Programming with Narrowing: a Tutorial

Sergio Antoy [1]

Computer Science Department
Portland State University
P.O. Box 751, Portland, OR 97207, U.S.A.

Abstract

Narrowing is a computation implemented by some declarative programming languages. Research in the last decade has produced significant results on the theory and foundation of narrowing, but little is published on the use of narrowing in programming. This paper introduces narrowing from a programmer viewpoint; shows, by means of examples, when, why and how to use narrowing in a program; and discusses the impact of narrowing on software development activities such as design and maintenance. The examples are coded in the programming language Curry, which provides narrowing as a first class feature.

1. Introduction

Narrowing is a programming feature found in some modern high-level declarative languages such as Curry (Hanus et al., 1995; Hanus, 2006) and $\mathcal{TOY}$ (Caballero and Sánchez, 2007; López-Fraguas and Sánchez-Hernández, 1999). These languages are often referred to as *functional logic* because they include both functions, as in the functional languages Haskell (Peyton Jones and Hughes, 1999) and ML (Milner et al., 1997), and logic variables, as in the logic language Prolog (ISO, 1995). Narrowing is the glue that allows the seamless integration of these features in a single computation paradigm by enabling the functional-like evaluation of expressions containing uninstantiated logic variables.

There exist also functional logic programming languages, such as Life (Aït-Kaci, 1990) and Escher (Lloyd, 1999), that do not provide narrowing, but use residuation for the same purpose. Narrowing and residuation are not in conflict and an elegant model (Hanus, 1997) let them coexist. A survey of functional logic languages, which includes a discussion on both narrowing and residuation, is in (Hanus, 1994), see (Hanus, 2007) for a more recent version.

Narrowing is more than a programming language feature. Originally introduced for theorem proving (Fay, 1979; Hullot, 1980) narrowing is nowadays applied in a variety of

Email address: `antoy@cs.pdx.edu` (Sergio Antoy).

[1] Supported in part by the National Science Foundation grants CCR-0110496 and CCR-0218224.

other areas, e.g., both partial evaluation (Albert et al., 2002; Alpuente et al., 1996, 1998, 2005b) and testing of programs (Christiansen and Fischer, 2008; Fischer and Kuchen, 2007, 2008), type-level inference (Sheard, 2007) and verification of cryptographic protocols (Meseguer and Thati, 2007).

The investigation of narrowing for programming goes back to (Dershowitz and Plaisted, 1988; Reddy, 1985). To our knowledge, (Reddy, 1985) is the first paper that identifies the potential of narrowing as a feature for programming. Early programming languages that offered narrowing to the programmer include K-Leaf (Giovannetti et al., 1991) and Babel (Moreno-Navarro and Rodríguez-Artalejo, 1992). Since then, significant theoretical results have been discovered. Nowadays, essential properties of narrowing such as soundness and completeness are known for several practical classes of programs, see (Antoy, 2005) for a survey, and optimal evaluation strategies have been discovered for some of these classes (Antoy, 1997; Antoy et al., 2000; Echahed and Janodet, 1997). Various active research efforts aim at efficient implementations (Antoy and Hanus, 2000; Antoy et al., 2001; Lux, 1999; Tolmach et al., 2004). The Curry homepage (Hanus, 2006) links all the current implementations of narrowing-based modern functional logic languages. For logic computations, the efficiency of narrowing is competitive with that of resolution (Robinson, 1965). For functional computations, there exist narrowing interpreters with a "pay-per-view" policy (Antoy et al., 2001)—if narrowing is not executed, i.e., the computation is functional, the efficiency of the narrowing interpreter is comparable to that of a functional interpreter. Despite these theoretical and practical successes, the potential of narrowing for programming remains underutilized and possibly poorly understood except for the specialist. One reason of this state of affairs is that narrowing is still a relatively young event in the programming languages landscape. Most publications on narrowing have emphasized its theory and foundations, while the discussion on its use in programming has lagged behind. This paper aims to correct this unbalance.

Narrowing is a computation best known for solving sets of equations possibly involving user-defined abstract data types. This remarkable property is used to great advantage for a variety of purposes as mentioned earlier. However, this property can also be inappropriately used, although this is not an intrinsic flaw of narrowing. For example, an essential step to decrypt an encrypted message is finding two prime factors of a large integer. This problem can be expressed by a small set of simple equations which in principle could be solved by narrowing. It would be foolish, though, to expect that narrowing is a viable approach to break a cipher. A *while* loop that looks for the prime factors of a number by trial and error would be more efficient—and equally useless for a good cipher. Dismissing narrowing as a programming language feature because it fails in situations of this kind is analogous to dismissing the *while* statement of imperative programming languages because it equally fails in the same situations and/or may lead to non-termination.

This paper presents narrowing from the programmer viewpoint. The presentation focuses on the use of narrowing. The emphasis is not on semantics or theoretical aspects, which abound in the literature, but on explaining narrowing to the non-specialist, presenting situations in which the use of narrowing is appropriate, and highlighting the advantages of its use. The methodology is inspired by Wirth (1971) in that "the creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques." Section 2 informally presents narrowing and summarizes the key results that ensure its most important properties. Section 3 outlines the programming language used for the examples and explains how narrowing is supported by the language and interacts with other programming features. Section 4 shows through examples the use of narrowing in programs. First we explain motivations and design, then we present executable code and finally we discuss differences with respect to alternative designs that do not use narrowing. Section 5 highlights some characteristics of a problem that suggest that using narrowing in the problem's solution may be convenient for the programmer. Section 6 offers the conclusion. An appendix gives a formal definition of narrowing and discusses some related notions such as correctness and strategies.

2. Narrowing

This section introduces narrowing from a programming viewpoint. The discussion is informal and often relies on intuition. A formal definition of narrowing and other related concepts introduced here is presented in the Appendix. We hope that the conceptual simplicity of the examples and the elegance and economy of their code will motivate the reader to undertake the effort required to digest the formal definition. We believe that a formal definition of narrowing is useful, but not essential for many programmers. By analogy, knowledge of axiomatic or denotational semantics would be useful, but generally not required, to programmers in an imperative language.

Rewriting is a special case of narrowing and, therefore, it is a good starting point for introducing the concept. Rewriting (Baader and Nipkow, 1998; Bezem et al., 2003; Dershowitz and Jouannaud, 1990; Echahed and Janodet, 1997; Klop, 1992; Plump, 1999; Sleep et al., 1993) is a computation defined by rules that describe how to transform expressions, also called terms or term graphs in this context. Without much stretching, the table for multiplying single-digit integers, which children learn in second grade, is a familiar example of rewriting. For example, 2×2 rewrites to 4. Obviously, the meaning drives the rules for rewriting, but the rules themselves are just syntax. They would stand without meaning, too. Rewriting is a viable tool for both describing and executing computations in both practical and theoretical situations, e.g., from chemical reactions to games to group theory.

The "things" being rewritten are *expressions* such as 2×2 of the previous example. The infix operators found in expressions have an explicit or implied precedence and associativity which generally will be left to the intuition. Expressions are rewritten according to *rewrite rules*. A rewrite rule is therefore a pair of expressions denoted with an arrow in between, e.g.:

$$2 \times 2 \to 4 \tag{1}$$

The multiplication table that children learn in elementary school consists of about 100 rules of this kind. A set of rewrite rules is a *rewrite system*.

For situations in which there exists an infinite number of expressions to rewrite, the rules contain *variables*. Variables must be distinguished from other symbols. Since the attention is on variables, we follow the syntax of the Prolog programming language (ISO, 1995). Identifiers that denote variables begin with an upper case letter. An alternative would be to explicitly state which identifiers denote variables, but the Prolog convention is more readable. Variables that occur only once in a rule are simply place-holders and do not need a name. They are called *anonymous* and are identified by an underscore. This too is only a convention intended to improve readability. Semantically, a variable stands for any expression, or any expression of the appropriate type, if one considers well-typed expressions only. For example, the following rewrite system defines common computations on stacks:

$$\begin{aligned} &top(push(E,\_)) \to E \\ &pop(push(\_,S)) \to S \end{aligned} \tag{2}$$

To show a practical rewrite with the above system, a notation for an empty stack, say *empty*, is needed, too. To rewrite an expression t, one has to *match* t with the left-hand side l of a rule. Matching is the process of finding what each variable in l stands for to make l equal to t. The value of a set of variables is called a *substitution*. For example, $top(push(1, pop(push(2, empty))))$ matches the left-hand side of the first rule, i.e., $top(push(E, \_))$. The substitution is 1 for E and $pop(push(2, empty))$ for the anonymous variable. Thus,

$$top(push(1, pop(push(2, empty)))) \to 1 \tag{3}$$

The result of this rewrite step, 1, is obtained by applying the matching substitution to the right-hand side of the rule, in this case E. Obviously, an infinite number of stacks match the rule's left-hand side and, hence, are rewritten by this rule. Alternatively:

$$top(push(1, pop(push(2, empty)))) \to top(push(1, empty)) \to 1 \tag{4}$$

Although this rewrite sequence is longer than (3), the extra step does not affect the final result. This independence of the order of evaluation does not hold for every rewrite system.

