The Specificity Rule for Lazy Pattern-Matching in Ambiguous Term Rewrite Systems

Richard Kennaway
School of Information Systems, University of East Anglia
Norwich NR4 7TJ, U.K.

Abstract

Many functional languages based on term rewriting (such as Miranda[1] and ML) allow the programmer to write ambiguous rule systems, with the understanding that rules will be matched against a term in the order in which the rules are written, and that the pattern-matching of a rule against a term proceeds from left to right.

This gives a precise semantics to such ambiguous systems, but it has disadvantages. It depends on the textual ordering of the program, whereas the standard theory of term rewriting has no such concept. As a result, equational reasoning is not always valid for this semantics, defeating the primary virtue of functional languages. The semantics also fails to be fully lazy, in that sometimes a non-terminating computation will be performed on a term which has a normal form.

We define a rule, called *specificity*, for computation in ambiguous term rewrite systems. This rule (really a meta-rule) stipulates that a term rewrite rule of the system can only be used to reduce a term which matches it, if that term can never match any other rule of the system which is more specific than the given rule. One rule is more specific than another if the left-hand side of the first rule is a substitution instance of the second, and the reverse is not true. Specificity captures the intuition underlying the use of ambiguity in ML and Miranda, while also providing lazy pattern-matching.

A natural generalisation of the idea provides a semantics for Miranda's lawful types.

1. Introduction

Van voorwaarts naar achter, van links naar rechts...
From forward to back, from left to right...
— Dutch nursery rhyme

The elegance and usefulness of functional programming lie in the fact that a functional program can be read as a piece of mathematics. In many functional languages, a program consists of a set of type declarations defining some domains of values, a set of axioms asserting that certain of these values are equal, and a term; executing the program amounts to proving that term equal to some term which possesses a printable representation. Proving properties of such a functional program may be performed by equational reasoning using those same axioms (together with induction principles).

Functional programming languages in this style include SASL [20], KRC [21], Miranda [22,23], ML [10], Lazy ML [1], Hope [6], Hope+ [18], Clean [5], and most recently, Haskell [11]. However, closer inspection reveals that for all of these languages, the simple picture sketched above is not accurate. An example is provided by everyone's favorite toy functional program: the factorial.

Example 1.

```
fac 0 = 1                    (1.1)
fac n = n * (fac (n-1))      (1.2)
```

[1] Miranda is a trademark of Research Software Ltd.

This is written in Miranda's syntax, but the point applies equally to the other languages mentioned. We assume that other rules are also present in the system to perform arithmetic. Here are the steps which a Miranda implementation performs to evaluate the expression (fac 2).

fac 2	→ 2*(fac (2−1))	(1.2)
	→ 2*(fac 1)	(arithmetic)
	→ 2*(1*(fac (1−1)))	(1.2)
	→ 2*(1*(fac 0))	(arithmetic)
	→ 2*(1*1)	(1.1)
	→ 2*1	(arithmetic)
	→ 2	(arithmetic)

But all is not as it appears. Consider the passage from 2*(fac (2−1)) to 2*(fac 1). This was performed by invoking the arithmetic rule for (2−1), replacing that expression by 1. However, if the rules are considered as a conventional term rewrite system, there is another possibility. We can apply rule (2) to the subterm (fac (2−1)), to obtain 2 * ((2−1) * (fac ((2−1)−1))). We could then apply the same rule to the subterm (fac ((2−1)−1)), obtaining 2 * ((2−1) * (((2−1)−1) * (fac (((2−1)−1)−1)))). If we now decide to do the arithmetic, then after a few reductions we get (2*(1*(0*(fac (−1))))). Depending on whether or not the rules for multiplication can rewrite a term 0*t to 0 without requiring t to be an integer, this term either reduces to 0, or cannot be reduced to any normal form.

Similarly, the term (fac 0) is intended to be rewritten to 1 by rule 1.1, yet it matches both 1.1 and 1.2, and could be rewritten to 0*(fac (0−1)) instead, which again will either rewrite to 0 or fail to terminate.

Thus the semantics of a program in Miranda, or any of the other languages mentioned above, is not simply the declarative semantics of a term rewrite system, but has an essential operational part: the reduction strategy which specifies which redex is to be reduced at each step. For lazy languages, such as Miranda, LML, and Haskell, the reduction strategy can be described as the following meta-rule. For the moment, we ignore Miranda's lawful types.

(1) The rules are to be matched against a term to be evaluated in the order in which they appear in the program.

(2) For each rule, the matching of its left hand side is performed from left to right, evaluating subterms as required by (3).

(3) If during pattern matching one attempts to match a constructor (i.e. a basic value or one of the tags of a user-defined type) in the pattern against a subterm whose principal function symbol is not a constructor, then the subterm is evaluated to constructor form before re-attempting the match. If the constructor form of the subterm has a different constructor than that appearing in the rule, the rule fails to match and the next rule is tried.

This "top-to-bottom left-to-right" pattern-matching strategy is common to Miranda, Lazy ML [LML], and Haskell. ML and Hope have "strict" semantics: all arguments to a function are evaluated before any pattern-matching is done. As a result, the order of pattern-matching within a rule has no effect on the semantics, but the order of rules is still significant. The above example behaves the same way in strict languages as it does in lazy languages. We are primarily interested in lazy semantics, and will not further discuss strict languages.

The functional programmer quickly becomes accustomed to this use of textual order, but it is not a trivial point. The presence of a particular strategy causes equational reasoning about the program to be not always valid. In the above example the problems stemmed from the ambiguity of the rule-systems.

However, the reduction strategy can cause problems even for regular (or, as it may shortly become known [Klo9?], *orthogonal*) rule-systems.

