5-16-2023

# Implementing a Functional Logic Programming Language via the Fair Scheme

Andrew Michael Jost
*Portland State University*

# Implementing a Functional Logic Programming Language via the Fair Scheme

by

Andrew Michael Jost

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

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Sergio Antoy, Chair
Steven A. Bleiler
Mark P. Jones
Tim Sheard

Portland State University
2023

Abstract

This document presents a new compiler for the Functional Logic programming language Curry based on a novel pull-tabbing evaluation strategy called the Fair Scheme. A simple version of the Fair Scheme is proven sound, complete, and optimal. An elaborated version is also developed, which supports narrowing computations and other features of Curry, such as constraint programming, equational constraints, and set functions.

The Fair Scheme is used to develop a new Curry system called Sprite, a high-quality, performant implementation whose aims are to promote practical uses of Curry and to serve as a laboratory for further research. An important aspect of Sprite is its integration with the popular imperative language Python. This combination allows one to write hybrid programs in which the programmer may move between declarative and non-declarative styles with relative ease. Benchmarking data show Sprite to be more complete than other Curry systems and competitive in terms of execution time, particularly for non-deterministic programs.

*Dedicated to A. and H. Always do your best.*

Acknowledgements

My deepest gratitude goes out to Sergio Antoy for patiently guiding me through the subject at hand, helping to develop the Fair Scheme and ICurry, and much more. Without his persistence and passion, and without the drive to share his considerable knowledge, this work would not have been possible. Special thanks go to Michael Hanus, whose seminal contributions to functional logic programming would be difficult to overstate. His conscientious efforts to develop and thoroughly document Curry over many years, plus his countless useful suggestions, bug fixes, and other improvements (especially with respect to ICurry) contributed significantly to this work. Thanks also go to the many people who developed and supported Curry, PAKCS and KiCS2, but particularly Bernd Braßel, whose dissertation greatly shaped my understanding of the issues surrounding Curry implementation. This work was supported by NSF grant #1317249.

**Table of Contents**

## List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

If the story of computer programming has a theme, it is that greater abstraction can lead to greater productivity[1]. Behind nearly every practical improvement to programming in the modern era lies a novel way to hide previously exposed complexity. In progressing from punchcards to assembly languages to high-level and scripting languages, for example, increasingly abstract language features have freed many programmers from troubling themselves much over the particulars of hardware architecture — or at least to not do so very often nowadays. Subroutines, functions, modules, packages, and package managers have similarly allowed programmers to structure code in increasingly abstract ways that promote reuse. To a large degree, advances such as these have led to the ability to do more, better and more reliably,

[1]This is by no means restricted to computer programming. To give one of endless examples, the greater efficiency of containerized shipping over earlier breakbulk methods is owed to the shipping container, a modular abstraction between packers and shippers. Its advent led to a sharp fall in worldwide transportation costs and brought about sweeping global economic changes [1].

with less code and less effort. In these and countless other cases, an advancement stems from a new hierarchical arrangement of components that conceals previously exposed details. This often takes the form of an interface that keeps separate concerns separated. The past eight decades have accordingly seen dramatic improvements in language design and programmer productivity to go along with dramatic increases in hardware capability.

The details hidden in this way are in many cases procedural. Languages following the popular *imperative* paradigm require programmers to provide a sequence of steps, called *instructions*, that when carried out cause a desired change in the program state. If a series of instructions appears multiple times, then a procedural abstraction called a function can be used to encapsulate it, thereby avoiding the need to spell it out repeatedly.

In contrast to this, languages belonging to the *declarative* paradigm hide some or all sequential aspects of programs[2]. Whereas the imperative approach is concerned with *how* to compute something, the declarative one is focused on *what* to compute. Using declarative principles, the *description* of a solution to some problem suffices to generate a program that solves it. By eliminating the need to supply step-by-step instructions, a declarative approach allows programmers to say less, which can make complex ideas easier to express compactly. Taken to its limit, this approach

[2]See [2] for a survey of declarative languages.

represents an abstraction with the remarkable ability to eliminate algorithms from computer programs! These still play a role behind the scenes, of course, but the point is they are hidden away so that a programmer can ignore them if he so chooses.

The most feature-rich, flexible subset of declarative languages is the *functional logic* [3] category, which comprises languages combining aspects of the two most important declarative programming paradigms: functional programming and logic programming. This combination can reduce development time, facilitate proofs of useful program properties [4, 5], provide relatively simple solutions to complex programs, and bring implementations closer to specifications [6].

One might favor a functional-logic programming language when the properties of a program are easy to describe, an algorithm is unknown or difficult to obtain, and correctness is of the utmost importance; or when one simply wishes to prioritize development effort over operational efficiency. Consider an October 2021 incident in which the U.S. company Facebook suffered a roughly five-hour global outage. According to a note released by the Facebook engineering team [7], this outage was caused when all of Facebook's data centers were inadvertently disconnected from the internet by an erroneous maintenance command. The note explains that "[Facebook] systems are designed to audit commands like these to prevent mistakes like this, but a bug in that audit tool prevented it from properly stopping the command." It is not difficult to imagine this being a case where:

1. the desired behavior is easily described in terms of properties;

2. operational efficiency is not a major concern; and

3. correctness is essential.

The suitability of a functional-logic solution to this problem can be explained in correspondence with each of these points. First, the appeal of programming in terms of properties — e.g., that no command should disconnect all data centers — is that it is maximally direct, and so involves less work and fewer superfluous details. Rather than contrive a sequence of steps whose overall effect on the program state has the required properties, a programmer expresses the properties directly and relies on a compiler algorithm to derive suitable instructions. Second, when weighed against the possibility of a global business disruption, whether the audit tool requires an instant, a second, or a minute to complete is likely of little consequence. The compiler, therefore, is not required to emit the most efficient possible sequence of instructions, which, generally speaking, would be next to impossible. Were efficiency paramount and the development budget unlimited, a different approach would likely deliver a better result, but such extremes are rare in practice. Third, by eliminating inessential program elements, each of which represents a potential error, a functional-logic implementation is more likely to be correct. Especially if the compiler algorithm can be proven correct, then a functional-logic approach can not only save development time, but eliminate a significant risk.

But the benefits of functional logic programming come at a cost, and these languages raise two significant practical issues. First, they greatly complicate compiler implementation. Since the source code of a functional logic program does not specify a sequence of instructions, one must be derived in order for the program to be run on sequential hardware. This becomes a responsibility of the compiler, and the way it is done is crucial. The target program, for instance, ought to always produce an answer if one exists and should only produce correct answers. Also, since the language implementation is entrusted with all operational aspects of a program, it must strive to be as efficient as possible. This is true even if operational efficiency is generally less important for functional logic applications, since a more efficient approach will always apply to more situations. How to accomplish all this is no small question.

Second, the functional-logic approach is niche, and, being so, struggles to attract a critical mass of users. While these languages deliver considerable benefits in certain cases, they are special tools befitting special circumstances, and as such are not especially well suited to everyday programming. Besides, being relatively new and experimental, functional-logic languages do not currently enjoy as much popular support as many other languages. In practical terms, this means that fewer libraries and introductory materials are available, the user community is smaller and less active, and programming techniques specific to this domain are less known. Given all this, we must evaluate critically what might motivate a prospective user to choose one of these languages.

This dissertation takes aim at both issues in the context of the functional-logic programming language Curry [8, 9]. It advances the state of the art in two ways. First, it develops a new compilation strategy called the Fair Scheme, for compiling Curry programs. The Fair Scheme can be used to implement a compiler that generates sound, complete, and optimal programs (according to concrete definitions of those terms). Second, it seeks to promote practical uses of Curry by providing a highly-performant, well-tested implementation based on sound engineering principles that is integrated into a more popular programming environment. This implementation, called Sprite, aims to overcome the most significant hurdles one would otherwise face when making practical use of Curry.

Oftentimes the best approach to a programming problem is to choose a library containing a pre-coded, pre-tested solution. From this point of view, the most important characteristic of a language might well be its popularity. This no doubt helps explain the persistence of certain languages that remain popular, despite years of competition from newer alternatives, that in some cases seem to offer compelling advantages. As a result, many programmers may rightly eschew undertaking a project in a language like Curry, since it could amount to a great deal of reinvention. But this presents a chicken-and-egg problem. A language may be unpopular in part due to its lack of comprehensive libraries, yet to justify the development of new libraries a language should be popular. In our estimation, the best way to confront this problem is to sidestep it. Rather than create new libraries in Curry, researchers should

focus their efforts on facilitating the use of existing libraries together with Curry. This can be achieved by integrating Curry into another, more popular programming environment.

To that end, an important contribution of this dissertation is the integration of Curry with Python [10]. Python is an extremely popular imperative programming language backed by an active community supporting an enormous number of projects. At the time of this writing, the official third-party Python library repository, PyPI [11], contains over 400,000 projects. These include stable libraries for machine learning [12–14], numerical and scientific computing [15–17], data analysis [18, 19], computer vision [20], graph analysis [21], HTTP servicing [22–24], micro-service creation [25], web scraping [26], and parser generation [27] — and these merely scratch the surface of what is available. The combination of Curry and Python allows one to write hybrid programs that encapsulate Curry functions and modules, and compartmentalize functional-logic computations apart from imperative ones. The result is an environment in which programmers are free to move between the imperative and functional-logic styles with relative ease. This not only makes Curry available to a very large group of programmers, it allows each one to take on as little or as much as he chooses and so migrate at his own pace. The philosophy behind this complementary approach places programmers first, based on the understanding that they know their problems best and should ultimately decide which programming style is best suited to any specific task.

To illustrate how this could promote practical uses of Curry, consider a program that converts images of handwritten or printed documents into a textual electronic format. This program must solve an *optical character recognition* (OCR) problem to extract text from images. This is far from trivial, so one might hope to rely on an existing software package. Fortunately, many OCR packages are freely available, though none is perfect. For example, when converting a certain image, the open-source OCR tool *Tesseract* [28] produces the following output:

```
Col la bo rates and Ensures Accounta bility
```

To make use of this one should remove the spurious whitespace. A first attempt at a general rule to do so might be the following:

*Rule: if removing spaces between a sequence of words containing an invalid word would produce a valid word, remove those spaces.*

According to this, the character sequence "`Accounta bility`" would be transformed into the word *accountability*. Provided a predicate to distinguish valid words from invalid ones, a programmer using the logic paradigm could encode this rule compactly with little effort. An equivalent imperative implementation could be longer, more difficult to understand, and more likely to contain errors. In addition, the logic approach scales better as the number of rules increases, so that a logic program that ensures many rules are satisfied simultaneously is more likely be manageable.

Tesseract interfaces with a variety of programming languages, including Python [29], but does not interoperate directly with Curry or, as far as we are aware, any logic programming language. Given that, an appealing implementation strategy could be to write a hybrid program in which one section uses Tesseract through an imperative interface to provide the OCR capability and a second section transforms the raw text created in this way via rules written in a logic (or functional-logic) style. With Sprite, one can do exactly that. A Python script can be written that first extracts text from images using the Tesseract Python interface and then transforms that text using Curry to ensure it meets certain criteria. The Curry code can be embedded directly in the Python script or loaded from a separate file via Python's native module system. We believe a hybrid capability such as this greatly reduces entry barriers for prospective Curry users and encourages a potentially large number of programmers to consider a functional-logic approach when they otherwise might not.

To set the stage for this work, the remainder of this chapter and the next chapter in its entirety present a brief synopsis of the history and principles of functional logic programming followed by an introduction to Curry.

## 1.2 Functional Logic Programming

During the 1980s and early 1990s, growing interest in combining the most appealing aspects of functional programming and logic programming led to the emergence of functional logic programming (see [2] for a survey). This effort was driven by a desire

within the logic programming community to improve the efficiency of logic languages, which typically rank behind functional languages in this regard due in part to their greater reliance on non-deterministic operational principles. Given the deterministic nature of most real-world programs, an implementation based on such principles often exhibits inferior performance by such measures as memory consumption and execution time. Researchers at the time perceived that a new language combining functional and logic principles could combine the benefits of logic programming with the efficiency of functional programming by relying on deterministic operational principles to evaluate the deterministic parts of a program.

Early work focused on introducing (deterministic) functions into the logic domain, and this resulted in an impressive proliferation of proposed functional logic languages, operational principles, and implementations (see [2], in particular Tables 1 and 2). Curry was introduced in a 1995 paper by Michael Hanus, Herbert Kuchen, and Juan José Moreno-Navarro [8], whose goal was to define a standard functional logic language that would help consolidate these efforts.

Seeing that functional logic programming is a combination of two deep-rooted paradigms, it is helpful to begin by discussing functional programming and logic programming separately. Historically speaking, functional programming is an evolution of combinatory logic, first investigated by Moses Schönfinkel in the 1920s, and lambda calculus, developed by Alonzo Church in the 1930s. Both theories represent early attempts to place computer programming on a rigorous foundation by expressing

computations in terms of mathematical logic. Despite their independent development and seemingly unrelated foundations, these theories are surprisingly similar in their explanatory power; so much so that they are, in fact, equivalent under a simple rule called extensionality [30]. Combinatory logic emerged from David Hilbert's ultimately unsuccessful effort to reformulate all of mathematics as a consistent logical system based on a finite set of axioms and inference rules. The American mathematician and logician Haskell Curry[3] continued its development through much of his career, and by 1947 had described one of the first high-level programming languages, a functional language based on combinatory logic. Lambda calculus is a formal system that describes computations in terms of function abstraction, function application, variable binding, and variable substitution.

