leftarrow f \,]\!]$ denotes the replacement of e_k by f in the given n-tuple.

The "$\equiv$", "$\sqsubseteq$" and "$\sqsupseteq$" symbols may all be interpreted as "=" during the first reading.

A1. $(f \circ g) \circ h \equiv f \circ (g \circ h)$

A2. $f \circ \mathrm{id} \equiv f$

A3. $\mathrm{id} \circ f \equiv f$

A4. $\bar{x} \circ f \sqsubseteq \bar{x}$

A5. $\langle k\ n \rangle \circ \prod_{i=1}^{n} e_i \sqsubseteq e_k$

A6. $\prod_{i=1}^{n} \langle i\ n \rangle \sqsubseteq \mathrm{id}$

A7. $\prod_{i=1}^{1} f_i \equiv f_1$

A8. $(\prod_{i=1}^{n} e_i) \circ f \equiv \prod_{i=1}^{n} (e_i \circ f)$

A9. $(p \to f\,; g) \circ h \equiv (p \circ h \to f \circ h\ ; g \circ h)$

A10. $h \circ (p \to f\,; g) \equiv (p \to h \circ f\,; h \circ g)$

A11. $\prod_{i=1}^{n} e_i \,[\![e_k \leftarrow (p \to f\,; g)]\!] \equiv (p \to \prod_{i=1}^{n} e_i \,[\![e_k \leftarrow f]\!]\,; \prod_{i=1}^{n} e_i \,[\![e_k \leftarrow g]\!])$

A12. $((p \to p_1\ ; p_2) \to f\,; g) \equiv (p \to (p_1 \to f\,; g)\,; (p_2 \to f\,; g))$

A13. $(\mathrm{not} \circ p \to f\,; g) \equiv (p \to g\,; f)$

A14. $(\mathrm{and} \circ [p, q] \to f\,; g) \sqsubseteq (p \to (q \to f\,; g)\,; g)$

A15. $(\mathrm{or} \circ [p, q] \to f\,; g) \sqsubseteq (p \to f\,; (q \to f\,; g))$

A16. $(p \to (p \to f\,; g)\,; h) \equiv (p \to f\,; h)$

A17. $(p \to h;\ (p \to f\,; g)) \equiv (p \to h\,; g)$

A18. $(p \to (q \to f\,; g)\,; h) \sqsupseteq (q \to (p \to f\,; h)\,; (p \to g\,; h))$

A19. $(p \to h;\ (q \to f\,; g)) \sqsupseteq (q \to (p \to h;\ f)\,; (p \to h;\ g))$

5.1.3. Comments

The axioms can be classified into several groups.

A1..A3 the set of FP function expressions under the operator of function composition forms a monoid

A4..A5 partial evaluations of the given expressions by a demand-driven evaluation rule

A6	valid as equality iff the argument is an n-tuple
A7	the definition of a 1-tuple
A8..A12	distributive laws
A13..A15	interpretations of Boolean operators
A16..A17	simplifications by subsumption
A18..A19	reshapings of decision trees; these are both special cases of the axiom $$(p \rightarrow (q \rightarrow a; b) ; (q \rightarrow c ; d)) \equiv (q \rightarrow (p \rightarrow a; c) ; (p \rightarrow b; d))$$ which may be viewed as asserting the equivalence of rotations of a decision table.

CHAPTER 6

Term Re-Writing Systems

6.1. Basics

A ***term*** (or ***tree***) ***re-writing system*** is a set of directed equations and a somewhat restricted "equals for equals" substitution mechanism. The system is applied to a term t, and produces a new term u, which is "equal to" t.

Given a set of equations **E**, we agree to limit the substitution mechanism to replacing one occurrence (in t) of the left-hand side of some equation by the corresponding instantiation of the right-hand side. We call such a substitution a ***reduction*** of t, and we say that t ***reduces*** (in one step) to u. A term which is irreducible (with respect to **E**) is called an ***E-normal form***.

For example, if we have the reduction rules (equations)

$$0 + x \ = \ x$$

$$succ(x) + y \ = \ x + succ(y)$$

then the computation 1+1=2 can be modelled by the reduction sequence

$$succ(0)+succ(0) \ \models \ 0+succ(succ(0)) \ \models \ succ(succ(0))$$

The final term, $succ(succ(0))$, is a normal form.

Term re-writing systems are sufficiently powerful to model all computable functions [Man74]. Our interest lies in the sub-class of such systems in which the reduction relation defines a partial order [KnB70].

6.2. Notation

We need to introduce some standard notation. The following definitions are adapted from Huet [Hue80].

$t \models u$ means that t reduces to u in one step.

$\models^0$ is the identity relation: $(t \models^0 u) \Leftrightarrow (t = u)$.

$\bullet$ is relation composition: $(t \models \bullet \models u) \Leftrightarrow \exists s : (t \models s) \wedge (s \models u)$

$\dashv$ is the inverse relation of $\vDash$: $(t \dashv u) \Leftrightarrow (u \vDash t)$.

For any relation $\vDash$, we define the auxiliary relations

$\vDash^{\varepsilon} \; = \; \vDash^{0} \cup \vDash$		*reflexive closure of* $\vDash$;
$\vDash\!\dashv \; = \; \vDash \cup \dashv$		*symmetric closure of* $\vDash$;
$\vDash^{i} \; = \; \vDash \bullet \vDash^{i-1}$	$(\forall i > 0)$	*i-fold composition of* $\vDash$;
$\vDash^{+} \; = \; \bigcup_{i=1}^{\infty} \vDash^{i}$		*transitive closure of* $\vDash$;
$\vDash^{*} \; = \; \bigcup_{i=0}^{\infty} \vDash^{i}$		*reflexive-transitive closure of* $\vDash$;
$\cong \; = \; \vDash^{*} \cap \dashv^{*}$		*equivalence under* $\vDash$;
$\sim \; = \; \vDash\!\dashv^{*}$		*congruence under* $\vDash$.

We also define the predicates

$$x \uparrow y \; = \; \exists w\colon \; (w \vDash x) \wedge (w \vDash y)$$

$$x \uparrow^{*} y \; = \; \exists w\colon \; (w \vDash^{*} x) \wedge (w \vDash^{*} y)$$

$$x \downarrow^{*} y \; = \; \exists z\colon \; (x \vDash^{*} z) \wedge (y \vDash^{*} z)$$

We say that the relation $\vDash$ is

(1) *acyclic* iff $\vDash^{+}$ is irreflexive (and then $\vDash^{*}$ is a partial ordering);

(2) *noetherian* iff there is no infinite sequence $t_1 \vDash t_2 \vDash t_3 \ldots$ (then $\vDash^{*}$ is well founded);

(3) *locally confluent* iff $\forall x,y\colon$ $(x \uparrow y) \Rightarrow (x \downarrow^{*} y)$. We call terms x and y satisfying $x \uparrow y$ a *critical pair* for the relation $\vDash$.

(4) *confluent* iff $\forall x,y\colon$ $(x \uparrow^{*} y) \Rightarrow (x \downarrow^{*} y)$.

A normal form is *canonical* iff it is not congruent ($\sim$) to any other normal form.

The expression $\Delta(\mathrm{E}, t)$ denotes the set of *all* normal forms for the E-congruence class containing t. The expression $\delta(\mathrm{E}, t)$ is used to denote *one* arbitrarily-chosen member of $\Delta(\mathrm{E}, t)$.

6.3. Complete Sets of Reduction Rules

A set of rules is ***complete*** iff it is confluent and noetherian. Completeness is a sufficient condition to ensure that all normal forms are canonical. If an equational theory admits a complete set of reduction rules, then the congruence of ground terms is decidable: two terms are congruent iff their canonical forms are identical.

6.3.1. Noetherian Relations

A relation is ***noetherian*** iff all reduction sequences terminate. Proving that a relation has this property can be quite difficult. The normal approach is to devise an auxiliary relation >, which is to be a well-founded strict partial ordering with the property that

$$[\forall t_1,t_2 \in T_\Sigma] \;\; (t_1 \models t_2) \Rightarrow (t_1 > t_2).$$

A large amount of very interesting work has been done, cataloguing the circumstances under which a partial ordering may be shown to be well-founded [Der82, DeM79, JoL82, HuL78, Lan79, LSn77, Pla78a, Pla78b]. An example is shown in Chapter 7, when we prove a particular set of rules to be noetherian.

6.3.2. Confluent Relations

Confluence may be regarded as a consistency criterion, asserting that the operation of reduction does not fundamentally alter the nature of a term. Confluence implies the following "Church-Rosser" property:

$$[\forall t_1, t_2 \in T_\Sigma] \;\; (t_1 \sim t_2) \Rightarrow (t_1 \downarrow^* t_2).$$

Any relation which is both locally confluent and noetherian is confluent. For relations with a finite set of rules **R**, local confluence is decidable. The test is known as the "Superposition Algorithm", and is credited to Knuth and Bendix [KnB70]. It is essentially a constructive proof of the local confluence.

Before we outline the algorithm, it is useful to introduce a formalization [Hul80] of what it means for a subexpression to occur at some point of a given expression.

6.3.2.1. Occurrences

We map the operators and variables of a term into sequences of non-negative integers, in such a way as to describe the "access path" traversed from the root to any given subtree. We use ε to represent the empty sequence, and "$\cdot$" to signify the operator which either appends an integer to the left end of an integer sequence, or concatenates sequences, depending on context. For any term t, the set of occurrences $Occ(t)$ is defined as the smallest set satisfying:

(i) $\varepsilon \in Occ(t)$

(ii) $[\forall \sigma \in \Sigma_n]\,[\exists i,\ 1 \le i \le n]\ \ \zeta \in Occ(t_i) \Rightarrow i \cdot \zeta \in Occ(\sigma \overset{n}{\underset{i=1}{\mathrm{X}}} t_i)$

The set of occurrences of a term is partially-ordered by prefix extraction: for all $\eta, \zeta \in Occ(t)$, $\eta \le \zeta \Leftrightarrow (\exists \vartheta)\ \eta \cdot \vartheta = \zeta$.