Example 2.

```
list ::= Nil | Cons num list
f Nil Nil          = 1          (3.1)
f Nil (Cons x y)   = 2          (3.2)
f (Cons x y) z     = 3          (3.3)
g Nil Nil          = 1          (3.4)
g (Cons x y) Nil   = 2          (3.5)
g z (Cons x y)     = 3          (3.6)
c x y z            = x z y      (3.7)
loop               = loop       (3.8)
```

This rule-system is orthogonal, and even strongly sequential. Since g is just f, but taking the arguments in the opposite order, and c is the argument-switching combinator, we might expect c g = f to hold. But with Miranda's top-to-bottom, left-to-right evaluation strategy, we find that (f (Cons 1 Nil) loop) fails to match 3.1 and 3.2, matches rule 3.3, and is reduced to 3, but (c g (Cons 1 Nil) loop) reduces first to (g loop (Cons 1 Nil)), and then attempted matching against 3.4 invokes evaluation of loop, which fails to terminate. Thus even for well-behaved rule-systems, the built-in reduction strategy complicates proofs of properties of programs.

Top-to-bottom left-to-right pattern-matching is useful in allowing the programmer to write default rules without having to explicitly exclude from the default case all the previous cases. However, we have seen that it muddies the semantics. It also results in the ordering of the arguments to a function playing two different and conflicting roles: it directs the pattern-matching, but the programmer may also want the order of arguments to reflect their meaning, grouping related arguments together.

We will describe an alternative reduction strategy, called *specificity*, which will allow programmers to write "default" rules as in the factorial example, but without sacrificing equational reasoning. The meaning of such systems will be described by a transformation into another system which does not have such ambiguities, and to which equational reasoning is applicable.

2. Specificity

For orthogonal strongly sequential systems, Huet and Lévy showed a long time ago how to perform lazy pattern-matching independently of textual ordering [12]. They describe a reduction strategy for such systems which always selects a "needed" redex, i.e. one which must be reduced to reach the normal form of the whole term. They prove that this strategy always finds the normal form of any term which has one. Example 2 is such a system; the semantics given to it by their strategy satisfies c g = f, as desired.

We take that work for granted, and will describe a semantics for ambiguous rule-systems such as example 1, by translating them into orthogonal, strongly sequential systems, to which the Huet-Lévy strategy may then be applied.

Intuitively, the reason that the term (fac 0) should be considered to match the rule (1.1) and not (1.2) is that (1.1) is "more specific" than (1.2). If the programmer had intended (fac 0) to match (1.2), it was superfluous to write the rule (1.1). The reason that (fac (2−1)) should not be considered to match (1.2) is that the subterm (2−1) is capable of further evaluation which may, for all we know (without performing some sort of look-ahead) result in 0, causing rule (1.1) to match. Reduction of (fac (2−1)) should therefore be postponed pending further evaluation of the subterm (2−1). This is the basic idea of specificity.

A simple generalisation of this idea can also be used to give a semantics for Miranda's lawful types.

3. Definitions and notations

We assume familiarity with the basic concepts of term rewriting (see, e.g. [13,14]), and will only define our notations and the key concepts in our treatment of specificity. A term has the form $F(t_1,\ldots,t_n)$, where $t_1,\ldots,t_n$ ($n\geq 0$) are terms. A subterm of a term may be specified by an *address*, a finite sequence of positive integers, in an obvious way. Given a term $t = F(t_1,\ldots,t_n)$ and an address $u = i{\cdot}v$ (i an integer, v an address), t/u is the term t_i/v. $\langle\,\rangle$ is the empty address; $t/\langle\,\rangle = t$. If u is an address of a subterm of t, $t[u:=t']$ is the term obtained by replacing that subterm of t by t'. An address u of t is *proper* if t/u is not a variable.

A *substitution* is a function σ from some finite set of variables to terms. $\sigma(t)$ is the result of replacing every occurrence of a variable x in t by $\sigma(x)$. An *address substitution* is a similar function defined on a finite set of addresses, and is applied to a term in the obvious way.

We write $t \leq_s t'$ if t' is a substitution instance of t. t and t' are *unifiable* if there is a t'' such that $t \leq_s t'' \geq_s t'$. We write $t \uparrow_s t'$.

These definitions extend to rewrite rules in the obvious way: if R_1 is $t_1 \rightarrow t'_1$ and R_2 is $t_2 \rightarrow t'_2$, then $R_1 \leq_s R_2$ iff for some substitution σ, $\sigma(t_1) = t_2$ and $\sigma(t'_1) = t'_2$.

A term is *linear* if no variable occurs more than once in it. A *closed* term is a term containing no variables. When dealing with linear terms, the identity of variables will often be unimportant. We may use the symbol • to indicate an occurrence of an unspecified variable, different from every other variable in the term, and $\underline{\bullet}$ to represent a tuple of distinct unspecified variables.

A *matching* of a term t into a term t' at address u of t' is a variable substitution σ such that $\sigma(t) = t'$/u. An address v of t' is *matched by* this matching if u≤v, v/u is an address of t, and t/(v/u) is not a variable.

3.1. Preorderings

We shall be defining several preorderings besides the $\leq_s$ defined above. For any preordering $\leq_x$, we define the associated relations. $\geq_x$, $<_x$, and $>_x$ in the obvious way. $\sqcup_x$ is the least upper bound operator (defined up to equivalence in the preordering). We also define:

$A \uparrow_x B \quad \Leftrightarrow \quad \exists C.\ (A \leq_x C) \wedge (B \leq_x C)$

$\Downarrow_x A \quad = \quad$ the set of $\leq_x$-minimal members of the set A.

$\Uparrow_x A \quad = \quad$ the set of $\leq_x$-maximal members of the set A.

For example, $t \uparrow_s t'$ means that t and t' are unifiable; when this is so, $t \sqcup_s t'$ exists and is their most general unifier.