Functional programming is based on the idea that computations can be carried out in terms of expressions and rules of evaluation rather than as sequences of instructions. Running a functional program amounts to evaluating an expression. Figure 1.1 provides a sketch of this idea, demonstrating how functional principles might be applied to sort a sequence of orderable objects

In functional languages, the construction and application of functions plays a central role. These languages feature first-class and higher-order functions, and loop by means of recursion (a definitional construct) rather than explicit control flow (a

[3]The languages Haskell and Curry featured prominently in this work are his namesakes.

$$\begin{array}{lcl} \text{sort}(d \cdot b \cdot c \cdot a) & \rightarrow & a \cdot \text{sort}(d \cdot b \cdot c) \\ & \rightarrow & a \cdot b \cdot \text{sort}(d \cdot c) \\ & \rightarrow & a \cdot b \cdot c \cdot \text{sort}(d) \\ & \rightarrow & a \cdot b \cdot c \cdot d \end{array}$$

FIGURE 1.1: Sorting a sequence using functional principles. Function *sort* identifies a minimum element, $m$, of its argument and evaluates to the concatenation of $m$ with a recursive application of itself to the sequence excluding $m$.

procedural construct). A *pure* functional style is one that permits only pure functions; that is, functions whose results are determined entirely by their arguments. Languages of this type may feature shared, immutable data, which can greatly simplify program analysis. In addition, being free of side effects and global state, they operate by a principle of *lazy* evaluation in which expressions are only evaluated when and if necessary. Such languages may be implemented in terms of symbolic manipulation, often through a formal mapping to a well-defined abstract machine, and very efficient implementations are known; for example, the spineless tagless G-machine (STG) [31].

The first mainstream functional programming language, LISP, was developed by John McCarthy in 1958 and first implemented shortly thereafter by Steve Russell [32]. LISP exemplifies the use of symbolic manipulation as a means of effecting computation. It is a homoiconic language, meaning the representation of a LISP program is itself data that can be manipulated by a LISP program. More generally, the ability of a language to process itself, sometimes referred to as metacircularity, is relatively common among functional languages. LISP has inspired a family of modern

languages including Scheme, ML and its derivatives Caml and OCaml, Erlang, Clean, and Haskell. Haskell, in particular, is relevant to this work because it provides a basic syntax that Curry extends only slightly.

Haskell originated at the 1987 conference on Functional Programming Languages and Computer Architecture held in Portland, Oregon, where a committee was formed to define a programming language that would integrate aspects of functional programming that enjoyed a "wide consensus" in order to "reduce unnecessary diversity" and facilitate the rapid sharing of ideas [33]. The work of this committee culminated in the publication of the Haskell Report in 1990, which defined the first version of the language. Since then, Haskell has grown in popularity both in academia and industry. Haskell is a statically- and strongly-typed polymorphic language with a modular structure that features a Hindley-Milner type inference system [34, 35]. Its most popular implementation is the Glasgow Haskell Compiler (GHC, [36]), whose development, led by Simon Peyton Jones, Simon Marlow, et al., has been supported for many years by Microsoft Research in Cambridge, England.

Logic programming is based on the idea that a single formalism suffices for both logic and computation [37]. Prominent logic families include Prolog, Datalog, and AnsProlog. A logic program is a collection of logical sentences expressing a set of facts and relationships pertinent to a problem domain. These are written as definite clauses of a simple kind, often Horn clauses (named for the $20^{\text{th}}$ century logician Alfred Horn) or some extension of those. The clausal approach to logic programming

makes use of automatic rules to achieve goals. For example, Stanford researcher Cordell Green in 1981 demonstrated how an inference rule called *resolution* could be exploited to solve the Tower of Hanoi puzzle [38]. That puzzle consists of a collection of pegs onto which several flat discs of different radii are stacked. The goal is to transfer a stack from one peg to another by a series of moves that each relocate a single topmost disc to the top of the stack residing on a different peg, subject to the rule that a larger disc is never stacked atop a smaller one.

A logic program sufficient to solve this puzzle need only describe the initial configuration, the logical relationships between valid configurations (i.e., the legal moves), and the desired ending configuration. Taken together, these define a domain-specific knowledge representation called the *solution space* in which possible solutions reside.

The advantages of this approach stem largely from avoiding details not directly related to the problem at hand. For example, an imperative implementation involving explicit loops would declare loop index variables, initialize them, modify their values, and test for loop termination. Nothing analogous need appear anywhere in a logic program. By keeping with a single formalism, logic programs can more closely approximate their specification.

Executing a logic program is equivalent to searching the solution space it defines. Programs can be divided into two parts called the *logic component* and the *control component*. The logic component, which is specified in source code, defines the solution space for the problem at hand while the control component defines the search

strategy. By varying the logic, different problems can be solved and by varying the control, programs may be executed in a variety of ways. Some logic languages fully separate these components, in which case source code is simply an order-independent set of clauses whose interpretation is strictly declarative. The control in such cases may be supplied entirely by the language implementation. In other languages – particularly Prolog – clauses also have a procedural interpretation. This mixes some aspects of control into the source code, allowing programmers to more directly influence the search strategy.

Search is a disjunctive process. Each choice corresponds to a branch in the search space[4]; any, some, or all of whose alternatives might lead to solutions. In this work, we shall only consider sequential search strategies. Consider a sequential procedure to satisfy the following Boolean proposition:

$$(a \vee b) \wedge \neg(a \wedge b) \tag{1.1}$$

There exist two solutions, $\{a \mapsto False, b \mapsto True\}$ and $\{a \mapsto True, b \mapsto False\}$. Being sequential, the search procedure must choose a variable to consider first. Let us arbitrarily assume it selects $a$. Either or both of its possible values might lead to solutions, so the search space bifurcates at this point. One way to proceed is by

[4]In this work, every search takes place through the solution space defined by a logic program. Therefore, the terms *solution space* and *search space* are interchangeable. We use *solution space* when writing about a certain program or problem, and prefer *search space* when discussing the search mechanism.

an approach called *backtracking*, which begins by assigning a speculative value to $a$. Next, the subspace induced by this assumption is fully explored and, afterwards, the search backtracks to $a$, selects a different value, and then continues in like fashion. Each assignment of $a$ leads to another branch, corresponding to variable $b$, that again bifurcates the space and results in another speculative assignment. This process repeats until every alternative of every variable is exhausted. By repeatedly applying this procedure the entire space is eventually explored, and one ultimately finds that, of the four possible assignments, two are solutions and two are not. This is summarized in Figure 1.2.

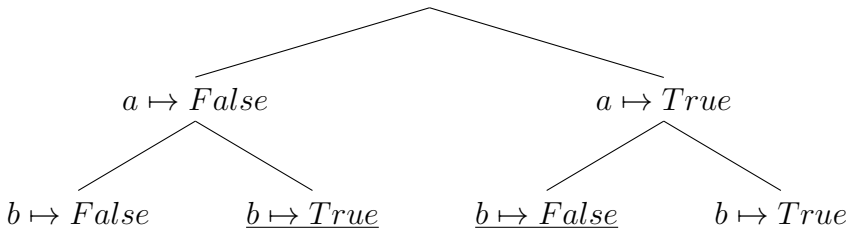



FIGURE 1.2: The search space induced by a Boolean proposition in two variables. Each branch represents the assignment of a value to one variable. A root-to-leaf traversal assigns variables in the order $a$, $b$. Distinct paths represent distinct assignments. Solutions to $(a \vee b) \wedge \neg(a \wedge b)$ are underlined. A backtracking search that considers variables in this order and assigns values in the order $False$, $True$ corresponds to a left-to-right traversal of the bottom rank.

A search space grows exponentially as the number of variables increases. For this reason it is convenient to use a more compact formulation. To this end we can

represent the space depicted in Figure 1.2 simply as $\{a, b\}$, where $a$ and $b$ are understood to be non-deterministic Boolean variables. In light of this, we may view the control component of a logic program as little more than an organized traversal of a non-deterministically-defined space. We shall see in coming chapters that a significant effort in implementing a Curry compiler involves devising a suitable search procedure. In contrast to the example above, such a procedure must consider dynamic search spaces that evolve as computations progress and that are, in general, infinitely-branching. A limitation of backtracking is the possibility that a speculative assignment to some variable, $x$, might lead to an infinite search, causing other values of $x$ to never be considered. Devising an adequate procedure in the face of this and proving it complete is a significant contribution of this work.

In the previous example, three arbitrary procedural choices arose: first, whether to consider $a$ before $b$ or vice versa and, additionally, whether to first consider $False$ or $True$ in the case of $a$ and again in the case of $b$. A tree-based diagram of the search space like that shown in Figure 1.2, along with the search procedure it implies, might be regarded as non-deterministic in the sense that it belongs to a family of closely-related ones that vary along these lines. There would seem to be no *a priori* reason to prefer one of these over another. To dispel any potential confusion, this sort of non-determinism — which relates to the search procedure itself rather than to the logic component of a logic program — is irrelevant to this work. We are concerned only with the non-determinism inherent to solution spaces of logic-like

programs. Figure 1.2 might give the impression that the details mentioned above have significance because, for example, the decision to consider $a$ before $b$ is built into the graphical depiction of the search space. This is merely an artifact of the representation. Another advantage of the non-deterministic formulation is that it avoids making these arbitrary choices, and so also avoids suggesting that there is any significance to them.

Programs may contain explicit and implicit representations of non-determinism. An important explicit representation is provided by *logic variables*[5]. These are variables, in the mathematical rather than imperative computer programming sense, that act as placeholders for unknown information. Truly, in a pure language such as Curry one cannot meaningfully have imperative variables, since they would represent references to mutable locations.

Logic variables represent one of the most appealing aspects of logic programming, since they provide the ability to compute with partial information. One may use them whenever something is unknown, difficult to determine, or not worth the effort to specify. This allows judicious programmers to trade execution time for development time. Prop. 1.1 makes for a very simple demonstration. It can be solved by the following Curry program:

```
main = (a || b) & (a && b) where a,b free
```

[5]More simply called *variables* when the context is clear.

Here, the infix operators `||` and `&&` are the ordinary Boolean connectives OR and AND, respectively, `&` is a constraint-satisfaction operator that restricts the program to solutions that satisfy both its left- and right-hand sides, and the *where* ... *free* clause indicates that $a$ and $b$ should be treated as logic variables. This program finds variable bindings that satisfy the given proposition without the programmer needing to specify a search procedure. The benefits are rather limited in this example, since this proposition is so easy to solve by hand, but in general they can be significant. For instance, consider the task of creating a small application that accepts an arbitrary Boolean expression as input and reports variable bindings that satisfy it.

The act of assigning concrete values to variables is called *instantiation*. How this is done in the course of program execution is a responsibility of the control component. Logic programming languages that rely on the resolution principle as originally proposed [39] suffer from the need to fully instantiate all variables. This can lead to unnecessary work being performed. An important refinement is the principle of *unification*, which allows clauses to be instantiated only insofar as they are needed. An advantage of this can be seen in satisfying the basic logical disjunctive proposition in two variables. As shown in Figure 1.3, a partial instantiation of variables can allow for more compact representations.

This will play an important role in the development of the Fair Scheme, developed in Chapter 3, which fully instantiates variables, and its extension, developed in

| *Without Unification* | *With Unification* |
|---|---|
| $\{a \mapsto True, b \mapsto False\}$ | $\{a \mapsto True\}$ |
| $\{a \mapsto True, b \mapsto True\}$ | $\{b \mapsto True\}$ |
| $\{a \mapsto False, b \mapsto True\}$ | |

FIGURE 1.3: Solutions to $a \vee b$ with and without unification. With unification, variables are instantiated only as far as is necessary to satisfy the proposition.

Chapter 4, which does so partially.

## 1.3 Organization of this Work

This dissertation presents the Fair Scheme (FS), a compilation strategy for functional-logic programming languages, and uses it as the basis of a new Curry system called Sprite.

Chapter 2 provides a suitable introduction to the Curry programming language, including its syntax (Sect. 2.1), semantics (2.2), and certain more advanced topics (Sect. 2.3).

Chapter 3 presents the initial version of the FS. The strong theoretical underpinnings allow us to provide formal definitions and rigorously prove three important properties of the FS: its soundness, completeness, and optimality.

Chapter 4 describes three extensions of the FS. Curry supports free variables, which are not bound to any specific value. Our initial approach eliminates these through a compile-time replacement rule and performs computations via rewriting

and pull-tabbing. As a result, the FS never produces any values that contain free variables. In Sect. 4.1, we introduce the first extension (FS-`x`), which provides an explicit representation for free variables and computes by the principle of narrowing rather than rewriting. This allows it to complete certain computations in fewer steps and more compactly represent certain sets of values. In Sect. 4.2 we add a mechanism for applying equational constraints to free variables with the second extension (FS-$\beta$). Finally, in Sect. 4.3, we introduce the third extension (FS-$\mathcal{S}$), which allows for evaluating set functions, an important feature of Curry.

Chapter 5 describes the full-featured Curry system, Sprite, enabled by these extensions. We discuss the compilation process used to transform Curry source code into an executable form through standard intermediate representations. Additionally, we present the data structures and algorithms used to create an efficient implementation of Curry. Our implementation deviates from the FS in certain ways to optimize performance, and we explain these deviations in detail.

Chapter 6 explores the use of Sprite. Sect. 6.1 presents the Sprite Python interface, while Sect. 6.2 presents an example application demonstrating how Sprite can help integrate Curry into an imperative programming environment.