For every $\zeta \in Occ(t)$, we denote "the subterm of t at ζ" by t/ζ, and we represent "the replacement of t/ζ by t'" as $t[\zeta \leftarrow t']$.

From the definitions, we see that for all $\eta, \zeta \in Occ(t)$, $\eta \le \zeta \Leftrightarrow t/\eta$ is a subterm of t/ζ.

6.3.2.2. The Superposition Algorithm

For all pairs $\alpha_1 \models \beta_1$ and $\alpha_2 \models \beta_2$ of (not necessarily distinct) rules in R, the algorithm attempts to unify α_2 with each subexpression α_1/ζ_i of α_1. For all (m.g.u.) substitutions μ_i obtained in this fashion, the term $\mu_i(\alpha_1)$ reduces in two ways: $\mu_i(\alpha_1) \models \mu_i(\beta_1)$, and $\mu_i(\alpha_1) \models \mu_i(\alpha_1[\zeta_i \leftarrow \beta_2])$. Both terms of this critical pair are then reduced to their respective **R**-normal form(s); if they have any such in common, then the pair "passes" the test. If all such critical pairs pass the test, then **R** is locally confluent. ▪

6.3.3. The Knuth-Bendix Completion Algorithm

Very often a noetherian relation is non-confluent (and therefore incomplete), but the addition of certain new rules may make it confluent without enlarging the congruence class of any term. We say that the new rules *complete* the relation.

Knuth and Bendix [KnB70] devised an algorithm which produces a completion of any relation **R** for which a well-founded strict partial ordering (>) can be given. Their algorithm is not guaranteed to terminate, since not all axiom sets have finite completions.

The Completion Algorithm takes any non-confluent critical pair x and y discovered by the Superposition Algorithm, and creates a new reduction rule $\delta(\mathrm{R}, x) \models \delta(\mathrm{R}, y)$, such that $\delta(\mathrm{R}, x) > \delta(\mathrm{R}, y)$. The new reduction is added to **R**, and the process repeats until the augmented rule set is confluent, or until the end of time, whichever comes first. It is possible that some new reductions will subsume old ones; in these cases, the old ones are eliminated. ▪

It is also possible for the algorithm to fail: there may arise critical pairs $[x,y]$ such that no term in $\Delta(\mathbf{R},x)$ is related by $>$ to any term in $\Delta(\mathbf{R},y)$. In such cases, the relation $>$ is shown to be insufficient; it is also possible that one choice of $>$ may yield a finite confluent set, while another may lead to the generation of an infinite set.

CHAPTER 7

Our Term Re-Writing System

7.1. Our Reduction Rules

We take the axioms A1..A15 as our reduction rules. When referring to the axioms in this role (i.e., as *directed* equations) we shall call them by the names R1..R15, and assign the name **R** to the set.

The presence of the distributive laws R8..R12 in **R** implies that all **R**-normal forms are decision trees, with the condition operators (if any) near the root, followed by construction operators, and then the compositions, with the primitive functions (or variables) at the leaves. For example, the (unique) **R**-normal form of the term

$$[(a \rightarrow b\ ;\ c), d] \circ e$$

is

$$(a \circ e \rightarrow [b \circ e\,,\ d \circ e];\ [c \circ e\,,\ d \circ e])$$

As noted previously, the n-tuple axioms A5, A6, A8, and A11 are actually axioms *schemes*, producing one axiom apiece for every $n \geq 1$. For any set of axioms **E**, we shall write $[\mathbf{E}]_m$ to signify the restriction of **E** to the subset containing no axioms with $n > m$.

7.2. Termination

We prove that the set **R** is noetherian by devising a strict partial ordering relation ($\supset$) which displays the following properties.

[a] $\supset$ is well-founded; that is, there is no infinite descending sequence of ground terms $t_1 \supset t_2 \supset t_3$ ....

[b] $[\forall t_1, t_2 \in T_\Sigma]\ \ (t_1 \models t_2) \Rightarrow (t_1 \supset t_2)$

The condition [b] is awkward because it quantifies over all ground terms. We would prefer that the termination property be deducible from a finite analysis of a finite set of re-write rules. Fortunately, the problem may be re-phrased. To show that **R** is noetherian, it is sufficient [HuO80] to show that the partial ordering ($\supset$) satisfies all three of the following conditions:

(1) $[\forall t_1, t_2 \in T_\Sigma]\ [\forall \sigma \in \Sigma_n]\ [\forall k,\ 1 \le k \le n]$

$$(t_1 \supset t_2) \ \Rightarrow\ \sigma(\overset{n}{\underset{i=1}{\mathrm{X}}} e_i \,[e_k \leftarrow t_1]) \supset \sigma(\overset{n}{\underset{i=1}{\mathrm{X}}} e_i \,[e_k \leftarrow t_2]);$$

(2) $[\forall t \in T_\Sigma]\ [\forall \sigma \in \Sigma_n]\ [\forall k,\ 1 \le k \le n]$

$$\sigma(\overset{n}{\underset{i=1}{\mathrm{X}}} e_i \,[e_k \leftarrow t\,]) \supset t;$$

(3) $[\forall \mu \in (T_\Sigma(V) \rightarrow T_\Sigma)]\ [\forall (\alpha \models \beta) \in \mathbf{R}]$

$$\mu(\alpha) \supset \mu(\beta).$$

We define such a relation by a slight variation on the *recursive path ordering method* [DeM79; independently re-discovered by this author]. First it is necessary to define an ordering on the operators of Σ:

$$\textit{atomic functions} \supset_\Sigma \textit{composition} \supset_\Sigma \textit{construction} \supset_\Sigma \textit{conditional}$$

We leave all the atomic functions incomparable to each other, with the exceptions that the selectors $\triangleleft\, i\ n \triangleright$ are *less* under $\supset_\Sigma$ than all other atomic operators, with the identity function **id** (a.k.a. $\triangleleft\, 1\ 1 \triangleright$) being the smallest of all:

$$\textit{other functions} \supset_\Sigma \triangleleft i\ n \triangleright \supset_\Sigma \mathbf{id}.$$

The relation $\supset_\Sigma$ is extended to an ordering $\supset$ on the (ground) terms of T_Σ by the following rules:

For all $\sigma_1 \in \Sigma_m$, and for all $\sigma_2 \in \Sigma_n$,

$$\sigma_2(\overset{n}{\underset{k=1}{\mathrm{X}}} g_k) \supset \sigma_1(\overset{m}{\underset{i=1}{\mathrm{X}}} f_i)$$

$$\Leftrightarrow$$

$$[\exists k, 1 \le k \le n]\ g_k \supseteq \sigma_1(\overset{m}{\underset{i=1}{\mathrm{X}}} f_i)$$

$$\vee$$

$$(\sigma_2 \supset_\Sigma \sigma_1) \ \wedge\ [\forall i, 1 \le i \le m]\ \sigma_2(\overset{n}{\underset{k=1}{\mathrm{X}}} g_k) \supset f_i$$

$$\vee$$

$$(\sigma_2 = \sigma_1) \ \wedge \ \mathop{\mathrm{X}}_{i=1}^{n} g_i \supset \mathop{\mathrm{X}}_{i=1}^{n} f_i$$

where $\mathrm{X}_{i=1}^{n} g_i \supset \mathrm{X}_{i=1}^{n} f_i$ denotes the lexicographic extension of $\supset$ to tuples, with the case $m = n = 0$ being (trivially) false.

This ordering may, in turn, be extended to possibly non-ground terms in $T_\Sigma(V)$ by augmenting the above disjunction with the fourth case

$$\sigma_1 \in V(\sigma_2(\mathop{\mathrm{X}}_{k=1}^{n} g_k)) \ \wedge \ n \neq 0.$$

where, if $m=0$, we now permit σ_1 to be chosen from $\Sigma_0 \cup V$.

If for all rules $\alpha \models \beta \in \mathbf{R}$, we have that $\alpha \supset \beta$, then $\mathbf{R}$ is noetherian. Our rule set $\mathbf{R} = \{\text{R1..R15}\}$ is shown to be noetherian by applying this test.