A *term rewrite system* (or *TRS*) is a triple $(\Sigma,\mathcal{R},\mathbf{T})$, where Σ is a set of non-variable function symbols, $\mathcal{R}$ is a set of rewrite rules, and $\mathbf{T}$ is a set of terms over Σ which is closed under reduction by $\mathcal{R}$ and by the subterm relation. That is, if t is in $\mathbf{T}$, all its subterms are in $\mathbf{T}$, and if $\mathrm{Red}(\mathcal{R})(t,t')$, then t' is in $\mathbf{T}$. We may refer to a TRS by its rule-set $\mathcal{R}$, considering Σ and $\mathbf{T}$ to be fixed. The specification of a set of terms $\mathbf{T}$ as a part of the system allows us to uniformly treat typed systems, where not all terms that could be formed from the function symbols are legal.

3.2. Mismatches, conflicts and orthogonality

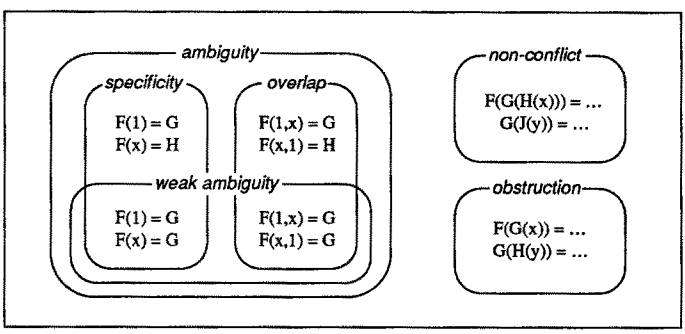
Given two open terms t and t', a *mismatch* of t and t' is an address u common to both terms, such that t/u and t'/u have different function symbols, neither being an identifier, and no proper initial segment of u has this property.

A rule R is *ambiguous with* a rule R' if $R \uparrow_{ls} R'$, and $R \neq R'$. The relation is symmetric. Some special cases are of importance. Firstly, if $R \uparrow_{s} R'$ (that is, the most general substitutions σ and σ' such that $\sigma l R = \sigma' l R'$ are such that also $\sigma r R = \sigma r R'$), then we say that R and R' are *weakly ambiguous*. Secondly, if $l(R) <_{ls} l(R')$, then we say that R is *less specific than* R', and write $R <_{ls} R'$. If R and R' are ambiguous, and neither is less specific than the other, we say that they *overlap*.

A rule R *obstructs* a rule R' *at* u if u is a nonempty proper address of l(R') and $l(R')/u \uparrow_{s} l(R)$.

Rules R and R' *conflict* if they are either ambiguous with each other or one obstructs the other. We call such conflicts respectively ambiguities and obstructions.

These various relations between rules, and examples thereof, are illustrated in the figure.



Note that it is possible for two rules to conflict with each other in more than one way. For example, rules F(F(x)) = ... and F(F(G(x))) = ... display both an obstruction and a specificity conflict with each other. In addition, F(F(x)) obstructs itself at the address 1.

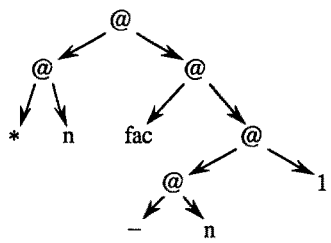
A TRS is *orthogonal* if its rules are left-linear, and none of its rules conflict. A set of terms is *orthogonal* if a rule-system, in which each of the terms is the left hand side of one rule, is orthogonal. Note that orthogonality of a TRS is not quite the same as orthogonality of the set of its left-hand sides; the only difference is that if two or more rules have identical left hand sides the system will not be orthogonal, although the set of its left hand sides might be.

3.3. Applicative and functional TRSs

We call a TRS $(\Sigma,\mathcal{R},T)$ *applicative* if every symbol in Σ has arity 0, except for one symbol having arity 2, which we will denote by @, and call *application*, but is usually not written explicitly. Otherwise, the system is called *functional*. Some term rewrite languages only allow applicative systems to be defined. SASL and KRC are examples. In these languages, the application symbol is not explicitly represented in the syntax, but is implied by juxtaposition. This is also true of Miranda, when algebraic types are not being used. A term such as n * (fac (n–1)) will look like this when applications are written explicitly:

@(@(*,n),@(fac,@(@(–,n),1)))

It is clearer when written as a syntax tree:



When quoting examples from Miranda, we shall follow its syntax, but when presenting "generic" examples, we follow the functional syntax, in which we would write the second factorial rule as Fac(n) = *(n,Fac(–(n,1))).

3.4. Operator-constructor systems and completeness

DEFINITION. An *operator symbol* of a TRS (F,$\mathcal{R}$,T) is a symbol which is the principal function symbol of the left-hand side of some rule of the system. A *constructor symbol* is a symbol which is not an operator. An *operator-constructor* system is one in which no operator appears as a sub-principal function symbol in the left hand side of any rule. A system is *complete for* a function symbol F if no closed normal form has F as its principal function symbol. A system is *complete* if it is complete for each of its operators. □

Applicative systems are in general not operator-constructor systems, as the application symbol appears in both principal and subprincipal positions in the left hand sides. However, they can be transformed into operator-constructor form. An example will show the general method.

An applicative rule:	$S\ x\ y\ z = (x\ z)\ (y\ z)$
With explicit application:	$@(@(@(S, x), y), z) = @(@(x, z), @(y, z))$
Transformed to op-con form:	$@(S, x) = S_1(x)$
	$@(S_1(x), y) = S_2(x, y)$
	$@(S_2(x, y), z) = @(@(x, z), @(y, z))$

In the transformed system, @ is an operator, and S, S_1, and S_2 are constructors.

Subject to some such transformation, Hope and ML programs are operator-constructor systems, as is Miranda, except for its lawful types (which we consider in section 7).

In Miranda, Hope, and ML, programs are, in effect, always complete. In Miranda, if one defines a function by Head(Cons(x, y)) = x, and tries to evaluate Head(Nil), a run-time error is detected and the program is terminated. We would say that a program which tries to evaluate Head(Nil) is not a program at all, any more than a syntactically erroneous program is.

Hope enforces completeness at compile-time: the programmer would be required to provide a rule to deal with Head(Nil).