Chapter 7 presents benchmarking results comparing the performance of Sprite with other popular Curry systems. Finally, Chapter 8 contains concluding remarks, including suggestions for future research to build on our work.

## Chapter 2

## The Curry Programming Language

The combination of functional and logic programming possesses advantages over either paradigm alone. From the point of view of a functional language, the introduction of logic features increases expressive power. The introduction of non-determinism can simplify programs, enable computations with partial data, and increase abstraction by hiding sequential aspects of programs. From the point of view of a logic language, deterministic functions improve operational efficiency, since deterministic parts of a computation can use more efficient principles.

The approach to integrating these paradigms has likewise been considered from both vantage points [2, Sect. 1], leading to two broad possibilities:

1. Logic aspects can be integrated into a functional language by a) admitting non-deterministic semantics into the language and b) replacing pattern matching with a more powerful principle, such as unification [40]; and

2. Functional aspects can be integrated into a logic language by augmenting the

resolution principle (or another logic principle) with a mechanism for function evaluation.

Curry embodies the first approach. It extends the pure functional language Haskell with non-deterministic constructs including logic variables, overlapping rules [41], functional patterns [42], and an explicit non-deterministic choice operator [43]. In this chapter, we introduce Curry in some detail. Because these languages are so similar, a working familiarity with Haskell is recommended.

Curry is a statically typed, lazy language with type inference. Expressions have a well-defined type known at compile time and such information can be used to find programming errors and perform optimizations. The inference system relieves programmers from having to explicitly specify types in many cases, though it is customary to provide type annotations for top-level functions and we shall follow this convention where it clarifies matters.

Curry is packed with high-level features, many of which are inherited from Haskell. It employs a modular structure that allows code to be separately compiled and linked. It sports a sophisticated type system complete with ad-hoc polymorphism, function overloading, type classes, and monadic input-output (I/O). It includes syntactic sugar for defining numeric ranges, constructing lists through comprehensions, and defining sequences of I/O actions. Fortunately, many of these features can be implemented in the compiler front-end. Type-incorrect programs, for example, are rejected before

the main part of the compiler is involved. Likewise, complex artifacts of the type system such as type classes are transformed into simpler ones before arriving at a representation the Fair Scheme need consider. Because of this we will focus on the fundamental features of Curry, safe in the knowledge that this does not lead to a practical limitation. Interested readers should consult the official Curry documentation for complete information about its features [9].

## 2.1 Syntax

We adhere to certain conventions in the discussion that follows. An identifier is a sequence of characters taken from the set of alphanumeric characters plus the underscore character (`_`) and not beginning with a digit. The identifier consisting of only a single underscore represents an unused, distinct, anonymous variable. Types and constructors begin with capital letters whereas functions and variables begin with lowercase ones. Constructors and functions are called *symbols*. An exception to the usual rules is that names are allowed to end with trailing single ticks. For instance, `f`, `f'`, and `f''` are valid, distinct function names. By convention, we shall use this to distinguish successive versions of a named entity, when needed. A second exception relates to the naming of operators: any function whose name consists entirely of non-alphanumeric characters, such as `++`, is an operator. The differences between operators and other functions are entirely syntactic.

### 2.1.1 Data Type Definitions

Data types are introduced with the `data` keyword. The simplest type can be defined as follows:

```
data Unit = Unit
```

This introduces a new type called `Unit` (the left-hand side) of which there is precisely one value called `Unit` (the right-hand side). The right-hand side symbol is called a *constructor* of this type. It is common to overload type and constructor names in this way, since they inhabit separate namespaces.

More complex types are composed through algebraic construction rules. A *sum type* represents a choice between alternatives. A sum type representing a Boolean value can be defined as follows:

```
data Bool = False | True
```

The vertical bar (`|`) signifies a disjunction, meaning that a value of type `Bool` is always exactly one of these constructors; i.e., either `False` or `True`. A *product type* represents a simultaneous combination. A pair of Booleans could be declared as the following product type:

```
data BoolPair = BoolPair Bool Bool
```

`BoolPair` comprises two independent Boolean values, so there are in total four values inhabiting this type. The above definition is inextensible: whenever a distinct pair

of types is needed, a separate data declaration similar to the one shown above would need to be written. Not only would that be repetitive, it would pollute namespaces and wreck interfaces, since it raises the question of who should be responsible for defining common types. To avoid these problems, Curry provides parametric type definitions. All possible pair types can be declared with the following definition:

```
data Pair a b = Pair a b
```

The type variables `a` and `b` represent the two types constituting any pair. This is an abstract definition that requires us to supply arguments for these variables in order to build a concrete type. That is done by simple juxtaposition: e.g., a pair of Booleans can be written `Pair Bool Bool`. To avoid the need to repeatedly type such things out, Curry gives programmers a way to define type aliases. The `type` keyword can be used as follows to create an alternate definition of `BoolPair`:

```
type BoolPair' = Pair Bool Bool
```

Sum and product types can be composed to form more elaborate types. We see this in the following definition of a singly-linked list:

```
data List a = Nil | Cons a (List a)
```

This says that a list is either empty (`Nil`) or the construction (`Cons`) of an element called the *head* and a list, called the *tail*, containing zero or more remaining elements,

where the element type is represented by the type variable, `a`. The recurrence of the list type, `List a`, indicates that the tail is a list whose elements have the same type as the head.

### 2.1.2 Built-in Data Types

Curry provides a few fundamental data types that cannot easily be defined using data type declarations. Within an algebraic type system, natural numbers can be defined following the Peano discipline. The nonnegative integers, for instance, can be defined in this way:

```
data Peano = Zero | Succ Peano
```

In this system, the first three numbers counting from zero can be written `Zero`, `Succ Zero`, and `Succ (Succ Zero)`, respectively. Although this formulation has practical uses, it can be inefficient because the costs in time and memory of representing a number are linear with respect to its magnitude. In addition, many arithmetic functions are more complex than necessary with this representation.

Unfortunately, no practical constant-space definition of numbers can be created using an algebraic declaration. The reason becomes clear if we attempt to write one out:

```
data Integer = 0 | 1 | 2 | ...
```

It is obvious that one cannot simply list all numbers in this way as separate constructors. Due to this, and also to make use of the efficient numeric representations native to hardware, Curry defines three *fundamental* data types: `Int` for integers, `Char` for characters, and `Float` for floating-point numbers. Each of these is equivalent to a large type sum, like the one depicted above, but is more efficiently implemented by the compiler and runtime system.

A string of digits such as `0`, `1`, or `42` is interpreted as an `Int`. Two sequences of digits separated by a decimal point, such as `3.1415`, specifies a `Float`. A `Char` is written as a character surrounded by single quotes, as in `'a'`. Characters can also be specified using an escape sequence consisting of a backslash (\) followed by a numeric or symbolic code. Using this, the null character (with numeric value zero) can be written `'\0'`, a newline can be written `'\n'`, and a backslash itself can be written `'\\'`.

Lists, tuples, and strings are afforded special syntax. A list type is written by enclosing its element type in square brackets. The type of a list of Booleans is thus written `[Bool]`. A list of values can be written as a comma-separated sequence of values enclosed in square brackets, such as `[]`, `[True,False]`, or `[1,2,3]`. A list may also be constructed as `a:b` for head `a` and tail `b`. Using this method, two of the above lists can be written `True:False:[]` and `1:2:3:[]`.

The zero-tuple, called *unit*, is written `()`. This is both its type name and its data

constructor. The type of a tuple with arity two or greater is written as a comma-separated sequence of types. For example, `(Bool,Int,Char)` is a triple containing a Boolean, integer, and character in that order. A tuple value is written as a comma-separated sequence of values enclosed within parentheses, as in `(True,False)`. Curry does not provide a one-tuple.

The string type is an alias for a list of characters; i.e.: `type String = [Char]`. Double-quoted strings are a quick notation for lists of characters. Thus, the string `"Curry"` and the list `['C', 'u', 'r', 'r', 'y']` are identical.

### 2.1.3 Function Definitions

Functions are defined by rules, each of which consists of a left-hand side (LHS) *pattern* and a right-hand side (RHS) *replacement* separated by the equal sign (`=`). A very simple function, which inverts a Boolean value, can be written as follows:

```
1 not :: Bool
2 not False = True
3 not True  = False
```

This introduces `not` as a function symbol and defines it by two rules. The first line declares the function and specifies its type. Since this information can usually be deduced by a compiler, we shall omit function declarations in most cases. The first rule matches when the argument is `False` and defines the replacement to be `True`. The second rule is complementary. The meaning of a function can in general be

determined by comparing its patterns against an expression to be evaluated and, if a match occurs, constructing the specified replacement and putting it in place of the original expression. We call the expression being evaluated the *candidate*. By this process, the expression `not False` is replaced with `True` according to the first rule. This is called a *rewrite step*, which we represent by writing `not False` $\rightarrow$ `True`. When a rule is used to perform a rewrite step, we say the rule is *fired*. We write $\rightarrow^*$ to denote a sequence of any number of rewrite steps.

A slightly more complex example further illustrates. The following function can be used to append two lists together:

```
1 append []     ys = ys
2 append (x:xs) ys = x : append xs ys
```

This takes two formal parameters, which we shall refer to as the first and second lists. According to the first rule, if the first list is empty, then the replacement is the second list. Otherwise, by the second rule, the head of the first list becomes the head of the replacement and the second argument is appended to the remaining elements to form its tail. In this case, pattern matching involves finding a correspondence between variables in the pattern and subexpressions in the candidate. We can define a match completely by the rule selected and a mapping from variables to subexpressions, called a *substitution*. To illustrate this, the steps taken to append `[3]` to `[1, 2]` are shown below:

| | Rule | Substitution |
|---|---|---|
| `append (1:2:[]) (3:[])` | $2^{\text{nd}}$ | `x` $\mapsto$ `1`, `xs` $\mapsto$ `2:[]`, `ys` $\mapsto$ `3:[]` |
| $\rightarrow$ `1 : append (2:[]) (3:[])` | $2^{\text{nd}}$ | `x` $\mapsto$ `2`, `xs` $\mapsto$ `[]`, `ys` $\mapsto$ `3:[]` |
| $\rightarrow$ `1 : 2 : append [] (3:[])` | $1^{\text{st}}$ | `ys` $\mapsto$ `3:[]` |
| $\rightarrow$ `1 : 2 : 3 : []` | | |

A function whose name consists entirely of non-alphanumeric symbols is called an operator. Curry supplies an operator whose behavior is identical to `append`. This operator, denoted `++`, can be defined as follows:

```
1  (++) []     ys = ys
2  (++) (x:xs) ys = x : xs ++ ys
```

Operators come in various fixities; e.g., the minus operator (`-`) may appear in certain languages in prefix or infix form, as in `-1` and `1 - 1`, respectively. There is also precedence between operators. Curry affords programmers control over these details but we shall not discuss them further, instead relying on the reader's intuition to discern the intended meaning.

A constructor expression contains only constructors and variables and a linear expression is one without repeated variables. Except for their leading function symbol, patterns in Haskell are linear constructor expressions, but Curry relaxes these constraints.

First, patterns in Curry can involve function applications. Such *functional patterns* [42] allow for direct representations of complex structures, which make possible high-level descriptions of queries and transformations that can lead to extremely concise

and obviously correct code. An example of this can be seen in the following definition of a function to determine the last element of a list:

```
last (_++[x]) = x
```

This function accepts one parameter, which is written as an expression that describes the construction of a list. This pattern matches any list with a last element $e$ and, when firing the rule, binds $e$ to the variable `x`. Compare this to the following definition without a functional pattern:

```
1 last' [x]    = x
2 last' (_:xs) = last' xs
```

The version involving a functional pattern is more concise and is arguably easier to understand.

Second, Curry allows repeated variables, which express logical equality between parameters. Pattern matching requires a substitution that maps each variable to exactly one expression, so repeated variables place constraints on the possible matches. We can see this at play in the following function:

```
find k (_++(k,v)++_) = v
```

This function has two parameters: a key, `k`, and a list of pairs. The repetition of `k` makes it so that this rule only matches a pair whose first element is `k`. The replacement is the second element of that pair. Accordingly, the following expression evaluates to `"banana"`:

```
find 'b' [('a',"apple"), ('b',"banana"), ('c','carrot')]
```

We will return to the question of what happens if the list contains multiple matching elements in Sect. 2.3.4.

Functions can be defined in parts with conditional rules. A *guard* is an optional predicate following the pattern and separated by the pipe character (`|`). A rule defined in this way is conditional on the predicate being satisfied. For instance, a function to determine the maximum of two integers can be defined as follows:

```
1 max a b | a < b     = b
2         | otherwise = a
```

The first part is used only when the predicate is satisfied; otherwise, the second part is used. The predicates are evaluated in order until a match (if any) is found. The final part may use the keyword `otherwise` to indicate it as the default. The comparison operators `<`, `<=`, `>`, `>=`, `==`, `/=` as well as the Boolean connectives `&&` (logical AND) and `||` (logical OR) are useful in defining predicates and carry their expected meanings.