7.3. Incompleteness

The Knuth-Bendix completion algorithm will supply a large number (∞) of extra rules, including

$$(p \to f\,;\, f) \ \models \ f \qquad \text{(R20)}$$

$$\langle 1\ 1 \rangle \ \models \ \text{id} \qquad \text{(R21)}$$

$$\prod_{i=1}^{n} (\langle i\ n \rangle \circ f) \ \models \ f \qquad \text{(R22)}$$

$$(\text{not} \to f\,;\, g) \ \models \ (\text{id} \to g\,;\, f) \qquad \text{(R23)}$$

$$(\text{and} \to f\,;\, g) \ \models \ (\langle 1\ 2 \rangle \to (\langle 2\ 2 \rangle \to f\,;\, g)\,;\, g) \qquad \text{(R24)}$$

$$(\text{and} \circ h \to f\,;\, g) \ \models \ (\langle 1\ 2 \rangle \circ h \to (\langle 2\ 2 \rangle \circ h \to f\,;\, g)\,;\, g) \qquad \text{(R25)}$$

$$(\text{or} \to f\,;\, g) \ \models \ (\langle 1\ 2 \rangle \to f\,;\, (\langle 2\ 2 \rangle \to f\,;\, g)) \qquad \text{(R26)}$$

$$(\text{or} \circ h \to f\,;\, g) \ \models \ (\langle 1\ 2 \rangle \circ h \to f\,;\, (\langle 2\ 2 \rangle \circ h \to f\,;\, g)) \qquad \text{(R27)}$$

The first new rule (R20) is the only one of independent interest. The derivation of R20 may help to shed light on its relationship with the computation rules implied by R4, R5, R10 and R11.

$$\begin{aligned} \bar{x} \ &\dashv \ \bar{x} \circ (p \to f\,;\, g) && \text{(by R4)} \\ &\models \ (p \to \bar{x} \circ f\,;\, \bar{x} \circ g) && \text{(by R10)} \\ &\models^2 \ (p \to \bar{x}\,;\, \bar{x}) && \text{(by R4, twice)} \end{aligned}$$

Or, more generally,

$$\begin{aligned} h &\dashv \lhd 1\ 2 \rhd \circ [h, (p \to f ; g)] && \text{(by R5)} \\ &\models \lhd 1\ 2 \rhd \circ (p \to [h,f] ; [h,g]) && \text{(by R11)} \\ &\models (p \to \lhd 1\ 2 \rhd \circ [h,f] ; \lhd 1\ 2 \rhd \circ [h,g]) && \text{(by R10)} \\ &\models^2 (p \to h ; h) && \text{(by R5, twice)} \end{aligned}$$

One other point is worth mentioning here. It seems that R20 may require the ability to recognize the extensional equality of two function expressions, which is of course undecidable. Fortunately, all that is really necessary is that the intensional relation $=_E$ be decidable.

The new axioms R25 and R27 subsume R14 and R15, so we omit the latter.

An exhaustive case analysis confirms that the rule set $\mathbf{R}^* = \{$R1..R13, R20..R27$\}$ is confluent modulo congruence classes induced by Boolean algebra. That is, $\mathbf{R}^*$ is not confluent, but the normal forms of any given term differ only in the arrangement of the *conditional operators*; the non-conditional sub-expressions are the same (although their order may be permuted). For instance, the term

$$[(p \to f_1;g_1),(q \to f_2;g_2)]$$

has two normal forms,

$$(p \to (q \to [f_1,f_2];[f_1,g_2]);(q \to [g_1,f_2];[g_1,g_2]))$$

and

$$(q \to (p \to [f_1,f_2];[g_1,f_2]);(p \to [f_1,g_2];[g_1,g_2]))$$

These $\mathbf{R}^*$-normal forms can be reduced to each other by the *selective* application of rules R16..R19 (which are *not* in $\mathbf{R}^*$). Of course, term rewriting systems are *not* permitted to be selective in their application of rules, and so we are left with these as distinct normal forms (if we exclude R16..R19), or, with non-termination (if we include them). For the same reasons, it is not possible to sort the predicates into alphabetical order, or otherwise choose a canonical representative of the set of all semantically-equivalent decision trees, without stepping outside of the framework of the term-rewriting system.

This author has proven that, for any $m \geq 2$, there does not exist a finite complete rule set containing $\lceil \mathbf{R}^* \rceil_m$. The proof is a generalization of the example given above. Reversing the associative grouping of composition

(R1) does not change matters.

7.4. Two Complete Subsets

The rule set $\mathbf{R}^* - \{R10,R11\}$ is complete. The confluence has been checked mechanically, and the termination property is demonstrated by the method given earlier.

The rule set $\mathbf{R}+ = \{R1..R8, R21..R22\}$ is also complete. It is the largest subset of $\mathbf{R}^*$ that contains no axioms involving the conditional operator; we shall find it useful in solving matching problems in the condition-free sub-algebra of FP.

CHAPTER 8

Soundness and Completeness

8.1. Soundness

All of the axioms are valid in the limited sense that for whatever arguments both sides of an axiom are defined (i.e., not $\perp$), both sides give the same answer. Thus, the left- and right-hand sides of each axiom are ***weakly equivalent*** ($\simeq$).

However, for some of the axioms, there exist arguments for which one side is defined, while the other is not. We define the relations of *approximates* ($\sqsubseteq$), *is approximated by* ($\sqsupseteq$), and *strong equivalence* ($\equiv$) in the standard way.

$$(f \sqsubseteq g) \Leftrightarrow [\forall x]\ [(f:x = \perp) \vee (f:x = g:x)]$$

$$(f \sqsupseteq g) \Leftrightarrow (g \sqsubseteq f)$$

$$(f \equiv g) \Leftrightarrow (f \sqsubseteq g) \wedge (f \sqsupseteq g)$$

For the FP computation rules, the axioms' left- and right-hand sides are related as shown in Chapter 5 [Bac78]. We shall consider our prospective pattern-matching algorithm Ω : [*Patterns* × *Targets*] → {*Substitutions*} to be *sound* iff the algorithmically-produced function is at least as well defined as the original function:

$$[\forall \mu \in \Omega(p, t)]\quad \mu(p) \sqsupseteq t.$$

We note that **R***-normalization and **R+**-canonicalization are both sound:

$$[\forall e \in Exp]\,[\forall \delta \in \Delta(\mathrm{R}^*, e)]\quad \delta \sqsupseteq e$$

$$[\forall e \in Exp]\,[\forall \delta \in \Delta(\mathrm{R}+, e)]\quad \delta \sqsupseteq e$$

8.1.1. A Partially-Ordered Set of FP Machines

Let us denote by **FP**(e) the *function* computed by the expression e under FP evaluation rules. The notation $f \sqsubseteq_{FP} g$ is used as a shorthand for **FP**(f) $\sqsubseteq$ **FP**(g), and analogously for $\sqsupseteq_{FP}$ and $\equiv_{FP}$.

We can extend the partial order $\sqsubseteq$ to ***abstract machines*** (computation rules) which map function expressions to functions. If Exp is the set of all valid FP function expressions, and M_1 and M_2 are machines,

$$M_1 \sqsubseteq M_2 \Leftrightarrow [\forall e \in Exp]\, M_1(e) \sqsubseteq M_2(e)$$

and similarly for $\sqsupseteq$, $\equiv$, and $\simeq$.

It is evident that by varying the computation rules of FP, a partially-ordered set $<\mathbf{M}, \sqsubseteq>$ of abstract machines may be created, satisfying the (weak) equality

$$[\forall \mathbf{m} \in \mathbf{M}] \quad \mathbf{m} \simeq \mathbf{FP}.$$

We wish to define an abstract machine under which the axioms are *equations*; so that all the $\sqsubseteq_{FP}$ and $\sqsupseteq_{FP}$ relations are raised to $\equiv$. The problem with using the axioms as they stand is that weak equality is not an equivalence relation (because $\simeq$ is not transitive), and so substituting "equals for equals" is not safe. For example, if we *were* permitted to freely substitute "weakly equals for weakly equals," then the trivial-but-true axiom

A0. $\bar{\bot} \sqsubseteq f$

would permit us to substitute $\bar{\bot}$ for any function (say, f). Further, it would allow us to continue by replacing the $\bar{\bot}$ with any function g, since $\bar{\bot} \simeq g$, and g may be completely unrelated to f.

We can effectively find two machines in **M** for which the axioms are all $\equiv$ relations. The simpler is the *less-defined FP* machine (ldFP). This machine's computation rule is derived by inspecting each $\sqsubseteq_{FP}$ and $\sqsupseteq_{FP}$ relation, and regarding the *less* defined side of each axiom as the common value for *both* sides. The (updated, modified) computation rule is then (re-)applied to all of the axioms, and the process repeated until a fixed-point is reached.

The less-defined side often contains (universally-quantified) function variables not occurring in the better-defined side, and so it is necessary to assume that all such variables are instantiated by $\bar{\bot}$. This has the effect of reducing the domain of the better-defined side to a sub-domain of the less-defined side, in this case the empty domain.

For example, axiom A5 gives us that

$$\langle 1\ 2 \rangle \circ [f, \bot] \equiv_{\mathrm{ldFP}} f$$

Since construction and composition are strict in FP, we have

$$\langle 1\ 2 \rangle \circ [f, \bot] \equiv_{\mathrm{FP}} \langle 1\ 2 \rangle \circ \bot \equiv_{\mathrm{FP}} \bot$$

and so we conclude that $f \equiv_{\mathrm{ldFP}} \bot$, for all function expressions f.

Of greater interest is the *more-defined FP* machine (**mdFP**), which uses the *better* defined side of each axiom as the value for both sides.

Any FP function expression, when 'run' on the **FP** machine, approximates the same expression run on the **mdFP** machine.

$$[\forall e \in Exp]\ \bot \equiv \mathbf{ldFP}(e) \sqsubseteq \mathbf{FP}(e) \sqsubseteq \mathbf{mdFP}(e)$$

In a machine with ***applicative-order*** computation rules, all arguments of a function are evaluated, even if the function does not need some of them under some circumstances. In a ***normal-order*** machine, arguments are evaluated if and only if they are needed.