ML would cause an exception to be raised on evaluating Head(Nil). This is not a run-time error, and is best described as saying that the compiler has automatically completed the programmer's incomplete rule set by causing Head to return an error value when the programmer's rules for Head do not match. (A formal semantics of ML exception handling along these lines is given in [9].)

3.5. Strong sequentiality

Strong sequentiality was defined by Huet and Lévy [12]. The definition is highly technical and there is not space to state it here. We shall briefly and informally describe the intuition behind the notion.

In an orthogonal system, Huet and Lévy showed that the any term having a normal form but not in normal form, at least one of the redexes of the term is "needed" — that is, every reduction of the term to normal form will at some point reduce at least one residual of that redex. They showed, furthermore, that any reduction strategy which reduces only needed redexes is normalising — it will find the normal form of any term which has one. The problem of lazy computation is thus reduced to the problem of finding needed redexes. Unfortunately, for general orthogonal systems, this is uncomputable. The reason is essentially that the only way in general to find needed redexes is to first reduce the term to normal form and see which steps of the reduction were in fact needed.

The problem is solved by seeking a stronger condition than orthogonality, which ignores the right-hand sides of the rules. Huet and Lévy showed that the following property of an orthogonal rule system $\mathcal{R}$ is decidable:

> for every term t having a n.f. but not in n.f., there exists a redex of t, such that for any rule-system having the same left-hand sides as $\mathcal{R}$, the redex is needed.

When $\mathcal{R}$ satisfies this condition, it is said to be *strongly sequential*. For such systems, there is also an algorithm for finding a needed redex in every term not in n.f. (Finding *all* the needed redexes of a term is still undecidable, however).

For orthogonal operator-constructor systems, the test for strong sequentiality is easily described. Take any linear, open term t, which is obtained from some left-hand side of $\mathcal{R}$ by replacing some subterms by new variables. Look at the set T of left-hand sides of $\mathcal{R}$ which are instances of t (N.B. not the reverse relation). Say that $\mathcal{R}$ is *strongly sequential at t* if T has the property that:

> if T has two or more members, then there is an address u of a variable occurrence in t such that every member of T instantiates u.

Then $\mathcal{R}$ is strongly sequential iff $\mathcal{R}$ is strongly sequential at every such term t. When this is the case, the addresses u found for such t encode a normalising reduction strategy for $\mathcal{R}$. But it would take us too far afield to describe this strategy.

Here is a well-known example of a non-strongly sequential rule system, known as "Berry's F" [4]:

Example 3.

F(x, 0, 1) = ...
F(1, x, 0) = ...
F(0, 1, x) = ...

This fails the above test, since all three rules are instances of F(a,b,c), yet neither a, nor b, nor c is instantiated by all three left-hand sides.

For example 2 of section 1, the terms which must be tested are f(x,y), f(Nil,x), f(x,Nil), f(x,Cons(y,z)), and similarly for g with the arguments reversed. In each of these terms, the occurrence of the variable x satisfies the condition. Thus the system is strongly sequential.

4. Specificity in operator-constructor systems

We can now formally define specificity. We deal first with a restricted class of TRSs.

4.1. DEFINITION. A TRS is *Type 0* if it is a complete, left-linear, operator-constructor system, and no two left-hand sides of $\mathcal{R}$ are identical (ignoring change of variable names). □

Programs in Miranda (without lawful types) and ML are Type 0 rewrite systems (except for the minor fact that they do not forbid rules having identical left-hand sides). Type 0 systems may be ambiguous, as

demonstrated by the examples in section 1, but they do not contain obstructions.

Given a Type 0 rule-system $\mathcal{R}$, we define another system Spec($\mathcal{R}$). The rules of Spec($\mathcal{R}$) will be substitution instances of the rules of $\mathcal{R}$; thus the reduction relation of Spec($\mathcal{R}$) will be a subrelation of the reduction relation of $\mathcal{R}$. As we wish to preserve the Type 0 property, the substitutions we apply to obtain the rules Spec($\mathcal{R}$) will be constructor substitutions, i.e. substitutions whose range consists only of constructor terms.

Given certain further conditions on $\mathcal{R}$, Spec($\mathcal{R}$) will be orthogonal and strongly sequential. We define Spec($\mathcal{R}$) to be the meaning of "$\mathcal{R}$ with the specificity rule".

4.2. DEFINITION. Let $\mathcal{R}$ be a Type 0 TRS. Spec($\mathcal{R}$) is the following rule-set:

$$\mathrm{Spec}'(R,\mathcal{R}) = \{\ \sigma(R) \mid \sigma \text{ is a linear constructor substitution} \wedge\ \forall R' \in \mathcal{R}.\ R' >_{ls} R \Rightarrow R' \text{ and } \sigma R \text{ are not ambiguous}\ \}$$

$$\mathrm{Spec}'(\mathcal{R}) = \bigcup\{\ \mathrm{Spec}'(R,\mathcal{R}) \mid R \in \mathcal{R}\ \}$$

$$\mathrm{Spec}(\mathcal{R}) = \Downarrow_s(\mathrm{Spec}'(\mathcal{R}))$$ □

This definition proceeds in two stages. First, for each R in $\mathcal{R}$ we define a set Spec'(R,$\mathcal{R}$) of instances of R, chosen so as not to be ambiguous with any rule of $\mathcal{R}$ more specific than R. Then we take the minimal members of all these (in general infinite) sets, with respect to the $\leq_s$ ordering (not the specificity ordering). This is Spec($\mathcal{R}$). This second step serves merely to eliminate redundant rules — both Spec'($\mathcal{R}$) and Spec($\mathcal{R}$) have the same reduction relation. While either may be used as a definition of the semantics of specificity, Spec($\mathcal{R}$) may also be used as a direct implementation.

The following basic properties of Spec($\mathcal{R}$) are easily proved.

4.3. THEOREM. Let $\mathcal{R}$ be Type 0.

(1) The reduction relation generated by Spec($\mathcal{R}$) is a subrelation of that generated by $\mathcal{R}$.

