Typed Higher-order Narrowing without Higher-order Strategies

Sergio Antoy and Andrew Tolmach

Portland State University

Abstract. We describe a new approach to higher-order narrowing computations in a class of systems suitable for functional logic programming. Our approach is based on a translation of these systems into ordinary (first-order) rewrite systems and the subsequent application of conventional narrowing strategies. Our translation is an adaptation to narrowing of Warren's translation, but unlike similar previous work, we preserve static type information, which has a dramatic effect on the size of the narrowing space. Our approach supports sound, complete, and efficient higher-order narrowing computations in classes of systems larger than those previously proposed.

1 Introduction

Functional logic languages generalize functional languages by supporting the evaluation of expressions containing (possibly uninstantiated) logic variables. Narrowing is an essential component of the underlying computational mechanism of an internationally supported unified functional logic language [4]. In recent years a considerable effort has been invested in the development of narrowing strategies. The results of this effort are satisfactory for first-order computations. Higher-order narrowing, the situation in which a computation may instantiate a variable of functional type, is still evolving.

The most fundamental question is what universe of functions should be considered as possible instantiations for variables of functional type. One attractive answer is to consider just those functions (perhaps partially applied) that are explicitly defined in the program already, or a user-specified subset of these functions. For example, consider the program

```
data nat = z | s nat
data list = [] | nat:list
compose (F,G) X = F (G X)
map F [] = []
map F (H:T) = (F H):(map F T)
```

and abbreviate `s z` with `1`, `s 1` with `2`, etc. The goal `map G [1,2] == [2,3]` would have the solution $\{$`G` $\mapsto$ `s`$\}$, the goal `map G [1,2] == [3,4]` would have the solution $\{$`G` $\mapsto$ `compose (s,s)`$\}$, but the goal `map G [1,2] == [2,4]` would have no solutions.

Authors' address: Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, U.S.A., {antoy,apt}@cs.pdx.edu.

This work has been supported in part by the National Science Foundation under grant CCR-9503383.

This approach is quite expressive, and also relatively easy for the programmer to understand. Moreover, there is a very reasonable approach to solving higher-order problems of this kind by translating them into equivalent first-order problems, in which unapplied and partially applied functions are represented by constructors and implicit higher-order applications are made into explicit first-order applications. This idea dates back to Warren [13] for logic programs and to Reynolds [11] for functional programs. Hanus [7, 8] shows how this idea works for dynamically typed languages. González-Moreno [5] adapts the same idea to untyped narrowing.

Our contribution is to define this transformation for *statically typed* source and target languages. We give a rigorous presentation for monomorphic programs and sketch an extension to polymorphic programs. The benefits of a typed source language are well known. The benefits of maintaining types during program transformation and compilation are becoming increasingly recognized in the functional programming community (e.g., [14, 15]); these include the ability to use efficient, type-specific data representations, the ability to perform optimizations based on type information, and enhanced confidence in compiler correctness obtained by type-checking compiler output. In functional logic programming, typing the target language has additional dramatic, immediate effects on the narrowing space; in particular, typing allows possibly infinite, extraneous branches of the narrowing space to be avoided. This obviously improves the efficiency of the language, and avoids run-time behaviors, especially for sequential implementations, that would be unintelligible to the programmer.

2 Background

The most recent and comprehensive proposals of higher-order narrowing strategies differ in both the domain and the computation of the strategy. González-Moreno [5] considers *SFL-programs*. Rules in these systems are constructor-based, left-linear, non-overlapping, conditional, and allow extra variables in the conditional part. A translation inspired by Warren [13] removes higher-order constructs from these systems and allows the use of (first-order) narrowing strategies for higher-order narrowing computations

Nakahara et al. [10] consider first-order *applicative* constructor-based orthogonal rewrite systems. A rule's left-hand side in these systems allows variables of functional type, but prohibits both the application of a variable to subterms and the presence of a function symbol in a pattern. The strategy proposed for these systems is computed by a handful of small inference steps, where "small" characterizes the fact that several inferences may be necessary to compute an ordinary narrowing step.

Hanus and Prehofer [6] consider *higher-order* inductively sequential systems. An operation in these systems has a definitional tree in which the patterns may contain higher-order patterns [9] as subterms and the application of variables to subterms. The strategy proposed for these systems is again computed by a handful of small inference rules that, when concatenated together, for the most part simulate a needed narrowing computation.

The domains of these strategies have a large intersection, but none is contained in any other. For example, applicative orthogonal systems include Berry's system, which is excluded from inductively sequential systems; higher-order inductively sequential systems allow higher-order patterns in left-hand sides, which are banned in SFL programs; and SFL programs use conditional rules with extra variables, which are excluded from applicative systems. All the above strategies are sound and complete for the classes of systems to which they are applied.

While first-order narrowing is becoming commonplace in functional logic programming, the benefits and costs of higher-order narrowing are still being debated. Factors limiting the acceptance of higher-order narrowing are the potential inefficiency of computations and the difficulty of implementations. In this paper we describe a new approach to higher-order narrowing that addresses both problems. Our approach is based on a program translation, similar to [3, 5, 11–13], that replaces higher-order narrowing computations in a source program with first-order narrowing computations in the corresponding target program. Our approach expands previous work in three directions.

(1) We use a translation [12] that preserves type information of the source program. Type information dramatically affects the size of the narrowing space of a computation.
(2) We present our technical results for the same class of systems discussed in [10]. We will argue later that our approach extends the systems considered in [5] and with minor syntactic changes extends the systems considered in [6], too.
(3) For a large class of source programs of practical interest [4], our approach supports optimal, possibly non-deterministic, higher-order functional logic computations [1] without the need for a higher-order narrowing strategy.

