Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

Spring 1-1-2012

The Basic Scheme for the Evaluation of Functional Logic Programs

Arthur Peters
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Programming Languages and Compilers Commons, and the Systems Architecture Commons

Recommended Citation

Peters, Arthur, "The Basic Scheme for the Evaluation of Functional Logic Programs" (2012). *Dissertations and Theses.* Paper 914.

10.15760/etd.914

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

The Basic Scheme for the Evaluation of Functional Logic Programs

by

Arthur Peters

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Sergio Antoy, Chair
Mark P. Jones
Bart Massey

Portland State University
© 2012

ABSTRACT

Functional logic languages provide a powerful programming paradigm combining the features of functional languages and logic languages. However, current implementations of functional logic languages are complex, slow, or both. This thesis presents a scheme, called the *Basic Scheme*, for compiling and executing functional logic languages based on non-deterministic graph rewriting. This thesis also describes the implementation and optimization of a prototype of the Basic Scheme. The prototype is simple and performs well compared to other current implementations.

ACKNOWLEDGMENTS

This thesis would never have been completed with out the help of my committee: Sergio Antoy, Mark Jones, and Bart Massey. I would like to specifically thank my adviser Sergio Antoy for his continuous stream of guidance and assistance. I would also like to thank Micheal Hanus for various correspondence and especially for helping me understand KiCS2. Finally, I would like to thank the various people, especially my mother Barbara Michener, who helped me edit this thesis into a form that can pass for good English.

CONTENTS

Abstract . . . . **i**

Acknowledgments . . . . **ii**

List of Tables . . . . **v**

List of Figures . . . . **vi**

1 Introduction **1**
- 1.1 Notational Conventions and Syntax 2

2 Declarative Programming **4**
- 2.1 Functional Programming 4
  - 2.1.1 Evaluation Order 5
- 2.2 Logic Programming 6
  - 2.2.1 Non-determinism 6
- 2.3 Functional Logic Programming 8
  - 2.3.1 Non-determinism 8
  - 2.3.2 Evaluation 8

3 Graph Rewriting for Functional Logic Evaluation **10**
- 3.1 Definitional Trees 13
- 3.2 Non-deterministic Functional Graph Rewriting 15
  - 3.2.1 Limited Overlapping Inductively Sequential Rewrite Systems 16
  - 3.2.2 Implementing non-determinism 18
- 3.3 Evaluation Strategies 24
- 3.4 Graph Rewriting Procedures 25

4 Implementing Functional Logic Languages **29**
- 4.1 Core Languages 29
- 4.2 Compiling to LOIS-IR 30
- 4.3 The Basic Scheme 32

4.3.1 Correctness of the Basic Scheme 37
4.3.2 Extensions To the Basic Scheme 40
4.4 ViaLOIS: Implementing the Basic Scheme 41
4.4.1 Complications of In-place Rewriting 42
4.4.2 Translation and Runtime System 42
4.4.3 Optimizations 44
4.5 Benchmark Results 45

5 Related Work 50

6 Conclusion 52
6.1 Future Work 52
6.2 Conclusion 54

References 56

Appendix A Benchmark Source Code 59
A.1 ChoiceIDs 59
A.2 PermSort 59
A.3 Tree 60
A.4 Sharing 61
A.5 Half 62

Appendix B ViaLOIS Source Code 64

للاستشارات المنارة

LIST OF TABLES

4.1 Lines of code (in thousands) of several Curry systems. Line counts exclude comments, blank lines, and the standard library. Built-in functions are included as part of the runtime. The parser and first compilation stage (conversion to FlatCurry) is omitted from the line counts as this part of the code is shared between all 4 implementations. [figure and caption from Antoy and Peters, 2012] 48

LIST OF FIGURES

1.1 The linear representation (a) represents the graph (b). 3
1.2 The notation I use for programs and expressions. 3

2.1 (a) shows a function to select an arbitrary element from a list. (b) shows a function to select an arbitrary element that is greater than 10 from a list. `membergt10` applies constraints to the non-determinism using the `if` statement and the explicit failure when the condition is not met. Both functions are non-deterministic in that it is not specified which element of the list will be returned just that it will fit the criteria. 7

3.1 (a) shows a constructor-based GRS that implements Peano addition. The symbols *suc* (successor) and *zero* are constructors. The symbol *add* is an operation symbol. (b) shows a series of rewrite steps (written as $\rightarrow$), rewriting an operation rooted expression (1) to head normal form (2) and finally to full normal form (3). 11
3.2 The GRS (a) is converted into the definitional tree (b). The boxed variable is the inductive node that must be evaluated to head normal form to allow a rule to be selected. The rule nodes each have a different pattern in place of w so it is easy to choose between them once w is in head normal form. 14
3.3 The rules defining the choice operator as used in limited overlapping GRSs. In such system this is the only operation with overlapping patterns in its rules. 16
3.4 Examples of overlapping operations converted into limited-overlapping form. 17
3.5 The bubbling transformation on an expression. The system must find a dominator to perform the transformation. $\Longrightarrow$ represents the bubbling step. 21

3.6 A simple example of pull-tab transformation (written Ξ) to show the “unzipping” process that gives it its name. The Just node is duplicated (“unzipped”) to allow the choice to be moved toward the root of the expression. 22

3.7 The pull-tab transformation (written Ξ) applied repeatedly to an expression. Each step moves a choice nearer the root and may duplicate a choice. The final expression is in non-determinism normal form. The notation $\mathbf{?}_2$ represents a choice with ID 2. 23

3.8 The procedure B produces a computation $\omega(B(e))$ that, if $e = d(g)$, performs pull-tab steps on $d(g)$ to bring any choices near the root of g above the application of d and then rewrites the applications of d with applications of v. 25

3.9 The resulting computation (a) and simulated computation (b) from the call $B(e)$ where $e = d(\mathbf{?}_2\,(3, 1))$. In the computation, procedure calls are shown as nodes with the actions resulting from them as children. The replacements and pull-tab steps are represented using the syntax introduced for describing the procedures. The procedure calls are also annotated (as a superscript) with the rule applied. The simulated computation is shown as a derivation annotated, in the right column, with the actions that perform each rewrite. All expressions are written in linear notation. 27

4.1 (a) shows how a case expression on an argument is moved into the pattern matching and hence encoded in the definitional tree. (b) shows how a case expression over a complex expression is translated into LOIS-IR by lifting the complex expression into a new operation f', so that the case expression can be encoded as the definitional tree of f'. (c) shows how a complex expression (a case in this example) that is an argument to an operation is lifted into a new operation. The symbol $\Rightarrow$ represents transformations performed by ViaLOIS to convert FlatCurry into LOIS. 31

4.2 Compilation of a *source* program into a *target* program consisting of 3 procedures: **N**, **H** and **A**. The syntax and semantics of the procedures is described in Section 3.4. The rules of **N** and **A** depend only on the set of operations and constructors. The rules of **H** are obtained from the definitional tree of each operation by the algorithm compile. The symbols c and f stand for a generic constructor and operation of the *source* program and i is a choice identifier. The call to a *target* procedure with some argument g systematically operates on the *trace* of g. [figure and caption based on Antoy and Peters, 2012] 34
4.3 An example of conversion of a LOIS system into a strict deterministic program by the Basic Scheme. The symbol member is an operation symbol and :: and [] are constructor symbols. 35
4.4 The evaluation of the expression Just(member(1 :: 2 :: [])), where Just is a constructor and member is the operation defined in Figure 4.3a, by the target program shown in Figure 4.3c combined with Figure 4.2. In each state of the computation, the left column is the call stack of the target program and the expression to the right is the state of the evaluation. The arrows from the procedures on the stack to the expression show which node is the argument of that procedure call; The transitions between the states are annotated with the procedure rules used to perform the step. The symbol $\overset{*}{\Longrightarrow}$ represents one or more computational steps (pull-tabs, replacements, and procedure calls). 36
4.5 A block diagram of the ViaLOIS compiler. The components FlatCurry2LOIS and LOIS2OCaml are the core of ViaLOIS and are custom. The components parsecurry (part of PAKCS [Hanus, 2008a]) and the OCaml Compiler [INRIA, 2011] are existing tools. 41
4.6 Because of the collapsing rule defining id, f (0 ?1) must be rewritten as shown in (c) instead of as shown in (b) because sharing is lost in (c). The symbols $\xrightarrow{*}$ and $\not\xrightarrow{*}$ represent correct and incorrect derivations respectively. 43
4.7 Benchmark Results. 🟩 *ViaLOIS* (based on the Basic Scheme), 🟥 KiCS2, ⬜ PAKCS, 🟦 MCC. 46

Chapter 1

INTRODUCTION

Lazy functional logic languages, such as Curry [Hanus, 2006] and $\mathcal{TOY}$ [Caballero and Sánchez, 2007], combine the features of lazy functional languages and logic languages — laziness from functional languages and free variables and non-deterministic execution from logic languages. However, functional logic languages are difficult to implement correctly, efficiently, and simply. Most current implementations rely on complex compilation schemes and runtime systems, but still produce slow programs. However, functional logic programming provides a concise and effective way to encode many algorithms. So, efficient and easy to understand implementations would be very useful for both real-world use and for research into FL programming and compilers.

My contribution is the development, with Sergio Antoy, of an evaluation scheme for functional logic programs called the *Basic Scheme* [Antoy and Peters, 2012] (Section 4.3). In addition, I contribute a prototype Curry system based on the Basic Scheme (Section 4.4) which shows the Basic Scheme is practical and easy to implement. As part of this prototype I describe a new intermediate language for representing functional logic programs. Sergio Antoy's contribution was primarily at the level of the formal model of the Basic Scheme. The work on the implementation is entirely my own.

The Basic Scheme is a simple technique that implements lazy, non-deterministic computation in a strict, deterministic language and performs well in real implementations. This provides a platform for both efficient practical implementations and research. The Basic Scheme requires only a few simple features in the target language: simple pattern matching on one symbol at a time, mutable records, and first-class function pointers.

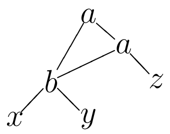
I provide an overview of functional and functional logic programming with a focus on implementation of such languages (Chapter 2). I provide an overview of graph rewriting as applied to formalizing and implementing functional logic languages (Chapter 3). I provide an overview of how functional logic languages are compiled and implemented including specific challenges of implementing functional logic languages and how they have been addressed in the past and how this issues are addressed by the use of a subclass of graph rewrite systems, called Limited Overlapping Inductively Sequential (LOIS) systems, in the Basic Scheme (Chapter 4). I also provide details of the implementation and performance of a prototype Curry system based on the Basic Scheme, called ViaLOIS because of its use of LOIS graph rewrite systems as an intermediate representation. This includes various small changes that where made to the formal Basic Scheme to allow efficient implementation (Section 4.4). I outline some interesting possibilities that could be the basis for future work (Chapter 6.1), including some discussion of the parallel evaluation of LOIS systems. Finally, I provide the conclusions that can be drawn from this work (Chapter 6.2). I also provide the source code of the benchmarks used and a link to complete source code of ViaLOIS in Appendix A and Appendix B, respectively.

1.1 NOTATIONAL CONVENTIONS AND SYNTAX

Graphs are written using a linear notation [Echahed and Janodet, 1997, Definition 4]. Informally, a graph is written $g : s(x_1, \ldots, x_k)$ where g is the root node of the graph, s is the symbol labeling g, and $x_1, \ldots, x_k$ are the successors of g

$a(g : b(x, y), a(g, z))$ means

(a) The linear representation of a graph



(b) The same graph in a graphical representation

Figure 1.1: The linear representation (a) represents the graph (b).

$\mathsf{increment}(x) \rightarrow x + 1$ (a) A rule

$\textbf{if } x == 2 \textbf{ then } 0 \textbf{ else } f(x)$ (b) An expression

$\textbf{let } x = f(y) \textbf{ in } (x, x)$ (c) **let** representing sharing

Figure 1.2: The notation I use for programs and expressions.

which can be either previously defined nodes or new nodes. For nodes that do not need to be referenced, the identifier may be omitted; for example $c(1)$ represents the same graph as $g : c(1)$. An example is shown in Figure 1.1. The notation $n : \_$ represents a node n that may be labeled with any symbol and may have any successors. Because colon is used in the linear representations of graph the cons operator for lists is written $x :: xs$.

I present programs using a combination of graph rewriting notation and functional programming notation. Operation rules are written using graph rewriting notation as shown in Figure 1.2a. Expressions are written using a functional notation, except that function parameters are in enclosed parentheses and comma delimited as shown in Figure 1.2b. **let** is used to represent sharing of a subexpression as shown in Figure 1.2c and does not imply any evaluation order. $\textbf{let } x = f(y) \textbf{ in } (x, x)$ is equivalent to the linear graph notation $(x : f(y), x)$.

Chapter 2

DECLARATIVE PROGRAMMING

Declarative programming encourages the programmer to focus on the problem they are trying to solve instead of the specifics of how to solve it. Functional programming allows the programmer to focus on the specific functional relationships between values without having to specify how the functions will be evaluated. Logic programming allows the programmer to describe the solution to the problem and leave finding a solution that matches that description to the language implementation. Functional logic programming merges these ideas providing an environment where functional relationships can be expressed as a description of a solution instead of a traditional equation.

A common and very powerful feature of modern declarative languages is pattern matching. Pattern matching allows the programmer to concisely enumerate a set of cases that should be handled differently by the program and to extract information from data structures.

2.1 FUNCTIONAL PROGRAMMING

Functional programming provides many useful abstractions, including higher-order functions. These allow algorithms to be implemented as functions and then composed to build more complex algorithms.

Functions are first-class values in functional programming, which allows for very elegant abstraction of algorithms. For instance, a `map` function can be defined that implements iteration over a list, but instead of doing a specific operation on each element it takes another function as a parameter and calls that function on each element.