(2) A term is a normal form of Spec($\mathcal{R}$) if and only if it is a normal form of $\mathcal{R}$.

(3) Spec($\mathcal{R}$) is Type 0.

(4) If $\mathcal{R}$ contains no specificity conflicts then $\mathcal{R}$ = Spec($\mathcal{R}$). □

From (1) and (2) it follows that any evaluation of a term to normal form in Spec($\mathcal{R}$) can be performed in $\mathcal{R}$. The converse is of course not true — the purpose of Spec($\mathcal{R}$) is to eliminate computations such as the example in which fac 1 = 0.

5. Examples

We consider the effect of Spec on the examples of section 1.

Example 1.

$$\mathrm{fac}\ 0 = 1 \qquad (1.1)$$

$$\mathrm{fac}\ n{:}\{\mathrm{INT}{-}0\} = n * (\mathrm{fac}\ (n{-}1)) \qquad (1.2)$$

The notation x:{INT–0} is intended to schematically express the infinite set of rules obtained from fac n = ... by substituting any non-zero integer for n. This rule-system is orthogonal and strongly sequential.

Example 2. This example, being already orthogonal, is not changed by Spec. As it is strongly sequential, the Huet-Lévy reduction strategy maintains the validity of the equational reasoning leading to the conclusion that c g = f is valid. More precisely, for any term t, (t (c g)) has normal form t' iff (t f) has normal form t'.

Spec($\mathcal{R}$) is not always as well-behaved as this. Firstly, overlap ambiguities in $\mathcal{R}$ may persist into Spec($\mathcal{R}$), as illustrated by "parallel or":

Example 4. Or(True, x) = True

Or(x, True) = True
Or(x, y) = False

Assuming that the only constructors that can appear as arguments to Or are True and False, Spec transforms this to:

Or(True, x) = True
Or(x, True) = True
Or(False, False) = False

The ambiguity between the first two rules is unaffected by the transformation. There is nothing surprising about this. Specificity is not a magic wand which will eliminate all ambiguities from a system. It is only intended to deal with those ambiguities resulting from the use of "default" rules.

A second reason for non-orthogonality of Spec($\mathcal{R}$) is more interesting. Here is another formulation of the Or function.

Example 5. Or(x, y) = True
Or(False, False) = False

Assuming that Or takes boolean arguments, Spec gives:

Or(True, y) = True
Or(x, True) = True
Or(False, False) = False

The first two rules are ambiguous with each other. Looking at the original rule set, this is only to be expected. To refute the possibility that a term Or(t,t') might match the rule Or(False,False) = ..., it is sufficient to either evaluate t far enough to discover that it is not False, or evaluate t' far enough to discover that it is not False. But it is not possible to tell in advance which should be evaluated. The specificity transformation cannot create sequentiality where none existed, but it makes the non-sequentiality of the original system explicit.

6. Conditions for orthogonality and strong sequentiality

To eliminate ambiguities from Spec($\mathcal{R}$), we must make further restrictions on $\mathcal{R}$.

6.1. DEFINITION. Let $t \leq_s t'$. $[t,t']$ is the set of terms t'' such that $t \leq_s t'' \leq_s t'$. A set of this form is called an *interval*. The interval is *thin* if $\leq_s$ is a total ordering on its members. □

6.2. EXAMPLE. (i) [F(•,•,•), F(•,G(•,H(J(•,•))),•)] = { F(•,•,•), F(•,G(•,•),•), F(•, G(•, H(•)),•), F(•,G(•,H(J(•,•))),•) }. This interval is thin.

(ii) [F(•,•),F(0,1)] = { F(•,•), F(0,•), F(•,1), F(0,1) }. This interval is not thin.

6.3. DEFINITION. $\mathcal{R}$ is *Type 1* if

(i) $\mathcal{R}$ is Type 0.

(ii) If t and t' are left hand sides of $\mathcal{R}$, and $t \sqcup_s t'$ exists, then it is (up to renaming of variables) also a left hand side of $\mathcal{R}$. Briefly, we say that $l(\mathcal{R})$ is $\sqcup_s$-*closed*.

(iii) Let T and T′ be two left hand sides of $\mathcal{R}$ such that $T \leq_s T'$ and the interval [T,T′] contains no other left-hand side of $\mathcal{R}$. Then the interval is thin. (We refer to this property by saying that $\mathcal{R}$ has *thin gaps*.) □

6.4. THEOREM. If $\mathcal{R}$ is Type 1 then Spec($\mathcal{R}$) is orthogonal.

PROOF (OUTLINE). Condition (ii) rules out the situation of example 3, by ensuring that whenever two rules R_1 and R_2 of $\mathcal{R}$ overlap, every term to which they both apply is also matched by a rule R_3 more

specific than both rules. This ensures that substitution instances of R_1 and R_2 which are not ambiguous with R_3 are not ambiguous with each other.

Condition (iii) rules out the situation of example 5, where there were two independent ways of instantiating a rule R_1 to make it unambiguous with R_2. □

We next consider strong sequentiality of Spec($\mathcal{R}$). In fact, all that is needed to ensure strong sequentiality of Spec($\mathcal{R}$) is that the set of maximally specific members of R be strongly sequential.

6.5. DEFINITION. A rewrite system $\mathcal{R}$ is Type 2 if it is Type 1, and $\Uparrow_{ls}\mathcal{R}$ is strongly sequential. □

6.6. THEOREM. If $\mathcal{R}$ is Type 2 then Spec($\mathcal{R}$) is strongly sequential.

PROOF (OUTLINE). From the Type 1 property we can show that without loss of generality, $\mathcal{R}$ may be assumed to be *dense*, by which we mean that if there exist a substitution σ and two rules R_1 and R_2 of $\mathcal{R}$ such that $lR_1 <_s \sigma lR_1 <_s lR_2$, then for some such substitution σ, σlR_1 is a left-hand side of $\mathcal{R}$. This is so because, given such R_1 , R_2, and σ, adding σR_1 to $\mathcal{R}$ does not change either Spec($\mathcal{R}$) or $\Uparrow_{ls}\mathcal{R}$.