3 Language

3.1 Basics

We describe our approach with reference to a monomorphically typed functional logic language L, whose abstract syntax is specified in Figure 1. For ease of presentation, we explicitly type all functions, constructors, and variables, but types could be inferred (and we therefore omit some typing information in the concrete examples in this paper). The abstract syntax is something of a compromise among the conventional notations for functional programs, logic programs, and rewrite systems. The concrete syntax of functional logic languages that could benefit from our approach could be much richer, e.g., it could allow infix operators, ad-hoc notation for numeric types and lists, nested blocks with locally scoped identifiers, anonymous functions, etc. Programs in languages with these features are easily mapped to programs in our language during the early phases of compilation. Thus, our approach is perfectly suited for contemporary functional logic languages, too.

A *program* is a collection of *constructor* declarations and *function* definitions. Constructors and functions are collectively called *symbols*; we use identifiers not beginning with upper-case letters for symbol names. Each symbol s has a unique associated non-negative arity, given by $ar(s)$. A function definition consists of one or more *rules* (not necessarily contiguously presented); each rule has a left-hand side which is a sequence of *patterns* and a right-hand side which is an *expression*. All the rules for a function of arity n must have n patterns. The patterns control which right-hand side(s) should be invoked when the function is applied; they serve both to match against actual arguments and to bind local variables mentioned in the right-hand side. Patterns are *applicative terms*, i.e., they are built from *variables*, fully-applied constructors, partially-applied functions, and *tuples*. This definition of "*applicative*" generalizes the one in [10], in that it permits partially-applied functions. In the logic programming tradition, we reserve identifiers beginning

(algebraic types)	$d := \textit{identifier}$	
(types)	$t := d$	(algebraic types)
	$\mid\ (t_1, \ldots, t_n)$	(tuples; $n \geq 0$)
	$\mid\ (t \to t)$	(functions)
(variables)	$v := \textit{identifier beginning with upper-case letter}$	
(constructors)	$c := \textit{identifier}$	
(functions)	$f := \textit{identifier}$	
(symbols)	$s := c \mid f$	
(problems)	$\textit{problem} := (\textit{program}, \textit{goal})$	
(programs)	$\textit{program} := \textit{dec}_1 \ldots \textit{dec}_m\ \textit{rule}_1 \ldots \textit{rule}_n$	$(m, n \geq 0)$
(goals)	$\textit{goal} := (v_1 : t_1, \ldots, v_n : t_n)\ e_1 == e_2$	(v_i disjoint)
(constr. decl.)	$\textit{dec} := c : t_1 \to \ldots \to t_n \to d$	$(n = ar(c))$
(function rules)	$rule := f\ p_1 \ldots p_n = e$	$(n = ar(f))$
(patterns)	$p := (v : t)$	(variable)
	$\mid\ (c\ p_1 \ldots p_n)$	(constructor; $n \leq ar(c)$)
	$\mid\ (f\ p_1 \ldots p_n)$	(function; $n < ar(f)$)
	$\mid\ (p_1, \ldots, p_n)$	$(n \geq 0)$(tuple)
(expressions)	$e := v$	(variable)
	$\mid\ s$	(constructor or function)
	$\mid\ (e_1, \ldots, e_n)$	(tuple)
	$\mid\ (e_1\ e_2)$	(application)

Fig. 1. Abstract syntax of functional logic language L. By convention, type arrows associate to the right and expression applications associate to the left. Each symbol s has a fixed associated arity, given by $ar(s)$. Only partially-applied function symbols are allowed in a pattern.

with upper-case letters for variables. Expressions are built from symbols, variables, tuples and binary application denoted by juxtaposition. Intuitively, function applications evaluate to the right-hand side of a matching rule, and constructor applications evaluate to themselves. Parentheses in expressions are avoided under the convention that application is left associative.

Functions (and constructors) are *curried*; that is, a function f of arity $ar(f) > 1$ is applied to only one argument at a time. This permits the manipulation of partially-applied functions, which is fundamental to expressing higher-order algorithms involving non-local variables. Unlike conventional functional and functional logic languages, which treat functions as black boxes, we permit matching against unapplied or partially-applied functions (or constructors). We delay the discussion of this feature until Section 6 when we compare our approach with [6]. It is often useful (particularly in presenting the result of our translation system) to describe uncurried functions that take their arguments "all at once;" this is done by specifying an arity-1 function taking a tuple argument.

A *problem* (p, g) consists of a program p and a goal g. A *goal* consists of a sequence of variable declarations followed by an *equation*, an expression of the form $e_1 == e_2$. Any variables appearing in e_1 or e_2 must appear in a declaration. There is no loss of generality in restricting goals to equations. A problem *solution* is a substitution θ (see Sect. 3.4) from the variables declared in the goal to applicative terms, such that $\theta(e_1)$ and $\theta(e_2)$ reduce to the same applicative term.

3.2 Typing

L is an explicitly typed, monomorphic language. Types include algebraic types, whose values are generated by constructors; tuple types; and function types. The typing rules are specified in Figure 2 as a collection of judgments for the various syntactic classes. The judgment $E \vdash_{class}$ *phrase* : $t \Rightarrow E'$ asserts that *phrase* of syntactic category *class* is well-typed with type t in environment E, and generates an environment E'; judgments for specific classes may omit one or more of these components. Environments map symbols and variables to types. Note that all functions are potentially mutually recursive.

Each variable declaration (in rules or goals) has an explicit type annotation; together with the typing rules these allow us to assume the existence of a well-defined function *typeof*(), mapping each (well-typed) expression or pattern to its type. A solution substitution $\{v_i \mapsto e_i\}$ is well-typed if and only if $\forall i, \mathit{typeof}(v_i) = \mathit{typeof}(e_i)$. In practice, we could generate the type annotations for L automatically by inference from a source program with missing or incomplete type annotations.

3.3 Relationship to Term Rewriting

A program p may be viewed as a rewrite system $(\Sigma, \mathcal{R})$ where

- Σ is a *signature*, i.e., a set of symbols (partitioned into functions and constructors), with associated types, consisting of those symbols that appear in p;
- $\mathcal{R}$ is the set of *rewrite rules* consisting of the function rules in p.