2.1.1 Evaluation Order

Functional programs can be evaluated either by *eagerly* (also called strictly) evaluating expressions as soon as possible or by *lazily* evaluating expressions only when their value is actually needed (such as for output).

Eager Evaluation

An eager language evaluates expressions as soon as it has enough information to do so. This is generally easier to implement as there is no need to store thunks and it is faster for cases where everything needs to be evaluated eventually. However, in cases where some value is never used, a eager strategy will evaluate that value whereas a lazy strategy will not. Because of this, `if-then-else` cannot be implemented as a function in a eager language: both the `then` and the `else` expression would be evaluated regardless of the value of the conditional. The unnecessary evaluation is both a performance problem and a semantic problem because even if the condition is `True`, non-termination or a fatal error while evaluating the `else` expression would prevent the program from completing.

Lazy Evaluation

Lazy evaluation refers to evaluation strategies in which evaluation in not performed until the value is actually needed. This type of strategy has a number of advantages including the ability to implement `if-then-else` as a function instead of as a primitive. Also in lazy languages, it is possible to define infinite data structures and, as long as only a finite number of elements are actually accessed, the program will complete in finite time because the rest of the data structure will be left unevaluated. However, implementing lazy evaluation has a cost. Unevaluated expressions, represented by thunks or continuations, must be stored until their value is needed.

2.2 LOGIC PROGRAMMING

Logic programming allows the programmer to describe the solution to a problem, but leave finding a solution that matches that description to the implementation. This makes it very easy to prototype an idea by simply writing a description of what you want the result to be and then running it. In many cases this will be slow compared to writing out a specific algorithm. However, the programmer time saved may outweigh the execution time increase.

In logic programming there is a concept of *failure*, which represents computations that do not produce values, for example `1/0`. Unlike in deterministic languages, failure in logic programs does not result in a crash, but instead is part of the normal execution of programs. Failure represents the lack of a result so I will write it as the bottom symbol $\perp$.

Instead of pattern matching, many logic programming languages implement unification; this works by taking two expressions and attempting to make them equal by instantiating free variables of both expressions to specific values. This is similar to matching a value against a pattern.

2.2.1 Non-determinism

In traditional logic programming languages, non-determinism is provided by free variables (also called logic variables). Free variables take on whatever value is needed by the computation. In cases where there are multiple ways for the computation to proceed the free variable will take on all values (either one at a time or all in parallel depending on how the system is viewed).

Another way to represent non-determinism is through explicit choices between

$$\begin{aligned} &\mathsf{member}(x :: xs) \rightarrow x \\ &\mathsf{member}(x :: xs) \rightarrow \mathsf{member}(xs) \\ &\mathsf{member}([\,]) \rightarrow \bot \end{aligned}$$

(a) member non-deterministically returns each element of its argument.

$$\begin{aligned} &\mathsf{membergt10}(l) \rightarrow \textbf{if } x > 10 \textbf{ then } x \textbf{ else } \bot \\ &\qquad \mathsf{where}\ x = \mathsf{member}(l) \end{aligned}$$

(b) membergt10 constrains the call to member so that only elements that are greater than 10 are allowed.

Figure 2.1: (a) shows a function to select an arbitrary element from a list. (b) shows a function to select an arbitrary element that is greater than 10 from a list. membergt10 applies constraints to the non-determinism using the if statement and the explicit failure when the condition is not met. Both functions are non-deterministic in that it is not specified which element of the list will be returned just that it will fit the criteria.

values. The choice operator, written **?**, is an explicit representation of non-deterministic choice; for example the expression $1 \,\mathbf{?}\, 2$ has exactly two possible values 1 and 2. The choice operator can be represented in as free variables and vice versa. However, I will use **?** in this thesis because it is used heavily in functional logic programming and the Basic Scheme in particular.

Non-determinism allows computation with incomplete information. The programmer provides a set of constraints on a value, but not a specific algorithm to find a value that fits these constraints. Conceptually, the system will non-deterministically provide some value that matches the constraints. For example, member in Figure 2.1a selects an arbitrary element of the list and membergt10 in Figure 2.1b selects an element of the list that is greater than 10. Equivalently non-deterministic operations can be viewed as returning a set of all possible values and the caller of the operation trying all the values to find the ones that result in successful evaluations.

In languages like Prolog that provide non-deterministic primitives as their only control structures, it is necessary to specify the strategy used to discover successful values. This is particularly important in languages that allow side-effects (such as Prolog), but it also a performance concern, even in side-effect-free code. In Prolog, and many other logic programming languages, non-determinism is handled by backtracking as discussed in Section 3.2.2.

2.3 FUNCTIONAL LOGIC PROGRAMMING

Functional logic programming combines functional programming with non-determinism as found in logic programming languages. Functional control structures like recursion and functional syntax are combined with logic variables and non-deterministic operations.

2.3.1 Non-determinism

In functional logic languages, non-determinism comes in two forms: free variables, and non-deterministic operations. Free variables in functional logic programs have the same semantics as they do in logic programs. Free variables and non-deterministic operations are equally expressive [Antoy and Hanus, 2006]. Any free variable can be encoded as a non-deterministic operation whose results are all the values of the type of the variable. These operations are called *generators* and are used to encode logic variables in the Basic Scheme.

Non-deterministic operations (also called non-deterministic functions) are functions except that they may have multiple possible return values of which one is chosen non-deterministically. Figure 2.1a shows a non-deterministic operation `member`, that will return an arbitrary value from the list given as its argument.

2.3.2 Evaluation

Functional nesting is when a function application is a parameter to another function, for example $f(g(x))$. For non-deterministic operations, there are two ways in which functional nesting could be interpreted. Either the non-deterministic choice is made when the call is made (called *call-time choice*), or it is made (possibly repeatedly and with different results) when the value is used in the function (called *use-time choice*). This distinction can be thought of as the difference between allowing non-deterministic values to be passed to function and restricting arguments to deterministic values. Arguments can be made for both call-time and use-time choice, but in general call-time choice is more intuitive, because the same non-deterministic value is not allowed to take on more than one value in a single computation. Call-time choice is the most common semantics and the one specified by the Curry language [Hanus, 2005, p 15], so call-time choice will be assumed for the rest of this thesis.

The function $\mathsf{eq}(x) \to (x == x)$ provides an obvious example of the different semantics. With call-time choice $\mathsf{eq}(x)$ will always return True, but with use-time choice True will always be among the results, but False will also be among the results if x has two or more possible values. For instance, $\mathsf{eq}(0 \mathbin{\textbf{?}} 1)$ will produce both True and False under use-time choice.

Call-time choice changes the semantics of the evaluation process so that sharing of a subexpression has a semantic meaning, and hence it is not valid to add or remove sharing. For instance, $\mathsf{eq}(0 \mathbin{\textbf{?}} 1)$ is not equivalent to $(0 \mathbin{\textbf{?}} 1) == (0 \mathbin{\textbf{?}} 1)$, but it is equivalent to $\textbf{let } x = 0 \mathbin{\textbf{?}} 1 \textbf{ in } x == x$. The semantic significance of sharing adds some noticeable complexity to the formalization of the Basic Scheme.

The same choice between lazy and eager evaluation that applies to functional programs also applies to functional logic programs. As long as sharing is properly maintained and choices are handled reasonably, either evaluation order can be used. However, the Curry language specifies lazy evaluation, and I will only address lazy functional logic languages in this thesis.

Chapter 3

GRAPH REWRITING FOR FUNCTIONAL LOGIC EVALUATION

Graph rewriting formalizes computation as a series of transformations on a graph. These transformations are called rewrites; some examples are shown in Figure 3.1. The graph represents the expression being evaluated by a rewrite system R. The functions defined in the program are represented as symbols in R. The semantics of the functions are represented as rewrite rules over these symbols. This set of rewrite rules and symbols is a *graph rewrite system* (GRS). In this chapter, I provide informal definitions sufficient for the development of the Basic Scheme. Echahed and Janodet [1997] provide formal definitions of these terms.

I will limit the discussion to *constructor-based GRSs* because they allow a simple definition of a value that results from a computation and the Basic Scheme is based on constructor-based GRSs. A constructor-based GRS distinguishes constructor symbols (used for data, for instance *suc* in Figure 3.1) from operation symbols (used for functions, for instance *add* in Figure 3.1) [Echahed and Janodet, 1997, Definition 22]. A GRS also contains a set of *variables* (used for binding operation arguments to results of a rewrite, for instance x and y in Figure 3.1).

An *expression* (also called a term graph) is a rooted graph, where each *node* is labeled with a constructor symbol or an operation symbol. Because expressions are graphs, standard graph notation is used with them and they may be represented in either linear or graphical form. A *constructor-rooted* expression is one whose root is labeled with a constructor symbol and similarly for *operation-rooted* and "x-rooted" in general. A *substitution* is a mapping $\sigma = \{x_1/g_1, \ldots, x_k/g_k\}$ meaning that, when the substitution is applied, each instance of the variable x_i will be

$$
\begin{aligned}
&add(suc(x), y) \rightarrow suc(add(x, y)) && \text{(suc-rule)}\\
&add(zero, y) \rightarrow y && \text{(zero-rule)}
\end{aligned}
$$

(a) A constructor-based GRS

$$
\begin{aligned}
&add(suc(zero), suc(zero)) && && (1)\\
&\quad \rightarrow suc(add(zero, suc(zero))) && \text{By suc-rule} && (2)\\
&\quad \rightarrow suc(suc(zero)) && \text{By zero-rule} && (3)
\end{aligned}
$$

(b) An evaluation in the GRS (a)

Figure 3.1: (a) shows a constructor-based GRS that implements Peano addition. The symbols *suc* (successor) and *zero* are constructors. The symbol *add* is an operation symbol. (b) shows a series of rewrite steps (written as →), rewriting an operation rooted expression (1) to head normal form (2) and finally to full normal form (3).

replaced by the expression g_i. I will write $\sigma(e)$ to refer to the substitution σ applied to the expression e. A *pattern* is an operation-rooted expression in which all nodes other than the root are variables or constructors (for example $add(zero, y)$ in Figure 3.1a). Variables are placeholders for unknown values; they are either bound by a pattern or free — we will only be concerned with bound variable as explained in Section 3.2.1. A pattern π *matches* an expression g if there is a substitution σ such that $\sigma(\pi) = g$.

I write $g[o \leftarrow p]$ to denote g with the node o *replaced* with the expression p. Formally, $g[o \leftarrow p]$ means changing every reference to o in the graph into a reference to p. In most cases this is equivalent to in-place updates, meaning updating the label and successors of o to be the same as the label and successors of p. However, because sharing is semantically significant functional logic languages, this is not always true (see Section 4.4.1)

An expression that contains only constructors is in *normal form* (also called, full normal form). However, there are normal forms that contain non-constructors, but in the context of constructor-based rewrite systems these normal forms are considered failures. I will implicitly simplify failures to the failure symbol $\perp$. An expression in normal form that is not a failure is called a *value*. A expression that is constructor-rooted, but may contain arbitrary subgraphs as its descendants is in *head normal form*. If an expression e can be rewritten to a value e', then e' is a *value* of e. For example, $suc(suc(zero))$ is a value of $add(suc(zero), suc(zero))$ by the derivation show in Figure 3.1b.

Rewrite steps are primitive operations in graph rewriting. To perform a rewrite, the system chooses a rule in the GRS, written $\pi \to r$ where π is the pattern of the rule and r is the right-hand side of the rule, and subexpression n in the expression g, such that π matches n. Section 3.3 discusses various ways to choose the rule and node; for the moment we will assume they are chosen arbitrarily. The replacement is $r' = \sigma(r)$ where σ is the substitution such that $\sigma(\pi) = n$. The result of the rewrite is a new expression $g' = g[n \leftarrow r']$. A rewrite step is written $g \rightarrow g'$. Figure 3.1b shows two rewrite steps.

A *derivation* of an expression e_0 is series of rewrite steps, $e_0 \rightarrow e_1 \rightarrow \cdots$. A derivation of an expression e can also be considered an evaluation of e.

3.1 DEFINITIONAL TREES

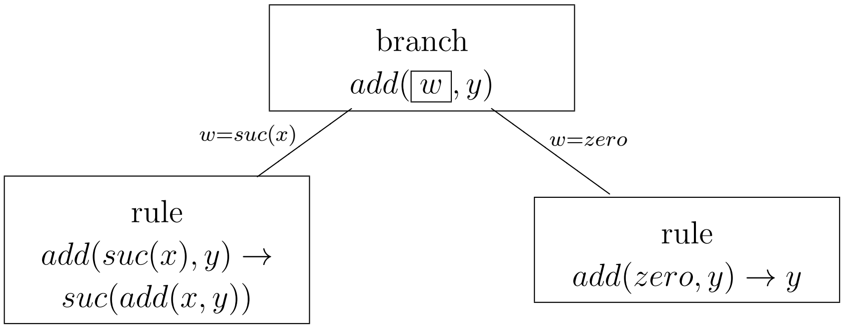
Definitional trees encode a set of rewrite rules defining an operation. An operation is *inductively sequential* if all of its rules can be encoded as a single definitional tree. The Basic Scheme uses definitional trees to deterministically and efficiently compute which subexpression of an operation application $f(n_1, \ldots, n_k)$ needs to be evaluated for $f(n_1, \ldots, n_k)$ to be rewritten and, if none needs to be evaluated, which rule can be applied to $f(n_1, \ldots, n_k)$. Figure 3.2 shows the rules of the operation *add* and the definitional tree that encodes them. The definitional tree tells us that to compute what rule to apply we must evaluate *add*'s first argument w to head normal form. The root constructor of w then tells us which rule to apply. Definitional trees can be viewed as a tree of nested *case* statements that match various parts of the pattern one at a time. I will not discuss how to compute definitional trees from an operation's rules, but Antoy [1992] develops an algorithm.