The symbols *pop*, *top*, *push* and *empty* are not all alike. The symbols *push* and *empty* construct stack instances. They are called (data) *constructors* and do not prompt any rewrite. They are like the numbers 2 and 4 in the initial example. In contrast, *pop* and *top* operate on stack instances. They are called *defined operations* because they are defined by rules, namely (2). They are like the multiplication symbol in the initial example. A rule defining an operation shows how to rewrite an expression where the operation is applied to arguments made up of constructors and variables only, as in (1) and (2). This property is referred to as the *constructor discipline* (O'Donnell, 1977, 1985). Rewrite systems with this discipline model computations in a natural way. Expressions containing constructors only abstract *data* and evaluate to themselves. In other words they are literal *values*. Expressions containing operations abstract *computations* that are executed by rewriting.

Narrowing comes into play when an expression to evaluate contains variables. A typical reason for computing with variables is lack of knowledge. If some value of an expression is not known, a variable takes its place. Thus, $F \times 2$ is a product where only the right factor is known and $push(1, S)$ is a stack where only the top element is known. Variables in rules differ from variables in expressions. As we said earlier, a variable in a rule stands for *any* value, whereas a variable in an expression to evaluate stands for *some* value. Often the value of the variable must become known to evaluate the expression in which the variable appears. Thus, the next issue to explore is how to rewrite, or evaluate, expressions that might contain variables. This is *narrowing*.

If an expression t contains variables, narrowing first guesses some substitution for some variables, then replaces these variables with their guessed values in the context of t, and finally rewrites the *instantiation* of t as usual. The instantiation by narrowing is pretty much uninformed, but utilitarian in the sense that only instantiations that promote rewrites are made. An instantiation of an expression t is found by *unifying* t with the left-hand side l of a rule. The unification is the process of finding what each variable in t and l stands for to make the two expressions equal. Loosely speaking, the expression matches the left-hand side and the left-hand side matches the expression simultaneously. If it is possible to unify t and l, any substitution unifying them is called a *unifier*. Thus, narrowing and rewriting behave identically on expressions containing no variable.

For example, consider again the rewrite system defining the rules for multiplying two single-digit integers. The expression $F \times 2$ could be narrowed by instantiating, e.g., F to 2 and rewriting 2×2 to 4. Any other digit would be an equally viable guess for F. However, 25 is not an acceptable guess because we assume only rules for multiplying digits; there is no rule in the system saying how to rewrite 25×2. Narrowing is a goal oriented activity similar to programming. Narrowing $F \times 2$ in isolation makes little sense except, perhaps, to generate a trivial example. During the execution of a meaningful program, an expression such as $F \times 2$ would be narrowed in some meaningful context for some meaningful purpose. This is the subject of Section 4.

A rewrite system specifies what are the steps, but not when and where to perform them. The latter is the task of a strategy. A *strategy* determines which subexpression, if any, of an expression should be evaluated and which variables, if any, of this subexpression should be instantiated. Strategies are highly technical and somewhat complicated. Thus, a good language should shield the programmer from certain details of a strategy. For example, Curry, which we will use for the code examples, stipulates that its implementations should provide all the values, in some arbitrary order, of an expression. This property is called *completeness*. Some implementations of Curry, e.g., (Antoy et al., 2005; Brassel and Huch, 2007) are complete, whereas others, e.g., (Hanus, 2008; Lux, 1999), for the sake of efficiency, provide only an approximation of the completeness. The completeness of an implementation effectively relieves the programmer from many concerns about the strategy. The mode in which all the values of an expression are presented to the user is implementation dependent. In typical interpreters, after a value is printed, the user is given the option to print another value or to end the computation.

The *narrowing space* of an expression t is the set of all the expressions obtained in zero or more narrowing steps from t. This space obviously depends on the strategy. For narrowing there exist viable strategies for various classes of rewrite systems well-suited for programming (Antoy, 2005). Loosely speaking, these strategies perform the minimum amount of work necessary to solve a problem, e.g., (Antoy, 1997; Antoy et al., 2000), i.e., they waste neither rewrites nor instantiations for *successful* (terminating in a value) computations. Unfortunately, it may be impossible to predict whether a computation will be successful, but this is a problem of computing, not a specific problem of narrowing. For many problems, a programmer should have some understanding of the narrowing space and in particular, of the order in which its elements are produced, e.g., depth-first or breadth-first, to predict the behavior of a program execution.

The extent to which narrowing is capable of computing is better described in terms of equations, which are a special case of expressions. As usual, an *equation* is a pair $t = u$ where t and u are expressions as well. The problem with an equation is to find instantiations for the variables occurring in t and/or u such that the instantiated sides of the equation have the same value. This is called *solving* the equation. In systems with the constructor discipline, an equation is considered solved only when its sides are rewritten to the same datum, i.e., an expression made up of constructors only. For example, $F \times 2 = 4$ is solved if and only if F is instantiated to 2 and the left side is rewritten to 4. On the same account, $pop(empty) = S$ has no solutions. Obviously, if S is instantiated to $pop(empty)$ the sides of the equation are equal, but they cannot be rewritten to a datum, since the expression $pop(empty)$ is not a stack. This is analogous to the fact that the equation $1/0 = N$ has no solutions because the expression $1/0$ is not a number. This notion of equality is called *strict*.[2]

Within the boundaries of the previous paragraph, narrowing offers a sound and complete procedure to solve equations. These equations can involve user-defined abstract data types such as the stack of a previous example. *Soundness* means that any instantiation of the variables of an equation computed while narrowing the sides of the equation to a same datum is a solution of the equation. *Completeness* means that if an equation has a solution, narrowing will find that solution, or a more general one, while narrowing the sides of the equation to a same datum.

An equation may have no solution or several solutions. If an equation has no solution, narrowing may be able to determine this fact or may run forever in a futile attempt to find a solution. Again, this is not a specific problem of narrowing, since both the existence of a solution of an equation and the termination of a computation are unsolvable problems. If an equation has several solutions, these solutions are found in some arbitrary order. Of course, only a complete strategy guarantees to produce all the solutions. Since narrowing guesses instantiations of variables, some guesses are likely to be wrong. The following example shows that this is not a significant problem.

The symbols *nil* and *cons* are the traditional constructors of the type *list*. Lists are equal to stacks as data structures. As types, they differ in the set of operations that manipulate the underlying structures. E.g., a very common operation on lists, absent for stacks, is *append*. The operation *append*, which concatenates two lists, is defined by the rules:

$$\begin{aligned} append(nil, Z) &\to Z \\ append(cons(X, Y), Z) &\to cons(X, append(Y, Z)) \end{aligned} \tag{5}$$

Let l be a list. Suppose that l is not *nil* and the problem is to compute the last element of a l. Instead of defining a new operation for this computation, we solve, by narrowing, the following equation:

$$append(X, cons(E, nil)) \approx l \tag{6}$$

[2] Unfortunately, the word "strict" has also a another meaning in programming languages. A procedure is *strict* if, operationally, it evaluates all its arguments whether or not the values of these arguments are necessary for the execution of the procedure. In our context, operations, including the equality, are not strict in this sense.

and the instantiation of E gives us the desired value.

The symbol "$\approx$", called *equality*, is defined by ordinary rewrite rules, which enable us to perform the entire computation within the realm of narrowing. Since the equality is defined for a variety of types, i.e., the symbol "$\approx$" is overloaded, it is convenient to present its definition using meta-rules. In (7) below, the meta-symbol c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule, "&" is a right-associative infix symbol, and *success* is the constructor of a singleton type that we call *success* as well.

$$\begin{aligned} c \approx c &\rightarrow \mathit{success} \\ c(X_1,\ldots,X_n) \approx c(Y_1,\ldots,Y_n) &\rightarrow (X_1 \approx Y_1) \ \& \ \cdots \ \& \ (X_n \approx Y_n) \\ \mathit{success} \ \& \ X &\rightarrow X \\ X \ \& \ \mathit{success} &\rightarrow X \end{aligned} \tag{7}$$

Implementations typically provide the equality as a primitive or builtin function because it operates on primitive or builtin types. E.g., it would be impossible to explicitly define the first rule above for each integer. Furthermore, implementations typically optimize this rule by unifying the two sides when one is a variable and the other is a value. This *ad hoc* behavior is observable only when both sides are variables and produces more compact solutions in this case. The examples discussed in this paper are unaffected by this behavior.

Adopting *success* as the type of the equality symbol, instead of the more traditional *Boolean*, simplifies both definitions and computations since we do not have to deal with computations that evaluate an equation to *false* and therefore do not solve it. With the equality rules of (7), a solution of an equation is simply obtained by evaluating the equation, i.e., by narrowing it to a value. The evaluation by narrowing of (6) instantiates X to some uninteresting value and E to the last element of the list.

Suppose, e.g., that in (6) l is *cons(1,cons(2,nil))*. We show the evaluation of *append(X,cons(E,nil))* $\approx$ *l*. ¿From the rules of *append*, the only guesses for X are *nil* and *cons*(X_1,X_2), where X_1 and X_2 are new variables. The first guess leads to the equation *cons(E,nil)* $\approx$ *cons(1,cons(2,nil))*. Using the equality rules this reduces to $E \approx$ *1* & *nil* $\approx$ *cons(2,nil)*. Further applications of the equality rules instantiate E to *1* and reduce the whole expression to *nil* $\approx$ *cons(2,nil)*. No equality rule is applicable to this equation. The equation cannot be narrowed to *success* and consequently solved. Thus, the first guess for X is wrong. The second guess leads to *cons*(X_1,*append*(X_2,*cons(E,nil)*)) $\approx$ *cons(1,cons(2,nil))*. Using the equality rules, X_1 is instantiated to *1* and the equation reduces to *append*(X_2,*cons(E,nil)*) $\approx$ *cons(2,nil)*. As before, the only guesses for X_2 are *nil* and *cons*(X_3,X_4). The first guess leads to *cons(E,nil)* $\approx$ *cons(2,nil)*. Using the equality rules this reduces to $E \approx$ *2* & *nil* $\approx$ *nil*. Further applications of the equality rules instantiate E to *2* and reduce the whole expression to *success*. This solves the equation. The second guess for X_2 leads to an equation that cannot be narrowed to *success*, i.e., has no solutions.[3]

[3] All the solutions of this equation are finitely determined, but in general it is undecidable whether an equation has a solution whether or not the computation is by narrowing.