In the terminology of [10], our system is an *applicative term rewriting system* ($\mathcal{A}$TRS) using our slightly generalized notion of *applicative term*. We do not specify other fundamental properties, such as left-linearity, or non-overlapping of rules, though each of these may be useful in practice. For simplicity of presentation, we prohibit extra variables on the right-hand sides of rules, but these could be added similarly to goal variables without fundamental difficulty, as could conditions to the rules.

3.4 Evaluation

This view of programs as rewrite systems defines the notion of evaluation for our system. We define the evaluation of variable-free expressions as ordinary rewriting. An expression is in normal form if it cannot be further rewritten. Orthogonal (left-linear and non-overlapping) programs will be confluent, but not necessarily terminating. Well-typed expressions enjoy the *subject reduction* property, which says that their type is invariant under reduction.

As noted above, a problem solution is a substitution from the variables declared in the goal to applicative terms. This definition doesn't indicate how a solution might be found.

$$\frac{\vdash_{\mathit{program}}\ \mathit{program} \Rightarrow E \quad E \vdash_{\mathit{goal}}\ \mathit{goal}}{\vdash_{\mathit{problem}}\ (\mathit{program}, \mathit{goal})}$$

$$\frac{\begin{array}{c}\vdash_{\mathit{dec}}\ \mathit{dec}_1 \Rightarrow E_1 \ \ \ldots \ \ \vdash_{\mathit{dec}}\ \mathit{dec}_m \Rightarrow E_m \\ E \vdash_{\mathit{rule}}\ \mathit{rule}_1 \Rightarrow E_{m+1} \ \ \ldots \ \ E \vdash_{\mathit{rule}}\ \mathit{rule}_n \Rightarrow E_{m+n} \\ E = E_1 + \ldots + E_{m+n}\end{array}}{\vdash_{\mathit{program}}\ \mathit{dec}_1 \ldots \mathit{dec}_m\ \mathit{rule}_1 \ldots \mathit{rule}_n \Rightarrow E}$$

$$\frac{E + \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\} \vdash_e e_i : t (i = 1, 2)}{E \vdash_{\mathit{goal}}\ (v_1 : t_1, \ldots, v_n : t_n)\ e_1 == e_2}$$

$$\frac{}{\vdash_{\mathit{dec}}\ c : t_1 \to \ldots \to t_n \to d \Rightarrow \{c \mapsto t_1 \to \ldots \to t_n \to d\}}$$

$$\frac{E(f) = t_1 \to \ldots \to t_n \to t \quad E \vdash_p p_i : t_i \Rightarrow E_i (1 \leq i \leq n) \quad E + E_1 + \ldots + E_n \vdash_e e : t}{E \vdash_{\mathit{rule}}\ f\ p_1 \ldots p_n = e \Rightarrow \{f \mapsto t_1 \to \ldots \to t_n \to t\}}$$

$$\frac{}{E \vdash_p (v : t) : t \Rightarrow \{v \mapsto t\}}$$

$$\frac{E(c) = t_1 \to \ldots \to t_n \to d \quad E \vdash_p p_i : t_i \Rightarrow E_i (1 \leq i \leq n)}{E \vdash_p (c\, p_1 \ldots p_n) : d \Rightarrow E_1 + \ldots + E_n}$$

$$\frac{E(f) = t_1 \to \ldots \to t_n \to t \quad E \vdash_p p_i : t_i \Rightarrow E_i (1 \leq i \leq m)}{E \vdash_p (f\, p_1 \ldots p_m) : t_{m+1} \to \ldots \to t_n \to t \Rightarrow E_1 + \ldots + E_m}$$

$$\frac{E \vdash_p p_i : t_i \Rightarrow E_i (1 \leq i \leq n)}{E \vdash_p (p_1, \ldots, p_n) : (t_1, \ldots, t_n) \Rightarrow E_1 + \ldots + E_n}$$

$$\frac{E(v) = t}{E \vdash_e v : t} \qquad \frac{E(s) = t}{E \vdash_e s : t}$$

$$\frac{E \vdash_e e_i : t_i (1 \leq i \leq n)}{E \vdash_e (e_1, \ldots, e_n) : (t_1, \ldots, t_n)} \qquad \frac{E \vdash_e e_1 : t_2 \to t \quad E \vdash_e e_2 : t_2}{E \vdash_e (e_1 e_2) : t}$$

Fig. 2. Typing rules for language L. The environment union operator $E_1 + E_2$ is defined only when E_1 and E_2 agree on any elements in the intersection of their domains.

To be more concrete, we define the evaluation of a problem to mean the computation of a solution substitution by a sequence of *narrowing* steps. Formally, a *substitution* is a mapping from variables to terms which is the identity almost everywhere. Substitutions are extended to homomorphisms from terms to terms. The narrowing relation on $\mathcal{R}$, denoted $\underset{\mathcal{R}}{\leadsto}$, is defined as follows: $e \underset{\mathcal{R}}{\leadsto}_{\theta,p,l=r} e'$, iff θ unifies $e_{|p}$ (the subterm of e at position p)

with the left-hand side l of some rule $l{=}r$ (with fresh variables), and $e' = \theta(e[r]_p)$ (where $e[r]_p$ is the result of replacing the subterm at position p of e by r). We will drop θ, p, or $l{=}r$ from this notation when they are irrelevant. When presented, the representation of θ will be restricted to the variables of a goal.