Definitional trees have 3 kinds of nodes: *branch*, *rule*, and *exempt*. Each contains a pattern. Branches represent choices between subtrees based on the runtime value at a specific *inductive node* in the pattern. Branch nodes have a set of subtrees such that there is exactly one subtree for every constructor that could appear at the inductive node. For example, the root node in Figure 3.2b represents the choice between two rules based on the value of w. Rule nodes represent rewrite rules for the operation. The pattern of the node is the pattern of the rewrite rule. Exempt nodes represent patterns that are not covered by any rule in the GRS. These expressions cannot be evaluated and are treated as failures in constructor-based GRSs.

$$add(suc(x), y) \to suc(add(x, y)) \qquad \text{(suc-rule)}$$

$$add(zero, y) \to y \qquad \text{(zero-rule)}$$

(a) a constructor-based GRS that implements Peano addition.



(b) the definitional tree encoding the rules for the *add* operation.

Figure 3.2: The GRS (a) is converted into the definitional tree (b). The boxed variable is the inductive node that must be evaluated to head normal form to allow a rule to be selected. The rule nodes each have a different pattern in place of w so it is easy to choose between them once w is in head normal form.

Given an expression $e = f(\ldots)$ and a definitional tree for the operation f, we can compute what rewrite should be performed by traversing the definitional tree from the root. The nodes are placed in the tree such that, if they are reached by following the branch nodes, then the pattern will match e. If a rule node is reached than that rule can be applied to e and if an except node is reached e can be rewritten to $\bot$. If a branch node is reached and the inductive node n is not in head normal form then n needs to be evaluated to head normal form.

Formally definitional trees are defined in terms of *partial definitional trees.*

Definition 3.1. *(Partial Definitional Tree)* A partial definitional tree, $\mathcal{T}$, is one of the following:

branch$(\pi, o, \bar{\mathcal{T}})$ where π is a pattern, o is a node in π called the inductive node.

$\bar{\mathcal{T}}$ is a set of partial definitional tree with exactly the patterns $\bar{\pi}$, where $\bar{\pi} = \{p \mid \text{for each } v \in s_o, p = \pi[o \leftarrow v_i]\}$ and s_o is the set of constructors in the kind of the variable at o in π.

rule$(\pi \rightarrow r)$ where π is a pattern, $\pi \rightarrow r$ is a rule, and r is an expression.

exempt(π) where π is a pattern.

Definition 3.2. *(Definitional Tree)* A partial definitional tree $\mathcal{T}_f$ is a definitional tree of an operation f if and only the root of $\mathcal{T}_f$ has a pattern $f(x_1, \ldots, x_k)$ where each x_i is a distinct variable.

3.2 NON-DETERMINISTIC FUNCTIONAL GRAPH REWRITING

Non-deterministic steps allow graph rewriting to be used to evaluate functional logic programs, but also increases the complexity of the model. In a non-deterministic GRS, a single expression may have more than one value. If the goal is to compute every possible value, then some method is needed to find and compute these values.

There are several differences from deterministic rewriting. Because failure is allowed in non-deterministic computations, we need a way to represent failure. This is done with a special constructor $\perp$ that propagates up through the expression whenever it labels a needed node. Also, sharing is semantically significant in non-deterministic GRSs that use call-time choice for the same reason discussed in Section 2.3.2.

In non-deterministic GRSs, rules are allowed to have overlapping patterns (for instance member in Figure 2.1a). When the patterns overlap, more than one rule may be applicable to the same expression and the system non-deterministically chooses which rule to apply. This produces a number of different derivations for the same expression. Some derivations may be *successful*, meaning that they result in a value.

$$x \mathbin{\textbf{?}} y \to x$$

$$x \mathbin{\textbf{?}} y \to y$$

Figure 3.3: The rules defining the choice operator as used in limited overlapping GRSs. In such system this is the only operation with overlapping patterns in its rules.

3.2.1 Limited Overlapping Inductively Sequential Rewrite Systems

This section defines a restricted class of non-deterministic GRSs called *limited overlapping inductively sequential GRSs* (*LOIS* systems). LOIS systems were described by Antoy [2011] and are used as the source program of the Basic Scheme because they allow non-determinism to be handled more simply than general non-deterministic GRSs.

A graph rewrite system is limited overlapping if the only rewrite rules with overlapping patterns are the rules for the choice operator **?**, shown in Figure 3.3. This is an implementation of the choice operator discussed in Section 2.2.1. Because **?** is the only operation with overlapping patterns it is the only operation whose rules need non-deterministic handling. Any GRS can be converted to a limited-overlapping system using the choice operator [Antoy, 2001]; Figure 3.4 provides examples. In cases where the patterns overlap, but are not identical, an auxiliary operation can be introduced that matches against the more specific pattern and fails on any other pattern.

In limited-overlapping systems **?** is often treated as a representation of non-determinism instead of as an operation whose rules can be applied. The choices are moved toward the root of the expression to allow the evaluation of parts of the expression without applying the rules of **?**.

It is useful to define two special normal forms for limited overlapping systems

$$\begin{array}{l} \mathsf{member}(x :: xs) \to x \\ \mathsf{member}(x :: xs) \to \mathsf{member}(xs) \end{array} \implies \mathsf{member}(x :: xs) \to x \mathbin{\textbf{?}} \mathsf{member}(xs)$$

(a) A simple operation with overlapping rules converted in limited-overlapping form.

$$\begin{array}{lcl} f(x :: xs) \to x & \implies & f(xs) \to hd(xs) \mathbin{\textbf{?}} g(xs) \\ f(xs) \to g(xs) & & hd(x :: xs) \to x \end{array}$$

(b) An operation with overlapping, but distinct, patterns in its rules converted in limited-overlapping form.

Figure 3.4: Examples of overlapping operations converted into limited-overlapping form.

to describe where the choices are in the expression. *Non-deterministic normal form* is an expression in which all the choices are near the root and underneath the choices are normal form expressions, so any path from the root there will contain zero or more choices at the beginning of the path and none there after. An expression in non-deterministic normal form (even if it contains failures) is called a *non-deterministic value*. A non-deterministic value is a set of zero or more deterministic values. *Non-deterministic head normal form* is an expression with choices near the root with head normal form expressions under them. These head normal form expressions may contain other choices. Examples of these normal forms are shown in Figure 3.7; every expression is in non-deterministic head normal form and the last is in non-deterministic normal form.

A GRS as a whole is inductively sequential if all its operations are inductively sequential (that is, they have definitional trees). A system that is both limited overlapping and inductively sequential is a LOIS system.

LOIS systems allow for efficient evaluation strategies (see Section 3.3) even though they are non-deterministic [Antoy, 2005]. As described LOIS systems do not allow for free variables. However, any LOIS system with free variables is equivalent to another LOIS system without free variables [Antoy and Hanus, 2006].

3.2.2 Implementing non-determinism

There are two basic axes that can be used to categorize implementations of non-determinism in deterministic systems (such as real computers): search strategy and binding strategy. A search strategy provides a way to try different non-deterministic choices in an expression to find a value. A binding strategy provides a way to find the possible bindings for any given free variable.

The most common search strategy is backtracking, which works by trying all possible values of a variable or non-deterministic operation, one at a time, until one is successful. If more than one value is needed, the search continues. Other search strategies include cloning, bubbling [Antoy et al., 2006], and pull-tabbing [Antoy, 2011]. These are discussed in more detail in the following sections.

In some cases, the binding of a free variable is easily derived because it is directly or indirectly stated to be equal to some other value. In this case, the free variable can simply be bound to that value. However, there are many cases where this is not possible, for instance $y = x + 1$ where x is free. In these cases a value (or set of possible values must) be found to bound the variable (x in the example above). There are two common methods for handling this: Residuation and narrowing.

Residuation [Hanus, 1992] works by suspending evaluation when a free variable cannot be bound and attempting to evaluate another expression in the program. This allows multiple expressions to bind variables for each other. When a variable that was residuated on is bound the suspended computation is resumed. However, as shown by Hanus [1992], there are programs for which residuation cannot bind all variables. Some systems, such as CLP(R) [Jaffar et al., 1992] and CLP(FD) [Codognet and Diaz, 1996], extend residuation with a constraint solver over a domain (real numbers and finite domains, respectively).

Narrowing [Antoy et al., 2000] allows variables to be non-deterministically bound without suspending the computation. If a free variable's binding is needed then the system will simply bind it non-deterministically to every possible value for that variable, or equivalently the free variable will be replaced with a non-deterministic choice between all possible values for that variable. Unlike residuation, narrowing will always find a successful value for a variable, if there is one and it is finite, simply because it will try every possible value.

Although residuation and narrowing theoretically produce the same results (with the exception of cases where residuation fails to find a value), in practice they perform quite differently. Residuation requires the suspension and resuming of computation whereas narrowing does not. Narrowing requires that all possible values of the variable be known so in practice it requires that the type be known, whereas residuation does not.

Backtracking

Backtracking works by evaluating the program deterministically, but when the system encounters a choice, then all possible values of the choice are tried in order. If a value causes the computation to fail, the system tries the next: essentially this performs a depth first search on the tree of choices to find successful values. This can become a problem for a number of reasons:

- If one non-deterministic choice prevents the program from terminating, then no more choices will ever be tried. This makes backtracking *incomplete* in the sense that, even if there is a terminating evaluation for the problem backtracking may never find it.
- It is difficult to share computations between non-deterministic branches, even if the computation is not effected by the choice.

However, backtracking is also one of the most efficient methods of handling non-determinism in situations where completeness is not needed and there is no computation shared between non-deterministic branches.

Cloning

Cloning implements non-determinism by copying the expression being evaluated when a choice is encountered and then evaluating each possible value for the choice in its own copy. Like backtracking, evaluation proceeds deterministically until a choice is encountered. Cloning has many of the same problems as backtracking and also tends to be slow because a copy of the state of the computation must be made or a persistent data structure needs to be used to allow multiple changed copies of the expression to exist. However, cloning allows parallel execution of non-deterministic branches and parallel cloning strategies are complete.

Bubbling

Bubbling [Antoy et al., 2006] only works on limited overlapping GRSs. In limited overlapping GRSs, we can keep all possible states of the expression in one larger expression. This is done by moving the choices toward the root of the expression and then evaluating the expressions underneath these choices. This is similar to cloning in that, if an operation is applied to a choice, then the application will be duplicated and evaluated for both sides of the choice. However, it has a number of advantages. For example, bubbling can dramatically reduce the amount of copying needed compared to cloning. Also, unlike backtracking, bubbling can be implemented in a way that allows concurrent evaluation of different sides of a choice.

Bubbling moves choices up to one of their dominators closer to the root of the expression. A dominator of a node is another node that is on every path from the node to the root. A bubbling system clones the paths from the dominator to the

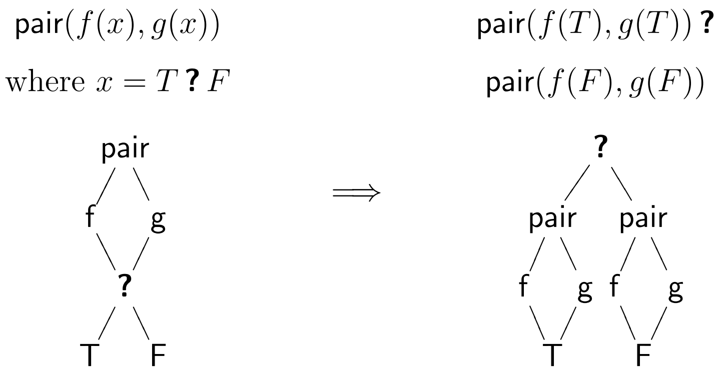


Figure 3.5: The bubbling transformation on an expression. The system must find a dominator to perform the transformation. $\Longrightarrow$ represents the bubbling step.

choice, but it does not clone the choice. This is not a local transformation and finding the dominator can be expensive because it requires traversing all paths back to the root of the graph. An example of this transformation on an expression is shown in Figure 3.5.

Because all the different choices are kept in the same expression, it is also possible to share the values of subexpressions between non-deterministic branches. For example, given an operation defined by the rule $f(x) \rightarrow \textbf{let } y = \mathsf{slow}(x) \textbf{ in } (y + 1)\,\textbf{?}(y+2)$, when a traditional backtracking or cloning system evaluations f 100 it will evaluates slow 100 twice — once for each side of the choice. However, in bubbling, the evaluation of slow 100 will be shared between the non-deterministic branches.

Pull-tabbing

The Basic Scheme uses a technique called pull-tabbing to handle non-determinism, which has several advantages over previous techniques. Pull-tabbing [Antoy, 2011] allows more control over when choices will be handled. Also, like bubbling, computations are shared between branches of the non-determinism, which can dramatically increase performance. Pull-tabbing can only be applied to limited overlapping

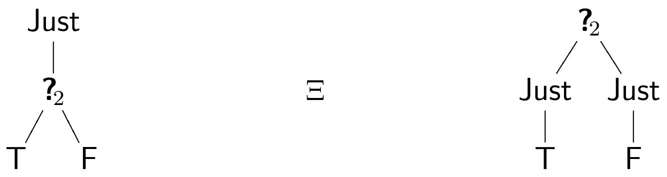


Figure 3.6: A simple example of pull-tab transformation (written Ξ) to show the "unzipping" process that gives it its name. The Just node is duplicated ("unzipped") to allow the choice to be moved toward the root of the expression.

systems because all non-determinism must be represented by the choice operator **?**.

Intuitively pull-tabbing moves choices up toward the root of the expression by "unzipping" the nodes above it. It is called pull-tabbing because the process resembles pulling on the choice and unzipping the expression as if it where a zipper and the choice were the tab of the zipper. An example of this is shown in Figure 3.6.

Pull-tabbing moves choices up to the root of the expression e by replacing symbols applied to a choice with a choice of symbol applied to the two branches of the original choice. This moves the choice toward the root of e and duplicates the symbol. A more complex example is given in Figure 3.7. The result e' of repeated pull-tab steps is an expression that has all the choices near the root. A pull-tab step is written $e \Xi e'$. Under the choices are deterministic expressions; each expression is the result of a different non-deterministic evaluation of the choices in e by the rules of **?**. However, choices may be moved up more than one path, resulting in the choice being duplicated. Because of this, choices are given an ID that is carried by all duplicates, so that the system can handle this case correctly.