It is sometimes useful to share bindings between parts of a function. Consider the following function that evaluates to the positive numbers of a list:

```
1 positives [] = []
2 positives (a:as) | 0 < a     = a : positives as
3                  | otherwise =     positives as
```

The repetition seen in the list tail can be eliminated by use of a *where* clause, which is a syntactic construct that introduces a shared binding into all parts of a rule. For example:

```
1 positives [] = []
2 positives (a:as) | 0 < a     = a : tail
3                  | otherwise =     tail
4     where tail = positives as
```

A *where* clause can also introduce auxiliary functions. Consider the following implementation of a function to evaluate polynomials by Horner's rule:

```
1 horner []     _ tot = tot
2 horner (a:as) x tot = horner as x (x * tot + a)
```

The first two parameters are a list of coefficients and the independent variable. The third parameter, `tot`, holds a running total. To evaluate a polynomial correctly, an initial total of zero must be supplied. For example, to evaluate the polynomial $3x^2 + 2x + 1$ at $x = 5$, one writes `horner [3,2,1] 5 0`. The need to supply the initial total is inconvenient. An improved definition that eliminates it through the use of a *where* clause is shown here:

```
1 horner' coeffs x = aux coeffs 0
2     where aux [] tot     = tot
3           aux (a:as) tot = aux as (x * tot + a)
```

In this version, the main computation involving the running total has been moved into an auxiliary function so as to eliminate the unnecessary parameter.

A third use of *where* is to declare free variables. This is done through a clause of the form "**where** *varname* **free**." The following function, for instance, represents an unknown value by evaluating to a free variable:

```
unknown = x where x free
```

A type annotation is sometimes required as a hint to the compiler. To specify the variable type, one writes, e.g., `(x::Bool) where x free`. In general, two colons (`::`) can be used in this way to supply a type annotation for any expression. Free variables are discussed in more detail in Sect. 2.3.1.

### 2.1.4 Higher Order Functions

A higher order function is one that accepts or produces a function. This treatment of functions as first-class objects is a hallmark of functional programming. A commonly used higher-order function known as `map` can be defined as follows:

```
1 map _ []     = []
2 map f (x:xs) = f x : map f xs
```

This applies a function, `f`, to each element of a list, producing a transformed list. We could use this to invert the elements of a Boolean list by mapping the `not` function over it, as in `map not [True, False]` $\rightarrow^*$ `[False, True]`. To write a higher-order function that produces a function we can specify a local function as the replacement, as the following example shows:

```
incBy n = f where f x = x + n
```

This returns a function to increment a number by the specified amount. For example, `incBy 1` is a function that increments by one, so we have `(incBy 1) 3` $\rightarrow^*$ `4`.

Another way to produce functions is through partial application, which binds certain arguments to fixed values, producing a function that takes fewer parameters. A technique called *currying*, which converts a function with several arguments into a chain of functions each taking one argument, is applied uniformly to simplify partial application. Consequently, one could write a function to evaluate the polynomial on Page 34 as `horner' [3,2,1]`. To apply this over a list of $x$-values, we write, e.g.:

```
map (horner' [3,2,1]) [0,1,2,4,8] →* [1,6,17,57,209]
```

Infix operators can be partially applied using a syntax natural to them. For example, `(+1)` produces an incrementing function by binding the right-hand argument of the binary *plus* (`+`) operator, as demonstrated in the following example:

```
map (+1) [1,2,3] →* [2,3,4]
```

This idea can also be extended to zero arguments to apply infix operators using prefix notation. We may, for example, write the following:

```
(:) True [] →* [True]
```

### 2.1.5 Expressions

We now turn our attention to the syntax for forming expressions. A *variable expression* is simply a variable. For instance, each RHS in the definition of `max` is a variable expression.

An *apply expression* consists of a leading constructor or function symbol followed by zero or more subexpressions. The leading symbol, $s$, is called the *root* and we say the expression is *rooted* by $s$ or is *$s$-rooted*. We may also refer to the symbol kind in saying that an expression is, e.g., constructor-rooted. To illustrate, the expression `append xs ys` that appears in a rule of `append` is a function-rooted expression that applies the symbol `append` to two subexpressions, the variable expressions `xs` and `ys`.

Literals such as `False` and `1` are degenerate cases in which the root symbol is nullary: that is, `False` is an application of the constructor symbol `False` to zero arguments. More complex literal expressions are (possibly nested) applications over constructor symbols with higher arity. These include the strings, lists, and tuples we have already seen.

The application of an infix operator is an apply expression. This can be seen clearly by parenthesizing the root symbol to write the expression in curried form. For instance, the RHS of the second rule of `append` could be written as follows to emphasize that its root is the list constructor:

```
(:) x (append xs ys)
```

Similarly, the key part of Horner's rule can be written in the following way to emphasize its structure:

```
(+) ((*) x tot) a
```

Bindings can be introduced by a *let expression*. These serve in the context of expressions a purpose similar to that of *where* clauses in function definitions. A function to normalize three numbers so that their sum is unity, for instance, can be written as follows:

```
1 normalize x y z = let norm = x + y + z
2                   in [x/norm, y/norm, z/norm]
```

The *let* construct here allows the normalizing constant to be defined once. This may also improve efficiency by allowing it to be computed once. Bindings introduced in this way are recursive, meaning they can refer to one another. This makes it possible to define infinite data structures. For instance, the infinite list `[True,False,True,...]` can be defined as shown here:

```
1 infList = let a = True  : b
2               b = False : a
3           in a
```

This example also demonstrates how multiple bindings can be introduced. The whitespace here is necessary to let Curry know that a continuation of the expression is intended.

Like a *where* clause, a *let* expression can be used to introduce free variables. An alternative version of `unknown` could thus be defined as follows:

```
unknown' = let x free in x
```

An *if-then-else expression* is syntactic sugar for conditional evaluation. With this we can redefine `max` more succinctly as shown here:

```
max' a b = if a < b then b else a
```

This has the same values as the following:

```
1 max'' a b = ifThenElse (a < b) b a
2     where ifThenElse cond x y | cond      = x
3                               | otherwise = y
```

The *if-then-else* syntactic construct is equivalent to the local function `ifThenElse` shown above, which conditionally evaluates to the consequent, `x`, when the condition is satisfied and to `y` otherwise.

Pattern matching can appear in an expression context through the use of a *case expression*. The grammatical construction "**`case`** *`discr`* **`of`** *`cases`*" matches a discriminator against a series of cases, where each case consists of a pattern and a replacement separated by a right arrow diglyph (`->`). Like the parts of a conditional function, the patterns are tried in textual order until the first match occurs. To demonstrate this, we can redefine `not` as follows:

```
1 not' x = case x of True  -> False
2                    False -> True
```

A *lambda expression* can be used to create an anonymous function. This takes the form "**`\`***`pat`* **`->`** *`repl`*," where *pat* is a linear constructor pattern and *repl* is a

replacement. This can simplify certain higher-order functions by eliminating the need to create a named local function just to return it. For instance, we could define `incBy` more succinctly using a lambda expression as follows:

```
incBy' n = \x -> x + n
```

Lambda expressions are useful when constructing arguments to higher-order functions. For example, a function to turn a boring list of strings into an exciting one could be defined as follows:

```
excite = map (\word -> word ++ "!")
```

This curries the `map` function with an anonymous function. The result is a function that accepts a list of strings and adds something exciting to each one.

## 2.2 Semantics

Now that we are familiar with their syntactic structure, we turn our attention to the meaning of Curry programs. We are particularly interested in what distinguishes Curry from Haskell, namely, the presence of non-determinism. Several approaches to formally capture the semantics of functional logic languages have been published [44–47]. To simplify matters we shall keep this discussion informal. The aim is to develop a model that allows readers to grasp Curry semantics well enough without venturing into arcane matters.

Our model is an unconventional one based on an analogy with the mathematical formulation of quantum mechanics (QM). What underlies this perhaps surprising choice is a remarkable similarity between the mathematics of non-deterministic physical systems and non-deterministic computations in Curry[1]. QM provides a mature framework for describing non-determinism and, as we shall see, its tools are well suited to the task at hand. Not only does this provide us a firm conceptual foundation, it lends confidence to our approach and may also point toward areas for further development wherever we note differences between existing semantic models and this analogy or where a similarity between these domains inspires us to borrow something. As is often the case, a shift in perspective can encourage new ways of thinking that lead to deeper understanding.

That said, it is important not to carry the analogy too far. What we present is not a rigorous formulation of Curry semantics, but rather an informal tool for explaining the behavior of Curry programs. A important difference here is that physical systems are constrained by conservation laws that do not exist in the computational domain. This means that Curry programs can exhibit behaviors with no physical analog. A concrete example is given in Sect. 2.3.2. Despite this, we believe the semantic model developed in the section, being both straightforward and distinctive, provides a useful way to understand Curry.

[1]It would be impossible to summarize the mathematics of QM here, so we shall introduce only the bits needed to construct the analogy. Interested readers are referred to, e.g., [48–50].

### 2.2.1 The Superposition Principle

One of the simplest non-deterministic functions imaginable is shown below:

```
1 binDigit = 0
2 binDigit = 1
```

Both rules match when evaluating the expression `binDigit` so this function evaluates to an arbitrary binary digit, either `0` or `1`. Since the result is an integer, the type of this function is `Int`. This might seem counterintuitive. Because there are two values, one might expect the result to be a collection of some sort. We can say with certainty that `binDigit` is multi-valued in the sense that it has two values and yet we see no evidence of that when it is evaluated. Our first task is to develop a precise understanding of this apparent contradiction. The principle of *superposition* will be instrumental in doing so. Superposition plays a central role in descriptions of non-deterministic physical systems by QM and can also greatly help us model non-deterministic computations.

Superposition is the ability of a system to be in multiple states at once. This is easily demonstrated through an example. Electrons have a measure of angular momentum called spin. When spin is measured, an electron is always found to be either spin-up or spin-down. Observable properties such as this are called *pure states* and we can denote these two as $|\uparrow\rangle$ and $|\downarrow\rangle$, respectively. The principle of superposition says there can exist states that are simultaneously spin-up and spin-down. These are called *mixed states* and we may write the simplest one as $|\uparrow\rangle + |\downarrow\rangle$. Mixed states cannot

be observed directly but their existence can be inferred experimentally. According to one popular interpretation of QM, the act of measurement causes a mixed state to collapse non-deterministically into a pure one.

Each aspect of the above account has an analog in the behavior of `binDigit`. Just as measuring an electron produces one of its pure states, evaluating `binDigit` yields one of its values. The values are thus analogous to pure states and we may write them as $|\texttt{0}\rangle$ and $|\texttt{1}\rangle$. Prior to its evaluation `binDigit` carries with it the potential to produce either value, so it corresponds to the state $|\texttt{0}\rangle + |\texttt{1}\rangle$. With this we may begin to see how the key to resolving the paradox above is to separate the values of an expression from the state it induces: although `binDigit` always evaluates to a singular binary digit, it induces a multi-valued state that has the potential to become either binary digit.

In QM, a state is a vector in a *Hilbert space*, a complex vector space with an inner product. Much of the mathematical complexity concerns the calculation of amplitude, which relates to the probability of a physical observation. When it comes to Curry semantics, we are only interested in determining the values produced by computations, not in their relative likelihood (if that can even be defined). For this reason, we can dispense with much of the mathematical complexity of QM. In particular, we shall ignore scalar factors, called weights, and shall not discuss inner products in detail. It is not difficult to imagine refinements that would make use of these, perhaps in estimating the probability of finding a value quickly or to otherwise

rank the prospective value of computational paths. This topic is beyond the scope of this work. We nevertheless find it useful to employ *Dirac notation*, where a state is written as $|s\rangle$, called a *ket*, in which the enclosed symbol is an identifier.

### 2.2.2 Program and Value Semantics

It is helpful to draw a distinction at this point between two representations that have arisen. One of these comprises the symbolic terms subject to pattern matching and rewriting. We call this the *program state* and say that it inhabits *program space*. Some form of this may reside in the memory of a computer running a Curry program. In simple cases the program state can be represented simply as an expression. A rewrite step represents a concrete change to this state and so we overload the $\rightarrow$ symbol to relate one program state to its successor. The second of these is the mathematical representation borrowed from QM. We call this a *value state*, since the pure states it is built up from correspond to Curry values. We say that value states inhabit *value space*.

For a type $T$, the set of all pure states corresponding to constructor expressions of $T$ constitute an orthogonal *basis* spanning the space of all possible value states of that type. For example, type `Int` forms an infinite-dimensional space with basis $\{|\texttt{0}\rangle, |\texttt{1}\rangle, |\texttt{-1}\rangle, |\texttt{2}\rangle, \ldots\}$. Some of the infinitely-many value states that can be formed from this include all integers, even integers, and prime numbers. To say the basis is orthogonal means its elements are pairwise non-overlapping. The overlap between

two states $|i\rangle$ and $|j\rangle$, written $\langle i|j\rangle$, can be understood as pattern matching with pattern $|i\rangle$ and candidate $|j\rangle$. The result is non-zero (taken nominally as unity) if and only if the pattern match succeeds. This can be tested with the following gadget:

```
case j of i -> ()
```

So, for example, $\langle \texttt{0}|\texttt{0}\rangle = 1$ because `case 0 of 0 -> ()` succeeds. Similarly, $\langle \texttt{0}|\texttt{1}\rangle = 0$ because `case 1 of 0 -> ()` does not. Let $\delta_{ij}$ represent the Kronecker delta, which equals zero where $i \neq j$ and unity where $i = j$. With this, the orthogonality condition can be written succinctly as follows:

$$\langle i|j\rangle = \delta_{ij}$$

This notation represents measurement in QM. For example, $\langle \uparrow | \psi \rangle$ gives the probability of measuring the spin-up property in state $|\psi\rangle$. In the computational analogy, it indicates whether a given Curry expression has a certain value. For instance, $\langle \texttt{0}|\texttt{binDigit}\rangle = 1$ and $\langle \texttt{2}|\texttt{binDigit}\rangle = 0$.