As habitual in functional logic languages with a lazy operational semantics, we define the validity of an equation as a strict equality on terms denoted by the infix operator ==. Because of the applicative nature of our systems, strict equality is defined by the families of rules

$$
\begin{array}{ll}
c == c = \mathit{true}, & \forall c \\
c\, X_1 \ldots X_n == c\, Y_1 \ldots Y_n = X_1 == Y_1 \wedge \ldots \wedge X_n == Y_n, & \forall c, ar(c) \geqslant n > 0 \\
(X_1, \ldots, X_n) == (Y_1, \ldots, Y_n) = X_1 == Y_1 \wedge \ldots \wedge X_n == Y_n, & \forall n > 0 \\
f == f = \mathit{true}, & \forall f, ar(f) > 0 \\
f\, X_1 \ldots X_n == f\, Y_1 \ldots Y_n = X_1 == Y_1 \wedge \ldots \wedge X_n == Y_n, & \forall f, ar(f) > n \geqslant 0 \\
\mathit{true} \wedge X = X &
\end{array}
$$

where c is a constructor and f is a function. (Recall that our language allows only partially-applied function symbols in a pattern.)

A sequence of narrowing steps $g \overset{*}{\underset{\mathcal{R}}{\rightsquigarrow}}_{\theta}\ \mathit{true}$, where $\mathcal{R}$ is the set of rules of p, is called an *evaluation* of (p, g) producing *solution* θ.

As a very simple example, consider the following problem, taken from [5, Ex. 1] (originally from [13]).

```
z : nat
s : nat -> nat
twice (F:nat->nat) (X:nat) = F (F X)
(G:(nat->nat)->(nat->nat)) G s z == s (s z)
```

A solution of this problem is the substitution $\{\texttt{G} \mapsto \texttt{twice}\}$. It is computed by the following evaluation.

$$\texttt{G s z == s (s z)} \rightsquigarrow_{\{\texttt{G}\mapsto\texttt{twice}\}} \texttt{s (s z) == s (s z)} \overset{+}{\rightsquigarrow}_{\{\}} \texttt{true}$$

Note that we still haven't suggested a strategy for choosing the appropriate sequence of narrowing steps. In fact, while a great deal is known about efficient strategies for first-order programs, we understand much less about higher-order ones. We will show shortly, however, that typing information can be a valuable guide to computing an efficient strategy. The main idea of this paper is to reduce the higher-order case to the first-order one while maintaining typability, as described in the next section.

4 Translation

The idea behind the translation is to encode all unapplied or partially-applied symbols[1] as constructors (called *closure constructors*), grouped into new algebraic data types (*closure types*), and replace all applications in the original program by applications of special new *dispatch functions*. As its name suggests, a dispatch function takes a closure constructor as argument, and, based on the value of that argument, dispatches control to (a translation

[1] Because constructors in the source program can be treated just like first-class functions (e.g., as arguments to higher-order functions), they are encoded just like functions.

of) the appropriate original function. The resulting program is well-typed in the strict first-order subset of the original language, so ordinary first-order narrowing strategies can be used to find solutions.

The main novelty introduced by the presence of types is that the translation constructs type-specific dispatch functions and associated closure-constructor types, one for each different function type encountered in the source program. (The obvious alternative—using a single dispatch function that operates over all closure constructors—is unattractive, because such a function could not be given a conventional static polymorphic type; some form of dependent typing would be required, and this would in general require dynamic type checking.) The translation essentially consists of two parts:

- generation of a set of new nullary or unary *closure constructors*, corresponding to partially applied functions and constructors in the source program, and a set of new unary *dispatch functions*;
- a syntax-directed transformation of the original program and goal, which replaces original constructor and function names by closure constructor names, original applications by dispatch function applications and original function definitions by equivalent uncurried function definitions.

The closure constructors and dispatch functions are then integrated with the translated program to produce the complete translated problem. Details of the translation are specified in Figures 3 and 4. Borrowing from [5], we denote closure constructors with identifiers prefixed by "#," closure types with the symbol "$" indexed by an (arrow) type, and dispatch functions with the symbol "@" indexed by a type.

In a translated program, all functions appearing in expressions are fully applied, so no applicative expression ever mentions a function. Thus, solutions of a goal under the translated program never mention the new *dispatch* functions, though they may mention the new $\#s$ constructors. Solutions may be translated back into a higher-order form that doesn't contain these constructors by translating each construction $\#s_0$ into the symbol name s, and each construction $\#s_k(e_1, \ldots, e_k)$ into the application $(s\ e_1 \ldots e_k)$. A complete inverse translation for expressions is given in Figure 5.

4.1 Example

Consider again the small example problem of the previous section. Our translation gives the following target problem

```
z : nat
s : nat -> nat
#twice0 : d1
#twice1 : d2 -> d2
#s0 : d2
twice (F:d2, X:nat) = @d2(F,@d2(F,X))
@d1(#twice0:d1, F:d2) = #twice1(F)
@d2(#twice1(F:d2), X:nat) = twice(F,X)
@d2(#s0:d2, X:nat) = s(X)
(G:d1) @d2(@d1(G,#s0),z) == @d2(#s0,@d2(#s0,z))
```

where

$$\mathtt{d1} = \$_{[\![\mathtt{nat}\to\mathtt{nat}]\!]\to[\![\mathtt{nat}\to\mathtt{nat}]\!]} = [\![(\mathtt{nat} \to \mathtt{nat}) \to (\mathtt{nat} \to \mathtt{nat})]\!]$$
$$\mathtt{d2} = \$_{[\![\mathtt{nat}]\!]\to[\![\mathtt{nat}]\!]} = [\![\mathtt{nat} \to \mathtt{nat}]\!]$$