As a first approximation to our "more defined" FP machine, let's look at a normal-order (demand-driven) FP machine, which we temporarily call **N**. (The underlying computation rules of **FP** are those of applicative-order (data-driven) evaluation.) In such a machine, the axioms A4, A5, and A6 are $\equiv_N$. Regrettably, this machine is insufficient. We find that A10 and A11 must become

A10. $h \circ (p \to f\,;\, g) \sqsupseteq_N (p \to h \circ f\,;\, h \circ g)$

A11. $\prod_{i=1}^{n} e_i \llbracket e_k \leftarrow (p \to f\,;\, g) \rrbracket \sqsupseteq_N (p \to \prod_{i=1}^{n} e_i \llbracket e_k \leftarrow f \rrbracket\,;\, \prod_{i=1}^{n} e_i \llbracket e_k \leftarrow g \rrbracket)$

That is, information is lost when going left-to-right. Also, the axioms A14, A15, A18 and A19 remain $\sqsubseteq_N$ or $\sqsupseteq_N$, and A20 is now $(p \to f\,;\, f) \sqsubseteq_N f$. (The relationships of axioms A4, A5, A10, and A11 with A20 were discussed briefly back in § 7.3.) From these inadequacies one can deduce the need for an "eager beaver" computation rule for conditions, namely, the ***parallel*** evaluation of ***all three*** subexpressions. One such rule for computing the value of a conditional function, applied to an argument x, is

$(p \rightarrow f\,;g) : x \;=$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (C_1)

> Simultaneously evaluate $p{:}x$, $f{:}x$, and $g{:}x$.
>
> If $p{:}x$ terminates with value **true**, then return $f{:}x$ and discard $g{:}x$.
>
> If $p{:}x$ terminates with value **false**, then return $g{:}x$ and discard $f{:}x$.
>
> If $f{:}x$ and $g{:}x$ both terminate with the same value, then return that value and discard $p{:}x$.
>
> In all other cases, return $\perp$.

With this computation rule (C_1), the demand-driven machine **N** becomes **mdFP**.

Actually, we can do somewhat better; this can be best illuminated by describing a new rule in the order-theoretic terms of denotational semantics [Sto77]. Consider all values in the ranges of f and g to be elements of a countably-based complete partial order $<D, \leq>$. Assume further that the computation of $f{:}x$ (and $g{:}x$) might emit some basis elements of the result as soon as they are available, rather than upon termination of the entire computation. We denote by $\bigsqcup f{:}x$ the least upper bound (l.u.b.) of the basis elements emitted (so far) by the computation of $f{:}x$, and similarly for $\bigsqcup g{:}x$. Then the conditional operator can emit intermediate results according to the computation rule:

$(p \rightarrow f\,;g) : x \;\equiv$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (C_2)

$$\begin{array}{ll} \bigsqcup f{:}x & \ldots \text{ if } p{:}x = \textbf{true}; \\ \bigsqcup g{:}x & \ldots \text{ if } p{:}x = \textbf{false}; \\ \bigsqcup f{:}x \sqcap \bigsqcup g{:}x & \ldots \text{ otherwise.} \end{array}$$

This accounts for cases wherein it is unnecessary for *any* of $p{:}x$, $f{:}x$, or $g{:}x$ to terminate. For instance, f and g may return (potentially infinite) sets as values, and subsequent operations upon these sets may only ask whether the sets contain some specific subset. In such circumstances, if $p{:}x$ has not yet terminated, it is unnecessary that the entire sets $f{:}x$ and $g{:}x$ be computed, if the desired subset is contained in the intersection of the partial results already returned by $f{:}x$ and $g{:}x$. (In this example, D is a powerset, the partial ordering relation $(\leq)$ is $\subseteq$, the basis elements are the finite sets in D, the least-upper-bound operation $(\sqcup)$ is $\cup$, and the greatest-lower-bound operation $(\sqcap)$ is $\cap$.)

In a series of papers [MaS75, MaS76], Manna and Shamir investigate the properties of the "optimal fixed-point" semantics of recursive function definitions. The optimal fixed-point is often more defined than the least fixed-point (which is the popular choice in the semantics literature). The relationship between their optimal fixed-point and our **mdFP** machine has not been explored.

The **mdFP** machine is too slow to be used to execute programs. We shall use it only to justify the intermediate steps of the pattern-matching process. The plan is to take an **FP** pattern-matching problem, regard it as an **mdFP** problem, solve it for the **mdFP** machine, and return as the **FP** answer only that subset of the **mdFP** answer which is *sound*; for each substitution returned in the answer, the target must approximate ($\sqsubseteq_{FP}$) the instantiated template.

We observe that we don't need all of the power of the C_2 rule; we can justify the pattern-matching algorithm with the weakest machine subsuming **FP** and satisfying our axioms.

8.2. Completeness

We may choose between several degrees of completeness for our prospective pattern-matching algorithm. We may demand that it find:

[1] one solution (if any exist);
[2] all solutions;
[3] all solutions, modulo an arbitrarily-chosen equivalence.

Clearly, option [1] demands the least effort to be expended. However, for the intended application of the algorithm (program transformations), some instantiations of the schema variables may be better than others; that is, some may satisfy the transformation constraints, while others may fail.

Option [2] offers the best chance to satisfy external constraints. Unfortunately, our algebra admits cases wherein complete solutions must contain infinite sets of m.g.u.'s. For example, if v_1 and v_2 are variables, then the set of substitutions calculated by $\Omega(v_1 \circ v_2, +)$ contains (amongst other things) the infinite family of m.g.u.'s

$$\{ <v_1 \leftarrow \text{id}>, <v_2 \leftarrow +> \}$$

$$\{ <v_1 \leftarrow \langle\!\langle 1\ 2 \rangle\!\rangle>, <v_2 \leftarrow [+, v_3]> \}$$

$$\{ <v_1 \leftarrow \lhd 1\ 2 \rhd \circ \lhd 1\ 2 \rhd>, <v_2 \leftarrow [[+,v_3],v_4]> \}$$

$$\{ <v_1 \leftarrow \lhd 1\ 2 \rhd \circ \lhd 1\ 2 \rhd \circ \lhd 1\ 2 \rhd>, <v_2 \leftarrow [[[+,v_3],v_4],v_5]> \}$$

etc.

We note that this difficulty was *not* caused by our infinite set of n-tuple axioms; only axioms for 2-tuples were used. Of course, similar infinite series *do* arise for every $n \geq 1$.

Option [3] (which of course subsumes [1] and [2]) may be viewed as offering a compromise between completeness and termination. We shall choose and characterize an equivalence relation (in which all partitions contain at least one finite member), after first presenting the algorithm.

CHAPTER 9

The Basic Algorithm

9.1. Partitioning the Problem

It is both possible and useful to separate the handling of the 'condition' functional from that of the other functionals. It is possible because all R*-normal terms are decision trees with all non-condition functional forms (and primitive functions) at the leaves; it is useful because the condition functional form is difficult to handle. We note that while the (condition-free) rule set R+ is confluent, the rule set R* is not; and the rule set R* ∪ {R16..R19}, which has enough power to express the equivalence of two decision trees, is not even noetherian.

9.2. Composition & Construction

We need a function $\Phi \in (\mathit{Patterns} \times \mathit{Targets}) \rightarrow \{\ \mathit{Substitutions}\ \}$ which does the pattern-matching operation on non-conditional expressions, returning the set of all substitutions which unify the pattern and the target.

We shall find it convenient to use $T_\Sigma(V)$ rather than T_Σ as the domain for targets, with the understanding that any variables introduced into the target are unique, and so do not occur in the pattern.

9.2.1. More Notation

We extend the operator of set union, to encompass the union of two set-producing functions:

$$f \cup g \ \equiv \ \lambda x\,.(f{:}x \cup g{:}x\,)$$

Thus, we are using the symbol "∪" as both a function and a functional form; the reader should be able to tell the difference from the context.

We define an infix operator @, which provides a convenient way to compose functions. Let S denote a set of widgets, and let f be a function that maps widgets to sets of doohickies:

$$S \,@\, f \;=\; \bigcup_{s \in S} f{:}s$$

The @ operator is sometimes more opaque than the notation that it displaces, but its use facilitates algebraic manipulations. The operator associates on the left:

$$(S \,@\, f \,@\, g) \;=\; ((S \,@\, f) \,@\, g)$$

It follows immediately from the definitions that @ distributes over union:

$$(S_1 \cup S_2) \,@\, f \;=\; (S_1 @ f) \cup (S_2 @ f)$$

$$S \,@\, (f \cup g) \;=\; (S @ f) \cup (S @ g)$$

We extend the notation slightly, to express n-fold @ composition.

$$S \overset{0}{\underset{i=1}{@}} f_i \;=\; S$$

$$(S \overset{n+1}{\underset{i=1}{@}} f_i) \;=\; (S \overset{n}{\underset{i=1}{@}} f_i) \,@\, f_{n+1}$$

We now define the infix operator $\triangleright$, which uses substitutions (possibly obtained from an earlier invocation of the pattern-match operator Φ) to provide environments in which subsequent pattern-match operations take place. For any $S \in \{Substitutions\}$ and $[p,t] \in T_\Sigma(V) \times T_\Sigma(V)$,

$$S \triangleright [p,t] \;=\; S \,@\, \lambda s\,.\left(\Phi[\delta(\mathrm{R+},s(p)),\delta(\mathrm{R+},s(t))] \,@\, \lambda s'.\{s \oplus s'\}\right)$$

The conventions of (left-)associativity and n-fold composition of $\triangleright$ are analogous to those for @.

In general, for all $\mu \in (S \triangleright [p,t])$, μ is a refinement of some substitution $s \in S$ (that is, there exists a substitution s' such that $\mu = s \oplus s'$), and of course μ meta-unifies p and t: $\mu(p) =_{\mathrm{R+}} \mu(t)$. We also notice that if p and t are R+-normal forms, then

$$\{\phi\} \triangleright [p,t] \;=\; \Phi[p,t].$$

9.2.2. The Algorithm Φ

We are almost ready to present a first draft of the algorithm Φ. First, however, we need a couple more auxiliary definitions.