The result of a function is a subspace whose basis is a subset of the basis of its result type. For instance, the type of function `binDigit` is `Int`, and this function induces the two-dimensional subspace with basis $B = \{|\texttt{0}\rangle, |\texttt{1}\rangle\}$. We may therefore say the basis of the expression `binDigit` is $B$ and that the values of `binDigit` are `0` and `1`. Computing the values of a Curry expression is equivalent to determining the basis of the subspace it induces.

These two representations provide complementary frameworks with which to understand the behavior and meaning of Curry programs. Program space offers a way to model structure and analyze the mechanics of computational steps. The meaning from this viewpoint is called *program semantics*. Value space allows us to reason about the values of a Curry program apart from its mechanisms. With this, one can reason about the validity of possible implementations and also about the effect that changing a program will have on the values it produces. The meaning of a program in this sense is called its *value semantics*.

The set of values (whether we can determine them ahead of time or not) is fixed at the outset of a program. If a computation is represented by a series of transformations in value space, then every transformation must be value preserving. For this reason, value states are most often equated. We shall find it useful to formulate value states in multiple ways and to think of them as evolving with respect to a variable that advances with the steps of a computation. This is no different from the way an equation would change form as we rearrange it through the application of algebraic rules, were we to parameterize our rearrangements with a time-like variable. The preserved quantity does not change, but our representation of it does. Central to this activity are *operators*, which allow us to model any computation as a value-preserving evolution in value space.

### 2.2.3 Operators

Operators are mathematical objects used to model state changes, such as the evolution in time of a physical system. For any operator, $\mathcal{O}$, the state that results from carrying out the process represented by $\mathcal{O}$ on an object in state $|s\rangle$ is written $\mathcal{O}\,|s\rangle$. For instance, the magnetic moment of an electron due to its spin causes it to be deflected when passed through a magnetic field. Suppose that spin-up electrons in a certain setup are deflected to the left and spin-down electrons are deflected to the right, and that we denote these as $|\leftarrow\rangle$ and $|\rightarrow\rangle$, respectively. Then we may write $\mathbf{X}\,|\uparrow\rangle = |\leftarrow\rangle$ to represent the position state of an up-spinning electron following this process, where $\mathbf{X}$ is an operator that models the state evolution. It may involve such things as the magnetic field strength and details of the measurement apparatus determining the final position. The elimination of an operator through a substitution such as the one above is called *reduction*. The probability of measuring the electron to the *left* in this system is given by $\langle\leftarrow|\mathbf{X}|\uparrow\rangle = 1$.

Functions evolve program states in much the same way operators evolve value states. To demonstrate this, let us express the electron-deflection setup in Curry as follows:

```
1 data Spin = Up | Down
2 emit = Up
3 emit = Down
4
5 data Position = Left | Right
```

```
6 position Up = Left
7 position Down = Right
```

Function `emit` represents the electron emission process, while `position` represents the rest of the system up through position measurement. For every Curry function, `f`, we can define a corresponding operator, **f**; and for every constructor expression, $c$, we can define a corresponding pure state, $|c\rangle$. This gives rise to a **position** operator and pure states $|\texttt{Up}\rangle$, $|\texttt{Down}\rangle$, $|\texttt{Left}\rangle$, and $|\texttt{Right}\rangle$. For any expression $e$, rewriting by a rule of `position` in $e$ advances the computation of $e$ in exactly the same way that reduction of the **position** operator evolves the value state corresponding to $e$. The table below illustrates this:

| Computational Step | Value State |
|---|---|
| `position Up` $\rightarrow$ `Left` | **position** $|\texttt{Up}\rangle = |\texttt{Left}\rangle$ |
| `position Down` $\rightarrow$ `Right` | **position** $|\texttt{Down}\rangle = |\texttt{Right}\rangle$ |

From this we can conclude, for instance, that the value state of `position Up` is $|\texttt{Left}\rangle$. In general, the value state indicates the values we should expect to obtain from evaluation.

One may wonder what the point of this is, since the function and operator seem to perform identical actions in this example. To be clear, the Curry value `Left` is something we hope to produce via a computation whereas the state $|\texttt{Left}\rangle$ is an intrinsic property of the original expression. One goal of a Curry compiler is to implement computations in a way that finds all values, which means producing every value that

corresponds to a pure state in the basis of a program's value state. These values, taken together, form the program's *value set*. A benefit of developing the mathematics of value space apart from program space is that it allows us to define value sets separate from any program mechanism. This is especially useful where it allows us to reason subjunctively about the effects of undecidable properties. For instance, we can consider the effects on the values of an expression of an endlessly-looping subexpression. The subexpression never produces a value, and so corresponds to an empty value state. But that fact cannot in general be determined from operational principles. The concept of value space gives us a framework within which to engage in *what-if* reasoning about such possibilities.

We can use this model to reason about how changing a program affects the values it produces. To give a trivial example, the expression `Up` corresponds to state $|$`Up`$\rangle$ and its value set is {`Up`}. To find the effect of applying the `position` function, as in `position Up`, we apply the **position** operator to $|$`Up`$\rangle$ and reduce to $|$`Left`$\rangle$, which leads us to conclude the value set of that expression is {`Left`}. With this, we can say that a computation of `position Up` ought to yield `Left` and check our program mechanism to ensure that it does.

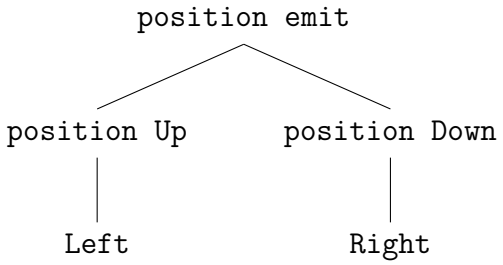
### 2.2.4 Additivity

Operators may act on mixed states. QM holds that, absent any measurement, an electron whose state is the superposition of up- and down- spins is simultaneously

deflected in *both* directions, putting its position into a superposed state. This is captured in the following equation:

$$\mathbf{X}\ (|\uparrow\rangle + |\downarrow\rangle) = |\leftarrow\rangle + |\rightarrow\rangle$$

Behavior such as this forms the physical basis of the famous "double-slit" phenomenon and others like it. A similar phenomenon is observed when evaluating the expression `position emit`. Rewriting is capable of deriving two values, `Left` and `Right`, as shown here:




The value state is therefore $|\texttt{Left}\rangle + |\texttt{Right}\rangle$. This can be derived in value space if we assume that **position** distributes over addition, i.e.:

$$\begin{aligned} \textbf{position}\ (|\texttt{Up}\rangle + |\texttt{Down}\rangle) &= \textbf{position}\ |\texttt{Up}\rangle + \textbf{position}\ |\texttt{Down}\rangle \\ &= |\texttt{Left}\rangle + |\texttt{Right}\rangle \end{aligned}$$

We can show that every operator in our formulation has this property, owing to the fact that every function is pure. First, the action of a pure function is inobservable in disjoint computations because by definition it cannot have side effects. This means that the act of distributing an operator cannot lead to its application in one context

contaminating another. Second, the effect of a pure function in a shared subexpression is identical in all contexts, since its action cannot depend on anything except its formal arguments. Thus, if we distribute an operator over two states that share a subexpression, $e$, then the effect of that operator on $e$ is identical in both contexts. This means that operator distribution never leads to spurious results due to the presence of shared subexpressions.

The validity of distributing operators over addition reflects a defining property of superposable systems: every operator, $\mathcal{O}$, in such a system is linear, which means that for arbitrary states $|s_1\rangle$, $|s_2\rangle$, and scalar $\alpha$, the following hold:

$$\mathcal{O}(\alpha\,|s_1\rangle) = \alpha\ \mathcal{O}\,|s_1\rangle \qquad \textit{(homogeneity)}$$
$$\mathcal{O}(|s_1\rangle + |s_2\rangle) = \mathcal{O}\,|s_1\rangle + \mathcal{O}\,|s_2\rangle \qquad \textit{(additivity)}$$

Since our model is unweighted the rule of homogeneity is degenerate.

The distribution of a process over non-deterministic value states by the rule of additivity corresponds to a transformation in program space called *pull-tabbing* [51] that will figure prominently in the Fair Scheme. Pull-tabbing is a mechanism for distributing deterministic computational steps over non-deterministic program states. Antoy et al. demonstrated its correctness in the context of functional logic programs [52]. Pull-tabbing is crucial to this work as it provides us the means to carry out potentially non-terminating and infinitely non-deterministically branching computations on sequential hardware in a way that is both sound and complete. These matters

will be dealt with in detail in Chapter 3, but for now it suffices to recognize that our semantic model is so far incomplete because it relies only on rewriting. Rewriting is not value-preserving in the presence of non-determinism. To see this, consider the rewrite step `position emit` → `position Up`, which occurs by a non-deterministic rule of `emit`. This causes the value `Right` to be lost. To address this shortcoming we will later introduce into the program domain a structural representation of non-determinism called *choice* that is subject to pull-tabbing.

### 2.2.5 Separability

Not all value states can be separated. To illustrate this, consider a system of two electrons. A possible result of measuring both spin states can be written $|\uparrow\rangle_0 |\downarrow\rangle_1$. This corresponds to the case when the first electron is spin-up and the second is spin-down. A product term such as this is called a *joint state*. As a shorthand, we can dispense with the subscripts and write this simply as $|\uparrow\downarrow\rangle$. If the electron spins are independent, the state of the entire system is a superposition over four pure states. This state is said to be *separable* because it can be factored into a product of independent terms that separately describe each electron, i.e.:

$$|\uparrow\uparrow\rangle + |\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle + |\downarrow\downarrow\rangle = (|\uparrow\rangle_0 + |\downarrow\rangle_0) \times (|\uparrow\rangle_1 + |\downarrow\rangle_1)$$

Now consider a configuration in which the spins are coupled. For instance, it is possible to generate a two-electron system where the total angular momentum is

zero, in which case each electron has a spin opposite the other. The corresponding state, $|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle$, can no longer be separated into a product. The electron spins are correlated in that a measurement of one completely determines the other. States with this property are called *inseparable* and these electrons are said to be *entangled*.

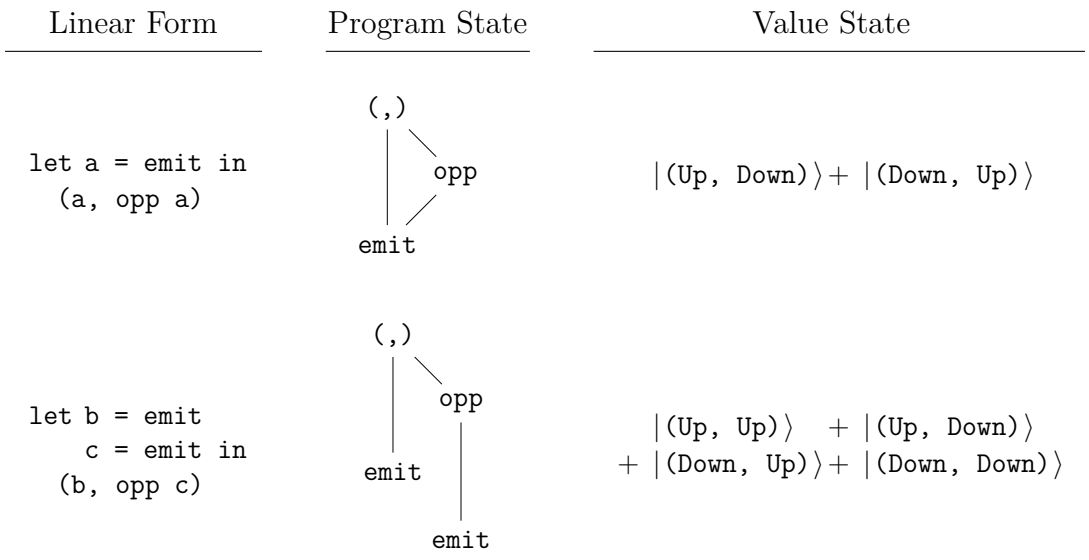



FIGURE 2.1: A comparison of inseparable and separable two-electron systems in Curry. The inseparable case (top) involves a single shared electron, represented by variable `a`. Its program state comprises a single node labeled `emit` and its value state is a superposition between two pure states. The separable case (bottom) involves two independent electrons represented by two variables, `b` and `c`. These appear as separate nodes labeled `emit` in the program state. The corresponding value state is a superposition between four pure states.

Inseparability in value space corresponds to sharing in program space. This is illustrated in Figure 2.1, which uses the following function to compute opposite spins:

```
1 opp Up   = Down
2 opp Down = Up
```

Shared subexpressions always arise from repeated variable references, such as those seen in `(a, opp a)`. This contains a shared subexpression precisely because `a` is referenced twice. The repetition can be more subtle. For instance, an application of `f x y = (x, opp y)` could result in a shared state though it does not directly involve a repeated variable. But in that case, the repetition would occur in another context such as `f a a`.