Formally, given an expression e with a subexpression $g : s(\dots, p : \mathbf{?}_i(x, y), \dots)$, where s is any symbol and x and y are arbitrary subexpressions, a pull-tab step $e \Xi e'$ is a replacement $e[g \leftarrow \mathbf{?}_i(s(\dots, x, \dots), s(\dots, y, \dots))]$. This duplicates the node g (note that there are two s-rooted symbols in the replacement) and moves

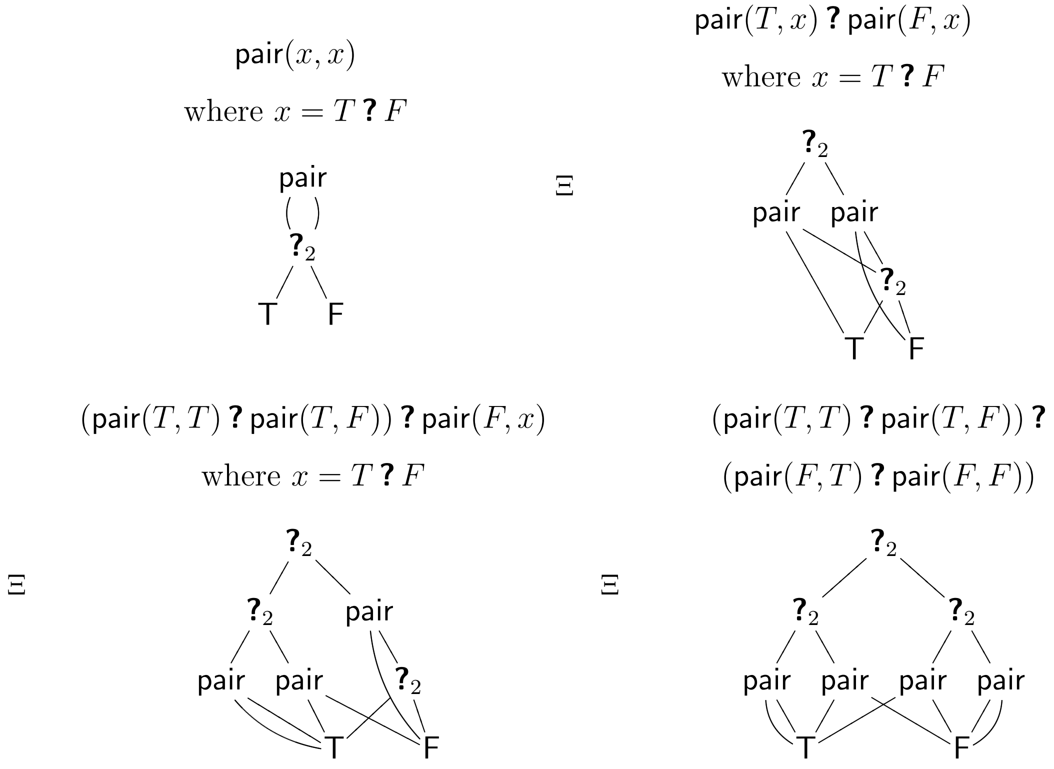

pair(x, x)
where x = T ? F
pair
?2
T
F
Ξ
pair(T, x) ? pair(F, x)
where x = T ? F
?2
pair
pair
?2
T
F
(pair(T, T) ? pair(T, F)) ? pair(F, x)
where x = T ? F
Ξ
?2
?2
pair
pair
pair
?2
T
F
(pair(T, T) ? pair(T, F)) ?
(pair(F, T) ? pair(F, F))
Ξ
?2
?2
?2
pair
pair
pair
pair
T
F

Figure 3.7: The pull-tab transformation (written Ξ) applied repeatedly to an expression. Each step moves a choice nearer the root and may duplicate a choice. The final expression is in non-determinism normal form. The notation $\mathbf{?}_2$ represents a choice with ID 2.

the choice towards the root of the expression.

In the resulting expression, we can define *consistent paths* as paths that consistently take the same branch (left or right) of choices with the same ID. Similarly, a *consistent derivation* is a derivation that always chooses the same branch when non-deterministically evaluating a choice with a given ID [Antoy, 2011, Definition 4]. Every value on a consistent path is the result of a consistent derivation. A *consistent value* is a value that is the result of a consistent derivation. Inconsistent values are ignored because they may not be correct in a system using call-time choice.

For example, the last expression of Figure 3.7 shows four values of which only $\mathsf{pair}(T, T)$ and $\mathsf{pair}(F, F)$ are consistent. These values can be reached by consistently following the same branch (left or right) when the traversal reaches choice with ID 2. However, the other values are inconsistent because they can only be reached by paths that go both left and right at choices with ID 2.

Formally, consistent paths are defined in terms of the *fingerprint* of the path. A path's fingerprint is a set of pairs of choice IDs and directions (Left or Right) that defines the choices that where traversed and the side that was taken at each choice. A consistent fingerprint contains at most one pair for each choice ID. The consistent values of Figure 3.7 have fingerprints of $\{(2, \text{Right})\}$ and $\{(2, \text{Left})\}$. However the inconsistent values have the fingerprint $\{(2, \text{Right}), (2, \text{Left})\}$ meaning that they did not consistently make the same choice for the choice ID 2.

3.3 EVALUATION STRATEGIES

To evaluate an expression to normal form, we need a *strategy* to find subexpressions to rewrite and rules that apply to those subexpressions. One strategy is to choose a subexpression randomly and then search for a matching rule in the set. However, this is inefficient because we may perform unnecessary work, if the rewrite we choose to do is never needed.

$$\begin{array}{lr}
B(g : d(p : \mathbf{?}_i\,(n_x, n_y))) = \mathcal{P}_g(g, p); B(L(g)); B(R(g)); & \text{B.1} \\
B(g : d(x)) = g[g \leftarrow v(x)]; & \text{B.2} \\
B(\_) = null & \text{B.3}
\end{array}$$

Figure 3.8: The procedure B produces a computation $\omega(B(e))$ that, if $e = d(g)$, performs pull-tab steps on $d(g)$ to bring any choices near the root of g above the application of d and then rewrites the applications of d with applications of v.

A *needed* rewrite is a rewrite that must be done to get the expression to normal form. For example, given the operation head defined by the rule:

$$\mathsf{head}(x :: xs) \rightarrow x$$

When evaluating $\mathsf{head}(f(x) :: g(y))$ to full normal form rewriting $f(x)$ is needed, but rewriting $g(y)$ would be wasted because head will never actually use the value of $g(y)$. A *needed node* is a node in an operation's patterns that must be brought to head normal form. For example, in $\mathsf{head}(x)$, x is needed because the rule for head requires that x has a specific constructor so it must be in head normal form to apply the rule.

3.4 GRAPH REWRITING PROCEDURES

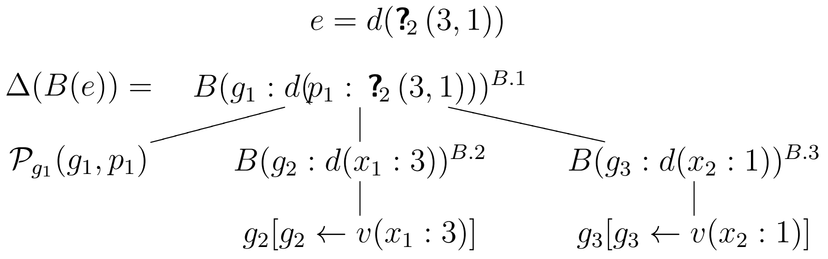
To formalize the Basic Scheme, we need a way to describe a specific deterministic rewriting process. I will use a set of procedures that compute a sequence of actions based on the state of the expression. This syntax and approach was developed by Sergio Antoy for use with the Basic Scheme [Antoy and Peters, 2012].

Procedures take an expression as an argument and return a sequence of actions to perform on a global state of the expression. Each action operates on a specific input state and produces an output state. Figure 3.8 shows a procedure that returns actions that replace all applications of $d(x)$ near the root of the expression with $v(x)$ and makes sure that x is not choice-rooted by performing pull-tab steps.

Procedures are defined using a sequence of rules where earlier rules have higher priority than later ones (in the same way as in deterministic functional languages). Each rule has a pattern for its argument. If the pattern matches the argument then the rule can be applied to derive a series of actions. However, no rule is applied if a rule of higher priority is also applicable. These procedures return a sequence of actions, not the result of applying those actions. The actions are applied as a side-effect. This is why the procedures are not referred to as functions. For example, in Figure 3.8, B.3 is only applied if B.2 cannot be applied so B.3 will never be applied to a d-rooted expression.

Each action is either a pull-tab step, a replacement, or a call to another procedure. Pull-tab steps, written $\mathcal{P}_g(d, s)$, pull the choice s up along a path to d as discussed in Section 3.2.2. Expression replacement is written using an overloading of the standard notation, $g[d \leftarrow s]$, meaning that all references to d in g are replaced with references to s. In this context, $g[d \leftarrow s]$ refers to the action of replacing d with d in g instead of the expression resulting from that replacement. Because of this, the action of replacing g itself is written $g[g \leftarrow s]$. Procedure calls are written as $Y(e)$ where Y is a procedure and e is its argument. Actions are terminated with ";", so a sequence of actions A_i is written $A_1; A_2; \ldots; A_k;$. The empty sequence of actions is written *null*. It is useful to refer to the left and right successors of choices explicitly without pattern matching; these are denoted $L(g)$ and $R(g)$ respectively, where g is a choice-rooted expression.

Actions form a finitely branching tree, called the *computation*, denoted $\Delta(A)$, where A is an action. Single actions (rewrites and pull-tabs) become leaves and procedure calls become branches. Specifically, if A is a rewrite or pull-tab step, then $\Delta(A) = A$. If A is a procedure call $Y(e)$ and there is a rule $Y(p) = A_1; \ldots; A_k$ where e matches p with a substitution σ and this rule is of higher priority than all other matching rules, then $\Delta(A) = (A, [\Delta(\sigma(A_1)), \ldots, \Delta(\sigma(A_k))])$. An example based on the procedure in Figure 3.8 is shown in Figure 3.9a.


$e = d(\boldsymbol{?}_2\,(3,1))$
$\Delta(B(e)) =$
$B(g_1 : d(p_1 : \boldsymbol{?}_2\,(3,1)))^{B.1}$
$\mathcal{P}_{g_1}(g_1, p_1)$
$B(g_2 : d(x_1 : 3))^{B.2}$
$B(g_3 : d(x_2 : 1))^{B.3}$
$g_2[g_2 \leftarrow v(x_1 : 3)]$
$g_3[g_3 \leftarrow v(x_2 : 1)]$

(a) The computation $\Delta(B(e))$.

	Derivation	$\omega(B(d(\boldsymbol{?}_2\,(3,1))))$
$d(\boldsymbol{?}_2\,(3,1))$	$\equiv c(\boldsymbol{?}_2\,(g_2 : d(3), g_3 : d(1)))$	$\mathcal{P}_{g_1}(g_1, p_1)$
	$\rightarrow c(\boldsymbol{?}_2\,(g_2 : v(3), g_3 : d(1)))$	$g_2[g_2 \leftarrow v(x_1 : 0)]$
	$\rightarrow c(\boldsymbol{?}_2\,(g_2 : v(3), g_3 : v(1)))$	$g_3[g_3 \leftarrow v(x_2 : 1)]$

(b) The simulated computation $\omega(B(e))$

Figure 3.9: The resulting computation (a) and simulated computation (b) from the call $B(e)$ where $e = d(\boldsymbol{?}_2\,(3,1))$. In the computation, procedure calls are shown as nodes with the actions resulting from them as children. The replacements and pull-tab steps are represented using the syntax introduced for describing the procedures. The procedure calls are also annotated (as a superscript) with the rule applied. The simulated computation is shown as a derivation annotated, in the right column, with the actions that perform each rewrite. All expressions are written in linear notation.

The *simulated computation* $\omega(A)$ resulting from an action A is the sequence of replacements and pull-tabs that are encountered in a depth-first, left to right traversal of the computation $\Delta(A)$. The semantics of this computation are simply the application of the replacements and pull-tabs in order. As will be shown later, if the procedures are properly defined, then the simulated computation can be considered a graph rewriting derivation. An example based on the procedure in Figure 3.8 is shown in Figure 3.9b.

The *trace* of a node n is either the node itself or a node m that replaced n during a rewrite [Antoy and Peters, 2012]. Intuitively the trace of a node is the same node after it is updated in place. All references to nodes are implicitly traces. So if n and m are nodes that may be the same, $B(m); B(n)$; will first apply the actions that result from B to m and then apply the actions of B to the trace of n, which may not be the original n because it may have been changed by the actions on m. In general this is equivalent to in-place replacement (changing the content of the node). However, in the case of collapsing rules, a simple in-place replacement is not sufficient. In the formal description, we will use the concept of a trace to work around this problem. However, in the implementation, we introduce a kind of indirection node that allows all replacements to be handled in-place.

Because traces are used systematically, each action A_i in a sequence $A_0; \ldots; A_k$ operates on the output of the previous action A_{i-1} or, in the case of A_0, on the initial state. So they can be considered a series of steps operating on the same mutable object, which is the intuitive model of the rewrite procedures. Similarly, in the computation tree, each action operates on the output of the previous action, where the *previous action* of A is defined as the action that appears before it in depth first traversal order: the preceding sibling of A or the previous action of the parent of A. The root of the computation operates on the input expression and has no previous action.

Chapter 4

IMPLEMENTING FUNCTIONAL LOGIC LANGUAGES

4.1 CORE LANGUAGES

To ease the implementation of a programming language a simple *core language* is often used. A core language is able to express everything that is expressible in the programming language, but is significantly simpler.