When $\mathcal{R}$ is Type 1 and dense, Spec($\mathcal{R}$) can be shown to consist of all the rules in $\Uparrow_{ls}\mathcal{R}$, together with some rules, each of whose left hand sides has the form σlR, where R is in $\Uparrow_{ls}\mathcal{R}$, and σ is an address substitution defined at exactly one address, its value there being a term of the form F(•) where F is a constructor. We already know that Spec($\mathcal{R}$) is orthogonal; the above characterisation of Spec($\mathcal{R}$), and that of strong sequentiality for op-con systems in section 3.5 implies that Spec($\mathcal{R}$) is strongly sequential. □

Examples 1 and 2 are Type 2. Example 3 is Type 1 but not Type 2. Examples 4 and 5 are Type 0 but not Type 1. In example 4, u_s-closure fails; in example 5, there is a thick interval between f • • and f 0 1.

The particular ways in which the Type 1 and Type 2 properties may fail to hold can be used to modify the original system in order to ensure orthogonality and strong sequentiality of the transformed system. In example 3, we must specify which argument of F should be evaluated first. For example, we may add an instance of the first rule of the form

$$F(2, 0, 1) = \ldots$$

which has the effect of specifying that F is to be strict in its third argument. In example 4, it is necessary to specify which rule is to be applied to a term of the form Or(True, •) u_s Or(•, True). In example 5, it is necessary to close the thick interval by adding a rule whose left-hand side is f 0 •, or one whose left-hand side is f • 1, or both.

We thus see that when a Type 0 system fails to be Type 2, it is possible to isolate the reasons for this failure. The programmer can use this information to add rules which explicitly inform the system of the preferred order of evaluation, in just those cases where the original system left this underspecified.

7. Specificity with sub-root conflicts

Our discussion so far covers operator-constructor systems, and hence the languages ML, Hope, and Haskell. But only a part of Miranda is covered, as a Miranda lawful type is defined by more general rewrite rules[2]. For example, consider the following Miranda definition of a type of lists of even integers:

Example 6.

```
elist ::= Enil | Econs num olist
Econs a x => x, odd(a)
```

[2] In the latest version of Miranda (version 2), lawful types have been removed from the language.

odd is a boolean predicate testing whether its argument is an odd integer. We shall not attempt to give here a semantics for guarded rules, but take the above rule as being a schematic representation for the set of instances where a is an odd integer.

A typical rule for operating on an elist might be:

ehead :: elist –> num
ehead (Econs a x) = a

The semantics of such a lawful type is that, viewed from 'outside', Enil and Econs are constructors, but whenever an attempt is made to pattern-match on an elist term, as with the ehead rule, the elist is first reduced to "head normal form" according to the laws for the type.

Another way of looking at this is to say that laws are no different from rewrite rules: because the rule for Econs obstructs the rule for ehead, the ehead rule may not be applied to an Econs node unless it is known that the Econs node could never in future match the Econs rule. This suggests that the definition of specificity may usefully be extended to such systems.

We modify the previous definition of Spec'(R,$\mathcal{R}$) by imposing different conditions on the substitutions to be applied to R. Firstly, they are weakened: since we are no longer working in an operator-constructor system, σ is not restricted to being a constructor substitution. (It must still be linear.) Secondly, they are strengthened: the condition that σR be non-ambiguous with any rule more specific than R remains, but we add a further condition that σR be chosen to as not to be obstructed by any rule of $\mathcal{R}$.

7.1. DEFINITION. Let $\mathcal{R}$ be a left-linear TRS. Spec($\mathcal{R}$) is the following rule-set:

$$\text{Spec}'(R,\mathcal{R}) = \{\ \sigma(l(R)) \mid \sigma \text{ is a linear substitution} \wedge \forall R' \in \mathcal{R}.\ R' \text{ does not obstruct } \sigma R \wedge \forall R' \in \mathcal{R}.\ R' >_{ls} R \Rightarrow R' \text{ and } \sigma R \text{ are not ambiguous}\ \}$$

$$\text{Spec}'(\mathcal{R}) = \bigcup\{\ \text{Spec}'(R,\mathcal{R}) \mid R \in \mathcal{R}\ \}$$

$$\text{Spec}(\mathcal{R}) = \Downarrow_s \text{Spec}'(\mathcal{R})$$

□

We are justified in calling this transformation by the same name as the one previously defined, because for Type 0 systems it is easy to show that it generates the same reduction relation as the Spec($\mathcal{R}$) of the earlier definition. (It is possible that it contains more rules than the previous definition would give, but the extra rules are incapable of applying to any term of the system.)

8. Examples

Applying Spec to the rules for Econs and ehead defined above yields the set consisting of all rules of the forms:

$$\text{Econs } o_1 \text{ (Econs } o_2 \text{ (...(Econs e x)...))} = \text{Econs e x}$$
$$\text{ehead (Econs e x)} = \text{e}$$

where o_1, o_2,... are odd integers (and there is at least one) and e is an even integer. These are infinite sets, useful as a definition of the semantics, rather than as an implementation.

On this example, Spec yields an orthogonal, strongly sequential system. But as for Type 0 systems, this is not always so.

Example 7.
F(G(x, y)) = ...
G(0, 1) = ...

Given a term of the form F(G(..., ...)), to apply the rule for F one must first ensure that the rule for G can never apply to the argument of F. This will require evaluating at least one of the arguments to G, but

not necessarily both. In general it is not possible to tell which argument must be evaluated.

9. Conditions for orthogonality and strong sequentiality

To ensure that Spec($\mathcal{R}$) is orthogonal, it is sufficient to define an appropriate generalisation of the thin gaps property.