Instead of solving an equation, one could define a specific (recursive) operation for computing the last element of a list. However, the above computation can be executed rather efficiently despite some (expected) wrong guesses. The next section presents a programming language in which the code of a viable operation for computing the last element of a list is based on equation (6).

3. Curry

The programming language used for the examples is Curry (Hanus, 2006). For the most part, a Curry program can be seen as a rewrite system with the constructor discipline. Programs are well typed, which means that operations are applied to meaningful values, e.g., the usual addition is not performed on stacks or, vice versa, *pop* is not applied to a number. A type and its constructors are introduced by a `data` declaration. For example, the type stack discussed in the previous section is defined by:

```
data Stack e = Empty | Push e (Stack e)                    (8)
```

The identifier `e` is a type variable, i.e., it stands for any type. In Curry, constructors start with an upper case letter and variables are in lower case. This convention is opposite to that made in the previous section. The reason is that here the attention is on data constructors because of the role they play in pattern-matching, thus we follow the syntax of Haskell (Peyton Jones and Hughes, 1999). We hope that following well-established, time-proved conventions is preferable, even if it may appear inconsistent at a first glance.

The operations *pop* and *top* are defined as in (2), but in many declarative languages, including Curry, the application of symbols is typically *curried*, i.e., denoted by juxtaposition. Currying supports partial application, i.e., a symbol of arity n is applied to m arguments with $m < n$. Partial application is necessary for higher-order functions, i.e., functions that take other functions as arguments. One of our examples (40) relies on this feature.

```
top (Push e _) = e
pop (Push _ s) = s                                         (9)
```

The computation of `top (Push 1 (pop (Push 2 Empty)))` is performed as in (3) rather than in (4) because Curry's evaluation strategy is *lazy*. Roughly speaking, an expression is evaluated only if the expression's value is needed to obtains the final result and (4) evaluates `pop (Push 2 Empty)` which is not needed in this sense. By contrast, an *eager* strategy would evaluate all the arguments of a function application before applying the function itself. The evaluation strategy is a design decision of the language. Narrowing is defined independently of the evaluation strategy. The strategy only determines where and when to apply a step.

Curry is a functional logic language. Similar to a modern functional language, it declares algebraic types by means of `data` declarations as in (8), and it defines operations on these types by means of defining rules as in (9). However, functional expressions may contain uninstantiated logic variables—examples will be proposed soon—and here is where narrowing comes into play.

If an expression cannot be evaluated (rewritten) because it contains an uninstantiated variable, the variable may be instantiated to enable a rewrite step. For example, the evaluation of (`top x`), where `x` is an uninstantiated variable, instantiates `x` to (`Push e` `s`), where `e` and `s` are new, fresh variables. Operations that are authorized to narrow their arguments, such as `top`, are called *flexible*. In some cases, some expressions should not be narrowed. Operations that are not authorized to narrow their arguments are called *rigid*. In Curry, whether or not to narrow an expression $f(t_1, \ldots, t_n)$ is a compile-time decision that depends only on f. By default, operations are flexible with a few exceptions. Some I/O actions are rigid because, e.g., it would make no sense to guess the value of an uninstantiated variable for printing. Arithmetic operations on numbers are rigid as well in most implementations because of the very large number of potential guesses, though the KICS implementation (Brassel and Huch, 2007) of Curry provides a different approach (Braßel et al., 2008).

Basic types, including integral and floating point numbers, Booleans, characters, tuples and `Success` are built-in. Ubiquitous types are represented in familiar notations, e.g., `"Hello world"` is a string and `[]`, `[0,1,2,3]` and `(x:y)` are lists, the latter with head `x` and tail `y`. The type `Success` is the result type of expressions used in conditions of defining rules. These expressions are referred to as *constraints*. Constraints differ from Boolean values: a Boolean expression reduces to either `True` or `False` whereas a constraint is checked for satisfiability. Rewrite rules in Curry are left-linear, may be overlapping, conditional and/or contain extra variables. Below, we explain these terms.

Left-linear means that each variable in the left side of a rule occurs only once. For example, the following rule is unacceptable since the variable `x` occurs twice in the left side:

```
member x (x:_) = True                                   (10)
```

Requiring left-linearity is only a language design choice. The requirement is neither a problem in theory nor in practice, see (13) below. In programs with the constructor discipline and strict equality, the semantics of non-left-linear rules is reduced to that of other left-linear rules (Antoy, 2001).

Overlapping means that more than one rule may rewrite the same expression. This is useful to code non-determinism in programs. For example, the following rules define the insertion of an element into a list at an unspecified position:

```
insert x y      = x:y
insert x (y:ys) = y:insert x ys                         (11)
```

Let *exp* be `insert 0 [1,2]`. The evaluation of *exp* non-deterministically yields any of three results: `[0,1,2]`, `[1,0,2]` or `[1,2,0]`. The programmer cannot control which result is produced. A non-deterministic operation is generally used in either of two ways. (a) A value produced by a non-deterministic operation is constrained for some specific purpose and the non-determinism may eventually disappear. Many of our examples follow this use. (b) All the values produced by a non-deterministic operation are lazily computed and processed together. This is accomplished with set functions (Antoy and Hanus, 2009). Any operation f implicitly defines an operation, denoted by $f_{\mathcal{S}}$, called *set function of* f that for any argument value(s) computes a set whose members are all and only the values computed by f on the same argument value(s). For example, $\texttt{insert}_{\mathcal{S}}$ `0 [1,2]` produces {`[0,1,2]`,`[1,0,2]`,`[1,2,0]`}. Set functions are automatically generated by a compiler or interpreter rather than being coded by a programmer. Obviously, the set function of an operation f is interesting only when f is non-deterministic.

Non-determinism and non-right-linear rewrite rules, such as the rule of `double` defined below (González Moreno et al., 1999), have a non-obvious interaction. For example, consider:

```
    coin = 0
    coin = 1                                                  (12)
    double x = x + x
```

The two occurrences of variable `x` in the right-hand side of the rule of `double` stand for the same expression. This convention is referred to as the *call-time choice semantics* (Hussmann, 1992) or, more simply, as *sharing*. Sharing implies that the right and left operands of "`+`" in any expression produced by an application of the rule of `double` are one and the same. Therefore, `double coin` has the values, 0 and 2, originating from the two values, 0 and 1 respectively, of `coin`. Sharing is nicely captured by considering expressions as graphs (Echahed, 2008; Echahed and Janodet, 1997; Plump, 1999; Sleep et al., 1993), in particular see (Echahed and Janodet, 1997, Def. 2, cond. 5), or by specialized narrowing calculi (González Moreno et al., 1999). Sharing the variables of non-right-linear rules is a necessary condition to ensure that the result of a computation does not depend on the order of evaluation, see (Antoy and Hanus, 2009) for details.

Conditional means that a rule has a conditional part in addition to the left and right sides. To better understand conditions, we first discuss the equality operation. In Curry, there are two kinds of equality. One, denoted by "`=:=`" and called *constrained*, is defined exactly as in (7). The other, denoted by "`==`", is the traditional *Boolean* equality. In addition to returning different types, the constrained equality is flexible and the Boolean equality is rigid. Intuitively, the constrained equality is invoked to solve an equation where as the Boolean equality is invoked to check whether an equation holds.

Going back to conditional rules, the conditional part defines a test that is performed after an expression has matched the left side of a rule. The test is an expression of type `Success`. The rule is fired if and only if the test succeeds. A Boolean test t is a short hand for t `=:=` `True`. For example, the following operation is an acceptable alternative to (10):

```
    member x (y:ys) | x==y      = True
                    | otherwise = member x ys                 (13)
```

where `otherwise` is a reserved word denoting the value `True`. For programs following the constructor discipline, conditional rules can be transformed into ordinary rules without changing the meaning of a program (Antoy, 2001).

Extra variables are variables that occur in the right side and/or the condition of a rule, but not in the left side. For example, the following operation computes the last element of a list, as described in the previous section:

```
    last l | p++[e]=:=l = e   where p,e free                  (14)
```

The extra variables, declared by a `free` clause in a `where` block, are `p` and `e`. The declaration, required by the syntax of Curry, is provided only for checkable redundancy, similar to the type declaration of an operation. These variables are not bound when an expression is matched to the rule's left side. The operation "`++`" denotes the concatenation of lists. The condition of operation `last` is evaluated by narrowing. The variables `p` and `e` are instantiated, if possible, to satisfy the condition.

The definition of `last` in (14) can be simplified using a *functional pattern* as follows:

```
last (_ ++ [e]) = e                                                    (15)
```

The latter is simpler, lazier and more efficient. A functional pattern extends ordinary patterns by allowing occurrences of defined operations such as "`++`" in this example. Interestingly for our discussion, the semantics of functional patterns (Antoy and Hanus, 2005) is based on narrowing, but the use of narrowing for this purpose goes beyond the scope of this paper.