FlatCurry [Hanus, 2008b] is a core language used in the compilation of Curry. It was initially developed for PAKCS, but has since been used as the input for many other implementations including KICS2 and ViaLOIS. FlatCurry has the same basic structure as Curry, but it removes a number of features such as anonymous functions. FlatCurry provides the following features: function and constructor application; function and constructor partial application; variable references; `case` expressions limited to matching on the root constructor of an expression; non-determinism both as free variables and non-deterministic operations; higher-order functions via partial applications and an `apply` function; and `let` expressions to bind variables. FlatCurry also provides information about the relationships between modules and the types declared. However, although the types of all the symbols defined in the module are specified, the actual code of the functions is untyped.

The LOIS intermediate representation or LOIS-IR (which is based on LOIS GRSs) is an even simpler core language that developed for use in ViaLOIS. LOIS-IR and how to generate it from FlatCurry is one of the contributions of this thesis. The LOIS-IR is, in effect, the definitional trees of all the operations in the program. Like FlatCurry, LOIS-IR is untyped in general, but contains some type information such as which types are in the system and which constructors belong to each of them. Unlike a true LOIS GRS, LOIS-IR allows: built-in types and constants; generators that represent every possible value of their type; partial application of operations and constructors to build data structures that can later be evaluated; and `let` expressions to represent sharing. Polymorphic functions can be translated into LOIS-IR because it is untyped.

The representation of an operation's definitional tree is similar to the definition of definitional trees in Definition 3.2 except that: each node only contains a pattern to match the inductive node of its parent; the inductive nodes are specified by referencing a variable that is bound by a pattern in the current or any ancestor branch node; and rules contain only the right-hand side of the rule because the pattern is implied by the path taken through the definitional tree. Also, LOIS-IR allows the compiler to tag a branch as incomplete so that any constructor that is not explicitly mentioned is assumed to be exempt. LOIS-IR also allows matching using literals of built-in types in addition to constructors.

4.2 COMPILING TO LOIS-IR

In FlatCurry, flow of control is described using limited `case` expressions that can be viewed as definitional trees. However, unlike FlatCurry, LOIS-IR does not allow branching on values other than arguments to functions and only allows this at the top-level of the operation, so single FlatCurry functions are split into multiple LOIS-IR operations whenever these features are encountered. For example, Figure 4.1 shows the conversion of several FlatCurry expressions into LOIS-IR.

FlatCurry expressions are divided into two classes for the conversion process. *Simple expressions* contain only variable references, constructors and operations that have simple arguments, choices with simple arguments, and `let` expressions involving only simple expressions. *Complex expressions* are expressions that are not simple. Simple expressions can be converted into LOIS-IR directly because

$$\begin{array}{l} f(x) = \textbf{case } x \textbf{ of} \\ \qquad \textsf{True} \to 1 \\ \qquad \textsf{False} \to 0 \end{array} \quad \Rightarrow \quad \begin{array}{l} f(\textsf{True}) \to 1 \\ f(\textsf{False}) \to 0 \end{array}$$

(a) case expression on a variable

$$\begin{array}{l} f(x) \to \textbf{case } g(x) \textbf{ of} \\ \qquad \textsf{True} \to 1 \\ \qquad \textsf{False} \to 0 \end{array} \quad \Rightarrow \quad \begin{array}{l} f(x) \to f'(g(x)) \\ f'(\textsf{True}) \to 1 \\ f'(\textsf{False}) \to 0 \end{array}$$

(b) case expression on a non-variable

$$\begin{array}{l} f(x) \to h(\textbf{case } x \textbf{ of} \\ \qquad \textsf{True} \to 1 \\ \qquad \textsf{False} \to 0) \end{array} \quad \Rightarrow \quad \begin{array}{l} f(x) \to h(f'(x)) \\ f'(\textsf{True}) \to 1 \\ f'(\textsf{False}) \to 0 \end{array}$$

(c) case expression inside an application

Figure 4.1: (a) shows how a case expression on an argument is moved into the pattern matching and hence encoded in the definitional tree. (b) shows how a case expression over a complex expression is translated into LOIS-IR by lifting the complex expression into a new operation f', so that the case expression can be encoded as the definitional tree of f'. (c) shows how a complex expression (a case in this example) that is an argument to an operation is lifted into a new operation. The symbol ⇒ represents transformations performed by ViaLOIS to convert FlatCurry into LOIS.

each simple FlatCurry structure maps directly to an equivalent LOIS-IR structure; complex expressions that cannot.

Complex FlatCurry expressions must be split into multiple different rules and operations in LOIS-IR. FlatCurry `case` expressions at the top level of a function that match against an argument are converted into branches in the definitional trees of the operation (Figure 4.1a). However, this does not work for `case` expressions that match against more complicated expressions; in this case a new operation is built to perform the `case` over an argument (Figure 4.1b). FlatCurry `let` expressions are simplified by lifting complex expressions being bound or in the body of the `let` into new LOIS-IR operations and then generating a LOIS-IR `let` expression of the simplified expression. Finally, `case` expressions nested inside function or constructor applications are lifted into separate LOIS-IR operations because all pattern matching in LOIS-IR must be done by a operation (Figure 4.1c).

LOIS-IR does not support free variables as such, but it does allow for generator functions which non-deterministically evaluate to every value of a type. Free variables in a FlatCurry program are converted into generators. Free variables and generators are equivalent as discussed in Section 2.3.1. FlatCurry does not have types for local variables (including free variables), so the generator is untyped and the specific generator to use must be chosen at runtime.

4.3 THE BASIC SCHEME

The *Basic Scheme* is formalized as a compilation process that converts a LOIS *source* system into a set of strict, deterministic rewrite procedures called the *target* program (see Section 3.4). This target program is easier to implement simply and efficiently than the source program. The Basic Scheme, arguments for its correctness, and extensions that allow better efficiency are some of the contributions of this thesis.

Three procedures are used: **N** (Normalize), **H** (Head normalize), and **A** (Adjust). They are formally defined in Figure 4.2.

- **H**(g) performs one rewrite or pull-tab step to bring g closer to head normal form. **H** is defined based on the definitional trees of the operations in the source system. **H** will either recursively call **H** to bring a needed node to head normal form (if the needed node is an operation) or will perform a rewrite (if a rule is applicable) or pull-tab step (if the needed node is a choice).
- **A**(g) performs pull-tab steps on g — which must already be constructor-rooted with successors in non-deterministic normal form — to bring all choices in the subexpressions of g to the root of g. The result of **A** is in non-deterministic normal form.
- **N**(g) brings the expression g to non-deterministic normal form. First, **N** brings g into head normal form using **H**, then **N** recursively calls itself to bring all of g's successors to non-deterministic normal form, then **N** calls **A** to pull all the choices to the root.

If any procedure encounters node labeled with $\perp$ in a needed position it will rewrite its argument to $\perp$. This is left implicit for clarity; these implicit rules will be referred to as **H**.$\perp$, **N**.$\perp$, and **A**.$\perp$.

Since **H** varies dramatically based on the system being compiled, it is presented as a algorithm that translates the definitional trees of the source LOIS system into the set of rewrite procedure rules for **H**. The rules are generated in most specific to least specific order. Each branch node is converted into a rule that examines the value at the inductive node defined by the branch node and either performs pull-tabs or recursively calls **H** on the inductive node. Each rule node is converted into a rule in **H** that performs the rewrite described by the rule. Each exempt node is converted into a rule in **H** that rewrites its argument to $\perp$. The resulting

$\mathbf{N}(\mathbf{?}_i\,(n_x : \_, n_y : \_)) = \mathbf{N}(n_x); \mathbf{N}(n_y);$ **N**.1

$\mathbf{N}(g : c(n_{x_1} : \_, \ldots, n_{x_k} : \_)) = \mathbf{N}(n_{x_1}); \ldots \mathbf{N}(n_{x_k}); \mathbf{A}(g);$ **N**.2

$\mathbf{N}(g : f(\_, \ldots, \_)) = \mathbf{H}(g); \mathbf{N}(g);$ **N**.3

$\mathbf{A}(g : c(p : \mathbf{?}_i\,(\_, \_), \_, \ldots, \_)) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1

$\mathbf{A}(g : c(\_, p : \mathbf{?}_i\,(\_, \_), \ldots, \_)) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1

$\vdots$

$\mathbf{A}(g : c(\_, \_, \ldots p : \mathbf{?}_i\,(\_, \_))) = \mathcal{P}_g(g, p); \mathbf{A}(L(g)); \mathbf{A}(R(g));$ **A**.1

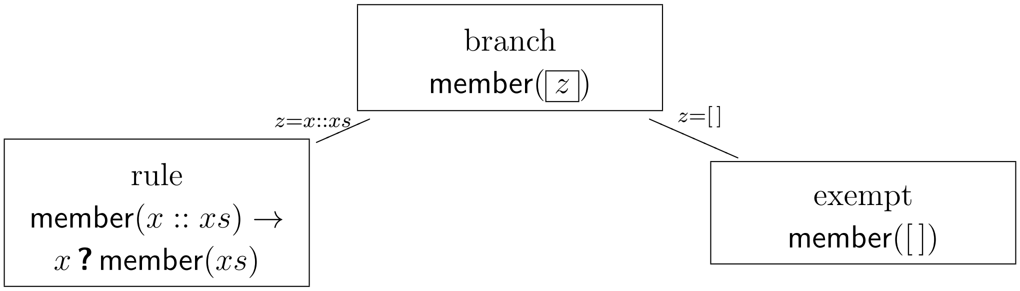
$\mathbf{A}(c(\_, \_, \ldots \_)) = null$ **A**.2

```
compile $\mathcal{T}$
    case $\mathcal{T}$
    when $branch(\pi, o, \bar{\mathcal{T}})$ then
        $\forall \mathcal{T}_i \in \bar{\mathcal{T}}$ compile $\mathcal{T}_i$
        output $\mathbf{H}(g : \pi[o \leftarrow p : \mathbf{?}_i\,(\_, \_)]) = \mathcal{P}_g(g, p);$    H.1
        output $\mathbf{H}(g : \pi) = \mathbf{H}(\pi|_o);$    H.2
    when $rule(\pi, l \to r)$ then
        output $\mathbf{H}(g : l) = g[g \leftarrow r];$    H.3
$\mathbf{H}(c(\_, \ldots \_)) = null$    H.4
```

Figure 4.2: Compilation of a *source* program into a *target* program consisting of 3 procedures: **N**, **H** and **A**. The syntax and semantics of the procedures is described in Section 3.4. The rules of **N** and **A** depend only on the set of operations and constructors. The rules of **H** are obtained from the definitional tree of each operation by the algorithm compile. The symbols c and f stand for a generic constructor and operation of the *source* program and i is a choice identifier. The call to a *target* procedure with some argument g systematically operates on the *trace* of g. [figure and caption based on Antoy and Peters, 2012]

$$\mathsf{member}(x :: xs) \to x \,\mathbf{?}\, \mathsf{member}(xs)$$

(a) The source LOIS system.



(b) The definitional tree representing (a).

$$\mathbf{H}_{\mathsf{member}}(g : \mathsf{member}(h : \mathbf{?}_i\,(\_, \_))) = \mathcal{P}_g(g, h) \qquad \mathbf{H}_{\mathsf{member}}.1$$

$$\mathbf{H}_{\mathsf{member}}(g : \mathsf{member}(h : f(\ldots))) = \mathbf{H}(h) \qquad \mathbf{H}_{\mathsf{member}}.2$$

where f is any operation symbol

$$\mathbf{H}_{\mathsf{member}}(g : \mathsf{member}(x :: xs)) = g[g \leftarrow x \,\mathbf{?}\, \mathsf{member}(xs)] \qquad \mathbf{H}_{\mathsf{member}}.3$$

$$\mathbf{H}_{\mathsf{member}}(g : \mathsf{member}([\,])) = g[g \leftarrow \bot] \qquad \mathbf{H}_{\mathsf{member}}.4$$

(c) The $\mathbf{H}_{\mathsf{member}}$ fragment generated from (b).

Figure 4.3: An example of conversion of a LOIS system into a strict deterministic program by the Basic Scheme. The symbol member is an operation symbol and :: and [ ] are constructor symbols.

H procedure is made up of a number of **H** *fragments* each implementing the rules for a single operation. These fragments are treated as separate procedures when it is convenient. The fragment implementing an operation f is called $\mathbf{H}_f$.

Figure 4.3 shows an example of a LOIS system and how it is compiled by the Basic Scheme. A complete evaluation in the target program is equivalent to performing all possible non-deterministic derivations in the source program. For example, in Figure 4.4 the result is a choice between three different values, each representing a possible non-deterministic derivation of the source program.