The predicate Is_Variable?(f) returns **true** if $f \in V$, and **false** otherwise.

The predicate Is_Tuple?(n, f) returns **true** if f is a function which returns an n-tuple as its result, and **false** otherwise. We recall that n-tuples may be generated by user-defined functions, as well as by construction operators.

The predicate Is_Const_Function?(f) returns **true** if f is a manifest constant function $\bar{o}$ for some object o, and **false** otherwise.

The symbols u and $v_{i,n}$ represent new variables created by the algorithm. They are assumed to be unique, and distinct from the new variables u' and $v'_{i,n}$ created by other (recursive) invocations of Φ, or by parallel computations within the same invocation. (The LISP function *gensym* performs a service such as this.)

The numbers running down the left-hand side of the algorithm just tag certain lines, to bind them to the corresponding comments on the next page.

```
letrec $\Phi\,[P,T]$ =
    if Is_Variable?($T$) then $\{\{<T \leftarrow P>\}\}$ else
    if Is_Variable?($P$) then $\{\{<P \leftarrow T>\}\}$ else
    case $P$ of
1)      atomic : if $P = T$ then $\{\phi\}$ else $\{\ \}$ fi;
        $\prod_{i=1}^{n} p_i$ :
            case $T$ of
                atomic : if Is_Tuple?($n, T$) then $\Phi[P, \prod_{k=1}^{n} \triangleleft k\ n \triangleright \circ T]$ else $\{\ \}$ fi;
                $\prod_{i=1}^{m} t_i$ : if $m=n$ then $\{\phi\} \overset{n}{\underset{i=1}{\triangleright}} [p_i, t_i]$ else $\{\ \}$ fi;
                $t_1 \circ t_2$ : if Is_Tuple?($n, T$) then $\Phi[P, \prod_{k=1}^{n} \triangleleft k\ n \triangleright \circ T]$ else $\{\ \}$ fi;
            esac;
        $p_1 \circ p_2$ :
            $\Phi[p_1, \text{"id"}] \triangleright [p_2, T] \quad \cup$
2)          $\Phi[p_1, T] \triangleright [p_2, \text{"id"}] \quad \cup$
            if Is_Variable?($p_1$)
3)              then $\bigcup_{n=2}^{\infty} \Phi\,[p_2, \prod_{i=1}^{n} v_{i,n}] \triangleright \bigcup_{k=1}^{n} [p_1, u \circ \triangleleft k\ n \triangleright] \triangleright [u \circ v_{k,n}, T]$
4)              else $\bigcup_{n=2}^{\infty} \Phi\,[p_2, \prod_{i=1}^{n} v_{i,n}] \triangleright \bigcup_{k=1}^{n} [p_1, \triangleleft k\ n \triangleright] \triangleright [v_{k,n}, T]$
            fi $\cup$
            case $T$ of
5)              atomic : if Is_Const_Function?($T$) then $\Phi[p_1, T]$ else $\{\ \}$ fi
6)              $\prod_{i=1}^{n} t_i$ : $\Phi[p_1, \prod_{i=1}^{n} v_{i,n}] \triangleright [\prod_{i=1}^{n} v_{i,n} \circ p_2, T]$;
                $t_1 \circ t_2$ :
7)                  $\Phi[p_1, t_1] \triangleright [p_2, t_2] \quad \cup$
8)                  $\Phi[p_1, v_1 \circ v_2] \triangleright [v_1, t_1] \triangleright [v_2 \circ p_2, t_2]$;
            esac;
    esac
    fi fi
end $\Phi$ ▪
```

9.2.3. Comments on Φ

The following comments are keyed to particular lines of Φ. Subsequent sections will give fuller treatments of some of the issues raised here.

1) If the atomic functions are identical, then return the identity substitution ϕ; otherwise return nothing (i.e., fail).

3) The matching attempts made by this line are based on rule R5. The n ranges from 2 to ∞; the lower bound excludes consideration of 1-tuples, while the upper non-bound models the infinite set of (1st-order) tuple axioms.

   We observe that, if p_2 is not a variable, the matching of p_2 with $\prod_{i=1}^{n} v_{i,n}$ will bind the (new) variables $v_{i,n}$ to sub-structures of p_2. So, by the time that the pair $[u \circ v_{k,n}, T]$ is matched, the variable $v_{k,n}$ has already been bound to some piece of p_2.

   We also see the phenomenon known as "variable-splitting" [Sti81], as exemplified by the new u variables. Compared with line 4, we see that some of the "variableness" of p_1 has persisted past the assignment of $\langle k\ n \rangle$; this provides extra flexibility in the matching of $u \circ v_{k,n}$ with T, vis-a-vis the matching of $v_{k,n}$ with T. Variable-splitting is required (for the sake of completeness) whenever an associative operator (in this case, "$\circ$") is modelled. Further examples may be found in lines 6 and 8.

   Finally, we note that if both p_1 and p_2 represent (unassigned) pattern variables, then u and $v_{k,n}$ are both still (unassigned) variables at the point where the final match of $[u \circ v_{k,n}, T]$ occurs, and this causes an infinite recursion. The infinite solution set shown at the end of Chapter 8 is generated here. (A similar loop arises if p_2 is of the form $v \circ \mathit{anything}$.)

4) By contrast, this line is well-behaved. Either p_1 or p_2 will supply the value of n, or the match will fail immediately.

5) This line is based on rule R4. In an actual implementation, this case would be merged with line 2.

6) Since both P and T are in R+-normal form, p_1 will match a tuple iff p_1 is a variable. (See rule R8.) The variable p_1 is "split" into n new variables $v_{i,n}$ $(1 \le i \le n)$.

7) Due to the fact that P and T are R+-normalized, the lines 7 and 8 are sufficient to give complete-but-finite matches of $p_1 \circ p_2$ with $t_1 \circ t_2$.

(See rule R1.)

8) Since both P and T are in R+-normal form, p_1 can match a composition iff p_1 is a variable. (See rule R1.) The variable p_1 is "split" into two new variables, v_1 and v_2.

9.2.4. Completeness vs Termination

The algorithm Φ, as presented, fails to terminate for two reasons. The first is evident from a casual inspection of the algorithm: the infinite number of tuple axioms is reflected in those little "∞" symbols in a couple of the quantifiers. The second reason is more subtle; Φ does in fact generate all of the solutions mentioned at the end of Chapter 8.

We claim that there is a finite subset of $\Phi[p,t]$ that contains all substitutions in which we are truly interested. This subset contains only ground substitutions of the template variables. (We note that both types of unboundedness mentioned above introduce unbounded numbers of free variables into the m.g.u.'s.) Furthermore, we are only interested in assignments $<v \leftarrow t>$ where the term t is in R+-normal form; the terms t, id$\circ t$, id$\circ t\circ$id, and so on, are not considered to be distinct. We say that Φ is *sufficiently complete* iff it generates all possible distinct ground substitutions.

We can modify Φ so that it terminates, and is sufficiently complete. To accomplish this, we shall use 2nd-order terms to represent n-tuples, with the idea that we can usually infer the correct value(s) of the free metavariable n from the context. That is, it is often the case that a 2nd-order tuple is matched against a 1st-order tuple occurring in the pattern (or the target), and so the value of n is immediately determined.

Sometimes, however, when the pattern contains multiple occurrences of a variable, the appropriate value for n cannot be determined by a local analysis of the (current) pattern and (current) target; rather, a global arbitration mechanism is needed. For example, to match the pattern $[v_1\circ v,\ [v_2\circ v,\ v_3\circ v]]$ with the target $[+,[-,\times]]$, any one of these ground substitutions is sufficient:

(i)	$v_1 \leftarrow +$,	$v_2 \leftarrow -$,	$v_3 \leftarrow \times$,	$v \leftarrow$ **id**
(ii)	$v_1 \leftarrow \triangleleft 1\ 3 \triangleright$,	$v_2 \leftarrow \triangleleft 2\ 3 \triangleright$,	$v_3 \leftarrow \triangleleft 3\ 3 \triangleright$,	$v \leftarrow [+,-,\times]$
(iii)	$v_1 \leftarrow \triangleleft 2\ 3 \triangleright$,	$v_2 \leftarrow \triangleleft 1\ 3 \triangleright$,	$v_3 \leftarrow \triangleleft 3\ 3 \triangleright$,	$v \leftarrow [-,+,\times]$
(iv)	$v_1 \leftarrow +\circ\triangleleft 1\ 2 \triangleright$,	$v_2 \leftarrow \triangleleft 2\ 2 \triangleright$,	$v_3 \leftarrow \times\circ\triangleleft 1\ 2 \triangleright$,	$v \leftarrow [\mathbf{id},-]$

as are many others. We see that, although neither the pattern nor the target contains a 3-tuple, some of these solutions require that v be instantiated by a 3-tuple; but taken ***individually*** the recursive pattern-matching sub-problems of $\Phi[v_1 \circ v,\ +]$, $\Phi[v_2 \circ v,\ -]$ and $\Phi[v_3 \circ v,\ \times]$ have no way of determining this. In general, a variable which occurs in the pattern k times might require instantiation by a k-tuple, to be able to show k "different faces" to the world. A size of less than k may also suffice, as demonstrated in the last substitution above (iv). A size of greater than k must inevitably leave free variables in the substitution.

We propose a simple mechanism that determines a reasonable upper bound on the value of the meta-variable n in a 2^{nd}-order tuple.