To complete this terse explanation, the type `Success` has no visible constructors, but there exists a predefined operation, `success`, that evaluates to the singleton value of this type. The constrained conjunction "`&`" evaluates its arguments concurrently in accordance with (7). This concurrent evaluation supports residuation. If during a computation, one operand of `&` residuates on a variable X, the computation continues with the evaluation of the other operand in hope that it will instantiate X so that the evaluation of the residuating operand could resume.

Curry has a variety of other syntactic and semantics features that make it a powerful general-purpose programming language. For a complete description see (Hanus, 2006). Below we mention only a few additional features that may help understanding the examples. Curry allows the compile-time definition of infix binary operators, such as "`++`" in (14), with user-defined precedence and associativity. Curry supports list comprehensions. Curry allows higher-order functions, but without higher-order narrowing.[4] Curry implements input/output declaratively using the monadic style.

4. Programming

This section focuses on the use of narrowing in programs. We present five problems. A problem consists of abstractions represented by numbers, character strings, lists, etc., that we call *elements* of the problem. To solve a problem, we need to compute the values of these elements. To compute these values, we capture some relationships between the elements into sets of equations. Narrowing computes these values by solving the equations. For the programmer, this will turn out to be simpler than designing an algorithm to compute the same values.

In discussing the specification of a problem we present an equation as a pair of expressions with the usual symbol "$=$" (math font) between them. When an equation is translated into Curry, the symbol "$=$" of an equation is translated into the constrained equality "`=:=`" defined in the previous section. We recall that in Curry, the symbol "`=`" (teletype font) is a syntactic separator that occurs in data declarations and rewrite rules.

The first two problems solve equations involving linear data structures. The first problem is very simple and makes a good introduction. The third and fourth problems present examples where narrowing solves equations involving symbolic arithmetic expressions represented by tree-like data structures. All the above problems have, in general, non-deterministic solutions. The fifth problem has a deterministic solution, yet narrowing contributes to a conceptually simple and elegant design. The code of all the examples has been compiled and executed using PAKCS (Hanus, 2008), a mainstream

[4] *Higher-order narrowing* refers to computations in which a narrowing step narrows a variable whose type is a function.

compiler/interpreter of Curry. Some consequences of using narrowing in the design and code of a program will be discussed later.

4.1. A Simple Example

The following problem (Problem E) is from the 2000 ACM Pacific NW Region Programming Contest (ACM, 2000). An age-old encryption technique "hides" a message m into a string of text t by embedding one character of m every n characters of t. For example, the message "`Hello World`" is hidden in "`aHaealalaoa aWaoaralad`" with embedding 2. The problem is to find n given m and t.[5]

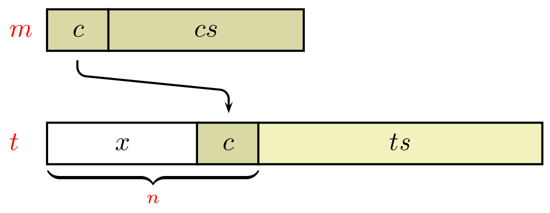
To code this problem into a program that uses narrowing, it is convenient to capture the relation between the elements of the problem, i.e., m, t, and n, as a system of equations. The condition that m and t are given and n is unknown is largely irrelevant. For the beginner, it might be easier to imagine that all the elements of the problem are alike and that the equations only "state" the relation between these elements.

Suppose that m is not empty and that c denotes the first character of m. The immediate condition between c and t is that c should be the n-th character of t. This condition leads to the following system of equations, where *append* and *cons* were defined in (5) and *length* computes the length of its argument:

$$\begin{cases} t = append(x, cons(c, ts)) \\ length(x) + 1 = n \end{cases} \tag{16}$$

In these equations, x represents the first $n-1$ characters of t and ts the characters of t past the first n.

Lists and other "linear" data types have an intuitive graphical or diagrammatic representation. This representation might help formulating the relation between the various elements of a problem. For the problem under discussion, a diagram equivalent to (16) is shown below. Often, it is easier to sketch the diagram before formulating the equations.



(17)

Equations (16) consider the first character of m only. Thus, in a program, it is natural to give a name to these equations and invoke them recursively for each following character of the message. This is shown in the first rule of (18). The only remaining concern to obtain a program is to terminate the recursion. If m is an empty string, then it is hidden

[5] The solution is non-deterministic. For example, "`xy`" is hidden in "`xyxzzy`" with embeddings 1 and 3. The formulation of the problem, which we borrowed from an ACM Contest, seems to ignore this possibility, though.

in every t and it does not define n. This is shown in the second rule of (18). The following program fragment implements the problem:

```
hide (c:cs) t n
   | t =:= x ++ c:ts & length x + 1 =:= n
   = hide cs ts n                                              (18)
   where x,ts free
hide [] _ _ = success
```

There are a couple of further remarks on the translation of (16) into (18). Operation `hide` should be initially invoked with the first argument different from empty. If the message is initially empty, it does not constrain the embedding. The program correctly reflects this condition. If `n` is an uninstantiated variable, it correctly remains uninstantiated. If the message is not empty and the embedding `n` is an uninstantiated variable—as per the problem's original formulation—then `n` is instantiated by the first invocation of `hide`. After that, `n` remains constant through the recursive invocations.

The integer addition operation, "`+`", is rigid in Curry, though some implementations (Brassel and Huch, 2007) make it flexible. Ideally, all the operations invoked for the satisfaction of a constraint should be flexible to ensure the completeness of the execution. In this case, the execution of (18) is complete because the first equation instantiates `x` to a list, thus `length x+1` is an integer and if `n` is free, it is instantiated to this integer.

We discuss narrowing-free solutions of all the problems presented in this paper. A "narrowing-free solution" is a program, coded in the same language and for solving the same problem, which computes the result without executing narrowing steps.

How can we code this problem without narrowing? We implement the same overall algorithm: check whether the first character of the message is the n-th character of the text and recur. The new version of `hide`, which we denote with `hide'`, has the same arguments, but its return type is Boolean. Without narrowing, there are no uninstantiated variables in the program. In particular, `hide'` is repeatedly called for every potential embedding `n`. The returned value reports whether, for a specific `n`, the message is indeed embedded in the text. The program uses the list index operation "`!!`" to extract the n-th character of the text and the `drop` operation to compute the suffix of the text passed to the recursive call.

```
-- invoke hide' m t x
-- for x in [0 .. length t `div` length m]

hide' (c:cs) t n = if t !! (n-1) == c                          (19)
                   then hide' cs (drop n t) n
                   else False
hide' [] _ _ = True
```

The narrowing-free version of the program is not much different from the narrowing-based version, but the operation `hide'` contains more details than a direct implementation of (17). In fact, our first attempt was incorrect because we had forgotten the detail that the n-th element of a list has index $n-1$ rather than n. A more important difference is that the narrowing-free version requires additional code to iterate calls to `hide'` for all the potential embeddings and to test the returned values. A more general and pervasive difference is discussed next.

In this problem, the operation `hide` was proposed to compute the embedding of a message in a text. The embedding is computed when the third argument of an invocation `hide` $m\ t\ n$ is a free unbound variable. If the message m is embedded in the text t, this invocation binds n to the embedding. However, this operation is much more versatile. For example, it can *extract* the message, if only the text and the embedding are given and the message is a free variable. Or it can *generate* the text, if only the message and the embedding are given. In the first case, i.e., to extract the message, the problem

	t	m	n	meaning
bound	yes	yes	no	find encoding
bound	yes	no	yes	extract message
bound	no	yes	yes	generate text

Fig. 1. Intended meaning of an invocation `hide` $t\ m\ n$ for various combination of bound and unbound (free) arguments.

statement does not define the length of the message. One could refine the problem with either an additional parameter defining the length of the message or the assumption that the message has maximum length. The latter can be accomplished by replacing the second argument of the second rule of `hide` with the empty list. In the second case, i.e., to generate the text, the characters of the text that do not originate from the message would remain uninstantiated variables. A simple additional equation could instantiate these variables to characters of some alphabet or, vice versa, verify that they are characters of the alphabet when `hide` is executed for a purpose different from the generation of the text.

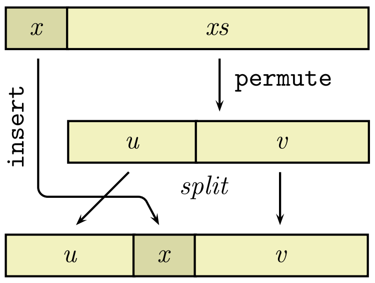
Finally, the above computations address the case in which two of the three elements of the problem are known because these computations have an intuitive practical meaning. However, computations in which two of the three elements are unknown do not require additional code either. For example, one can execute `hide` to extract the messages for all possible embeddings.

This versatility of narrowing is sometimes referred to as "running functions backward" since the arguments of an operation application can be computed from its returned value. A concrete use of this feature will be presented shortly.

4.2. Narrowing Lists

The n-queens problem is a popular puzzle. Simply stated, it requires to place n queens on an $n \times n$ chess board so that the queens cannot capture each other. A typical implementation of this problem represents a placement of queens on the board as a permutation of $1, 2, \ldots, n$, where the i-th element of the permutation is the row of the queen placed in column i. With a generate-and-test architecture, the program enumerates the permutations of $1, 2, \ldots, n$ and tests whether each permutation represents a safe placement of the queens.

The following diagram shows how to compute a permutation of a list. Informally, first one permutes the tail of the input list, then non-deterministically splits the result into a prefix and a suffix, and finally inserts the head of the input list between them.