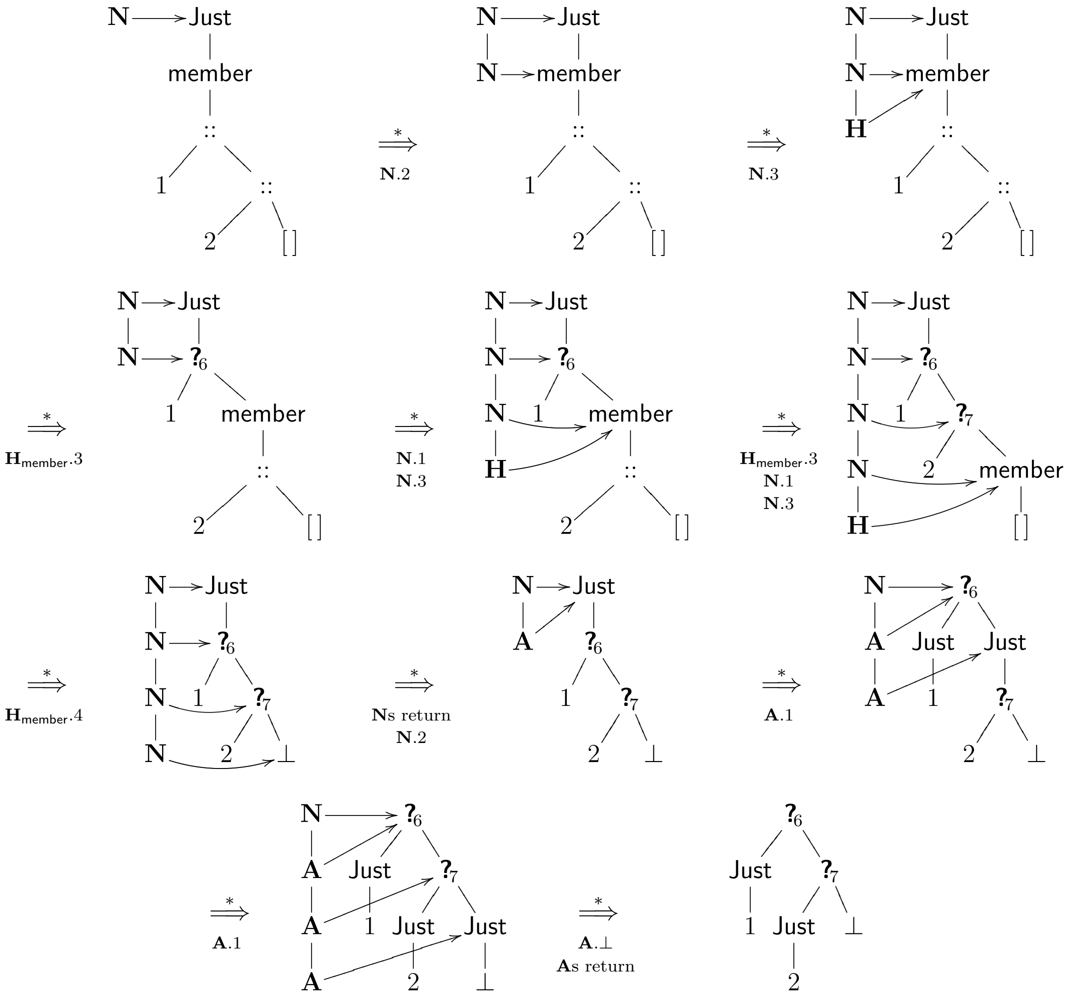


Figure 4.4: The evaluation of the expression Just(member(1 :: 2 :: [])), where Just is a constructor and member is the operation defined in Figure 4.3a, by the target program shown in Figure 4.3c combined with Figure 4.2. In each state of the computation, the left column is the call stack of the target program and the expression to the right is the state of the evaluation. The arrows from the procedures on the stack to the expression show which node is the argument of that procedure call; The transitions between the states are annotated with the procedure rules used to perform the step. The symbol $\overset{*}{\Longrightarrow}$ represents one or more computational steps (pull-tabs, replacements, and procedure calls).

Together, the target procedures **N**, **H**, and **A**, provide a concrete implementation of source LOIS system. The target procedures can be evaluated eagerly without losing the lazy semantics of the LOIS system. The implementation is not dependent on the evaluation order of the target language because the LOIS functions are represented as data.

4.3.1 Correctness of the Basic Scheme

In this section, I provide theorems that show the correctness of the Basic Scheme and the outlines of proofs for these theorems. The theorems assume that $\mathbf{N}(e)$, $\mathbf{A}(e)$, and $\mathbf{H}(e)$ terminate on their argument.

Lemma 4.1 states that any computation in the target program simulates some pull-tabbing derivation in the source program.

Lemma 4.1 (Simulation)**.** *Let S be a* LOIS *system, T the program obtained from S by the Basic Scheme, e an expression of S, and Y a procedure of T. If $\Delta(Y(e))$ is finite, then $\omega(Y(e))$ is a pull-tabbing derivation of e in S, i.e., $e \Rightarrow t_1 \Rightarrow \ldots t_n$, for some $n \geqslant 0$.* [from Antoy and Peters, 2012, Lemma 1 (Simulation)]

Proof outline of 4.1. By structural induction over the computation $\Delta(A)$ where A is an action over e.

Base case: If A is a pull-tab step or rewrite, then $\omega(A)$ is a pull-tabbing derivation of length one. If A is *null* or an uninterpreted procedure application then $\omega(A)$ is a pull-tabbing derivation of length zero.

Inductive case: A must be a procedure application so $\Delta(A) = (Y'(e_0), B)$, where e_0 is the state of the expression before A is applied, $B = a_1, a_2, \ldots, a_k$, and a_i is a sub-computation of $\Delta(Y'(e_0))$. By induction, each a_i is a derivation of e_0. Specifically, a_1 is a derivation of e_0 with a result e_1. Traces are used at every step of the computation so a_2 must be a derivation of e_1 with a result e_2 and so forth. Every a_i is a derivation of e_{i-1} with a result e_i. So $\omega(Y'(e_0))$ (the concatenation of $\omega(a_1), \omega(a_2), \ldots, \omega(a_k))$ is a derivation $e_0 \overset{*}{\Rrightarrow} e_1 \overset{*}{\Rrightarrow} \ldots \overset{*}{\Rrightarrow} e_k$. e_0 must by a subexpression of e and so any derivation of e_0 is also a derivation of e. Therefore, $\omega(A)$ is a pull-tabbing derivation of e.

If the lemma is true for a computation $\omega(A)$ where A is any action, then it is true for a computation $\omega(A)$ where $A = Y(e)$. □

Theorem 4.2 states that:

1. The Basic Scheme is sound — the target program does not derive any values that are not derivable in the source program.

2. The Basic Scheme is complete — from any state of the computation in the target program it is possible to derive any value derivable in the source program.

Theorem 4.2 (Correctness). *Let S be a* LOIS *system, T the program obtained from S by the Basic Scheme, e_0 an expression in S, and $\mathbf{N}$ the procedure from T. So, $\omega(\mathbf{N}(e_0)) = e_0 \Rrightarrow e_1 \Rrightarrow \ldots$*

1. *For each e_i, if $e_i \overset{*}{\rightarrow} v$ is a consistent derivation in S, then $e_0 \overset{*}{\rightarrow} v$ is a consistent derivation in S.*

2. *For each e_i, if $e_0 \overset{*}{\rightarrow} v$ is a consistent derivation in S, then $e_i \overset{*}{\rightarrow} v$ is a consistent derivation in S.*

[based on Antoy and Peters, 2012, Proposition 1 (Correctness)]

Proof outline of 4.2. By Lemma 4.1, $e_0 \overset{*}{\Rrightarrow} e_k$ defines a pull-tabbing derivation of e_0 in S that never applies the rules of **?**. Therefore, points (1) and (2) are identical to (2) and (1), respectively, of Antoy [2011, Theorem 1]. □

Lemma 4.3 (Normal Form Result of **A**). *If $\mathbf{A}(c(n_1, \ldots, n_k))$, where every n_i is a non-deterministic value, terminates then the last expression of the derivation $\omega(\mathbf{A}(c(n_1, \ldots, n_k)))$ is a non-deterministic value.*

Proof outline of 4.3. By induction over the number of choices in the successors of g.

Base case: If no n_i has a choice at its root, then $\mathbf{A}(c(n_1, \ldots, n_k)) = \mathit{null}$ and $c(n_1, \ldots, n_k)$ is already in non-deterministic normal form.

Inductive case: If some n_i is choice-rooted, $\mathbf{A}(g : c(\ldots, n_i : \mathbf{?}_i\,(\_, \_), \ldots)) = \mathcal{P}_g(g, n_i); \mathbf{A}(L(g)); \mathbf{A}(R(g));$, where n_i is the first successor of g that is choice rooted. $\mathcal{P}_g(g, n_i)$ pulls a single choice above the c, so g is choice-rooted and $L(g)$ and $R(g)$ are c-rooted. Also $L(g)$ and $R(g)$ contain one fewer choice in their ith successor than g originally did. By induction, $\mathbf{A}(L(g))$ and $\mathbf{A}(R(g))$ bring $L(g)$ and $R(g)$ respectively into non-deterministic normal form. So g is a choice between non-deterministic values and hence a non-deterministic value itself. ☐

Theorem 4.4 (Normal Form Result)**.** *If* $\mathbf{N}(e)$ *terminates, the last expression of the derivation* $\omega(\mathbf{N}(e))$ *is in non-deterministic normal form.*

Proof outline of 4.4. By induction on the expression e.

Base case: If $e = c()$ where c is an arbitrary constructor or $e = \perp$ then e is already a non-deterministic value and $\omega(\mathbf{N}(e))$ is empty because $\mathbf{N}(e) = \mathbf{A}(e) = \mathit{null}$ or $\mathbf{N}(e) = \mathit{null}$, respectively.

Inductive case: There are three cases.

1. If $e = x \,\mathbf{?}\, y$, then $\mathbf{N}(e) = \mathbf{N}(x); \mathbf{N}(y)$. By induction this will bring x and y to non-deterministic normal form and, by definition, a choice of non-deterministic values is a non-deterministic value.
2. If $e = c(n_1, \ldots, n_k)$ where c is an arbitrary constructor and $k > 0$, then $\mathbf{N}(e) = \mathbf{N}(n_1); \ldots; \mathbf{N}(n_k); \mathbf{A}(e);$. By induction and the premise that $\mathbf{N}$ terminates, this will bring $n_1, \ldots, n_k$ into non-deterministic normal form. By Lemma 4.3, $\mathbf{A}(e)$ will bring e to non-deterministic normal form.
3. If $e = f(\ldots)$ where f is an arbitrary operation, then $\mathbf{N}(e) = \mathbf{H}(e); \mathbf{N}(e);$.

H(e) will perform a rewrite or pull-tab step on e to bring it closer to head-normal form, so when **N**(e) is called recursively e will be at least one step closer to head normal form. The recursive call to **N**(e) will repeatedly call **H**(e) until e is in head normal form. Once e is in head normal form **N**(e) will bring e to non-deterministic normal form by case 2. □

4.3.2 Extensions To the Basic Scheme

The Basic Scheme evaluates all possible combinations of choices, but it also clones choices and will evaluate expressions that can only be reached by inconsistent paths. We extend the Basic Scheme to use fingerprints to avoid evaluating expressions that are only needed on branches that will not produce consistent values. To do this we need a new normal form: *consistent normal form* is non-deterministic normal form except that it only requires subexpressions to be in non-deterministic normal form if they are on consistent paths. Evaluating an expression to consistent normal form is sufficient to get the values because all consistent values will be in normal form and those are the only useful values.

An implementation can take advantage of this by passing a fingerprint to each procedure and any procedure called with an inconsistent fingerprint will return *null*. This avoids wasting time evaluating expressions on inconsistent paths and will bring the expression to consistent normal form because every consistent path will still be fully evaluated. I believe that Theorem 4.2 and Theorem 4.4 can be extended to show that consistent normal form evaluation is sound, complete and results in a consistent normal form.

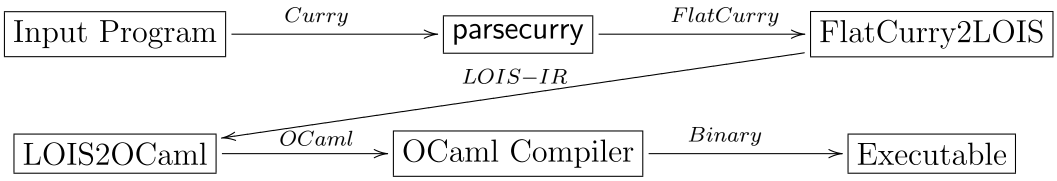


Figure 4.5: A block diagram of the ViaLOIS compiler. The components FlatCurry2LOIS and LOIS2OCaml are the core of ViaLOIS and are custom. The components `parsecurry` (part of PAKCS [Hanus, 2008a]) and the OCaml Compiler [INRIA, 2011] are existing tools.

4.4 VIALOIS: IMPLEMENTING THE BASIC SCHEME

ViaLOIS [Peters, 2012b] is a prototype implementation of Curry that uses the Basic Scheme. Its name comes from its use of LOIS-IR as an intermediate representation during compilation. ViaLOIS is one of the contributions of this thesis. It is available online at `http://web.cecs.pdx.edu/~amp4/vialois`. I chose to use OCaml [INRIA, 2011] because it is eager and provides powerful pattern matching (which is not required, but makes the implementation easy). Also OCaml is a mature language with a compiler that produces very efficient code. ViaLOIS has two stages and a single intermediate language. ViaLOIS takes FlatCurry as input and converts it to LOIS-IR, using the techniques described in Section 4.2. ViaLOIS then compiles the LOIS-IR into an OCaml program using the Basic Scheme. Each constructor becomes an OCaml constant that contains the constructor's name, arity, and a pointer to the type's generator function. Each operation f becomes an OCaml constant with name, arity, and an OCaml function that implements the fragment $\mathbf{H}_f$. The translation of operations is discussed in more detail in Section 4.4.2. The ViaLOIS runtime library provides implementations of $\mathbf{N}$ and $\mathbf{A}$. This process along with the pre-existing components that were used are shown in Figure 4.5.

ViaLOIS represents expressions as a graph of mutable records each representing a node and containing a symbol and a list of successors. The symbol is either: an operation symbol, a constructor symbol, a choice containing an ID, a special symbol representing a value of a built-in type, or an indirection node as defined in Section 4.4.1.

4.4.1 Complications of In-place Rewriting

In-place rewriting avoids the expense of redirecting all the pointers to the original node to a new version as is done in the formalization of graph rewriting. However, an additional kind of node is needed to maintain sharing in collapsing rules. *Indirection nodes* act as a pointer to another node. An indirection node pointing to a node n is written $\mathsf{ind(n)}$. Collapsing rules rewrite the function application to an indirection node. This ensures that the argument is not copied, breaking sharing. For example, in the program shown in Figure 4.6a, $\mathsf{id}(x)$ cannot be replaced in-place by a copy of x because then $\mathsf{f}(x)$ would return two copies of x which would lose sharing, which is semantically significant as well as important for performance. Instead $\mathsf{id}(x)$ rewrites to $\mathsf{ind}(x)$, so $\mathsf{f}(x)$ returns a pair with two indirect references to the same shared node. An example is show in Figure 4.6.