Each variable repeated in a textual representation appears in the corresponding program state as a node with multiple predecessors. Such nodes are for this reason called *shared*. An expression corresponding to a program state is called its linear form and the conversion of a program state into that is called *linearization*. Note that linearization can involve variable introduction. For instance, to properly retain the meaning in going from the topmost program state in Figure 2.1 to the corresponding linear form requires the introduction of a variable to represent the shared subexpression. In general, each incoming edge represents an occurrence, so every node of higher in-degree requires a corresponding variable to express its multiple occurrences linearly. On the other hand, in the separable case no variables are necessary. Two expository variables are shown in Figure 2.1 but they are unnecessary: the linear form could just as well be written as `(emit, opp emit)`. This demonstrates that multiple linearizations are possible. We may omit variables from a linear form when they are functionally irrelevant; in particular, this arises when the shared part is simply a constructor expression (see next example).

The presence of repeated variables or, equivalently, shared nodes causes the corresponding value state to be inseparable. This is because shared subexpressions are entangled, which manifests in value space as inseparability. To illustrate, consider the following rewrite step according to the rule `emit = Up`:

```
let a = emit in (a, opp a)    →    (Up, opp Up)
```


The two subexpressions of the pair, which represent two entangled electrons, are correlated in that a single rewrite step affects them both. Replacing a single shared node in the program state has this effect. In value space, the consequence of correlation is that certain joint states — the ones corresponding to uncorrelated variance — are excluded. This precludes factorization because the missing terms mean the mathematical "square" is incomplete. To summarize, up to certain degenerate cases we shall not consider, the following three objects are identical:

- In a linear form, a multiply-referenced variable;
- In program space, a node with multiple predecessors;
- In value space, an inseparable state.

### 2.2.6 Analysis of Curry Programs

The semantic framework we have now developed helps us understand different aspects of Curry. The structure of expressions can be understood through their program states, and computations can be expressed as transformations in that domain. This view not only helps us grasp the meaning of programs, but also informs our implementation of a Curry compiler and runtime system because it tells us precisely which data structures and what transformations over those data structures are required. The values of an expression can be understood by examining its value state. Operator mathematics gives us a computational system in value domain, where transformations are subject to the constraint that they be value-preserving. Non-determinism can be modeled as the ability to choose from among multiple overlapping rules and its effect in value space can be predicted by the principle of superposition.

This yields a constructive method for analyzing Curry programs. At the bottom, each innermost expression corresponds to a value state[2]. Moving outwards, function application constructs a larger program state whose meaning is found by applying the corresponding operator to the previous value state. Computational steps are carried out by rewriting in program space and operator reduction in value space. For example, to analyze `position emit`, one begins with `emit`. This gives rise to a superposition between either spin state: $\psi_0 = |\texttt{Up}\rangle + |\texttt{Down}\rangle$. Moving outwards, the

[2]A nullary function is considered a reference to its RHS to avoid applying an operator to nothing.

application of `position` leads to $\psi_1 =$ **position** $\psi_0 = |\texttt{Left}\rangle + |\texttt{Right}\rangle$. We may simulate a computational step by choosing a rule of `position` and carrying out a rewrite step, though we must be careful to remember that eventually all rules need to be applied to all states in order to find all values.

## 2.3 Additional Topics

We complete our introduction to Curry with a discussion of a few matters beyond the basic syntax and semantics. We begin with the various forms of non-determinism that arise in Curry programs.

### 2.3.1 Representations of Non-determinism

Among the several ways of expressing non-determinism in Curry are overlapping patterns, a choice operator, and free variables. These arose from attempts by many researchers to couch principles of logic programming in a Haskell-like syntax. Although it may not be apparent at first, all of these are semantically equivalent [41]; this fact will be used as we develop the Fair Scheme to greatly simplify our implementation. The addition of logic features, interestingly, involves little syntactic change. Rather, these new capabilities emerge mostly as a natural consequence of extending the value space to include superposition.

We have already seen how overlapping patterns embody non-determinism. The two rules defining `binDigit`, for instance, both match the same (trivial) pattern, so

we could choose either one in a rewriting computation involving that symbol. As discussed in the previous section, overlapping patterns are interpreted in value space as a superposition over all possible matches. Compare this with Haskell, which has no concept of superposition and so handles overlapping patterns by simply trying them in order until the first match occurs.

This difference means we sometimes need to adapt function definitions. In Haskell one must be careful to arrange rules in a sequence that ensures the proper behavior when considering overlaps. In Curry, on the other hand, one should avoid accidental overlaps that give rise to unintentional non-determinism. Figure 2.2 illustrates this by comparing the definitions of `zip` in Haskell and Curry. Note that `zip` is a binary function that converts two lists into a list of pairs, where each output element contains corresponding input elements. We could use this as follows: `zip [1,2] [3,4]` $\rightarrow^*$ `[(1,3),(2,4)]`.

**Haskell**

```
1 zip []     _      = []
2 zip _      []     = []
3 zip (a:as) (b:bs) =
4     (a, b) : zip as bs
```

**Curry**

```
1 zip []     _      = []
2 zip (_:_)  []     = []
3 zip (a:as) (b:bs) =
4     (a, b) : zip as bs
```

FIGURE 2.2: Comparison between definitions of `zip` in Haskell and Curry. The Haskell definition involves overlapping rules that would result in this function being non-deterministic in Curry (i.e., consider a match against `zip [] []`). The Curry definition uses a more specific pattern.

Another way to express non-determinism is through the built-in *choice* operator, written `?` and defined by the following rules:

```
1  x ? _ = x
2  _ ? y = y
```

Application of this operator creates an explicit superposition between its arguments. Choices are the primitive expression of non-determinism in Curry, since any function with overlapping rules can be rewritten as a non-overlapping function that applies choices. For example, we could redefine `binDigit` as follows:

```
binDigit' = 0 ? 1
```

Although the choice operator can be defined by the ordinary rules shown above, it plays a special role in the Fair Scheme as the structural representation of superposition in program space. For that reason, as will be discussed in the next chapter, its rules are never applied in a computation by the Fair Scheme.

The introduction of an "ambiguity" operator such as choice into programming can be traced to McCarthy [53] and the introduction of choice itself into Curry is owed to Antoy [43]. Choices are a natural way to express non-determinism. Early versions of Curry, lacking choices, resolved non-deterministic computational branches through a principle called *narrowing* [40] in which free variables are instantiated with a particular function rule in mind. These "just so" instantiations are done carefully to ensure the successful match of a certain pattern. Although simple evaluation

strategies based on narrowing are known to be complete and optimal [54], these have the same limitation as purely rewriting strategies, stemming from the fact that narrowing steps are by themselves incomplete. Therefore, an additional mechanism that ensures all narrowing steps are eventually taken is needed to ensure completeness.

The absence of choice perhaps contributed to a fragmented early semantics of Curry that found it divided into separate functions and predicates and allowed narrowing only in predicates (e.g., see [55]). Choices help make possible a tighter integration between the functional and logic portions of Curry programs.

A third way to express non-determinism is through the use of free variables. Variable bindings can occur in rule patterns, *where-clauses*, and the introductory parts of *let-expressions*. Any variable referenced without first being bound in a surrounding context is free. Such a variable would be treated as an error in Haskell, but in Curry these are interpreted as unknown values to be deduced. The value of a free variable is a superposition over all values belonging to its type. For example, a Boolean free variable corresponds to the value state $|\texttt{False}\rangle + |\texttt{True}\rangle$. The value of a free variable with type `[Bool]` is an infinite sum over all Boolean lists.

However, it turns out that programmers sometimes make mistakes. Thus it is unwise to impute meaning to what could be an accident. Curry, therefore, requires free variables to be introduced by the *free* keyword in most cases. This is recalled in the following function definition, which uses a free variable to define the pairs `(False, True)` and `(True, False)`:

```
boolPair = (x, not x) where x free
```

To drive home the equivalence between the ways of expressing non-determinism just discussed we shall write this function in two additional ways to demonstrate each method. Using a choice, we may write this as follows:

```
boolPair' = (x, not x) where x = False ? True
```

The order of arguments to `?` is not relevant because we are only ever interested in the values of expressions but not their order. With overlapping rules we can write the following:

```
1 boolGen = False
2 boolGen = True
3 boolPair'' = (x, not x) where x = boolGen
```

This latter example defines a function `boolGen` to generate all Boolean values. A function whose value state is identical to a free variable — which is to say a sum over all pure states of the proper type — is called a *generator function*. It is possible to write such a function for any type, including those with infinitely many pure states, simply by providing a rule for each constructor in which every argument is a generator function of the appropriate type. For example, a generator for a list of Booleans can be written as follows:

```
1 boolListGen = []
2 boolListGen = boolGen : boolListGen
```

This suggests an approach to implementing computations in the presence of free variables, which is to replace them with generator functions. We will examine this idea in more detail as we develop the Fair Scheme.

### 2.3.2 Failure

Curry semantics permit a state with no basis — i.e., a superposition of zero states. An expression with this state has no values. Though this might at first seem a useless concept, this special state, called *failure*, comes up frequently and can be quite useful. We represent failure as $\perp$, though we must be careful not to confuse this with the bottom type, written in the same way, which is a type inhabited by nothing. Failures play a role analogous the number zero in addition, since they can be added to any value state without changing anything.

A computation fails when it does not produce a value. We have so far informally used the term *value* without a precise definition. We will attend to that in the next chapter where we discuss the properties of rewriting systems in more detail, but for now it suffices to say that an expression containing irreducible function symbols is not a value. One kind of failure, then, is the inability to complete a pattern match. We see this possibility in the `head` function, shown below, which evaluates to the head of a given list:

```
head (a:_) = a
```

Since the head of an empty list is not defined, no rule matching an empty list is provided. A function such as this that is not defined for all possible arguments is called *partial*. Accordingly, the evaluation of `head []` fails. In Haskell this situation is an error that causes the program to terminate. In Curry, though, a more natural response is simply to ignore the failure, since it adds nothing to the relevant value state. The sense in this is illustrated by the next example.

One expects the expression `head boolListGen` to correspond to the value state $|\texttt{False}\rangle + |\texttt{True}\rangle$. We might interpret this as asking the question, what head elements might we observe in an arbitrary Boolean list? The existence of empty lists does not invalidate this question. But if pattern-matching failures were interpreted as errors then it would have no values, since the first rule of `boolListGen` leads to the irreducible expression `head []`.

Failures highlight a limitation of our semantic analogy. The physical analog of an empty value state is nothingness, i.e., a state whose basis is empty. No linear operator can transform such a state into one with a nonempty basis. There exists no constructor expression $c$ such that $\langle c|\perp\rangle$ is nonzero. In a Curry computation, by contrast, certain valueless expressions can be transformed into values by function application. To demonstrate this, we can define the following function to obtain the first element of a pair:

```
fst (a, _) = a
```

Now consider the expression `fst (0, head [])`. The pair is a failure because there exists no constructor expression $c$ such that $\langle c \,|\, \texttt{(0, head [])} \rangle$ is nonzero. But the application of `fst` gives us $\langle \texttt{0} | \textbf{fst} | \texttt{(0, head [])} \rangle = 1$, and so `0` is a value of that expression. By this we can see that Curry functions are not perfectly analogous to linear operators.

### 2.3.3 Constraint Programming

The practice of ignoring failure makes possible a programming style called *constraint programming* in which programs are written by describing the constraints solutions should adhere to. Viable answers are produced through a search process. This paradigm, first proposed by Jaffar and Lassez in 1987 [56], is rooted in logic programming, where the search capabilities inherent to many such languages make it a natural fit.

In Curry, a constraint may be expressed as a guard expression involving one or more free variables. In the RHS, those free variables take on all and only the values satisfying the constraint. As a trivial example, consider the following function, which finds a Boolean substitution that satisfies the logical AND relation:

```
satAnd | x && x = x where x free
```

Since this rule has no default part, it fails when the constraint is not satisfied. When the constraint is met — that is, when `x` is bound to `True` — this function produces

the value of `x`.

The essential mechanic of this pattern is captured by the *constrained expression* operator [57, Sect. 3.1], an infix binary operator whose first argument is a constraint and whose second argument is an arbitrary expression. It is defined by the following rule:

```
True &> x = x
```

This function evaluates to its right argument provided its left argument is satisfied and fails otherwise. With this, `satAnd` could be defined as shown here:

```
satAnd' = x && x &> x where x free
```

Curry, in addition, provides an *equational constraint* operator (`=:=`) that evaluates to `True` when its arguments are equal and fails otherwise. This can be defined as follows:

```
x =:= y | x == y = True
```

This gives us a third way to define the `last` function introduced in Sect. 2.1.3. In the next definition, the function pattern that selects the final list element is moved into the RHS:

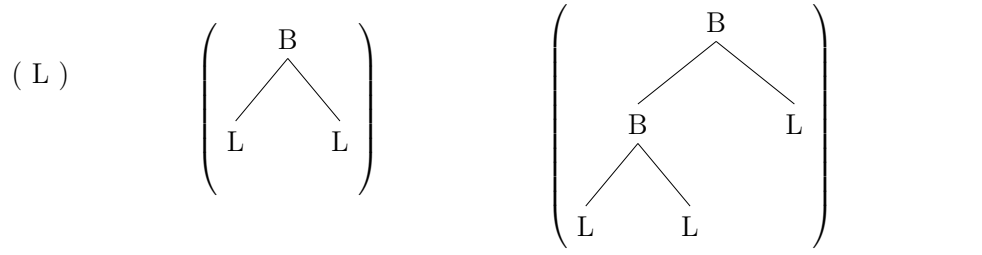
```
last'' l = let x free in l =:= (_++[x]) &> x
```

A slightly more complex example further illustrates the usefulness of this approach. A binary tree is defined as either a *branch*, which contains two subtrees, or a *leaf*. This can be defined in Curry with the following data definition:

```
data bTree = Branch bTree bTree | Leaf
```

A few examples are depicted below:

`Leaf` `Branch Leaf Leaf` `Branch (Branch Leaf Leaf) Leaf`




A function to generate all binary trees up to a specified height, `h`, can be written as follows:

```
1 bTree _          = Leaf
2 bTree h | h > 0 = Branch (bTree (h-1)) (bTree (h-1))
```

We can use constraint programming to refine this in order to generate trees that meet whatever criteria we choose. For instance, a binary tree is symmetric if its left and right branches are equal. We could implement a constraint to enforce this as follows:

```
1 symmetric Leaf         = True
2 symmetric (Branch l r) = l =:= r
```

Using this, a function to select symmetric binary trees can be written simply as shown below:

```
sbTree tree | symmetric tree = tree
```

To generate all symmetric binary trees up to height $h$, one could write this:

```
sbTree (bTree h)
```

A tree is balanced if and only if every branch it contains is balanced, and a branch is balanced if and only if 1) its subtrees are balanced, and 2) the difference in height between its subtrees is no more than one. A function to determine the height of a tree can be written as follows:

```
1 height Leaf         = 0
2 height (Branch l r) = 1 + max (height l) (height r)
```

We can implement a constraint for balanced trees as shown here:

```
1 balanced Leaf = True
2 balanced (Branch l r)
3     | balanced l && balanced r &&
4       abs (height r - height l) <= 1 = True
```

Similar to the previous example, a function to select balanced binary trees can be written as follows:

```
bbTree tree | balanced tree = tree
```

While techniques exist to reduce the computational complexity of object generation in constraint programming, these aspects have not been addressed in the previous examples, as their discussion falls outside the scope of this work.