(20)

Narrowing easily splits a list p into two sublists u and v by solving the following equation:

$$p = append(u, v) \tag{21}$$

If n is the length of p, equation (21) has $n + 1$ solutions for u and v. The following operation, `permute`, codes the algorithm sketched by (20) to compute permutations. As expected, the fact that equation (21) may have several solutions makes program (22) non-deterministic, i.e., `permute` returns any of the permutations of its argument:

```
permute [] = []
permute (x:xs) | permute xs =:= u ++ v
               = u ++ x:v
               where u,v free
```

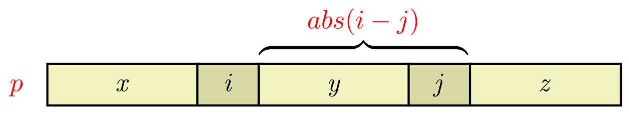
(22)

Narrowing slightly simplifies coding this problem into a program. Without narrowing we would have to define and invoke an operation for splitting a list into two sublists. This is an example of running a function backward mentioned earlier. The arguments, u and v, of a concatenation are determined from the result. There exist also a formulation of `permute` that does not split a list explicitly, but uses the non-deterministic insertion of an element in a list shown in (11).

```
permute [] = []
permute (x:xs) = insert x (permute xs)
    where insert x y = x:y
          insert x (y:ys) = y : insert x ys
```

(23)

The representation of a placement as a permutation of the rows of the board ensures that no two queens can be in the same row or the same column. Thus, two queens capture each other only if they are on a diagonal. Being on a diagonal translates into the condition that the distance between two queens' rows is the same as the distance between the queens' columns. The diagram expressing this condition is shown below, where the list p represents a placement as a permutation of $1, 2, \ldots, n$, the operation *abs* denotes the absolute value function and the variables i and j are the rows of two queens capturing each other:



(24)

This diagram is formalized by the following system of equations, where p is a placement of the queens on the board and x, y and z are sublists of p.

$$\begin{cases} p = append(x, cons(i, append(y, cons(j, z)))) \\ abs(i-j) = length(y) + 1 \end{cases} \tag{25}$$

The operation `unsafe` defined below names the above system of equations. A functional pattern "extracts" the components of the argument, defined in the first equation of (25), that are used in the second equation of (25).

```
unsafe (_ ++ i:y ++ j:_) = abs (i-j) =:= length y + 1                (26)
```

For some permutation p, if the system of equations represented by `unsafe` p has a solution, p is *not* a solution of the puzzle. Narrowing finds any solution of (26). Hence, some p is a solution of the puzzle if and only if `unsafe`$_{\mathcal{S}}$ p, the set function of `unsafe` applied to p, produces the empty set.

Narrowing conceptually simplifies coding this problem into a program. A narrowing-free Curry program that implements the same algorithm as (26) must execute nested iterations over the columns of the board, which are represented by indexes of a permutation. For any two columns, the corresponding rows are computed with the index operation. The simplest code we could think of for this computation is a list comprehension. As in previous examples, the type of `unsafe'`, the narrowing-free version of `unsafe`, is Boolean.

```
unsafe' p = or [ abs (p!!i-p!!j) == j-i
                | i<-[0..n-2], j<-[i+1..n-1] ]                        (27)
            where n = length p
```

The narrowing-free program contains more details, e.g., the bounds of the iterations over the indexes of a permutation, than the narrowing-based program. It less directly implements (24) and it has a somewhat more imperative flavor. There exist other algorithms for the n-queens that could be coded in, or adapted to, Curry, e.g., (Bird and Wadler, 1988, pages 161–165). The amount of detail in the corresponding programs is very similar to (27).

4.3. Narrowing Trees

The examples discussed in the previous sections show the applicability of narrowing to problems involving lists. A list is a simple structure for representing a collection of elements and a collection of elements is at the core of many problems. To solve these problems, often the programmer formalizes relationships between the values of some elements and/or their positions in the collection. Diagram (24) is typical in that it formalizes and relates both these aspects. For example, Problem D of (ACM, 2000) requires swapping two cards of a deck. The diagram depicting the relation between the elements of this problem is very similar to (24), except for a different condition on the positions of some elements in the list.

In this section, we show that the applicability of narrowing is not limited to lists; narrowing can solve systems of equations involving any algebraically defined data type. These equations, as for lists, formalize relationships between components and/or their positions in composite structures. The problem is to simplify a symbolic arithmetic expression, such as $1*(x+0)$, implemented by a tree-like structure. In our implementation, a symbolic expression is a value of the following type.

```
data Exp = Lit Int
         | Var [Char]
         | Add Exp Exp                                        (28)
         | Mul Exp Exp
```

The non-deterministic operation `reduce` defines a handful of *elementary reduction pairs.* Obviously, many more are possible, but the following ones suffice to make our point. A pair of expressions (r, s) is an elementary reduction pair if and only if there is a reduction of r into s.

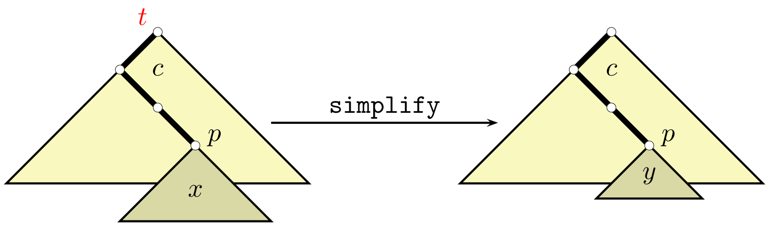
```
reduce (Add (Lit 0) x) = x
reduce (Add x (Lit 0)) = x
reduce (Mul (Lit 1) x) = x
reduce (Mul x (Lit 1)) = x
reduce (Mul x (Lit 0)) = Lit 0                                (29)
reduce (Mul (Lit 0) x) = Lit 0
reduce (Add (Lit n) (Lit m)) = Lit (n+m)
reduce (Mul (Lit n) (Lit m)) = Lit (n*m)
reduce (Add x y) | x == y = Mul (Lit 2) x
```

At the core of our design is an operation, called `replace`, similar to the concatenation of lists, but operating on symbolic expressions. The concatenation operation makes a list out of two lists. The `replace` operation makes an expression out of two expressions referred to as *context* and *replacement.* It also takes an additional argument referred to as the *position.* The position is a possibly empty sequence of positive integers identifying the replacement in the context.

```
replace _ [] x = x
replace (Add l r) (1:p) x = Add (replace l p x) r
replace (Add l r) (2:p) x = Add l (replace r p x)             (30)
replace (Mul l r) (1:p) x = Mul (replace l p x) r
replace (Mul l r) (2:p) x = Mul l (replace r p x)
```

The operation "`++`" defined earlier is similar to `replace` in the following sense. Its first argument is the context and its second argument is the replacement. It does not take the position argument because the position of the replacement is always that of *nil* in the context. It is easy to define a replace-like function for any tree-like type.

To simplify an expression t, we locate in t at some position p some subexpression x such that, for some y, (x, y) is an elementary reduction pair. Then, the result of the simplification of t will be equal to t except for y in the place of x. The following diagram depicts the situation we are presenting.



(31)

The context c does not play any significant role except abstracting all the expressions that differ from each other only at the position p. Diagram (31) is formalized by the following system of equations:

$$\begin{cases} t = \mathit{replace}(c, p, x) \\ \mathit{reduce}(x) = y \\ \mathit{simplify}(t) = \mathit{replace}(c, p, y) \end{cases} \tag{32}$$

Finally, the simplification operation, `simplify`, is a straightforward implementation of (31) and (32) defined using a functional pattern.

```
simplify (replace c p x) = replace c p (reduce x)                    (33)
```

The application of `simplify` to `Mul (Lit 1) (Add (Var "x") (Lit 0))` yields either `Add (Var "x") (Lit 0)` or `Mul (Lit 1) (Var "x")`, non-deterministically. A further application of `simplify` to either of these expressions yields `Var "x"`.

If an expression t cannot be simplified, `simplify` t simply fails; otherwise, it non-deterministically executes a single reduction step. The application of repeated reduction steps to an expression until no more reduction steps are available can be controlled using the set function of `simplify`.

As we mentioned earlier, when computations are executed by narrowing, operations become more versatile. This fact can be verified by the intended use of each of the two occurrences of `replace` in `simplify`. In the functional pattern, `replace` is invoked to find in the argument of `simplify` a subexpression that can be simplified, whereas in the body, `replace` is invoked to indeed replace that subexpression with its simplification.

How can we code this problem without narrowing? It would seem reasonable to use pattern matching to determine whether an expression is the first component of an elementary reduction pair. Other alternatives are considerably more complicated. The use of pattern matching leads to an operation, which we denote with `simplify'`, that traverses an expression to simplify it. The argument of `simplify'` is matched against the first component of a pair, whereas the corresponding second component makes up the right-hand side. E.g.,

```
simplify' (Add x (Lit 0)) = x
... -- other Add-rooted patterns
simplify' (Add x y) = Add (simplify' x)                    (34)
                          (simplify' y)
...
```

Merging the simplification rules with the traversal of an expression is not an appealing design because it leads to more complicated and less modular code. Another complication is to detect when an expression cannot be further simplified. The operation `simplify` fails, but the same technique does not seem applicable for controlling `simplify'` and additional machinery becomes necessary.

There exist techniques to alleviate some of these difficulties, but they require somewhat "compiling" the elementary simplification pairs into the program.