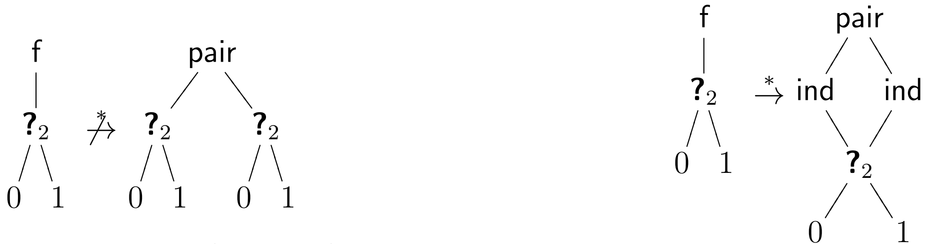
Every time a node is referenced, the runtime must check if it is an indirect node and, if so, operate on the target. Chains of indirect nodes can form and each one in the chain must be dereferenced. Whenever an indirection node is traversed, ViaLOIS “flattens” chains of indirection nodes by updating each one to point to the target instead of another intermediate indirection node. This helps reduce the cost, but does not remove it.

4.4.2 Translation and Runtime System

The translation of LOIS-IR into OCaml follows the Basic Scheme very closely. The **N** and **A** functions are implemented generically in the runtime because constructors and operations are easy to differentiate at runtime regardless of the program

$$\mathsf{id}(x) \to x$$
$$\mathsf{f}(x) \to \mathsf{pair}(\mathsf{id}(x), \mathsf{id}(x))$$

(a) A program that requires indirection nodes.



(b) An incorrect rewrite due to mishandling of collapsing rules.

(c) A correct rewrite. ind represents an indirection node.

Figure 4.6: Because of the collapsing rule defining id, f (0 ?1) must be rewritten as shown in (c) instead of as shown in (b) because sharing is lost in (c). The symbols $\xrightarrow{*}$ and $\not\xrightarrow{*}$ represent correct and incorrect derivations respectively.

that is being evaluated. For each operation f, a function $\mathbf{H}_f$ is generated that implements the fragment denoted by the same notation; the runtime dispatches calls to $\mathbf{H}$ to the appropriate $\mathbf{H}_f$ function. In ViaLOIS, $\mathbf{H}$ brings its argument to head normal form instead of performing one step as described in the Basic Scheme.

$\mathbf{H}_f$ is generated from the corresponding definitional tree by building a tree of match statements in OCaml (the OCaml equivalent of case). These match statements perform the exact same function as the rules of $\mathbf{H}$ in the formal definition. Each branch node generates a match expression that will be nested inside a parent match expression.

These match expressions handle cases where choices or operations are at the inductive node in addition to dispatching to appropriate nested expressions generated by other definitional tree nodes. When the inductive node of an expression e, where e is the expression that $\mathbf{H}$ was called on, is a choice $p : \mathbf{?}_i\,(x, y)$, a pull-tab step is performed. The path from the root of the operation to the inductive node is rebuilt twice, producing two new expressions $a = e[p \leftarrow x]$ and $b = e[p \leftarrow y]$, then e is rewritten in-place to be $\mathbf{?}_i\,(a, b)$. When the inductive node of an expression e is an operation application $n : f(\ldots)$, then $\mathbf{H}$ is called on n and then $\mathbf{H}$ is called on e again to complete the rewrite of e.

Each exempt node in the definitional tree generates a call that rewrites e to $\perp$, where e is the expression on which $\mathbf{H}$ was called. Each rule node generates a call that rewrites e appropriately. In the case of non-collapsing rules, a new expression r is built and e is rewritten in-place to r. In the case of collapsing rules where the right-hand side is a variable x, e is rewritten in-place to $\mathsf{ind}(x)$.

$\mathbf{H}$ dispatches to the specific $\mathbf{H}_f$ functions using a function pointer stored in the operation symbol for f, and $\mathbf{N}$ and $\mathbf{A}$ work by checking what type of symbol a node has without needing to look at the specific name or other properties of the symbol. This design allows easy and efficient dispatch over the symbol of nodes and their successors.

4.4.3 Optimizations

ViaLOIS supports built-in types and operations as specialized symbols that carry a value. This allows for efficient 32-bit integers for instance. Operations over these types are implemented by manually writing the $\mathbf{H}$ fragment for them except that, instead of performing LOIS style pattern matching, it will simply unpack the built-in value (such as an integer), perform some operation on it and then rewrite the operation application to the built-in value that was computed. The integer module takes only 70 lines of code; mostly in a file implementing operations over integers, but with a few lines in the core to declare and handle the special symbol.

In ViaLOIS, instead of performing a single step toward head normal form, $\mathbf{H}$ brings its argument all the way to head normal form by calling itself if it rewrites the root to a new operation instead of a constructor and when a rewrite is done on a subexpression. Also in this case, the call is directly to the $\mathbf{H}_f$ of the appropriate operation since it in known at compile time.

ViaLOIS implements fingerprints as described in Section 4.3.2. The fingerprints are represented using a mutable value in each choice ID that stores which side has been traversed for that choice ID (left, right, or none). This is a major performance advantage because all updates and lookups on the fingerprint are very fast.

4.5 BENCHMARK RESULTS

To evaluate the performance and simplicity of ViaLOIS, I tested it against three other Curry implementations.

- PAKCS [Hanus, 2008a] is a mature Curry implementation that compiles Curry source code into Prolog and hence handles non-determinism using backtracking and suffers from poor performance in deterministic computation.
- KICS2 [Braßel et al., 2011] is a recent implementation that compiles Curry code into Haskell code and implements non-determinism using pull-tabbing. It also makes a concerted effort to detect and take advantage of determinism for efficiency. These benchmarks were run on KICS2 version 0.1.
- MCC [Lux, 2007] is an older implementation of Curry that compiles Curry code to virtual machine code. The virtual machine is implemented in C and uses backtracking to implement non-determinism, but has much better deterministic performance than Prolog based implementations.

For these benchmarks, ViaLOIS, even in its current unoptimized form, is as fast or faster than PAKCS on all benchmarks and as fast or faster than KICS2 on many benchmarks, as shown in Figure 4.7. MCC is faster than ViaLOIS on most benchmark (except Sharing), but it uses a custom virtual machine written in C giving it a performance advantage over the other implementations simply because

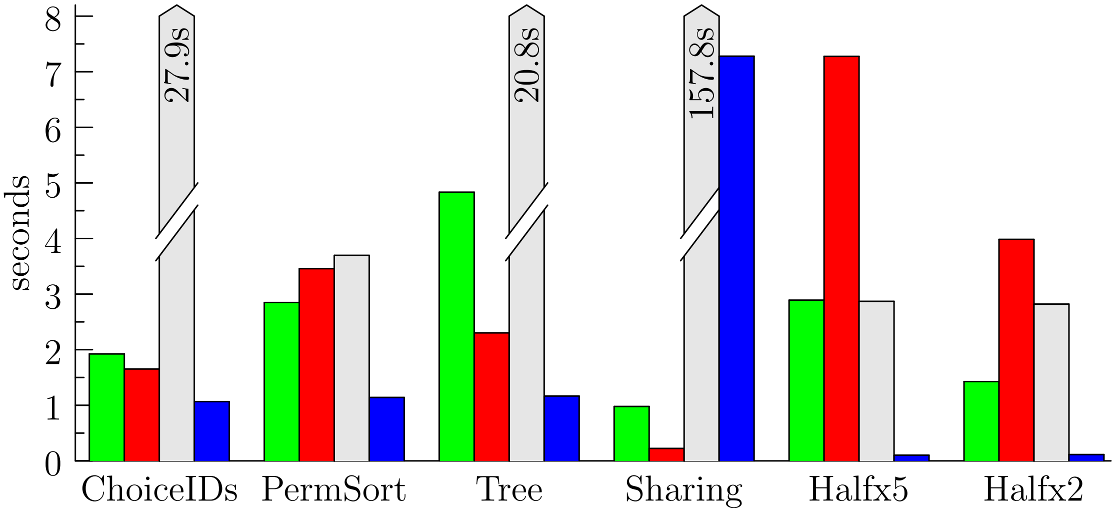


Figure 4.7: Benchmark Results. 🟩 *ViaLOIS* (based on the Basic Scheme), 🟥 KiCS2, ⬜ Pakcs, 🟦 Mcc.

it can take advantage of low-level optimizations not possible in Haskell, OCaml, or Prolog. ViaLOIS requires much less code to implement, giving it a major advantage in simplicity as shown by the lines of code in various implementations of Curry shown in Table 4.1.

The benchmarks were run ten times on each implementation and median eight were averaged to produce the presented values. The code run on each version was nearly identical except for variation in the `main` function to compensate for variations in the implementations handling of non-determinism at the top level of the program. The benchmarks were chosen to test a number of different performance characteristics and to demonstrate that ViaLOIS can perform well.

ChoiceIDs tests the performance of large numbers of non-deterministic choices over built-in types, most of which fail with very little computation. The program non-deterministically generates 262144 integers searching for $262144 - 1$. Due to the left-to-right evaluation order of the implementations being tested every choice will be traversed.

PermSort tests the non-deterministic performance over algebraic data types by performing a permutation sort over a list of 13 integers resulting in over 6×10^9 possible values.

Tree tests the performance of deterministic manipulation of data structures and recursive function calls. This program implements a simple binary tree and inserts 200,000 pseudo-random numbers into it and then counts the number of elements in the tree.

Sharing tests the performance of computations where an expensive computation is shared over different non-deterministic branches. This performs a permutation sort over a list of 5 numbers that are generated by a small version of the Tree benchmark. This benchmark favors the pull-tabbing based implementations because backtracking implementations must repeat the expensive generation of the numbers each time a new non-deterministic branch is tried.

Halfx5 tests the performance of computations where a non-determinism value is shared between different parts of computation. This computes $x = 1500/2$ by solving the equation $1500 = x+x$ and then computes the sum $x+x+x+x+x$. Peano numerals are used. This benchmark shows the performance problems associated with duplicating choices when a choice is shared.

Halfx2 is the same as Halfx5 except that the sum computed is $x + x$. The improvement in the performance of this benchmark over Halfx5 in KiCS2 and ViaLOIS is due to the duplication of choices caused by pull-tabbing. This benchmark combined with Halfx5, shows that ViaLOIS is roughly twice as fast as KiCS2 at pull-tabbing and choice handling.

Although these benchmarks are small, artificial programs, they still test the performance of important parts of the Curry implementations. These benchmarks are based on the benchmark set used to test KiCS2 Halfx2 and Halfx5 are taken

	Compiler	Runtime
ViaLOIS	0.56 (Curry)	0.75 (OCaml)
KiCS2	4.6 (Curry)	1.6 (Haskell)
PAKCS	4.7 (Prolog)	3.3 (Prolog)
MCC	4.3 (Haskell)	9.6 (C)

Table 4.1: Lines of code (in thousands) of several Curry systems. Line counts exclude comments, blank lines, and the standard library. Built-in functions are included as part of the runtime. The parser and first compilation stage (conversion to FlatCurry) is omitted from the line counts as this part of the code is shared between all 4 implementations. [figure and caption from Antoy and Peters, 2012]

directly from benchmark sets developed for KiCS2. The source code for the benchmarks is provided in Appendix A.

As a simple metric of code complexity Figure 4.1 shows the number of lines of code in various implementations of Curry. ViaLOIS is much smaller than any of the other implementations, which supports the idea that it is simpler than the other implementations. Although ViaLOIS is not a complete implementation of Curry, I believe that a complete implementation of Curry based on the Basic Scheme would still be simple and concise, because additions such as IO support and new built-in types have required only 100-200 lines of code. The primary missing features are function patterns and free variables of built-in types. Function patterns should be easy to implement by translating them into currently supported features. Free variables of built-in types will be more complicated because to support them some form of residuation will be needed, but Antoy and myself have some ideas of how to implement this simply.

The difference in size between KiCS2 and ViaLOIS is partly due to the complicated handling of choices used in KiCS2. In ViaLOIS each choice is handled consistently, but in KiCS2 choices can be in 4 different states. Each of these cases must be handled separately whenever choices are handled.

Chapter 5

RELATED WORK

Graph rewriting is the most common formalization of functional logic computation, but it is not the only formalization. Constructor-based conditional rewrite logic [Gonzlez-Moreno et al., 1999] formalizes functional logic programs as statements in a formal logic. Gonzlez-Moreno et al. also provide a term rewriting calculus for computing solutions to programs written in this logic. Some other work has also focused on term tree rewriting instead of graph rewriting such as Hanus [1997]. However, unlike graph rewriting, term rewriting introduces some problems in handling of non-determinism in call-time choice languages, such as Curry. These problems are caused by the inability to represent sharing in a term that is not a graph.

Graph rewriting is also heavily used to formalize functional computation and extensive work has been done in this context (for instance, Peyton Jones [1992]). In general, this work is not applicable to functional logic computation because it assumes determinism, but Section 6.1 discusses possible useful ideas.

In addition to pull-tabbing [Antoy, 2011] used in the Basic Scheme, there are other techniques to handle non-determinism in programs. These techniques are described in Section 3.2.2. The notable techniques are backtracking as used in Prolog, cloning, and Bubbling [Antoy et al., 2006].

Several techniques have been used to implement functional logic programming languages. Compilation to Prolog is used in PAKCS [Hanus et al., 1995, Hanus, 1996]. Prolog provides non-determinism, so it need not be implemented in the Curry runtime, however Prolog is eager in its evaluation so implementations built on Prolog must implement laziness. Also, these Curry implementations suffer from performance problems because Prolog implementations are slow compared to many other languages when doing deterministic computation. However, Prolog based implementations often do very well on non-deterministic benchmarks because Prolog is based on backtracking which is very efficient in non-deterministic computation where there is little or no sharing of subexpressions.