$$
\begin{aligned}
[\![d]\!] &= d \\
[\![(t_1, \ldots, t_n)]\!] &= ([\![t_1]\!], \ldots, [\![t_n]\!]) \\
[\![t_1 \rightarrow t_2]\!] &= \$_{[\![t_1]\!] \rightarrow [\![t_2]\!]}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathit{dec}_1 \ldots \mathit{dec}_m\ \mathit{rule}_1 \ldots \mathit{rule}_n]\!] &= [\![\mathit{dec}_1]\!] \ldots [\![\mathit{dec}_m]\!]\ \mathit{newsigs}\ [\![\mathit{rule}_1]\!] \ldots [\![\mathit{rule}_n]\!]\ \mathit{newrules} \\
&\quad \text{where } \mathit{newsigs} = \mathit{nsigs}(\mathit{dec}_1) \ldots \mathit{nsigs}(\mathit{dec}_m) \\
&\qquad\qquad\qquad \mathit{nsigs}'(\mathit{rule}_1) \ldots \mathit{nsigs}'(\mathit{rule}_n) \\
&\quad \text{and } \mathit{newrules} = \mathit{nrules}(\mathit{dec}_1) \ldots \mathit{nrules}(\mathit{dec}_m) \\
&\qquad\qquad\qquad \mathit{nrules}'(\mathit{rule}_1) \ldots \mathit{nrules}'(\mathit{rule}_n) \\
[\![(v_1 : t_1, \ldots, v_n : t_n)\ e_1\ \ \mathtt{==}\ \ e_2]\!] &= (v_1 : [\![t_1]\!], \ldots, v_n : [\![t_n]\!])\ [\![e_1]\!]\ \ \mathtt{==}\ \ [\![e_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![c : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow d]\!] &= c : ([\![t_1]\!], \ldots, [\![t_n]\!]) \rightarrow d && \text{(if } ar(c) = n > 0\text{)} \\
&= c : d && \text{(if } ar(c) = n = 0\text{)}
\end{aligned}
$$

$$
[\![f\ p_1 \ldots p_n\ \ =\ \ e]\!] = f([\![p_1]\!], \ldots, [\![p_n]\!]) = [\![e]\!]
$$

$$
\begin{aligned}
\text{(patterns)} \qquad [\![(v : t)]\!] &= (v : [\![t]\!]) \\
[\![(c\ p_1 \ldots p_n)]\!] &= (c([\![p_1]\!], \ldots, [\![p_n]\!])) \\
[\![f]\!] &= \#f_0 \\
[\![(f\ p_1 \ldots p_n)]\!] &= (\#f_n([\![p_1]\!], \ldots, [\![p_n]\!])) \\
[\![(p_1, \ldots, p_n)]\!] &= ([\![p_1]\!], \ldots, [\![p_n]\!])
\end{aligned}
$$

$$
\begin{aligned}
\text{(expressions)} \qquad [\![v]\!] &= v \\
[\![s]\!] &= \#s_0 && \text{(if } ar(s) > 0\text{)} \\
[\![s]\!] &= s && \text{(if } ar(s) = 0\text{)} \\
[\![(e_1, \ldots, e_n)]\!] &= ([\![e_1]\!], \ldots, [\![e_n]\!]) \\
[\![(e_1\ e_2)]\!] &= @_{[\![\mathit{typeof}(e_1)]\!]}([\![e_1]\!], [\![e_2]\!])
\end{aligned}
$$

Fig. 3. Translation Rules. The notation for translation is overloaded to work on each syntactic class. The definitions of *nsigs*(),*nsigs*′(),*nrules*(), and *nrules*′() are given in Figure 4.

An evaluation of the translated problem produces the following (incomplete) narrowing sequence:

$$
\begin{aligned}
&\mathtt{@_{d2}(@_{d1}(G,\#s_0),z)\ \ ==\ \ @_{d2}(\#s_0,@_{d2}(\#s_0,z))} \\
&\quad \rightsquigarrow_{\{\mathtt{G} \mapsto \mathtt{\#twice_0}\}}\ \mathtt{@_{d2}(\#twice_1(\#s_0),z)\ \ ==\ \ @_{d2}(\#s_0,@_{d2}(\#s_0,z))} \\
&\quad \rightsquigarrow_{\{\}}\ \mathtt{twice(\#s_0,z)\ \ ==\ \ @_{d2}(\#s_0,@_{d2}(\#s_0,z))} \\
&\quad \rightsquigarrow_{\{\}}\ \mathtt{@_{d2}(\#s_0,@_{d2}(\#s_0,z))\ \ ==\ \ @_{d2}(\#s_0,@_{d2}(\#s_0,z))}
\end{aligned}
$$

The solution substitution $\{\mathtt{G} \mapsto \mathtt{\#twice_0}\}$ to the translated problem is mapped by the inverse translation back to the substitution $\{\mathtt{G} \mapsto \mathtt{twice}\}$, which is a solution of the original problem.

4.2 Discussion

We have tried to present the translation in the simplest and most uniform way possible. As a result, the translated code is substantially more inefficient than necessary; this can be corrected by using a more sophisticated translation or by post-translation optimization,

$$
\begin{aligned}
nsigs(s : d) &= \{\} \\
nsigs(s : t_1 \to \ldots \to t_n \to d) &= \left\{ \begin{array}{l} \quad \#s_0 : [\![t_1 \to \ldots \to t_n \to t]\!] , \\ \quad \#s_1 : [\![t_1]\!] \to [\![t_2 \to \ldots t_n]\!] , \\ \quad \ldots , \\ \#s_{n-1} : ([\![t_1]\!] , \ldots , [\![t_{n-1}]\!]) \to [\![t_n \to t]\!] \end{array} \right\} \\
nsigs'(f\ p_1\ \ldots\ p_n\ =\ e) &= nsigs(f : typeof(f))
\end{aligned}
$$

$$
\begin{aligned}
nrules(s : d) &= \{\} \\
nrules(s : t_1 \to \ldots \to t_n \to d) &= \left\{ \begin{array}{l} @_{[\![t_1 \to \ldots \to t_n \to t]\!]}(\#s_0, v_1 : [\![t_1]\!]) \ = \ \#s_1(v_1), \\ @_{[\![t_2 \to \ldots \to t_n \to t]\!]}(\#s_1(v_1 : [\![t_1]\!]), v_2 : [\![t_2]\!]) \ = \ \#s_2(v_1, v_2), \\ \ldots , \\ @_{[\![t_n \to t]\!]}(\#s_{n-1}(v_1 : [\![t_1]\!] , \ldots , v_{n-1} : [\![t_{n-1}]\!]), v_n : [\![t_n]\!]) \ = \\ \hfill s(v_1, \ldots , v_n) \end{array} \right\} \\
nrules'(f\ p_1\ \ldots\ p_n\ =\ e) &= nrules(f : typeof(f))
\end{aligned}
$$