4.4. Non equational narrowing

The examples discussed in the previous sections show the applicability of narrowing to solve equations. We recall that an equation is an expression of type `Success` whose leading symbol is the *constrained equality* operator "`=:=`". It is possible to narrow an expression of any user-defined type whose leading symbol is any user-defined operation.

The problem of this section, which relies on definitions of the previous section, is to find a common subexpression in an expression. The solution is straightforward: x is a common subexpression of t if it occurs at two distinct positions p and q of t. The relationship between p and q can be strengthened. Since the problem calls for common subexpressions of an expression, it must be that the position of one subexpression is to the left of the position of the other subexpression. We denote the relationship "is to the left of" among positions with "$\prec$". Without loss of generality we assume that p is to the left of q. A set of conditions defining the solution follows.

$$\begin{cases} t = \mathit{replace}(c, p, x) \\ t = \mathit{replace}(c, q, x) \\ p \prec q \end{cases} \tag{35}$$

We remark that the third condition, which is essential to specify the problem that we are discussing, is not an equation. We need to conjoin, with the constrained conjunction operator "`&`", the conditions of (35). Hence, the return type of the comparison of p and q must be `Success`. A Curry infix operation, denoted by "`<:`" which implements "$\prec$", is defined below.

```
(1:_) <: (2:_) = success
(1:x) <: (1:y) = x <: y                                    (36)
(2:x) <: (2:y) = x <: y
```

The problem is implemented by the operation `commonSubexp` which directly encodes (35) in Curry. No simplification rules are used for this problem and no actual replacement of expressions takes place.

```
commonSubexp t | replace c p x =:= t &
                 replace c q x =:= t &
                 p <: q                                    (37)
               = x
               where c,p,q,x free
```

Coding a simple narrowing-free version of `commonSubexp` does not appear a simple task. One option would be to code nested traversals of the input expression in order to compare subexpressions pairwise. This problem is similar to that posed by `unsafe` in a previous section, but in this case there is no "expression comprehension" notation to ease the task. Another option would be to collect, e.g., in a list, all the subexpressions of the input expression and look for a repeated subexpression. Both options are not as close to the specification as the narrowing-based program. They are less straightforward and elegant, require more code and contain substantially more details.

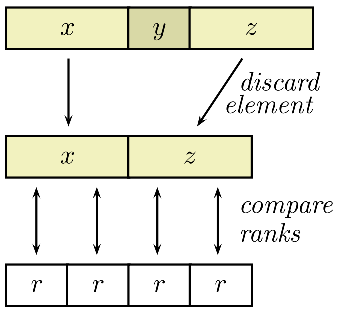
4.5. Deterministic Computations

The problems discussed in the previous sections have, at least in principle, non-deterministic results. Narrowing seems a natural choice to solve problems of this kind since narrowing computations are non-deterministic as well. In this section, we show that narrowing can be conveniently applied also to problems with a deterministic result.

The problem proposed in this section is to determine whether a poker hand features a four-of-a-kind. Several other game combinations, such as two-of-a-kind, three-of-a-kind, full-house, etc. would be similarly processed. A *card* is represented by its *suite* and *rank*. The operation `rank` returns the rank of a card.

```
data Suit = Clubs | Spades | ...
data Rank = Ace | King | ...
data Card = Card Rank Suit                                 (38)
rank (Card r _) = r
```

A set of cards is represented by a list. Thus, a poker *hand* is a list of five cards. The narrowing-based algorithm discards one non-deterministically chosen card from the hand and it checks whether the ranks of the four remaining cards are the same. The diagram of this algorithm is:



(39)

The Curry code is a straightforward implementation of the diagram. The operation `four` takes a hand and it succeeds if and only if the hand features a four-of-a-kind. Variations that return the rank, `r`, and/or the fifth card, `y`, and/or the four cards, `x++z`, witnessing the four-of-a-kind score are immediate. A Boolean outcome can be obtained with the set function of `four`

```
four (x++[y]++z)
   = map rank (x++z) =:= [r,r,r,r]                    (40)
   where r free
```

Several narrowing-free algorithms come to mind to solve the same problem.

Sort and test. Sort the hand according to rank and test whether the first or last four cards have the same rank. The double test is necessary because the rank of the four-of-a-kind may either precede or follow the rank of the "fifth" card, i.e., the card that does not contribute to the four-of-a-kind. This algorithm can be adapted to other game combinations, but it is not as convenient for some combinations, e.g., two-of-a-kind.

Rank counters. For each rank, count how many cards of the hand have that rank, then test whether one counter is four. This algorithm is not as terse as the others, but it generalizes easily to other game combinations. It has the drawback that the fifth card of the hand is not immediately accessible. Further code is needed to determine it.

Decision tree. Pick a card of the hand and compare the rank of some cards of the hand with the rank of the picked card and make suitable decisions. This algorithm is likely to be efficient, but possibly more tedious and error-prone to code, and it cannot be parameterized for other game combinations.

There is also a narrowing-free algorithm inspired by the narrowing-based algorithm. This algorithm iterates over the cards of a hand. At each iteration, the algorithm removes one card from the hand and it checks whether the remaining cards all have the same rank. This algorithm requires the programmer to code some operations that are not invoked in the narrowing-based algorithm and are not likely to be found in a library, e.g., an operation that removes the n-th element of a list, and an operation that checks whether all the elements of a list are equal. Further code is necessary to control the iteration over the elements of a hand and to "glue" together the above functions. Finally, if an application calls for some additional information beside a boolean outcome, e.g., the rank of the four-of-a-kind, additional code would be needed.

5. Highlights of When, Why and How to Narrow

In this section, we highlight some general situations in which the programmer should consider employing narrowing to solve a problem. We have seen that the problems discussed in the examples have both narrowing-based and narrowing-free implementations. Thus, *when* to use narrowing is largely a matter of preference and convenience. Many narrowing-based implementations were conceptually simpler and textually shorter than their narrowing-free counterparts. Below we abstract some conditions that are likely to provide these benefits and thus suggest to look for narrowing-based solutions.

Frequently, programmers are given a specification rather than an algorithm and are called to design and code programs satisfying the specification rather than implementing an algorithm. The specification establishes relationships between the elements of a problem—sometimes without specifying how some elements are identified or should be computed. All the programs proposed in the previous section witness to some degree this practice. E.g., the specification of the problem of Sect. 4.4 is: *x is a common subexpression of t if and only if x occurs at two positions p and q of t such that p is to the left of q.* Neither the common subexpression nor its positions are precisely identified in the specification. Likewise, the specification of the problem of Sect. 4.5, which was left to the intuition, is: *a poker hand features a four-of-a-kind if and only if there exists a rank r such that four cards of the hand have rank r.* In this case, too, neither the rank nor the four cards are precisely identified in the specification. All the other problems of the previous section exhibit similar characteristics.

Our examples show that sometimes the specification of a problem is a relation among elements of the problem in which some of these elements may be existentially quantified. Furthermore, some of these elements must be identified, and their values computed, to solve the problem. We have seen in Section 2 that narrowing evaluates expressions containing uninstantiated variables that stand for unknown values. Consequently narrowing allows us to treat certain specifications as executable programs. This is the reason *why* for certain problems narrowing is a very convenient programming feature.

The final aspect of this section is a discussion of *how* narrowing can be employed to "execute" a specification. Specifications come in various degrees of formality. E.g., "*x is a common subexpression of t if and only if x occurs at two positions p and q of t such that p is to the left of q*" is clear and precise, but it does not formally define the meaning of key concepts such as "occur" or "to the left of". To employ narrowing, the programmer must define and code types and operations to formalize the elements that play a role in the specification. E.g., "position" was declared as a sequence of positive integers, "occur" was abstracted and generalized by the operation `replace` and "to the left of" was defined by the operation "`<:`".

To solve a programming problem, the programmer must encode in the implementation language certain elements of the problem's specification. This encoding seems an unavoidable effort whether or not a program makes use of narrowing. Rather, we argue that in narrowing-based programs this effort is likely to be smaller than in narrowing-free programs because narrowing both computes with incomplete information and runs functions backward.

6. Conclusion

Rewriting is a model of computation. It consists of rewrite rules that may be used to describe problems in mathematics, engineering, and everyday events. The application of these rules is seen as a computation or deduction. Narrowing generalizes rewriting by applying these rules when some information about a problem is unknown.

Narrowing supports programming at a very high level of abstraction. Programmers sometimes sketch diagrams to express the relationships between the elements of a problem. Then, they translate these diagrams into programs. Narrowing makes this translation largely superfluous because it has the potential to "execute" the diagrams or their corresponding specifications. This approach to programming becomes particularly convenient when some elements of a specification are not precisely identified or explicitly given a value.

The solutions of the problems of Section 4 show that code employing narrowing is generally more declarative, conceptually simpler and textually shorter than equivalent code that does not employ narrowing. The code is more declarative because its structure is a direct encoding of relations, often equations, that specify a problem or program. Fewer symbols (operations and/or variables) need to be coded or invoked because solving a relation or an equation for different variables is equivalent to execute different operations which, in other programming styles or paradigms, must be independently coded. With narrowing there are fewer dependencies to understand in a program because both there are fewer symbols in the program and the dependencies between these symbols are more explicit.

To the extent shown in Section 4, designing, coding, testing and documenting software are faster because both less code is produced and the code is more declarative. A software artifact that employs narrowing can be developed more quickly and at a lower cost because it is simpler, shorter and it has a faster life cycle. For the same reasons, it is less likely to contain undetected errors and it is easier to maintain.