9.1. DEFINITION. $\mathcal{R}$ is *irredundant* if there do not exist rules R_0 and R_1 in $\mathcal{R}$ and a nonempty address u of lR_0 such that $iR_0/u \geq_s lR_1$.

A substitution σ is *thin* if $\sigma(x)$ is thin whenever it is defined.

An *obstruction* is a tuple (t,u,t') where t and t' are terms, u is a proper address of t, and $t/u \uparrow_s t'$. With any obstruction is associated a substitution: that σ having the smallest domain such that $\sigma(t/u) = t/u \sqcup_s t'$. The obstruction is thin if σ is thin.

Given two obstructions of the form (t,u,t'), (t',v,t''), if u·v is a proper address of t, then there is an obstruction (t,u·v,t''), This is the *composition* of the two given obstructions.

Given a set of terms T, an obstruction of T is an obstruction (t,u,t') such that t and t' are in T. It is *minimal* if it is not the composition of two obstructions of T.

T is said to have *thin gaps* if every minimal obstruction of T is thin.

A rule system $\mathcal{R}$ is *Type 3* if it is left-linear and irredundant, and the set of its left hand sides is closed under $\sqcup_s$ and has thin gaps. □

Note that all Type 0 systems are irredundant, and that for Type 0 systems, the above definition of thin gaps coincides with the earlier one.

The example in the last section did not have thin gaps, for the obstruction (F(G(x,y)),⟨1⟩,G(0,1)) has the associated substitution [x:=0, y:=1], which is not thin.

The set { F(G(H(•))), H(J(K(•))) } has thin gaps.

The set { F(G(0,1)), G(•,•) } is not irredundant. Note that specificity demands that a term of the form F(G(0,1)) may not be reduced unless the subterms matched by this left hand side are all in head normal form. But because of the left hand side G(•,•), this can never be the case. Thus the rule whose left hand side is F(G(0,1)) is superfluous.

The $\sqcup_s$-closure (whose definition does not need to be changed) and thin gaps properties will play the same role for left-linear systems as they did for Type 0 systems. Failure of $\sqcup_s$-closure implies that the system contains ambiguities which specificity will not eliminate. Non-thin gaps cause ambiguities in the transformed system (in the first of the above examples, specificity will add rules for both F(G(1,•)) and F(G(•,0))).

9.2. THEOREM. If $\mathcal{R}$ is Type 3 then Spec($\mathcal{R}$) is orthogonal.

PROOF (OUTLINE). It is immediate from the definition of Spec that Spec($\mathcal{R}$) can contain no obstructions. If R_1 and R_2 are rules of Spec($\mathcal{R}$) which are ambiguous with each other, then (in the same way as for the proof of the same proposition for Type 1 systems) the $\sqcup_s$-closure property implies that they must be instances $\sigma_1(R)$ and $\sigma_2(R)$ of the same rule R of $\mathcal{R}$. However, the thin gaps property rules this out. □

When is Spec($\mathcal{R}$) strongly sequential?. A natural generalisation of the Type 2 property suffices.

9.3. DEFINITION. Let $\mathcal{R}$ be Type 3. Let E($\mathcal{R}$) be the following set of terms:

1. E($\mathcal{R}$) contains all the left-hand sides of $\mathcal{R}$.
2. For any T and T' in E($\mathcal{R}$) and nonempty proper address u of T, if Funct(T/u) = Funct(T') then T[u:=T'] ∈ E($\mathcal{R}$).

3. $E(\mathcal{R})$ is as small as possible subject to (1) and (2).

We call $E(\mathcal{R})$ the set of *extended left hand sides* of $\mathcal{R}$.

R is *Type 4* if every orthogonal subset of $E(\mathcal{R})$ is strongly sequential. □

For Type 1 systems, $E(\mathcal{R})$ is just the set of left hand sides of $\mathcal{R}$, and Type 4 is equivalent to Type 2. Note that we cannot simply follow the definition of Type 2 and define Type 4 in terms of the set of maximally specific members of $E(\mathcal{R})$, since $E(\mathcal{R})$ may contain unbounded $\leq_{ls}$-ascending sequences. The situation is illustrated by the pair of left hand sides F(G(•)) and G(F(•)), which give rise to extended left-hand sides F(G(•)), G(F(•)), F(G(F(•))), G(F(G(•))), F(G(F(G(•)))), G(F(G(F(•)))), etc. For the example of even lists, $E(\mathcal{R})$ happens to be the same as $l(\mathcal{R})$. An example from [19] in which $E(\mathcal{R})$ is much larger than $l(\mathcal{R})$ is a type of ordered lists:

```
olist  ::=  Onil | Ocons num olist
Ocons a (Ocons b x)   =>   Ocons b (Ocons a x), a<b

ohead :: olist -> num
ohead (Ocons a x)   =   a
```

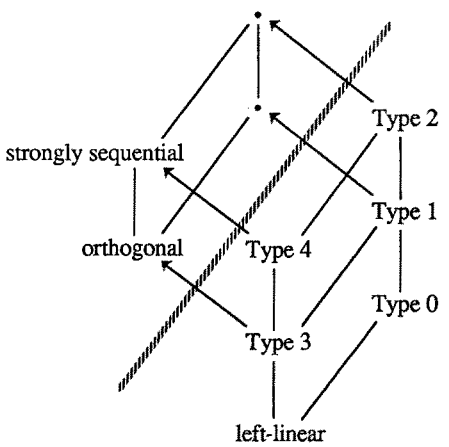
Computation of $E(\mathcal{R})$ for this example is left as an exercise.

9.4. THEOREM. If $\mathcal{R}$ is Type 4 then $Spec(\mathcal{R})$ is strongly sequential.

PROOF. Similar to theorem 6.6. □

10. Summary

The following diagram shows the relations between the various types of TRS. This is a partial order: a line from one class to another indicates containment of the higher class in the lower. Least upper bounds in the diagram are set intersections. Specificity conflicts only occur below the shaded line. The arrows indicate the action of the specificity transformation.