We return our attention to the line labelled "3)" in the algorithm Φ. Given that the union quantifiers have been replaced by the direct use of 2^{nd}-order terms (i.e. $\prod_{i=1}^{n} v_{i,n}$ and $\langle k\ n \rangle$) with the free meta-variables n and k, it is sufficient that whenever p_2 is of the form 'v' or '$v \circ$*anything*', the pattern-matching sub-problem $\Phi[u \circ v_{k,n},\ T]$ be *suspended*, and the (unevaluated) *suspension* returned in the answer.

We name the suspensions after the leading variable in p_2, by saying that $\Phi[u \circ v_{k,n},\ T]$ and $\Phi[u \circ (v_{k,n} \circ \mathit{anything}),\ T]$ are $v_{k,n}$-*suspensions*. We define an operator $\#(v_{k,n},\ s)$, which counts the number of $v_{k,n}$-suspensions in the substitution s.

When the *entire* (top-level) matching operation is done, then the function *ARBITER* is invoked on the resulting set of substitutions (suspensions and all) to return the desired answer: a finite sufficiently-complete set of ground substitutions.

```
letrec $ARBITER\ (S) \equiv$
    if $(\exists s \in S)\ (\exists v_{k,n})\ \#(v_{k,n}, s) \geq 1$
        then $ARBITER\ (S - \{s\}) \cup arbiter\ (s)$
        else $S$
    fi
end
where $arbiter\ (s) \equiv$
    if $(\exists\ v_{k,n})\ \#(v_{k,n}, s) \geq 2$
        then $ARBITER \left[ \bigcup_{n=2}^{\#(v_{k,n},s)} \bigcup_{k=1}^{n} RESTART\ (s) \right]$
        else $\{\ \}$
    fi
```

The function $RESTART(s)$ returns the set of all substitutions generated by resuming the evaluation of the $v_{k,n}$-suspensions in s, with the n and k now bound to the values indicated by the quantifiers. It is expected (in fact, guaranteed) that after chugging on for a while the computation will produce not only new ground substitutions but also substitutions containing new suspensions. These new suspensions are *not* automatically restarted, although they might be resumed later by recursive invocations of *ARBITER*.

The function *ARBITER* does eventually terminate, when all of the $v_{k,n}$-suspensions in every $s \in S$ have $\#(v_{k,n},s)=1$. *ARBITER* splits variables which have multiple occurrences, until all suspensions involve unique variables. At this point we stop, because continuing would either introduce free variables into the final substitutions, or generate substitutions that contain 1-tuples $[f]$ (a.k.a. f) and the selector $\triangleleft\ 1\ 1 \triangleright$ (a.k.a. **id**), which immediately **R+**-reduce to already-discovered substitutions.

9.3. Condition

As mentioned in the introduction to this chapter, the functional form of condition is hard to deal with. To do it justice, this author feels that we must incorporate knowledge of Boolean algebra into our equational theory (hence the axioms A13..A15), and generally model, in decision-tree form, the various Boolean axioms such as commutativity, associativity, idempotence, and subsumption. Theories modelling such axioms cannot be both noetherian and confluent; $\mathbf{R^*} \cup$ {R16..R18} is not noetherian, essentially because R17 and R18 simulate the commutativity of logical conjunction.

We desire a pattern-matching algorithm that is conversant with these various equivalences.

We have such an algorithm. It always terminates, with a complete set of solutions (assuming of course that Φ does also). However, the algorithm was not designed to serve as the basis for an actual implementation; it was intended that it serve as a benchmark of the inherent difficulties involved in obtaining completeness. It is *incredibly* slow, partly because it adopts an excessively low-level viewpoint on the problem, partly because it returns solutions which are in some sense equivalent, and partly because some matching problems actually have vast quantities of distinct solutions.

9.3.1. Notation & Terminology

In the subsequent sections, we shall assume that all terms are **R***-normalized; this means that each one may be viewed as a decision tree with all of the condition operators near the root and all non-condition subterms at the leaf positions. We model a decision tree (i.e., term) containing n distinct predicates as an n-dimensional hypercube (decision table). In such a cube, the each of the n axes is labelled by a predicate, and the 2^n points inside the hypercube are condition-free subterms.

The expression $|f|$ denotes the number of distinct predicates in the term f; or, alternatively, the number of dimensions in the hypercube f.

The expression $\pi(f,n)$ represents the n^{th} axis label for the hypercube f, in some arbitrary enumeration of the $|f|$ axes. Similarly, the expression $\xi(f,n)$ denotes the n^{th} point in the cube, in some arbitrary enumeration of the $2^{|f|}$ points.

9.3.2. Least (and other) Upper Bounds of Hypercubes

When matching a pair $[p_0, t_0]$ of hypercubes, we require that both have the same dimensionality. In practice, the target is usually much larger than the pattern. The first step, then, is to find some q_0 such that both the pattern p_0 and target t_0 may be converted into q_0-dimensional hypercubes. All else being equal, the smallest such q_0 would be the preferred one. However, in order to generate all solutions to the matching problem, we must actually generate a (rather large) set of pairs $[p_i, t_i]$ of q_i-dimensional hypercubes.

Observation: For any term f with $v \in V(f)$ and $v_1, v_2, v_3 \notin V(f)$, the term obtained from f by substituting $(v_1 \rightarrow v_2 ; v_3)$ for all occurrences of v has at least one more predicate than does f. (If v occurs more than once in f, or occurs in a predicate, then the new term may have *several* more predicates than f.) Reducing the new term to any $\mathbf{R}^*$-normal form then converts it into a decision tree (hypercube).

For example, performing this substitution on the term $[v\circ\times,\ v\circ\div]$ results in the new term

$$[(v_1 \rightarrow v_2\,;\, v_3)\circ\times,\ (v_1 \rightarrow v_2\,;\, v_3)\circ\div],$$

whose $\mathbf{R}^*$-normal forms are the decision trees

$$\begin{array}{l}
((v_1\circ\times) \rightarrow \\
\quad\quad ((v_1\circ\div) \rightarrow \\
\quad\quad\quad\quad [v_2\circ\times,\ v_2\circ\div]; \\
\quad\quad\quad\quad [v_2\circ\times,\ v_3\circ\div])\ ; \\
\quad\quad ((v_1\circ\div) \rightarrow \\
\quad\quad\quad\quad [v_3\circ\times,\ v_2\circ\div]; \\
\quad\quad\quad\quad [v_3\circ\times,\ v_3\circ\div])\ )
\end{array}$$

and

$$\begin{array}{l}
((v_1\circ\div) \rightarrow \\
\quad\quad ((v_1\circ\times) \rightarrow \\
\quad\quad\quad\quad [v_2\circ\times,\ v_2\circ\div]; \\
\quad\quad\quad\quad [v_3\circ\times,\ v_2\circ\div])\ ; \\
\quad\quad ((v_1\circ\times) \rightarrow \\
\quad\quad\quad\quad [v_2\circ\times,\ v_3\circ\div]; \\
\quad\quad\quad\quad [v_3\circ\times,\ v_3\circ\div])\ )
\end{array}$$

which contain the new predicates $(v_1\circ\times)$ and $(v_1\circ\div)$.

Our strategy, then, will be to first increase the size of the pattern, through exhaustive instantiation of its variables (as above), yielding new pairs $[p_i,\ t_i]$, until each pattern p_i has *at least* as many (distinct) predicates as does its target t_i. Since the size of a pattern may increase in an irregular manner, it may not be possible to have all of the inflated patterns p_i possess exactly $|t_i|$ predicates. Therefore, the second phase of the process will be to increase the size of each target to match the size of its pattern. (The inflation of a target *is* a smooth process; it will always be possible to get an exact match.)

In the subsequent function definitions, it is assumed that the symbols v_0, v_1, v_2, and v_3 represent new variables every time the algorithm encounters them; some mechanism like LISP's *gensym* is presumed.

9.3.3. Our Pattern-Matching Function Ω

At this point, we introduce our function $\Omega \in$ *Patterns* × *Targets* → {*Substitutions*}, and explain the nature of its constituent parts in the subsequent sections.

$\Omega[p,t]$ =

$$\bigcup_{n=0}^{|t|-|p|} \left[\{[p,t]\} \mathop{@}_{i=1}^{n} PUMP \right] \text{ @ } FILTER \text{ @ } EXPAND \text{ @ } EMBED \text{ @ } MATCH$$

9.3.3.1. PUMP

The function *PUMP* inflates a pattern term p, by applying the instantiation trick mentioned above, substituting $(v_1 \to v_2\,; v_3)$ for all occurrences of some variable $v_k \in V(p)$. Each resulting term p_k is then converted to a decision tree by reducing it to any one of its $\mathbf{R}^*$-normal forms.

PUMP is only employed, of course, when $|t| - |p| > 0$.

As noted previously, instantiating one variable v_k in p may cause the creation of an arbitrary number of new predicates in p_k, perhaps enough that no further pumping instantiations are required: $|p_k| \geq |t|$. On the other hand, it is possible that all but one of these new predicates may be matched against "don't care" predicates of the target; for any instantiation, we can only guarantee that *one* new predicate will be matched against a "real" predicate of the target. (If *all* new predicates resulting from a pumping instantiation are matched against "don't care" predicates, then pumping on v_k was superfluous.) Thus, the n-fold composition of *PUMP*, applied to p, produces patterns which are guaranteed to match at least $n+|p|$, but no more than $|t|$, of the "real" predicates in t.

let $PUMP[p,t]$ =

$$\bigcup_{v \in V(p)} \textbf{let } \mu = \{<v \leftarrow (v_1 \to v_2; v_3)>\} \textbf{ in } \{[\delta(\mathbf{R}^*, \mu(p)), t]\}$$

end

9.3.3.2. FILTER

The function *FILTER* weeds out undersized patterns, ones possessing fewer predicates than their (target) mates.

```
let $FILTER[p,t]$ =
    if $|p| \geq |t|$ then $\{[p,t]\}$ else $\{\,\}$ fi
end
```

9.3.3.3. EXPAND

The function *EXPAND* maps pairs of hypercubes $[p_i, t_i]$ where $|p_i| \geq |t_i|$ to pairs $[p_i, t_i']$ such that $|p_i| = |t_i'|$. If $|p_i| > |t_i|$, then we demand that t_i' exhibit the following behavior:

(1) $t_i' \equiv_{\mathrm{mdFP}} t_i$;

(2) $|t_i'| > |t_i|$.