### 2.3.4 Non-Linear and Function Patterns

As discussed in Sect. 2.1.3, Curry patterns may involve function applications, repeated variables, or both. We shall now take a deeper look into the meaning of and mechanisms behind these features. To begin, we recall the definition of `last`:

```
last (_++[x]) = x
```

This can be understood as a recipe for the compiler, instructing it how to build patterns dynamically. This allows the rule to be adapted at runtime to fit any given list. For example, the table below shows the computed patterns and corresponding substitutions that arise in a few simple cases:

| Expression | Computed Pattern | Subst. |
|---|---|---|
| `last [0]` | `last [x]` | $\texttt{x} \mapsto \texttt{0}$ |
| `last [0,1]` | `last [_,x]` | $\texttt{x} \mapsto \texttt{1}$ |
| `last [0,1,2]` | `last [_,_,x]` | $\texttt{x} \mapsto \texttt{2}$ |

Repeated variables can be implemented via constraints. Consider the following constraint function that is satisfied when two lists have equal head elements:

```
sameHead (a:_) (a:_) = True
```

For this rule to match, a consistent substitution must be found. A simple way to do this is through the introduction of a constraint. The idea is to allow the occurrences of the repeated variable to vary independently but make the rule conditional on them being equal. The following possible implementation is suggestive of this approach:

```
sameHead' (a0:_) (a1:_) = a0 =:= a1
```

Repeated pattern variables can always be eliminated in this way, and the constraints created can appear in rule conditions or by prepending an application of `&>` to the RHS.

A functional pattern can expand to multiple matching rules. In that case the usual semantics apply, meaning that the dynamically generated rules overlap and lead to a superposition. To illustrate this, recall the definition of `find`:

```
find k (_++(k,v)++_) = v
```

Now consider the following expression to find digits that begin with the letter $t$:

```
find 't' [('z',0), ('o',1), ('t',2), ('t',3), ('f',4)]
```

The rules generated for this expression are like those shown here:

```
find k [(k',v), _, _, _, _] | k =:= k' = v
find k [_, (k',v), _, _, _] | k =:= k' = v
find k [_, _, (k',v), _, _] | k =:= k' = v
find k [_, _, _, (k',v), _] | k =:= k' = v
find k [_, _, _, _, (k',v)] | k =:= k' = v
```

This behaves like any other rule set: here, the third and fourth rules match so the expression above corresponds to $|2\rangle + |3\rangle$.

### 2.3.5 Set Functions

The application of a function induces a value set (see 2.2.3). The set corresponding to a function in Haskell is expected to contain exactly one element, but in Curry it can take any size. For failures, this set is empty. For deterministic functions (provided they succeed), it contains exactly one value. The value set corresponding to a function with non-determinism may contain zero, one, or more values. Set functions [58] provide a way to collect these values and work with them as a singular entity. By encapsulating the non-determinism of a function, one can test for the success of computations, find minimum or maximum values within a solution space, or compute the intersection or union among values of disjoint computations, among other things. Set functions therefore find practical use in applications that need to test whether any solution exists or that seek to compare solutions. Set functions have also been used to define extensions to Curry such as default rules [59]. The Sprite Curry system developed later provides an implementation of set functions, so we shell discuss them in some detail here.

Let `f`$_S$ denote the set function of function `f`. A Curry implementation might supply higher-order functions[3] for obtaining `f`$_S$ from `f`. If `f` produces a result of type $r$, then `f`$_S$ is a function that accepts the same number and type of arguments as `f`

[3]Note that these may act as compiler directives rather than as ordinary functions.

and produces a set over elements of type $r$. An application of $\texttt{f}_\texttt{S}$ evaluates to a set containing the values that the corresponding application of `f` would produce.

To illustrate, recall `binDigit` (see Sect. 2.2.1) and consider the corresponding set function $\texttt{binDigit}_\texttt{S}$. This evaluates to a set containing `0` and `1`. For simplicity, we use surrounding curly braces to represent sets in Curry, and so we write the result of $\texttt{binDigit}_\texttt{S}$ as `{0,1}` despite the fact that is not a valid syntactic construct in Curry[4]. In practice, the most popular Curry implementations supply an abstract interface to value sets that allows them to be computed lazily, meaning there is no proper set type exposed to programmers but rather several functions that operate on objects of this abstract type. This approach can be used to, e.g., efficiently test whether a value set is empty or to encapsulate non-terminating computations.

As discussed in [60] and summarized in [61], an important subtlety lies in the handling of any non-determinism present in the arguments to a set function. A semantic rule called *strong encapsulation*, similar to Prolog's `findall`, calls for encapsulating all non-determinism encountered during the evaluation of a set function. That approach suffers from a serious shortcoming, which is that the computed sets depend on incidental aspects of the evaluation strategy. We shall therefore rely on another principle called *weak encapsulation* that says a set function $\texttt{f}_\texttt{S}$ encapsulates only the non-determinism arising from `f` itself, and not from the arguments it is applied to.

[4]Curly braces have a reserved meaning related to object layout that we will not discuss.

To see this distinction, consider the application of $\texttt{incr}_S$ to `binDigit`, where `incr` is defined as `incr i = i + 1`. Since `incr` is deterministic there is nothing for $\texttt{incr}_S$ to capture, so the sets it produces always contain exactly one value. The non-determinism in this example comes from the argument, `binDigit`, which is not captured by $\texttt{incr}_S$. Accordingly, there are two values: `{1}` and `{2}`. Now, instead consider the application of $\texttt{adj}_S$ to `binDigit`, where `adj i = (i-1) ? (i+1)`. This involves non-determinism both in the function and its argument. The non-determinism of `adj` is encapsulated, giving rise to sets containing two elements. The non-determinism of `binDigit` is not encapsulated, so this expression produces two values: `{-1,1}` when `binDigit` evaluates to `0` and `{0,2}` when it evaluates to `1`.

When several non-deterministic arguments are involved, resolving them results in a combinatorial explosion. For example, consider a binary "butterfly" function defined by the following rule: `bfly x y = (x+y) ? (x-y)` In the evaluation of the expression $\texttt{bfly}_S$ `binDigit binDigit`, the non-determinism of the arguments leads to four values:

$\texttt{bfly}_S$ `0 0` $\rightarrow^*$ `{0}` $\qquad$ $\texttt{bfly}_S$ `0 1` $\rightarrow^*$ `{1,-1}`

$\texttt{bfly}_S$ `1 0` $\rightarrow^*$ `{1}` $\qquad$ $\texttt{bfly}_S$ `1 1` $\rightarrow^*$ `{2,0}`

Note that, despite what the above might suggest, set function arguments are evaluated lazily.

These results can be derived in value space. Recall that carrying out a computation in program space is equivalent to finding the basis of the corresponding value state (see Sect. 2.2.2). Define BASIS $(|\psi\rangle)$ to be the union of the basis of state $|\psi\rangle$ with $\{|\bot\rangle\}$, where $|\bot\rangle$ is the failure state. The reason for including failure will be made clear shortly. With this, the values that arise from applying a unary set function $\mathtt{f}_\mathcal{S}$ to an argument with value state $|\psi\rangle$ can be defined as follows:

$$\mathbf{f_S}\ |\psi\rangle\ =\ \sum_{|\widehat{\psi}\rangle \in \text{BASIS}(|\psi\rangle)} \text{BASIS}\left(\mathbf{f}\ |\widehat{\psi}\rangle\right) \tag{2.1}$$

Here, each application of BASIS produces a value set encapsulating the non-determinism of one application of `f` to a deterministic argument. The overall result is a superposition over value sets in which each term arises from $|\widehat{\psi}\rangle$ taking a pure state from the basis of the argument or failure. We define an empty value set to have the value $|\bot\rangle$. Using this, the values of $\mathtt{adj}_\mathcal{S}$ `binDigit` can be derived as shown below:

$$\begin{aligned}
\mathbf{adj_S}\ (|\mathtt{0}\rangle + |\mathtt{1}\rangle) &= \text{BASIS}\,(\mathbf{adj}\ |\mathtt{0}\rangle)\ +\ \text{BASIS}\,(\mathbf{adj}\ |\mathtt{1}\rangle)\ +\ \text{BASIS}\,(\mathbf{adj}\ |\bot\rangle) \\
&= \text{BASIS}\,(|\mathtt{-1}\rangle + |\mathtt{1}\rangle)\ +\ \text{BASIS}\,(|\mathtt{0}\rangle + |\mathtt{2}\rangle)\ +\ |\bot\rangle \\
&= |\mathtt{\{-1,1\}}\rangle\ +\ |\mathtt{\{0,2\}}\rangle
\end{aligned}$$

To see why failure is included in the definition of BASIS, consider the following constant function: `const1 _ = 1`. Since the argument is ignored, it need not have any value for an evaluation of `const1` to succeed. For example, `const1 (head [])`

evaluates to `1`. To avoid missing values that arise in cases such as this, it is necessary to include a term corresponding to failure in the summation.

This inclusion does not lead to spurious values. We so far have not provided the formal definitions required to prove this rigorously, particularly the concept of *need* (see Def. 3.4.1). We can present an informal argument by using "need" in an informal, intuitive way. We say the parameter of `const1` is not needed in the sense that a value is obtained even if the supplied argument is a failure. A parameter is not needed when it does not participate in pattern matching and is not used meaningfully in the RHS (including as part of the replacement), in which case a lazy evaluation strategy would have no reason to inspect it. With reference to Eq. 2.1, one way a spurious value could arise would be if the basis of $|\psi\rangle$ were empty and $\mathbf{f}\,|\perp\rangle$ produced a value not implied by $|\psi\rangle$. But $\mathbf{f}\,|\perp\rangle$ cannot produce a value unless the parameter to `f` is unneeded because inspecting $|\perp\rangle$ would result in failure. This is precisely the sort of value we wish to include, e.g., as in $\texttt{const1}_S$ `(head [])`. The second way a spurious value could arise would be if the basis, $B$, of the argument $|\psi\rangle$ were non-empty and $\mathbf{f}\,|\perp\rangle$ produced a value different from $\mathbf{f}\,|\widehat{\psi}\rangle$ for some $|\widehat{\psi}\rangle \in B$. As just argued, $\mathbf{f}\,|\perp\rangle$ cannot produce any value if the parameter is needed. Alternatively, if the parameter is not needed, then `f` returns the same value for every argument because to produce a different value would require inspecting the argument to make a distinction. Therefore the inclusion of failure does not produce a distinct element in the value set.