11. Directions for further work

It is important to establish the complexity and the practicality of the reduction strategy we have described. In the operator-constructor case, calculating the transformed system Spec(R) and applying the

Huet-Lévy strategy will work, but a more direct implementation will very likely be more efficient. In the more general case of Type 4 systems, this is not always possible, as Spec(R) can be infinite (even if the set of function symbols is finite). Here, a more direct implementation is necessary.

The definition of specificity should be extended to guarded rules. For the particular example of section 7, we were able to sidestep this by considering the guard as shorthand for a set of rules, but this is not possible in general.

12. Background

Rule systems with various notions of explicit priorities among or conditions on rules have been studied in [2,3]. However, for many of these notions, the reduction relation is uncomputable or not uniquely defined. This is because of a problem of circularity: to say "this rule shall not be applied to a term unless that rule could never be applied" makes reference to the terms which the given term could be reduced to, by the reduction relation that one is attempting to define. It is necessary to prove that the circular definition is well-defined. In many cases it is not, or if it is, it may be uncomputable. In earlier attempts to formalise the concept of specificity, we encountered the same problem. We avoid it here by restricting attention to the left hand sides of the rewrite rules. We leave open the question of whether information about the right hand sides of the rules can usefully be employed in a more refined definition of specificity.

Laville [15,16] has given a formal semantics for the top-to-bottom, left-to-right strategy by means of a transformation similar to that described here.

Turner [22] defined the semantics of Miranda's laws by translation into an operator-constructor system. Each constructor of a lawful type is replaced by a pair of symbols: an operator and a constructor. Each use of the lawful constructor as the principal function symbol of the left hand side of a law is replaced by the new operator, and every other use is replaced by the new constructor. In addition, a "default" rule of the form $F' x_1 \ldots x_n = F'' x_1 \ldots x_n$ is added (where F' is the operator and F'' the constructor replacing some lawful constructor F). Thompson [19] has studied the problem of verification of Miranda programs which use laws, using Turner's transformation to define the semantics, and giving conditions under which equational reasoning is valid. The pattern-matching is still top-to-bottom, left-to-right. However, this does not affect the examples he studies, which, after transformation, happen to be left-sequential [17], for which top-to-bottom, left-to-right pattern-matching is normalising; it is not clear whether his results need to be modified to deal with strongly sequential but non-left-sequential systems such as our Example 2.

It should be possible to show that for left-sequential Miranda programs the semantics we have given coincides with Turner's.

Hope+ [7,18] uses a weaker form of specificity than that described here, called "best-fit" pattern-matching. This is used only to avoid dependency on the order of rules; within a rule, pattern-matching is performed left-to-right, and as a result is not fully lazy.

In a somewhat different vein, Dactl0 [8] is a language of graph rewriting in which specificity conflicts are allowed, and resolved by always choosing the most specific rule. However, the situation is much simpler than for the functional languages considered here, because in Dactl0, pattern-matching and evaluation are independent — in choosing which rule to apply, all that is relevant is the set of rules which could be applied immediately, not the rules which might apply at some time in the future.

References

1. A. Augustsson. A compiler for Lazy ML, ACM Conf. on Lisp and Functional Programming, 1984.

2. J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Priority rewrite rules, Report CS-R8407, Centrum voor Wiskunde en Informatica, Amsterdam, 1982.

3. J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. J. Comp. Sys. Sci., **32**, no.3, 323–362, 1986.

4. G. Berry. Stable models of typed lambda-calculi, in: G. Ausiello and C. Böhm., eds., Proc. 5th Int. Conf. on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol.62 (Springer, 1978)

5. T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean: a language for functional graph rewriting, Report, Computing Science Department, University of Nijmegen, 1987.

6. R.M. Burstall, D.B. MacQueen, and D.T. Sannella. HOPE: an experimental applicative language, in: Proc. 1st ACM Lisp Conference, 136–143, Stanford, 1980.

7. A.J. Field, L.S. Hunt, and R.L. While. Best-fit pattern matching for functional languages. Internal report, Department of Computing, Imperial College, London, 1989.

8. J.R.W. Glauert, J.R. Kennaway, and M.R. Sleep. DACTL: a computational model and compiler target language based on graph reduction. ICL Technical Journal, **5**, 509-537, 1987.

9. K. Hammond. Implementing Functional Languages for Parallel Machines, Ph.D. thesis, School of Information Systems, University of East Anglia, 1988.

10. R. Harper, R. Milner, and M. Tofte. The definition of Standard ML, Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

11. P. Hudak et al. Report on the Functional Programming Language Haskell. Draft Proposed Standard, 1988.

12. G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems, INRIA report 359, 1979.

13. J.W. Klop. Term rewriting systems: a tutorial, Bull. EATCS, no.32, 143–182, June 1987.

14. J.W. Klop. Term rewriting systems, *in* S. Abramsky, D. Gabbay, and T. Maibaum (eds.) *Handbook of Logic in Computer Science*, (Oxford University Press, in preparation).

15. A. Laville. Lazy pattern matching in the ML language, INRIA report 664, 1987.

16. A. Laville. Comparison of priority rules in pattern matching and term rewriting, INRIA report 878, 1988.

17. M.J. O'Donnell. Equational Logic as a Programming Language, MIT Press, 1985.

18. N. Perry. Hope+. Internal report, Department of Computing, Imperial College, London, 1988.

19. S. Thompson. Lawful functions and program verification in Miranda, Science of Computer Programming, to appear, 1989.

20. D.A. Turner. SASL language manual. University of St. Andrews, 1979.

21. D.A. Turner. Recursion equations as a programming language, *in* J. Darlington, P. Henderson, and D.A. Turner (eds.) Functional Programming and its Applications: an Advanced Course, Cambridge University Press, 1982.

22. D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud (ed.), Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, vol.201, Springer-Verlag, 1985.

23. D.A. Turner. Miranda language manual. Research Software Ltd., Canterbury, U.K., 1987