Virtual machines provide better performance and more control over the evaluation strategy compared to targeting Prolog, but then non-determinism and laziness must be implemented. In the case of Mcc [Lux, 2007], non-determinism is implemented much the same way it is in Prolog using backtracking. The use of C allows many optimizations beyond what was performed for ViaLOIS.

A newer implementation, KiCS2, compiles to Haskell [Braßel et al., 2011]. Haskell provides laziness and a very fast runtime, however non-determinism must be implemented in the runtime. KiCS2 is currently the fastest Curry implementation available in many benchmarks. KiCS2 uses pull-tabbing to handle non-determinism with extensions to allow Prolog like unification of expressions and constraint solving for built-in types. KiCS2 was the first Curry implementation to use pull-tabbing to handle non-determinism and has shown that pull-tabbing has major advantages in some cases.

Chapter 6

CONCLUSION

6.1 FUTURE WORK

Finding the relationship between the Basic Scheme and abstract rewriting machines, such as the Spineless Tagless G-Machine [Peyton Jones, 1992], would provide insight into which of the many optimizations that have been applied to those machines would be applicable in a non-deterministic environment. The Spineless Tagless G-Machine has allowed lazy functional languages such as Haskell to dramatically increase their performance with no change in their semantics or ease of use. If a similar virtual machine were developed for functional logic programming languages it would revolutionize the field of functional logic implementation techniques.

In functional logic programs there are two kinds of natural parallelism that can be exploited. “And” parallelism is present when multiple successors of a node can be seen to be needed at the same time and hence evaluated in parallel. This is the case for constructors being evaluated to normal form and of operations that pattern match on multiple arguments. This is called “and” parallelism because both branches of the parallelism must complete for the computation as a whole to finish. “Or” parallelism is present when two or more non-deterministic choices are available. This occurs for every non-deterministic choice. This is called “or” parallelism because only one of the branches must return for the computation as a whole to finish.

LOIS effectively encodes both “and” and “or” parallelism. The Basic Scheme could be extended to support parallel evaluation of LOIS systems. Because every rewrite operation only needs local information, it may be possible to implement rewriting very efficiently and with very little synchronization between threads. For these techniques to be useful, there must be parallelism present in real world code. So a study of the parallelism present in real-world code would be needed.

A Basic Scheme implementation that generates C instead of OCaml would allow low-level optimizations to be explored. This could increase the performance of the Basic Scheme to match MCC except without the performance problems related to backtracking. A C-based implementation would also allow parallel implementation techniques to be explored at a very low level such as the technique I present in Peters [2012a].

The performance of ViaLOIS is partly limited by the garbage collector used by OCaml. The OCaml garbage collector used a write barrier to allow the collector to run in parallel with the program. However, in ViaLOIS, nodes are mutated frequently so the write barriers are a major performance issue. A custom garbage collector could use knowledge of the expression structure and rewrite process to reduce or remove the need for barriers and increase performance. Also the garbage collector could replace pointers to indirection nodes with pointers to their targets. This could be a major advantage in some cases.

Handling of logic variables in the Basic Scheme is limited to algebraic data types and does not work well in practice for large built-in types. A technique is needed to implement generators over built-in types efficiently in ViaLOIS. It may be possible to implement a lazy generator function that produces values only as needed or as requested; this would allow generators to work over large built-in types. Some of the techniques developed by Braßel [2011] on KICS2 may also be helpful.

6.2 CONCLUSION

The Basic Scheme is a novel abstract compilation technique. An intermediate representation (LOIS) of a functional logic program is transformed into three target procedures that execute simple manipulations of a graph representing the state of a computation of an expression. These rewrite procedures are easily mapped to procedures of a concrete programming languages. I have described this compilation technique by formalizing LOIS and the Basic Scheme transformation and precisely defining the target procedures.

The target procedures implement a traversal of a state of the computation of an expression to find a subexpression that can be replaced using either a rewrite step or a pull-tab step. Rewrite steps occur when an operation application is encountered that matches a rule in the original rewrite system. This rewrite will be a valid rewrite in the original rewrite system. Pull-tab steps occur when an operation or constructor application is encountered that has a choice as a successor. The pull-tab step will not destroy any information and every result that was possible before the pull-tab step will still be possible after the pull-tab step.

LOIS graph rewrite systems contribute to the simplicity of the Basic Scheme in a number of ways. Limited-overlapping means that all non-determinism is represented as explicit choices allowing pull-tabbing to be used. Inductively sequential means that the Basic Scheme avoids any unnecessary computation because it is always known what nodes in an expression must be evaluated next. Finally the inclusion of explicit failures allows unsuccessful non-deterministic choices to be eliminated efficiently.

If the procedures produced by the Basic Scheme terminate they will produce every result that is possible in the original LOIS graph rewrite system. However, the Basic Scheme will not terminate if any derivation in the original graph rewrite system is infinite. Specifically, because of the left bias present in the current description of the Basic Scheme, if the left branch of a choice does not terminate, the other branch will never be evaluated, and terminating results on the branch will never be found. A non-biased version of the Basic Scheme would allow all terminating results to be found in finite time even in the presence of infinite derivations. A parallel or interlaced implementation of the Basic Scheme could actually implement this non-biased evaluation.

The simplicity of the Basic Scheme simplifies proving its properties. I proved the soundness of the Basic Scheme by showing that any computation in the Basic Scheme is a set of valid derivations in the original LOIS graph rewrite system. This shows that every result produced by the Basic Scheme is a valid result. I proved the weak completeness claim that a terminating computation in the Basic Scheme computes every possible result from the original graph rewrite system. This shows that no results are lost by the Basic Scheme if it terminates.

The prototype implementation of the Basic Scheme (ViaLOIS) provides a concrete example of the simplicity and power of the Basic Scheme. ViaLOIS closely follows the Basic Scheme in its implementation by using an slightly modified form of LOIS graph rewrite systems as an intermediate representation and then compiling this intermediate representation into OCaml code that implements the graph rewrite procedures. This two step compilation process is simple and easy to implement and modify. ViaLOIS performs well on various functional logic benchmarks, showing the Basic Scheme does not have prohibitive performance overhead and may be appropriate for future practical implementations. ViaLOIS validates the formalism provided by the Basic Scheme by showing the Basic Scheme is not just an abstract idea, but a technique that can be used to implement real programming languages on real hardware.

REFERENCES

S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, Sept. 1992. Springer LNCS 632.

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

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

S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.

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

S. Antoy and A. Peters. Compiling a functional logic language. In *Proceedings of the Eleventh International Symposium on Functional and Logic Programming (FLOPS'12)*, Kobe, Japan, May 2012. Springer LNCS. To appear.

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

S. Antoy, D. Brown, and S.-H. Chiang. On the correctness of bubbling. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications*, pages 35–49, Seattle, WA, Aug. 2006. Springer LNCS 4098.

B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs.* PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.

B. Braßel, M. Hanus, B. Peemller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In H. Kuchen, editor, *Functional and Constraint Logic*

Programming, volume 6816 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22530-7. doi: http://dx.doi.org/10.1007/978-3-642-22531-4_1. 10.1007/978-3-642-22531-4_1.

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

P. Codognet and D. Diaz. Compiling constraints in clp(fd). *The Journal of Logic Programming*, 27(3):185 – 226, 1996. ISSN 0743-1066. doi: 10.1016/0743-1066(95)00121-2.

R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at `ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz`.

J. Gonzlez-Moreno, M. Hortal-Gonzlez, F. Lpez-Fraguas, and M. Rodrguez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40(1):47 – 87, 1999. ISSN 0743-1066. doi: 10.1016/S0743-1066(98)10029-8.

M. Hanus. On the completeness of residuation. In *JICSLP'92*, pages 192–206, 1992.

M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *LOPSTR '95: Proceedings of the 5th International Workshop on Logic Programming Synthesis and Transformation*, pages 252–266, London, UK, 1996. Springer-Verlag.

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

M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005. Available at `http://www.informatik.uni-kiel.de/~mh/publications/reports/`.

M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/`.

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

M. Hanus. Flatcurry: An intermediate representation for Curry programs. Available at `http://www.informatik.uni-kiel.de/~curry/flat/`, 2008b.

M. Hanus, H. Kuchen, and J. J. Moreno-Navarro. Curry: A truly functional logic language. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, Portland, Oregon, 1995.

INRIA. The OCaml programming language. Available at `http://caml.inria.fr/ocaml/index.en.html`, 2011.

J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, May 1992. ISSN 0164-0925. doi: 10.1145/129393.129398. URL `http://doi.acm.org/10.1145/129393.129398`.

W. Lux, editor. *The Münster Curry Compiler*, 2007. Available at `http://danae.uni-muenster.de/~lux/curry`.

A. Peters. The Random Traversal Technique for parallel evaluation of functional programs. In *21st International Workshop on Functional and (Constraint) Logic Programming (WFLP'12)*, Nagoya, Japan, May 2012a. EasyChair. To appear in Work in Progress session.

A. Peters. ViaLOIS. Available at `http://web.cecs.pdx.edu/~amp4/vialois`, Apr. 2012b.

S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(02):127–202, 1992. doi: 10.1017/S0956796800000319.

Appendix A

BENCHMARK SOURCE CODE

The following listing are the benchmark programs used for testing ViaLOIS. For benchmarks results and descriptions see Section 4.5.

A.1 CHOICEIDS

This file is called `benchmarks/choiceIdStress.curry` in the ViaLOIS distribution.

```
number d x = if d == 0 then
                x
             else
                (number d' (x*2)) ? (number d' (x*2+1))
   where d' = d - 1

findn  v  n | v == n = success

main = let n = number 18 0 in findn n (262144 - 1)
```

A.2 PERMSORT

This file is called `benchmarks/PermSortrand.curry` in the ViaLOIS distribution.

```
insert  x  []  = [x]
insert  x (y:ys) = x:y:ys ? y : ( insert  x  ys)

perm [] = []
perm (x:xs) = insert  x (perm xs)
```

```
sorted :: [Int] -> [Int]
sorted []          = []
sorted [x]         = [x]
sorted (x:y:ys) | x <= y = x : sorted (y:ys)

psort xs = sorted (perm xs)

main = psort  [12,1,2,13,9,8,7,11,4,6,3,10,5]
```

A.3 TREE

This file is called `benchmarks/tree_insert.curry` in the ViaLOIS distribution.

```
-- Carefully selected constants to make sure we actually have
-- enough numbers in the cycle to add a new one each insertion .
m = 39916801
a = 1664525
b = 1013904223
rnd x = (a * x + b) `mod` m

data BT = Leaf | Branch Int BT BT
 insert x Leaf = Branch x Leaf Leaf
 insert x (Branch y l r) | x < y = Branch y (insert x l) r
                         | y < x = Branch y l ( insert x r)
                         | x == y = Branch y l r

count Leaf = 0
count (Branch _ l r) = 1 + count l + count r

 tree_loop n x t = if n==0 then t
    else tree_loop (n-1) (rnd x) ( insert (x`mod`200000) t)

 iterations  = 200000
someseed = 24642

-- count so it does not print a big tree
```

```
main = count (tree_loop  iterations  someseed Leaf)
```

A.4 SHARING

This file is called `benchmarks/nondet_sharing.curry` in the ViaLOIS distribution.

```
-- Carefully selected constants to make sure we actually have
-- enough numbers in the cycle to add a new one each time.
m = 39916801
a = 1664525
b = 1013904223
rnd x = (a * x + b) `mod` m

data BT = Leaf | Branch Int BT BT
 insert  x Leaf = Branch x Leaf Leaf
 insert  x (Branch y l  r) | x < y = Branch y (insert  x l)  r
                           | y < x = Branch y l ( insert  x r)
                           | x == y = Branch y l r

count Leaf = 0
count (Branch _ l  r) = 1 + count l + count r

 tree_loop  n x t = if n==0 then t
    else tree_loop  (n-1) (rnd x) ( insert  (x`mod`10000) t)

 iterations  = 15000
someseed = 24642

h i = count (tree_loop  iterations  (someseed+i) Leaf)

g = head []

f i = (h i)

 linsert  x [] = [x]
 linsert  x (y:ys) = x:y:ys ? y : ( linsert  x ys)
```

```
perm [] = []
perm (x:xs) = linsert x (perm xs)

sorted :: [Int] -> [Int]
sorted []        = []
sorted [x]       = [x]
sorted (x:y:ys) | x <= y = x : sorted (y:ys)

psort xs = sorted (perm xs)

main = psort [f 1, f 2, f 3, f 4, f 5]
```

A.5 HALF

This file is called `benchmarks/hanus/half.curry` in the ViaLOIS distribution. The code below is for Halfx2, but Halfx5 is identical except for the replacement of x+x with x+x+x+x+x.

```
-- Examples for duplicating non-deterministic computations caused
-- by free variables:

data Peano = O | S Peano

toPeano :: Int -> Peano
toPeano n = if n==0 then O else S (toPeano (n-1))

fromPeano :: Peano -> Int
fromPeano O = 0
fromPeano (S x) = fromPeano x + 1

equal :: Peano -> Peano -> Bool
equal O O = True
equal (S p) (S q) = equal p q
equal (S _) O = False
equal O (S _) = False
```

```
add :: Peano -> Peano -> Peano
add O     p = p
add (S p) q = S (add p q)

half y | equal (add x x) (toPeano y) = fromPeano x where x free

main = let x = half 1500 in x+x
```

Appendix B

VIALOIS SOURCE CODE

The ViaLOIS source code is too large to include as a listing. It is available online at `http://web.cecs.pdx.edu/~amp4/vialois`. This website contains the complete source code, documentation on how to build and test ViaLOIS, and an example of the OCaml code that is generated by ViaLOIS.