Fig. 4. Definition of closure constructors and dispatch functions.

$$
\begin{aligned}
[\![v]\!]^{-1} &= v \\
[\![\#s_0]\!]^{-1} &= s \\
[\![\#s_k(e_1, \ldots , e_k)]\!]^{-1} &= (s\ e_1 \ldots e_k) \\
[\![c]\!]^{-1} &= c \qquad\qquad \text{(for other constructors } c\text{)} \\
[\![(e_1, \ldots , e_n)]\!]^{-1} &= ([\![e_1]\!] , \ldots , [\![e_n]\!])
\end{aligned}
$$

Fig. 5. Inverse Translation Rules.

using standard simplification techniques. For example, we replace *all* source applications by dispatch function applications, whereas in fact *only* applications whose operators are variables need to be dispatched; when the operator is a known symbol, the dispatch function call could be inlined. As another example, translated functions are called in only one place, from within the (type-appropriate) dispatch function, so they could be inlined there without increasing code size. Note, however, that one cannot in general perform *both* these transformations simultaneously without risking a blow-up in the size of the translated program.

Our translation relies fundamentally on having a monomorphic source program, so that each source function can be assigned unambiguously to a closure-constructor type and associated dispatch function in the target. It is obviously desirable to extend the translation to polymorphic problems. One straightforward approach is to generate (automatically) multiple *specialized* versions of the program's polymorphic functions, one for each type at which the function is used, before applying the translation [12]. This method works for problems that combine polymorphic programs with monomorphic goals. For example, given the program

```
u : foo
v : bar
id (Z:α) = Z
```

where α is a type variable, we can solve the goals

```
(X:foo, Y:bar) (id X, id v) == (u,Y)
(F:foo->foo,G:bar->bar) (F u, G v) == (u,v)
```

by first specializing `id` into two functions

```
id_foo (Z:foo) = Z
id_bar (Z:bar) = Z
```

and then translating to first-order and solving in the normal way. For the latter goal, we obtain the solution $\{\texttt{F} \mapsto \texttt{id\_foo}, \texttt{G} \mapsto \texttt{id\_bar}\}$; we can transform both the monomorphic function names appearing in the solution back into `id` for presentation to the user. Note that the type annotations on variables `F` and `G` are needed to guide the specialization process.

This approach fails to handle goals with polymorphic function-typed variables. For example, given the above program, consider the goal

```
(F u, F v) == (u,v)
```

Here $\{\texttt{F} \mapsto \texttt{id}\}$ is the (sole) solution to the polymorphic goal; no single monomorphic instance can possibly be a solution. Solving this problem appears to require dynamic (runtime) type manipulation. For example, Hanus [8] has shown that a system using explicit type annotations on all terms together with runtime type unification can express the Warren transformation using a single polymorphic dispatch function; there are static analyses that permit the runtime types to be omitted *provided* there are no function-valued goal variables [7]. If there *are* function-valued variables, some runtime type operations will need to remain, and it is not clear what level of static type checking can be guaranteed to the programmer; we are currently investigating this question.

Our translation is presented as operating on an entire program at one time, but this is not a fundamental requirement. If the source program is given incrementally, the translation can be performed incrementally too, provided that it is possible to add new constructors to an existing algebraic datatype and new rules for an existing dispatch function on an incremental basis.

5 Correctness

In this section we formulate the correctness of our translation and sketch its proof. Intuitively, a translation of a source problem into a target problem is correct if solutions of the source problem can be computed by means of the target problem, i.e., if the following diagram commutes:

$$\begin{tikzcd}
\text{source problem} \arrow[r, "\text{translate}"] \arrow[d, "\text{solve}"] & \text{target problem} \arrow[d, "\text{solve}"] \\
\text{solution} & \text{solution} \arrow[l, "\text{inverse translate}"']
\end{tikzcd}$$

More specifically, it should make no difference whether we compute the solutions of a source program directly or through its translation. To formalize this idea, we need the following notation and definitions. If θ is a substitution, we define $[\![\theta]\!]$ as the substitution that, for any variable v, $[\![\theta]\!](v) = [\![\theta(v)]\!]$. We say that a substitution θ is *applicative* if, for any variable v, $\theta(v)$ is an applicative term. Two substitutions are *equivalent* if each one can be obtained by the other through a renaming of variables. If θ is a substitution and g is a goal, then $\theta_{|\mathcal{V}ar(g)}$ denotes the *restriction* of θ to the variables of g. If P is a problem, then $solve(P)$ is the set of its solutions. If S is a set, then $[\![S]\!] = \{[\![x]\!] \,|\, x \in S\}$. The comparison of two sets of substitutions is always implicitly intended modulo equivalence of substitutions.

Let P be a source problem. We say that a translation $[\![\,]\!]$ is *complete* iff, for every source problem P, $solve(P) \subseteq [\![solve([\![P]\!])]\!]^{-1}$. We say that a translation $[\![\,]\!]$ is *sound* iff, for every source problem P, $solve(P) \supseteq [\![solve([\![P]\!])]\!]^{-1}$.

Note that the soundness and completeness of a narrowing strategy used to compute a problem solution are not an issue for the soundness and completeness of a translation, though we envision that a sound and complete strategy will be used for the target program.