Note

The Portable Document Format version of this paper, available from the author homepage at `http://www.cs.pdx.edu/~antoy/homepage/publications.html` contains active links to the code of the programs referenced in the text:

`e.curry`	Problem E of the 2000 ACM Pacific NW Region Programming Contest, Sec. 4.1
`queens.curry`	Solve the n-queens puzzle, Sec. 4.2
`simplify.curry`	Simplify symbolic or arithmetic expressions, Sec. 4.3
`common.curry`	Find common subexpressions of an expression, Sec. 4.4
`poker.curry`	Find if a poker hand scores a four-of-a-kind, Sec. 4.5

Acknowledgements

I am grateful to Michael Hanus for his comments and corrections on a preliminary version of this paper, to Olivier Danvy for his encouragement to complete an early version of this paper, and to Mark Jones for discussing narrowing-free solutions of some problems of Section 4.

A. Appendix

Various approaches have been proposed to model the functional logic computations discussed in this paper. Prominent among them are: (1) a rewriting logic (González Moreno et al., 1999) that addresses subtle aspects of equality and sharing, (2) operational semantics (Alpuente et al., 2005a; Tolmach et al., 2004) based on *heaps* and *stores* specifically developed for the interaction of non-determinism and sharing, and (3) term (Baader and Nipkow, 1998; Bezem et al., 2003; Dershowitz and Jouannaud, 1990) and graph (Echahed, 2008; Echahed and Janodet, 1997; Plump, 1999; Sleep et al., 1993) rewriting. Rewriting is the framework for the definition of narrowing and it has been particularly fruitful for the development of both evaluation strategies (Antoy, 2005; Antoy et al., 2000; Echahed, 2008; Echahed and Janodet, 1997) and program transformations (Antoy, 2001; Antoy and Hanus, 2005, 2006) that support efficient implementations.

This appendix reviews key concepts of rewriting, presents a formal definition of narrowing, and sketches an efficient evaluation strategy for narrowing computations. Source Curry programs allow a rich set of syntactic and semantic features, e.g., conditional rules, partial application and functional patterns, that are not found in the rewrite systems that we present below. The class of systems that we define plays the role of an easy-to-execute *core language* to which source programs are mapped via semantic-preserving transformations. This class is well-suited for reasoning about computations, e.g., for addressing properties such as soundness, completeness and optimality of narrowing strategies, and for implementing functional logic languages.

A.1. Definition of Narrowing

A *signature* Σ is a non-empty, finite set S of *symbols* together with an *arity* function $a : S \to \mathbb{N}$ that for each s in S defines the number of arguments that s expects. Let $\mathcal{X}$ be a countably infinite set of *variables*. A term is a tree (or graph) whose nodes are labeled by symbols and/or variables. Graphs model better than trees the condition that a variable is a singleton element in an expression, and consequently can have at most one instantiation, but the treatment of graphs is more complicated. Trees serve the same purpose with some conventions or conditions, such as the call-time choice semantics (Hussmann, 1992), which are not as elegant, but keep the presentation simpler. Therefore, we define terms as trees.

The set $\mathcal{T}(\Sigma, \mathcal{X})$ of *terms* over $\Sigma \cup \mathcal{X}$ is inductively defined as follows: x is a term for every $x \in \mathcal{X}$, and if $f \in \Sigma$, $a(f) = n$, and $t_1, \ldots t_n$ are terms, then $f(t_1, \ldots t_n)$ is a term. Ill-typed terms can be banned using a many-sorted signature, but the much simpler arity function suffices for defining both narrowing and strategies.

An *occurrence* or *position* is a sequence of positive integers identifying a subterm in a term. For every term t, the empty sequence identifies t itself. For every term of the form $f(t_1, \ldots, t_n)$, the sequence $i \cdot p$, where i is a positive integer not greater than n and p is a position, identifies the subterm of t_i at p. The subterm of t at p is denoted by $t|_p$ and the result of *replacing* $t|_p$ with s in t is denoted by $t[s]_p$. A term t is *linear* iff any variable occurs in t at most once.

The set of symbols of a signature Σ is partitioned into two disjoint sets: $\mathcal{C}$, whose elements are called *constructors*, and $\mathcal{D}$, whose elements are called *operations*. The terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$ are called *constructor terms*. A term $t = f(t_1, \ldots t_n)$, with $n \geqslant 0$, is called a *pattern* iff t is linear, $f \in \mathcal{D}$ and $t_1, \ldots t_n$ are constructor terms. A *rewrite system* $\mathcal{R}$ is a set of *rewrite rules*, pairs of terms written $l \to r$, where l is a pattern. There are also rewrite systems whose rules left-hand sides are not patterns, but they are not interesting for our discussion.

A *substitution* σ is a mapping $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ such that $\{x \in \mathcal{X} \mid \sigma(x) \neq x\}$, the *domain* of σ, is finite. Substitutions are extended to terms by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$, for every term $f(t_1, \ldots, t_n)$. A term u is an *instance* of a term t iff there is a substitution σ with $u = \sigma(t)$. In this case, we write $t \leqslant u$. A term u is a *variant* of a term t iff $t \leqslant u$ and $u \leqslant t$. A variant of a rule R is *fresh* iff every variable in R does not occur in any term to which R is applied. A substitution σ is a *unifier* of terms t and u iff $\sigma(t) = \sigma(u)$.

A *narrowing step* of a term t according to a rewrite system $\mathcal{R}$ is a triple $\langle l \to r, p, \sigma\rangle$ such that $l \to r$ is a fresh variant of a rule of $\mathcal{R}$, p is a position of t such that $t|_p$ is not a variable, and σ is a unifier of $t|_p$ and l. A term t *narrows* to a term u with step $\langle l \to r, p, \sigma\rangle$ iff $u = \sigma(t[r]_p)$.

A.2. Overview of the Strategy

A narrowing computation is the repeated transformation by narrowing steps of a term until no further step is applicable. The computation of a term t is *successful* iff it produces a term u in $\mathcal{T}(\mathcal{C}, \mathcal{X})$, i.e., u is a constructor term, which is called a *value* of t. In general, many steps are applicable to a term t. In particular, according to our definition, there could be an infinite number of unifiers. Executing all these steps to ensure that all the values of t are obtained could be unfeasible or computationally prohibitive. A narrowing strategy is the policy or algorithm that selects which steps to execute in t. A "good" strategy should produce the steps that ensure that all the values of t are computed and it should do so without producing steps that do not contribute to the computation of any value.

Good strategies are known for classes of rewrite systems that impose some conditions on their rewrite rules (Antoy, 2005). These conditions are imposed only on the core language into which source programs are transformed for execution and hence are not a significant problem for a programming language. The *inductively sequential* rewrite systems (Antoy, 1992) with extra variables are an adequate class for the core language (Antoy and Hanus, 2006). In the inductively sequential systems, the rules defining every operation are organized in a hierarchical structure called a *definitional tree* (Antoy, 1992). A formal presentation of definitional trees goes beyond the scope of this paper. Rather, we sketch on an example the idea behind this concept and how this concept is applied to determine the steps of a term.

Not every operation has a definitional tree. However, every rewrite system as defined earlier can be transformed into a semantically-equivalent rewrite system in which every operation has a definitional tree (Antoy, 2001). For example, the operation, defined below, that merges two lists has a definitional tree. The right-hand sides of the rules are irrelevant, thus we omit some details.

```
merge [] y = y
merge (x:xs) [] = x:xs                                    (A.1)
merge (x:xs) (y:ys) = if x<=y then ...
```

Operation `merge` makes an initial distinction on its first argument. The cases on this argument are empty list and non-empty list. In the non-empty list case, operation `merge` makes a subsequent distinction on its second argument. The cases on this argument are again empty list and non-empty list. A definitional tree of the operation `merge` encodes these distinctions, the order in which they are made, and the cases that they consider. Thus, e.g., the evaluation of $t =$ `merge(merge(`t_1`,`t_2`),`t_3`)` by a strategy that uses definitional trees will start with the evaluation of `merge(`t_1`,`t_2`)`. If it is the empty list, t will be reduced to t_3 according to the first rewrite rule. If it is a non-empty list, and only in this case, t_3 will be evaluated to determine whether it is an empty or non-empty list and consequently the second or third rule should be applied.

Evaluations guided by definitional trees resemble evaluations guided by case expressions found in some modern functional languages (Peyton Jones and Hughes, 1999), but there are significant differences. Case expressions are coded by the programmer, whereas definitional trees are inferred, by a simple algorithm (Antoy, 2005; Barry, 1996) from the rewrite rules of an operation. Evaluations via definitional trees impose neither a top-to-bottom precedence among the rewrite rules nor a left-to-right precedence among the arguments. Definitional trees allow both the evaluation of terms with unbound variables, which supports narrowing and, with some variation or extension, the definition of operations with overlapping rules, which supports non-determinism (Antoy, 2005).

References

ACM, 2000. ACM Pacific NW Region Programming Contest. Available at `http://icpc.baylor.edu/past/icpc2001/regionals/PacNW00/` [Accessed 16 March 2009].

Aït-Kaci, H., 1990. An overview of LIFE. In: Schmidt, J. W., Stogny, A. A. (Eds.), Proceedings of the Workshop on Next Generation Information System Technology. Springer LNCS 504, pp. 42–58.

Albert, E., Hanus, M., Vidal, G., 2002. A practical partial evaluator for a multi-paradigm declarative language. Journal of Functional and Logic Programming 2002 (1).