These goals are easily attained by letting $t_i' = (v_0 \rightarrow t_i \ ; t_i)$ for some new variable v_0. The new variable will eventually be matched against a corresponding "don't care" predicate of the pattern. The expansion of the target is repeated until the sizes of the terms are equal.

```
letrec $EXPAND[p,t]$ =
    if $|p| > |t|$
        then $EXPAND\ [p, (v_0 \rightarrow t \ ; t)]$
        else $\{[p,t]\}$
    fi
end
```

9.3.3.4. EMBED

For every pair $[p_i, t_i]$ of q_i-dimensional hypercubes, there are $q_i! \times 2^{q_i}$ distinct ways to embed one into the other. The factor of $q_i!$ is the number of ways to put the axes of p_i in 1-1 correspondence with the axes of t_i; and the factor of 2^{q_i} is the number of ways to match the "minimal" element of p_i (i.e., the unique case that is selected when all predicates yield **false**) with that of t_i. Our algorithm will, of course, have to try *all* of these possible permutations, for the sake of completeness. Things could have been worse; if our terms were merely unstructured *sets* of 2^{q_i} subterms rather than decision tables, then there would be $2^{q_i}!$ embeddings of p_i into t_i.

We let $Perm(p)$ denote the set of all $|p|! \times 2^{|p|}$ permutations of the hypercube p.

$$EMBED[p,t] = \bigcup_{p' \in Perm(p)} [p',t]$$

9.3.3.5. MATCH

For each pair $[p_i, t_i]$ of fully-aligned hypercubes, the procedure is simple. We need only run through all $|p_i|$ pairs of predicates, and all $2^{|p_i|}$ pairs of points (condition-free subterms), applying the function Φ to each such pair and accumulating the set of substitutions as we go.

$$MATCH[p,t] = \{\phi\} \overset{|p|}{\underset{n=1}{\triangleright}} [\pi(p,n),\pi(t,n)] \overset{2^{|p|}}{\underset{n=1}{\triangleright}} [\xi(p,n),\xi(t,n)]$$

9.3.4. Time Complexity

The complexity of the algorithm Ω is so bad that in the next chapter we suggest using NP-complete heuristics (!), to speed up the matching.

CHAPTER 10

Enhancements

10.1. Eliminating Template Variables

The effort expended by the algorithm Φ is proportional to the number of distinct variables occurring in the pattern; each variable gives Φ the chance to find more solutions. Therefore, it is advantageous to formulate the schemata carefully, avoiding unnecessary generalizations.

For example, we have already encountered the transformation schema

$$\textbf{define } f \equiv (p \rightarrow g \; ; h \circ [i, f \circ j])$$

with the enabling conditions:

(1) $g \circ j \sqsubseteq g$

(2) $[\exists h'] \; (/ \; h \; g) \equiv (\backslash \; h' \; g)$

If both constraints are satisfied, then the definition of f may be stated non-recursively:

define $f \equiv$

$$\langle 1\ 2 \rangle \circ (\textbf{while } \text{not} \circ p \circ \langle 2\ 2 \rangle \textbf{ do } [h' \circ [\langle 1\ 2 \rangle, i \circ \langle 2\ 2 \rangle], j \circ \langle 2\ 2 \rangle]) \circ [g, \text{id}]$$

When this example was given earlier, instead of (1) we demanded that g be a constant function [Coo66, DaB76, KiS81], and in place of (2) we used the simpler but more stringent constraint that h had to be associative and commutative [DaB76]. In such cases $h' \equiv h$. Condition (2) is equivalent to the following requirement:

(2′) $(\exists h' \in \mathit{Exp}) \; (\forall a, b, c \in \mathit{Exp})$

$$h \circ [a, h' \circ [b, c]] \equiv h' \circ [h \circ [a, b], c] \;\wedge\; h \circ [a, g] \equiv h' \circ [g, a]$$

This version (2′) is due to Cooper [Coo66], but was re-discovered by Kieburtz & Shultis [KiS81]; in both cases it was motivated by the special case wherein g is constant.

We call the functions h and h' a pair of *associative duals* with respect to g. Perhaps the most famous such functions (in which $h \neq h'$) are the list constructors over a domain D:

$$h \equiv \text{appendL} \in D \times D^* \to D^*$$
$$h' \equiv \text{appendR} \in D^* \times D \to D^*$$
$$g \equiv \bar{\varepsilon} \in D^*$$

so that, for instance,

$$\text{appendL}:[4,\varepsilon] = <4> = \text{appendR}:[\varepsilon,4]$$

and

$$\text{appendL}:[3,\text{appendR}:[<4>,5]] = <3,4,5> = \text{appendR}:[\text{appendL}:[3,<4>],5]$$

It is the case that the choice of i in the 'factorial template' does not affect the satisfaction of the constraints. We note that the subterm

$$h \circ [i, f \circ j]$$

may be factored into

$$(h \circ [i_1 \circ \triangleleft 1\ 2 \triangleright, \triangleleft 2\ 2 \triangleright]) \circ [i_2, f \circ j]$$

where $i_1 \circ i_2 \equiv i$. From the point of view of satisfying schema constraint (2), one factorization of i is as good as any other.

Proposition:

$$(/\ h\ g) \equiv (\backslash\ h'\ g) \iff$$
$$\forall i_1\ (/\ \ h \circ [i_1 \circ \triangleleft 1\ 2 \triangleright, \triangleleft 2\ 2 \triangleright]\ \ g) \equiv (\backslash\ \ h' \circ [\triangleleft 1\ 2 \triangleright, i_1 \circ \triangleleft 2\ 2 \triangleright]\ \ g)$$

Proof:

($\Rightarrow$)

The terms $h \circ [i_1 \circ \triangleleft 1\ 2 \triangleright, \triangleleft 2\ 2 \triangleright]$ and $h' \circ [\triangleleft 1\ 2 \triangleright, i_1 \circ \triangleleft 2\ 2 \triangleright]$ are shown to be associative duals with respect to g by direct substitution into the equalities of constraint 2', combined with the simplification of both sides of each equation to R+-canonical form.

$$\begin{aligned}
&(h \circ [i_1 \circ \triangleleft 1\ 2 \triangleright, \triangleleft 2\ 2 \triangleright]) \circ [a, (h' \circ [\triangleleft 1\ 2 \triangleright, i_1 \circ \triangleleft 2\ 2 \triangleright]) \circ [b, c]] \\
&\equiv\ h \circ [i_1 \circ a, h' \circ [b, i_1 \circ c]] \\
&\equiv\ h' \circ [h \circ [i_1 \circ a, b], i_1 \circ c] \\
&\equiv\ (h' \circ [\triangleleft 1\ 2 \triangleright, i_1 \circ \triangleleft 2\ 2 \triangleright]) \circ [(h \circ [i_1 \circ \triangleleft 1\ 2 \triangleright, \triangleleft 2\ 2 \triangleright]) \circ [a, b], c]
\end{aligned}$$

and

$$\begin{aligned}&(h\circ[i_1\circ\langle 1\ 2\rangle,\ \langle 2\ 2\rangle])\circ[a,g]\\ \equiv\ & h\circ[i_1\circ a,\ g]\\ \equiv\ & h'\circ[g,\ i_1\circ a]\\ \equiv\ & (h'\circ[\langle 1\ 2\rangle,\ i_1\circ\langle 2\ 2\rangle])\circ[g,a].\end{aligned}$$

($\Leftarrow$)

Trivial; choose $i_1 \equiv \mathrm{id}$. Then

$$\begin{aligned}&h\circ[\mathrm{id}\circ\langle 1\ 2\rangle,\ \langle 2\ 2\rangle]\\ \equiv\ & h\circ[\langle 1\ 2\rangle,\ \langle 2\ 2\rangle]\\ \equiv\ & h\circ\mathrm{id}\\ \equiv\ & h\end{aligned}$$

and similarly for h'. ▪

Therefore, we may arbitrarily assign $<i_2 \leftarrow \mathrm{id}>$ when formulating the transformation schema, eliminating the variable altogether. We conclude that the schema templates should be:

$$\textbf{define}\ f \equiv (p \rightarrow g\ ;\ h\circ[\mathrm{id},\ f\circ j])$$

and

$$\textbf{define}\ f \equiv \langle 1\ 2\rangle \circ (\textbf{while}\ \mathrm{not}\circ p\circ\langle 2\ 2\rangle\ \textbf{do}\ [h',\ j\circ\langle 2\ 2\rangle])\circ[g,\ \mathrm{id}]$$

with the same constraints (1) and (2) as above.

10.2. Impossible Hypercube Embeddings

At the stage where we are about to *MATCH* (i.e., apply Φ to) a pair of q_i-predicate decision trees, we may first do some pre-processing to attempt to invalidate the match (or rather, the chosen embedding) before getting down to the nitty-gritty details. We have two suggestions:

(1) We have already noted that if *all* of the new predicates created by a single pumping instantiation of a pattern variable v are paired against don't-care predicates of the target, then pumping on v accomplished nothing. This suggests that we keep track of the genealogy of each predicate, and decline to pair all members of any family against phantoms.