We express the completeness of our translation as:

Let θ be a solution (applicative substitution) of a problem (p, g), where p is an $\mathcal{A}$TRS. There exists a substitution θ' such that $[\![g]\!] \overset{*}{\underset{[\![p]\!]}{\rightsquigarrow}}_{\theta'}$ true and $[\![\theta']\!]^{-1}_{|\mathcal{V}ar(g)}$ is equivalent to θ.

The proof of the completeness of our translation is in two parts. First, one proves the completeness claim when all the steps of a computation are rewriting steps by induction on the length of the computation. Then, one proves the completeness claim of a narrowing computation by lifting the completeness result for the corresponding rewrite computation.

The intuition behind the completeness of our translation is that a computation in the source program is "simulated" by a computation in the target program. For example, the evaluation presented in Section 4.1 simulates the corresponding evaluation presented in Section 3.4.

Each application in the source program is replaced by at most two applications in the target program (before optimization). Consequently, the derivation in the target program takes more steps than in the source program, but only by a constant factor. Moreover, all the target program steps are first-order. In addition, the optimized translation of a first-order source application is just itself. Thus, there is no loss of efficiency in the target program for purely first-order narrowing computations in the source program.

We express the soundness of our translation as:

Let (p, g) be a problem, where p is an $\mathcal{A}$TRS. If $[\![g]\!] \overset{*}{\underset{[\![p]\!]}{\rightsquigarrow}}_{\theta}$ true, then $[\![\theta]\!]^{-1}_{|\mathcal{V}ar(g)}$ is a solution of (p, g).

The proof of the soundness of our translation stems directly from González-Moreno's work [5]. His translation of a problem can be obtained from our translation of the same problem by collapsing all our dispatch functions into a single untyped dispatch function. Some consequences of this difference are discussed in Section 6. For example, his translation of Example 4.1 yields the following first-order program (where we have added type annotations):

```
twice (F:nat->nat, X:nat) = @(F,@(F,X))
@(#twice₀:(nat->nat)->(nat->nat), F:nat->nat) = #twice₁(F)
```

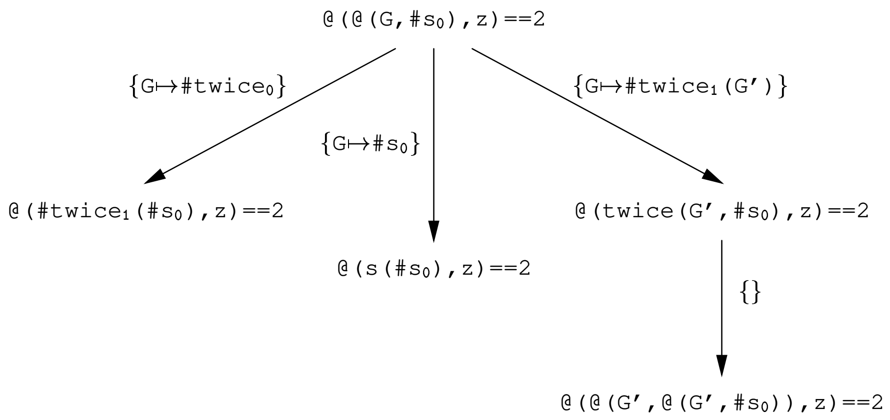

Fig. 6. Portion of the search space of `@(@(G,#s0),z)==2`

```
@(#twice1(F:nat->nat), X:nat) = twice(F,X)
@(#s0:nat->nat, X:nat) = s(X)
(G:(nat->nat)->(nat->nat)) @(@(G,#s0),z) == @(#s0,@(#s0,z))
```

Therefore, every solution (substitution) of a problem computed via our translation is also computed by González-Moreno's translation of the same problem. Thus, [5, Th. 1] directly implies that our translation is sound.

6 Related Work

The benefits of our approach to higher-order narrowing computations can be better understood by comparing it with related work.

6.1 Smaller Narrowing Space

The approach to higher-order narrowing closest to ours is [5]. The major difference between these approaches is that our target programs are well-typed whereas the target programs in [5] are not. We show the effects of this difference on the narrowing space of the example discussed in Section 5. Figure 6 shows a portion of the goal's narrowing space computed with the translation proposed in [5], where 2 is an abbreviation of $s\ (s\ z)$. The same portion of narrowing space generated by our target program contains only the left branch. Both middle and right branches of Figure 6 contain ill-typed terms. The right branch is infinite due to

$$\texttt{@(G,\#s}_0\texttt{)} \overset{+}{\rightsquigarrow} \texttt{@(G',@(G',\#s}_0\texttt{))} \overset{+}{\rightsquigarrow} \texttt{@(G''@(G'',@(G'',\#s}_0\texttt{)))} \overset{+}{\rightsquigarrow} \ldots$$

These conditions, neither of which arises with our translation, have far reaching consequences. An increase in the branching factor of the nodes of a narrowing space implies an exponential growth of the number of nodes that may be generated during a computation. The presence of infinite branches in the search space implies that sequential implementations of complete narrowing strategies may become operationally incomplete. Even if these extreme consequences can be avoided in many cases, we expect in most cases a substantial slowdown of narrowing computations that discard type information. This observation is also made in [10].

Of course, even branches containing only type-correct terms may be infinite, making the use of sequential implementations problematic. We believe, however, that the behavior of such implementations will be much easier for the programmer to understand if they are guaranteed to proceed in a well-typed manner. Gratuitous alteration of the size or finiteness of the narrowing space which cannot aid in finding solutions is surely unacceptable.