Alpuente, M., Falaschi, M., Vidal, G., 1996. Narrowing-driven partial evaluation of functional logic programs. In: ESOP '96: Proceedings of the 6th European Symposium on Programming Languages and Systems. Springer-Verlag, London, UK, pp. 45–61.

Alpuente, M., Falaschi, M., Vidal, G., 1998. A unifying view of functional and logic program specialization. ACM Comput. Surv., 9.

Alpuente, M., Hanus, M., Lucas, S., Vidal, G., 2005a. Specialization of functional logic programs based on needed narrowing. Theory and Practice of Logic Programming 5 (3), 273–303.

Alpuente, M., Lucas, S., Falaschi, M., Vidal, G., Hanus, M., 2005b. Specialization of functional logic programs based on needed narrowing. Theory Pract. Log. Program. 5 (3), 273–303.

Antoy, S., Sep. 1992. Definitional trees. In: Kirchner, H., Levi, G. (Eds.), Proceedings of the Third International Conference on Algebraic and Logic Programming. Springer LNCS 632, Volterra, Italy, pp. 143–157.

Antoy, S., Sep. 1997. Optimal non-deterministic functional logic computations. In: Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97). Springer LNCS 1298, Southampton, UK, pp. 16–30.

Antoy, S., Sep. 2001. Constructor-based conditional narrowing. In: Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01). ACM, Florence, Italy, pp. 199–206.

Antoy, S., 2005. Evaluation strategies for functional logic programming. Journal of Symbolic Computation 40 (1), 875–903.

Antoy, S., Echahed, R., Hanus, M., Jul. 2000. A needed narrowing strategy. Journal of the ACM 47 (4), 776–822.

Antoy, S., Hanus, M., Mar. 2000. Compiling multi-paradigm declarative programs into Prolog. In: Proceedings of the Third International Workshop on Frontiers of Combining Systems (FroCoS 2000). Springer LNCS 1794, Nancy, France, pp. 171–185.

Antoy, S., Hanus, M., Sep. 2005. Declarative programming with function patterns. In: 15th Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2005). Springer LNCS 3901, London, UK, pp. 6–22.

Antoy, S., Hanus, M., Aug. 2006. Overlapping rules and logic variables in functional logic programs. In: Twenty Second International Conference on Logic Programming. Springer LNCS 4079, Seattle, WA, pp. 87–101.

Antoy, S., Hanus, M., Sep. 2009. Set functions for functional logic programming. In: Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009). Lisbon, Portugal, pp. 73–82.

Antoy, S., Hanus, M., Liu, J., Tolmach, A., Sep. 2005. A virtual machine for functional logic computations. In: Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004). Springer LNCS 3474, Lubeck, Germany, pp. 108–125.

Antoy, S., Hanus, M., Massey, B., Steiner, F., Sep. 2001. An implementation of narrowing strategies. In: Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01). ACM Press, Florence, Italy, pp. 207–217.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Barry, B., 1996. Needed narrowing as the computational strategy of evaluable functions in an extension of Gödel. Master's thesis, Portland State University.

Bezem, M., Klop, J. W., de Vrijer (eds.), R., 2003. Term Rewriting Systems. Cambridge University Press.

Bird, R., Wadler, P., 1988. Introduction to Functional Programming. Prentice Hall, New York, NY.

Braßel, B., Fischer, S., Huch, F., 2008. Declaring numbers. Electron. Notes Theor. Comput. Sci. 216, 111–124.

Brassel, B., Huch, F., Oct. 2007. The Kiel Curry System KiCS. In: Seipel, D., Hanus, M. (Eds.), Preproceedings of the 21st Workshop on (Constraint) Logic Programming (WLP 2007). Würzburg, Germany, pp. 215–223, technical Report 434.

Caballero, R., Sánchez, J. (Eds.), 2007. TOY: A Multiparadigm Declarative Language (version 2.3.1). Available at `http://toy.sourceforge.net`.

Christiansen, J., Fischer, S., 2008. EasyCheck - test data for free. In: Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008). Springer LNCS 4989, pp. 322–336.

Dershowitz, N., Jouannaud, J., 1990. Rewrite systems. In: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science B: Formal Methods and Semantics. North Holland, Amsterdam, Ch. 6, pp. 243–320.

Dershowitz, N., Plaisted, D. A., 1988. Equational programming. In: Hayes, J. E., Mitchie, D., Richards, J. (Eds.), Machine Intelligence 11. Claredon Press, Oxford, Ch. 2, pp. 21–56.

Echahed, R., 2008. Inductively sequential term-graph rewrite systems. In: Graph Transformations, 4th International Conference (ICGT 2008). Springer, LNCS 5214, Leicester, UK, pp. 84–98.

Echahed, R., Janodet, J. C., 1997. On constructor-based graph rewriting systems. Tech. Rep. 985-I, IMAG.

Fay, M. J., 1979. First-order unification in an equational theory. In: Proc. 4th Workshop on Automated Deduction. Academic Press, Austin (Texas), pp. 161–167.

Fischer, S., Kuchen, H., 2007. Systematic generation of glass-box test cases for functional logic programs. In: PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming. ACM, New York, NY, USA, pp. 63–74.

Fischer, S., Kuchen, H., 2008. Data-flow testing of declarative programs. In: ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. ACM Press, New York, NY, USA.

Giovannetti, E., Levi, G., Moiso, C., Palamidessi, C., 1991. Kernel LEAF: a logic plus functional language. The Journal of Computer and System Sciences 42, 139–185.

González Moreno, J. C., López Fraguas, F. J., Hortalá González, M. T., Rodríguez Artalejo, M., 1999. An approach to declarative programming based on a rewriting logic. The Journal of Logic Programming 40, 47–87.

Hanus, M., 1994. The integration of functions into logic programming: From theory to practice. Journal of Logic Programming 19&20, 583–628.

Hanus, M., 1997. A unified computation model for functional and logic programming. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97). pp. 80–93.

Hanus, M. (Ed.), 2006. Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`.

Hanus, M., 2007. Multi-paradigm declarative languages. In: Proceedings of the International Conference on Logic Programming (ICLP 2007). Springer LNCS 4670, pp. 45–75.

Hanus, M. (Ed.), 2008. PAKCS 1.9.1: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs`.

Hanus, M., Kuchen, H., Moreno-Navarro, J. J., 1995. Curry: A truly functional logic language. In: Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming. Portland, Oregon, pp. 95–107.
URL `citeseer.ist.psu.edu/hanus95curry.html`

Hullot, J. M., 1980. Canonical forms and unification. In: Proc. 5th Conference on Automated Deduction. Springer LNCS 87, pp. 318–334.

Hussmann, H., 1992. Nondeterministic algebraic specifications and nonconfluent rewriting. Journal of Logic Programming 12, 237–255.

ISO, 1995. Information technology - Programming languages - Prolog - Part 1. General Core. ISO/IEC 13211-1, 1995.

Klop, J. W., 1992. Term Rewriting Systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (Eds.), Handbook of Logic in Computer Science, Vol. II. Oxford University Press, pp. 1–112.

Lloyd, J. W., 1999. Programming in an integrated functional and logic language. Journal of Functional and Logic Programming 1999, 1–49.

López-Fraguas, F. J., Sánchez-Hernández, J., 1999. TOY: A multiparadigm declarative system. In: Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99). Springer LNCS 1631, pp. 244–247.

Lux, W., 1999. Implementing encapsulated search for a lazy functional logic language. In: Proc. 4th Fuji Intl. Symposium on Functional and Logic Programming (FLOPS '99). Springer LNCS 1722, pp. 100–113.

Meseguer, J., Thati, P., 2007. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher Order Symbol. Comput. 20 (1-2), 123–160.

Milner, R., Tofte, M., Harper, R., MacQueen, D., 1997. The Definition of Standard ML - Revised. MIT Press.

Moreno-Navarro, J. J., Rodríguez-Artalejo, M., 1992. Logic programming with functions and predicates: The language BABEL. Journal of Logic Programming 12, 191–223.

O'Donnell, M. J., 1977. Computing in systems described by equations. Springer LNCS 58.

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

Peyton Jones, S., Hughes, J. (Eds.), 1999. Haskell 98: A Non-strict, Purely Functional Language. Available at `http://www.haskell.org/onlinereport/`.

Plump, D., 1999. Term graph rewriting. In: H. Ehrig, G. Engels, H.-J. K., Rozenberg, G. (Eds.), Handbook of Graph Grammars. Vol. 2. World Scientific, pp. 3–61.

Reddy, U. S., 1985. Narrowing as the operational semantics of functional languages. In: International Symposium on Logic Programming. IEEE Computer Soc. Press, pp. 138–151.

Robinson, J. A., 1965. A machine-oriented logic based on the resolution principle. J. ACM 12 (1), 23–41.

Sheard, T., 2007. Type-level computation using narrowing in Ωmega. Electron. Notes Theor. Comput. Sci. 174 (7), 105–128.

Sleep, M. R., Plasmeijer, M. J., van Eekelen, M. C. J. D. (Eds.), 1993. Term Graph Rewriting Theory and Practice. J. Wiley & Sons, Chichester, UK.

Tolmach, A., Antoy, S., Nita, M., Sep. 2004. Implementing functional logic languages using multiple threads and stores. In: Proc. of the 2004 International Conference on Functional Programming (ICFP). ACM, Snowbird, Utah, USA, pp. 90–102.

Wirth, N., Apr. 1971. Program development by stepwise refinement. Communications of the ACM 14 (4), 221–227.