(2) It is clear that if there exist two (or more) identical points (condition-free subterms) in the pattern hypercube, then in order that there may be any chance at all of matching the entire cubes, we must pair these identical pattern subterms with target subterms that are likewise identical (any substitution applied to identical terms yields identical

terms). We also note that our cube-inflation procedures do indeed sometimes introduce such identical twins into the pattern and (independently) into the target. If the twins do not "line up", then the match can be aborted immediately.

10.3. Careful Expansion of Hypercubes

It is clearly evident that the most exorbitant expenditure of resources occurs in the generation of myriad pairs of hypercubes. In some cases, including those noted above, much of this effort is unnecessary and therefore wasted. We would prefer to take advantage of various symmetries in the normalization process, to predict in advance which expansions of hypercubes would be fruitless to pursue.

10.4. Application-Specific Constraints

In the intended application of the pattern-matcher, namely the recognition of instances of certain types of recursive definitions, we can impose a useful restriction on the values that may be assigned to variables. If our template is a function definition, say **define** $f \equiv Q(f)$ for some form (higher-level function) Q, then we prohibit the assignment of terms containing f to any variables occurring in Q. This allows us to immediately discard many hypercube embeddings, in which a target subterm contains an occurrence of f, and the corresponding pattern subterm does not. The justification of this restriction is that the presence or absence of the parameter f is critical in distinguishing one recursive schema from another, and the template instantiator is only permitted to fiddle with "inessential" differences between the function intensions. This idea may be generalized, to involve *sets* of distinguished parameter symbols.

CHAPTER 11

Summary

11.1. What Has Been Accomplished

The algorithm Ω pattern-matches FP templates with programs, returning sufficiently-complete sets of unifying ground substitutions.

The pattern-matching function Φ, modified as described, can serve as the basis for an actual implementation. Consideration of the problems introduced by the (rather ambitious) requirements for handling the condition operator have clarified some issues in the mind of this author.

11.2. What Will Be Attempted

It will probably be useful to further modify Φ to use a "lazy evaluation" paradigm [HeM76], to produce substitutions one-at-a-time rather than all-or-nothing. That way, a program transformer can keep requesting the next substitution until some transformation's constraints are satisfied, and then stop.

The method of handling the condition functional form used in Ω, based on inflations of decision trees/hypercubes, will be retired, due to its profligate use of resources. Work on its successor Ω' is in progress; the approach being developed now resembles AI goal-directed heuristic searching.

The algorithm Ω' will be implemented, probably in LISP. A rudimentary program transformation system is planned, although we do not presently have a theorem prover (constraint verifier) available to us.

11.3. Acknowledgements

The author is grateful to Professor Richard B. Kieburtz for sharing his advice, ideas, and perspectives.

References

[Bac78]
BACKUS, J. *Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs.* Communications of the ACM 21 8 (1978), pp 613-641.

[Bal81]
BALZER, R. *Transformational implementation: an example.* IEEE Transactions on Software Engineering SE 7 1 (1981), pp 3-14.

[Bir77]
BIRD, R.S. *Notes on recursion elimination.* Communications of the ACM 20 6 (1977), pp 434-439.

[BuD77]
BURSTALL, R.M. and DARLINGTON J. *A transformation system for developing recursive programs.* Journal of the ACM 24 1 (1977), pp 44-67.

[Coo66]
COOPER, D.C. *The equivalence of certain computations.* Computer Journal 9 (May 1966), pp 45-52.

[Dar81]
DARLINGTON, J. *An experimental program transformation and synthesis system.* Artificial Intelligence 16 (1981), pp 1-46.

[DaB76]
DARLINGTON, J. and BURSTALL R.M. *A system which automatically improves programs.* Acta Informatica 6 (1976), pp 41-60.

[Der82]
DERSHOWITZ, N. *Orderings for term-rewriting systems.* Theoretical Computer Science 17 (1982), 279-301.

[DeM79]
DERSHOWITZ, N. and MANNA, Z. *Proving termination with multiset orderings.* Communications of the ACM 22 8 (1979), pp 465-476.

[FaM82]
FATEMAN, R.J. and MOSES, J. **MACSYMA Primer for VAX/UNIX** University of California at Berkeley (1982).

[Fea82]
FEATHER, M.S. *A system for assisting program transformation*. ACM Transactions of Programming Languages and Systems 4 1 (1982), pp 1-20.

[Fod80]
FODERARO, J.K. **The FRANZ LISP Manual**. University of California at Berkeley (1980), Chapter 12.

[HeM76]
HENDERSON, P. and MORRIS, J.H. *A lazy evaluator*. Proceedings of the 3rd ACM Symposium on the Principles of Programming Languages (1976), pp 95-103.

[Hue80]
HUET, G. *Confluent reductions: abstract properties and applications to term rewriting systems*. Journal of the ACM 27 4 (1980), pp 797-821.

[HuL78]
HUET, G. and LANKFORD, D.S. *On the uniform halting problem for term rewriting systems*. Rapport Laboria 283, IRIA (1978).

[HuO80]
HUET, G. and OPPEN, D.C. *Equations and rewrite rules: a survey*. Stanford University TR STAN-CS-80-785 (1980).

[Hul80]
HULLOT, J. *Canonical forms and unification*. **Lecture Notes in Computer Science**. Springer-Verlag (1980), pp 319-334.

[Ive62]
IVERSON, K. **A Programming Language**. Wiley (1962).

[JoL82]
JOUANNAUD, J. and LESCANNE, P. *On multiset orderings*. Information Processing Letters 15 2 (1982), pp 57-63.

[KiS81]
KIEBURTZ, R.B. and SHULTIS, J.C. *Transformations of FP program schemes*. 1981 Conference on Functional Programming Languages and Computer Architecture, A.C.M. (Oct 1981), pp 41-48.

[KnB70]
KNUTH, D.E. and BENDIX, P. *Simple word problems in universal algebras*. **Computational Problems in Abstract Algebra**, Pergamon Press (1970), pp 263-297.

[Lan79]
LANKFORD, D.S. *On proving term rewriting systems are noetherian.* Louisiana Tech. University, Report MTP-2 (1979).

[LaB77]
LANKFORD, D.S. and BALLATYNE, A.M. *Decision procedures for simple equational theories with commutative-associative axioms: complete sets of commutative-associative reductions*. University of Texas, Mathematics Dept Technical Report (Aug 1977).

[LSn77]
Lipton, R.J. and SNYDER, L. *On the halting of tree replacement systems*. Proceedings of the Waterloo Conference on Theoretical Computer Science, University of Waterloo (1977), pp 43-46.

[LiS76]
LIVESEY, M. and SIEKMANN, J. *Unification of A+C terms (bags) and A+C+I terms (sets)*. Intern. Ber. Nr. 5/76, Institut fur Informatik I, Universitat Karlsruhe, W. Germany (1976).

[LoF81]
LONDON, P. and FEATHER, M.S. *Implementing specification freedoms*. USC/ISI Research Report #81-100 (1981).

[Mak77]
MAKANIN, G.S. *The problem of solvability of equations in a free semigroup*. Soviet Akad. Nauk SSSR 233 2 (1977).

[Man74]
MANNA, Z. **Mathematical Theory of Computation**. McGraw-Hill Inc. (1974).

[MaS75]
MANNA, Z. and SHAMIR, A. *The optimal fixedpoint of recursive programs*. Proc. Symp. on Theory of Computing, A.C.M. (1975).

[MaS76]
MANNA, Z. and SHAMIR, A. *The theoretical aspects of the optimal fixed-point*. SIAM Journal of Computing 5 3 (1976), pp 414-426.

[MaW81]
MANNA, Z. and WALDINGER, R. *Deductive synthesis of the unification algorithm*. Science of Computer Programming 1 (1981), pp 5-48.

[PaK82]
PAIGE, R. and KOENIG, S. *Finite differencing of computable expressions*. ACM Transactions on Programming Languages and Systems 4 3 (1982), pp 402-454.

[PeS81]
PETERSON, G.E. and STICKEL, M.E. *Complete sets of reductions for some equational theories*. Journal of the ACM 28 2 (1981), pp 233-264.

[PeB82]
PETTOROSSI, A. and BURSTALL, R.M. *Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique*. Acta Informatica 18 (1982), pp 181-206.

[Pla78a]
PLAISTED, D. *Well-founded orderings for proving termination of systems of rewrite rules*. University of Illinois, Computer Science TR 78-932 (1978).

[Pla78b]
PLAISTED, D. *A recursively-defined ordering for proving termination of term rewriting systems*. University of Illinois, Computer Science TR 78-943 (1978).

[Rob65]
ROBINSON, J.A. *A machine-oriented logic based on the resolution principle*. Journal of the ACM 12 1 (1965), pp 32-41.

[Ste78]
STEELE, G.L. **RABBIT: A Compiler for SCHEME**. Ph.D. Thesis, M.I.T. (1977).

[Sti81]
STICKEL, M.E. *A unification algorithm for associative-commutative functions*. Journal of the ACM 28 3 (1981), pp 423-434.

[Sto77]
STOY, J.E. **Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory**. M.I.T. Press (1977).

[SMR75]
STRONG, H.R. and MAGGIOLO-SCHETTINI, A. and ROSEN, B.K. *Recursion structure simplification*. SIAM Journal of Computing 4 3 (1975), pp 307-320.