6.2 Expanded programs

Our overall approach has a significant difference with respect to both [6, 10]. We reduce higher-order narrowing computations to first-order ones by means of a program translation that is largely independent of both the class of source programs and the narrowing strategy applied to these programs. This decoupling opens new possibilities. For example, there is no published work on higher-order narrowing in both constructor based, weakly orthogonal rewrite systems and inductively sequential, overlapping rewrite systems. It is easy to verify that our translation preserves many common properties of rewrite systems, including weak orthogonality and inductive sequentiality. Since sound, complete, and efficient first-order narrowing strategies are known for both weakly orthogonal rewrite systems [2] and inductively sequential, overlapping rewrite systems [1], our approach immediately provides a means for higher-order narrowing computations in these classes.

6.3 Efficiency

Another difference of our approach with respect to both [6, 10] is the granularity of the steps. Both [6, 10] perform narrowing computations by means of inferences whereas our approach promotes the use of strategies that perform true narrowing steps. Generally, we expect an overhead when a narrowing step is broken into many inferences.

A noteworthy feature of our approach is that when no variable of function type is instantiated, higher-order narrowing computations do not cost significantly more than first-order narrowing computations. This feature is essential for powerful functional logic languages. For example, the current proposal for Curry defines the dispatch function as *rigid*, thus excluding higher-order narrowing computations from the language, "since higher-order unification is an expensive operation" [4, Sect 2.6]. Our translation is a step toward lifting exclusions of this kind. Indeed, the above feature extends (at least for monomorphically typed programs) the behavior of modern narrowing strategies in that when no variable is instantiated, narrowing computations should not cost significantly more than functional computations.

6.4 Partial Applications in Patterns

There is a significant difference between [6] and the other higher-order narrowing approaches referenced in this paper. In [6], the patterns of a rule may consist of or contain higher-order patterns [9]. For example, [6, Ex. 1.2] defines a higher-order function diff, where $\mathit{diff}(F, X)$ computes the differential of F at X in the form of a higher-order pattern. The rules of diff include higher-order patterns for symbols such as sin, cos, and

ln. Although intuitively these symbols stand for functions sine, cosine, and logarithm, the program defines neither the symbols nor the functions by means of rewrite rules. Our approach supports and extends this feature. The following definitions, where for readability we omit type declarations, allow us to express polynomial functions.

```
const N _ = N
x X = X
plus F G X = F X + G X
times F G X = F X * G X
```

For example, the function $x^2 + 1$ would be expressed using the above rules as

```
plus (times x x) (const 1)
```

The following program defines our *diff* function. Similar to [6], narrowing allows us to use *diff* to compute symbolic integration, although *diff* alone would be inadequate in most cases. By contrast to [6], we use *diff* to compute both the symbolic derivative with respect to x of a polynomial and the differential of a polynomial at $x = X$ — in our framework the former evaluates to a function and the latter evaluates to a number.

```
diff (const _) = const 0
diff x = const 1
diff (plus F G) = plus (diff F) (diff G)
diff (times F G) = plus (times (diff F) G) (times F (diff G))
```

For example

```
plus (times x x) (const 1) 2 ~*~> 5
diff (plus (times x x) (const 1)) 2 ~*~> 4
diff (plus (times x x) (const 1)) ~*~>
   plus (plus (times (const 1) x) (times x (const 1)))
        (const 0)
```

A “simplification” function with rules such as the following ones would be useful to improve the latter and necessary to compute non-trivial symbolic integration.

```
simpl (plus F (const 0)) = F
simpl (times F (const 1)) = F
...
```

Our approach eases understanding the appropriateness of this unusual programming style. Intuitively, both functions and constructors of the source program become (closure) constructors of the target program, hence function symbols in patterns of the source program need not be harmful. Indeed, higher-order programming is, loosely speaking, allowing a function f to be the argument of a function g. A rule of g has generally a variable, of functional type, to match or unify with f, but there is no reason preventing the use of the symbol f itself in the rule of g. Furthermore, since we do not allow fully applied functions in patterns, the use of functions in patterns does not compromise the confluence of the program, although the lack of confluence would not be an issue for our translation in any case.

7 Conclusion

We have presented a translation from source to target programs that allows us to perform higher-order narrowing computations with first-order narrowing strategies. This has several noteworthy advantages.

Our approach immediately provides sound, complete and efficient higher-order narrowing computations for large classes of systems for which no sound, complete and/or efficient higher-order narrowing strategies were known.

Our approach refines previous translation attempts by preserving in the target program type information of the source program. It is easy to verify that even in trivial examples this has a dramatic effect on the size of the narrowing space.

Our approach allows and justifies the presence of function symbols in the patterns of a rewrite rule. This feature extends the use of higher-order patterns, increases the expressive power of a language and simplifies metaprogramming tasks.

Acknowledgments

The authors are grateful to J. C. González Moreno and M. Hanus for discussions and suggestions on the topics of this paper.

References

1. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the 6th Int. Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, Sept. 1997. Springer LNCS 1298.
2. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In *Proc. of the 14th Int. Conference on Logic Programming (ICLP'97)*, pages 138–152, Leuven, Belgium, July 1997. MIT Press.
3. J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proc. 2nd International Conference on Functional Programming*, pages 25–37, June 1997.
4. Curry: An integrated functional logic language. M. Hanus (ed.), Draft Jan. 13, 1999.
5. J.C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. GULP' 93*, pages 569–585, Gizzeria Lido, IT, Oct. 1993.
6. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. 7th International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.
7. Michael Hanus. Polymorphic higher-order programming in prolog. In *Proc. 6th International Conference on Logic Programming*, pages 382–397. MIT Press, June 1989.
8. Michael Hanus. A functional and logic language with polymorphic types. In *Proc. International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 215–224. Springer LNCS 429, 1990.
9. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
10. K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher-order functional logic programming. In *Proc. 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pages 97–114. Springer LNCS 982, 1995.
11. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.
12. A. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.
13. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.
14. *Proc. Workshop on Types in Compilation (TIC97)*, June 1997. Boston College Computer Science Technical Report BCCS-97-03.
15. *Proc. Second Workshop on Types in Compilation (TIC98)*. Springer LNCS 1473, March 1998.