Extending Eq. 2.1 to functions that accept more arguments is straightforward, as each additional argument simply adds another level to the summation:

$$\mathbf{f}_{\mathbb{S}}\ |\psi_1\rangle \ldots |\psi_n\rangle \;=\; \sum_{|\widehat{\psi}_1\rangle \in \textsc{basis}(|\psi_1\rangle)} \;.\;.\;. \sum_{|\widehat{\psi}_n\rangle \in \textsc{basis}(|\psi_n\rangle)} \textsc{basis}\left(\mathbf{f}\ |\widehat{\psi}_1\rangle \ldots |\widehat{\psi}_n\rangle\right) \tag{2.2}$$

The values of $\texttt{bfly}_{\mathbb{S}}$ `binDigit binDigit` can be derived as follows according to this equation:

$$\begin{aligned}
&\mathbf{bfly}_{\mathbb{S}}\ (|\texttt{0}\rangle + |\texttt{1}\rangle)\ (|\texttt{0}\rangle + |\texttt{1}\rangle) \\
&\quad= \textsc{basis}\,(\mathbf{bfly}\ |\texttt{0}\rangle\ |\texttt{0}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\texttt{0}\rangle\ |\texttt{1}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\texttt{0}\rangle\ |\bot\rangle) \\
&\qquad\ \textsc{basis}\,(\mathbf{bfly}\ |\texttt{1}\rangle\ |\texttt{0}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\texttt{1}\rangle\ |\texttt{1}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\texttt{1}\rangle\ |\bot\rangle) \\
&\qquad\ \textsc{basis}\,(\mathbf{bfly}\ |\bot\rangle\ |\texttt{0}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\bot\rangle\ |\texttt{1}\rangle) \;+\; \textsc{basis}\,(\mathbf{bfly}\ |\bot\rangle\ |\bot\rangle) \\
&\quad= \textsc{basis}\,(|\texttt{0}\rangle) \;+\; \textsc{basis}\,(|\texttt{1}\rangle + |\texttt{-1}\rangle) \;+\; \textsc{basis}\,(|\texttt{1}\rangle) \;+\; \textsc{basis}\,(|\texttt{2}\rangle + |\texttt{0}\rangle) \\
&\quad= |\texttt{\{0\}}\rangle \;+\; |\texttt{\{1,-1\}}\rangle \;+\; |\texttt{\{1\}}\rangle \;+\; |\texttt{\{2,0\}}\rangle
\end{aligned}$$

It is possible to define infinite value sets. Consider the following function:

```
greater n = n ? greater (n+1)
```

Although the value of, say, $\texttt{greater}_{\mathbb{S}}$ `0` is well defined, it is infinite; hence it cannot be computed in finite time. This is similar to the way that infinite data structures (e.g., see `infList` on Page 38) can be defined but never fully realized. Like some other Curry systems, Sprite provides an abstract set type that allows programs to interact lazily with value sets. Like other computations set functionss represent a possibly infinite lazy computation.

Now that we have a firm understanding of Curry, we can move on to discussing our evaluation strategy. In the next chapter, we will introduce the initial version of the Fair Scheme, which compiles Curry into a well-defined class of rewrite systems that is suitable for building an efficient implementation. We will formally define this class and our compilation scheme, allowing us to rigorously define and prove important properties of our strategy.

# Chapter 3

## The Fair Scheme

*Note: The Fair Scheme was first described by this author with Antoy [62]. This chapter parallels that work, though here the notation has been simplified, the proofs revised, and the discussion extended.*

In this chapter we begin the work of implementing a Curry system by developing the Fair Scheme (FS), an evaluation strategy for functional logic programs. The FS defines a suitable representation of programs and outlines a series of transformations that can be used to compute their values. The initial version developed in this chapter eagerly replaces free variables with choice-based expressions and subsequently performs rewrite and pull-tab steps. Computations of this form produce only ground values in which all logic variables are reduced to constructor expressions (e.g., as shown in Figure 1.3, left). Although lacking a unification mechanism, this version serves as an important first step in developing a computational mechanism that can be proven correct and optimal. In the next chapter, we will develop three extensions to the FS that, among other things, allow for lazy variable replacement,

which admits free variables into values. Our focus at this stage is on developing a computational mechanism that is simpler and more suitable for proving the properties we are interested in, rather than on optimizing the exact form that values take.

An evaluation strategy dictates the flow of control in compiled programs. This determines crucial properties including their performance, resource usage, and correctness. The branching nature of functional-logic computations raises subtle issues that make correctness challenging to guarantee. In particular, when a computation on a sequential computer forks, one branch that leads to a value may "wait in line" behind another that never terminates. If the productive branch is never afforded an opportunity to proceed, then its value will never be produced. The FS prescribes step-by-step transformations in program space that avoid this and other correctness problems. Additionally, the FS is optimal in the sense that every step it takes is unavoidable (in a precise technical sense that we shall define).

Several well-known evaluation strategies for functional and logic languages exist. The spineless, tagless G-machine (STG) [31], for instance, serves as the basis of the Glasgow Haskell Compiler (GHC), [36], a popular implementation of Haskell. Variations on the principle of resolution likewise undergird implementations of Prolog [63, 64]. These strategies can and have been used as the basis for functional logic languages, including Curry. Two popular Curry systems, KiCS2 [65] and PAKCS [66], transform Curry programs into functional and logic programs, respectively. Compiled programs are evaluated by Haskell or Prolog using the strategies those targets provide.

This approach is convenient for implementers since it avoids having to develop an entirely new evaluation strategy, compiler, or runtime system. A major drawback, though, is that it requires features of either the functional or logic domain to be simulated at a high level in the other. For instance, when PAKCS maps a Curry program to Prolog, the deterministic portions of the program must be evaluated using the rules of resolution. Conversely, when KiCS2 maps a Curry program to Haskell, the non-deterministic aspects of the program must be simulated in a functional context. Experience shows that the results are suboptimal. For instance, PAKCS typically evaluates deterministic Curry programs (which are simply Haskell programs) much slower than does GHC. And, conversely, KiCS2 can perform relatively poorly on many non-deterministic programs as compared to PAKCS [47, 67, 68]. These results are confirmed by our measurements (see Chapter 7).

In addition to affecting efficiency, giving up control over the evaluation strategy impacts correctness. A well-behaved functional or logic strategy may only be approximately correct in the joint functional-logic domain, as the following program illustrates:

```
1 loop = loop
2 goal = loop ? True
```

Function `loop` is an infinite loop; hence, it has no values. The expression `goal` has one value, `True`, but neither KiCS2 nor PAKCS reports it. Both implementations exhibit a left bias in that they start by evaluating the left alternative of the choice, `loop`.

Neither has a mechanism to give the right-hand side of the choice "fair" attention, so neither can proceed when the evaluation of `loop` goes on forever. Since termination is undecidable, one cannot expect a runtime system to reliably detect loops in order to take corrective action. But an implementation tailored to the needs of functional logic programs can avoid problems such as this through the careful design of a suitable evaluation strategy.

These issues suggest that a custom evaluation strategy for functional logic programs may yield better results. By incorporating directly into the evaluation strategy everything needed to implement Curry — including choices, free variables, constraints, set functions, and failures — one decouples it from whatever target domain might otherwise be used for its implementation, thereby avoiding the peculiarities of that domain. The hope is that, by committing to the substantial work of defining this strategy and implementing a new compiler and runtime system, we will be rewarded with an improved Curry system.

After introducing the notation and other preliminaries in Sect. 3.1, we discuss in Sect. 3.2 the approach used by the FS to implement non-determinism. In Sect. 3.3, we give a precise definition of the FS, and in Sect. 3.4 we state and prove its correctness and optimality.

## 3.1 Background

The FS is built on the theory of constructor-based graph rewrite systems [69–72]. This work follows the standard notations with additions specific to functional logic programming following Braßel [47] and Antoy et al. [52, 62, 73]. The notation is simplified where extra details not needed for this work can be discarded. In this framework, expressions are represented as graphs, and computations are performed by rule-based graph transformations.

The use of term graphs instead of terms is motivated by efficiency considerations. The key difference between these two representations is that subexpressions in term graphs can be shared, as discussed in Sect. 2.2.5, whereas those in term rewriting systems cannot (since expressions in term rewriting systems are trees). Sharing subexpressions in term graphs allows computational steps to be shared directly, which avoids repeating arbitrary amounts of computation and also saves space.

The benefits to operational efficiency can be substantial. To illustrate this point, consider the following invocation of the `horner'` function (defined on Page 34):

```
horner' coeffs (a+b)
```

If the rules of horner were applied before reducing the term `a+b` in a term rewriting system, this would result in $n - 1$ copies of addition, where $n$ is the number of coefficients given. Each of these additions would be evaluated separately, even though they all produce identical results. The use of term graphs ensures that the result

of this addition is shared, allowing it to be evaluated at most once and avoiding unnecessary computation. The use of term graphs, therefore, is crucial to the ability of the FS to prevent redundant steps, which is an important aspect of the optimality property that we will prove in Sect. 3.4.3.

### 3.1.1 Constructor-Based Rewriting Systems

We follow the definition of constructor-based rewriting systems established by Echahed and Janodet [71]. We recall certain essential aspects needed for this work. A constructor-based signature, $\Sigma = \mathcal{C} \cup \mathcal{F}$, is a union of constructor and function symbols. Symbols are used to label graph nodes, and a node has as many successors as the arity of the label. Symbols with arity zero, one, and two are referred to as nullary, unary, and binary, respectively. Constructors are grouped into families called *types* in which the constructors of each type are sorted in an arbitrary but fixed order. Let $\mathcal{X}$ be an infinite set of nullary variable symbols such that $\Sigma \cap \mathcal{X} = \emptyset$ for all signatures.

An expression is a finite, acyclic, single-rooted graph that satisfies the following conditions[1].

1. Every node is labeled by a symbol taken from $\Sigma \cup \mathcal{X}$;

2. The outgoing edges emanating from each node are ordered; and

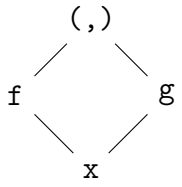
3. The out-degree of each node equals the arity of its label.

[1]This definition is greatly simplified. See [71] or [52, Def. 1] for a complete definition.

An expression that contains no function symbols is called a *constructor expression* and one that contains no variables is said to be *ground*. Since nodes and expressions exist in a bijection mapping each expression to its root, we use those terms interchangeably. We prefer to speak of expressions in most cases, reserving nodes for when the discussion centers on graphs. Only well-typed expressions are considered. In a compiler system built on the FS, a front-end component performs type-checking and rejects ill-typed programs.

We write $e|_s$ to indicate the subexpression of $e$ at node $s$. Note that this implies $e|_s = s$ for all suitable $e$ and $s$. Every expression is trivially a subexpression of itself. We may also say that $e$ is a context in which $s$ appears. The $i^{\text{th}}$ successor of $e$ is written $e_i$. We shall assume this is well-formed wherever it appears: i.e., that $i$ is less than the arity of the label of $e$. An expression is said to be *linear* when the subgraph it roots is a tree. Given expressions $e|_s$ and $t$, the replacement of $s$ by $t$ is written $e[s \leftarrow t]$. The next example demonstrates.

**Example 3.1.1** (**Replacement**)**.** Let expression $e$ be defined as shown below:




$e$

Let $f$ and $x$ be the subexpressions of $e$ labeled `f` and `x`, respectively, let $h =$ `h`, and let $u =$ `f h`. Two replacements in $e$ are shown below:


$e[x \leftarrow h]$ $\qquad$ $e[f \leftarrow u]$

– *end example.*

A *rewrite rule*, $l \rightarrow r$, consists of a pattern and an expression in which every variable (reference) that appears in the right-hand side also appears (i.e., is introduced) in the left-hand side. A pattern is a linear expression whose root is labeled by a function symbol and that otherwise contains only constructor and variable symbols.

**Example 3.1.2.** Function `head` (see Sect. 2.3.2) can be represented by the rewrite rule `head (a:_)` $\rightarrow$ `a`. The pattern is rooted by a node labeled with the function symbol `head` and otherwise contains only nodes labeled by a constructor symbol (`:`) or variable (`a` and `_`). Variable `a`, referenced in the replacement, is introduced in the pattern. – *end example.*

A *substitution*, $\sigma$, is a well-typed mapping from variables to expressions. We write $e\sigma$ to denote the application of $\sigma$ to an expression $e$, which replaces in $e$ each variable in the domain of $\sigma$ with the corresponding expression from $\sigma$. For example, taking $e$ to be the expression from Exam. 3.1.1 and using the substitution $\sigma = \{\texttt{x} \mapsto \texttt{h}\}$,

the application $e\sigma$ is equivalent to the replacement $e[\texttt{x} \leftarrow \texttt{h}]$, shown in the example. A substitution is *complete* for an expression if it replaces every variable in that expression. An instantiation of a rewrite rule, $l \rightarrow r$, is a pair $(l\sigma,\ r\sigma)$, where $\sigma$ is a complete substitution for the left-hand side. Two instances of the rule defining `head` are (`head [1]`, `1`) and (`head [True, False]`, `True`). The rule is applied in $e$ by choosing a (function-rooted) subexpression, $e|_f$; finding an instantiation of the rule such that $f = l\sigma$; and then performing the replacement $e[f \leftarrow r\sigma]$.

A *rewrite system*, $\mathcal{R} = \langle \Sigma, \rightarrow_\rho \rangle$, consists of a signature and a rewrite relation, $\rightarrow_\rho$, defined over a set of rewrite rules, $\rho$, conforming to that signature. The rewrite relation is formed as follows. A rewrite rule induces a binary rewrite relation over all its instantiations. The collection of all pairs over all instances constitutes the rewrite relation for that rule, which may be finite or infinite. We write $\rightarrow_\rho$ to represent the union of all such relations over every rule in $\rho$, or simply $\rightarrow$ when the rule set is implied.

In the context of a rewrite system, the application of a rewrite rule is called a *rewrite step*. Given an expression $e$, a *reducible expression* or *redex* of $e$ with respect to $\mathcal{R}$ is a subexpression of $e$ at which a rewrite step of $\mathcal{R}$ can be performed. A finite or infinite sequence of rewrite steps, $e_0 \rightarrow e_1 \rightarrow ...$, is called a *rewriting computation* or *derivation*. Each $e_i$ is said to be a *state* of the computation. A *normal form* is an expression containing no redexes. A *value* is a normal form that is also a ground constructor expression. The transitive closure of the rewrite relation ($\rightarrow^*$) can be

used to write the derivation of a value, as in $e \to^* v$.

**Example 3.1.3** (**Rewrite System**)**.** Consider the following Curry definitions:

```
1 data Bool = False | True
2 id x = x
3 not True = False
4 not False = True
```

This defines constructor symbols $\mathcal{C} = \{\texttt{False}, \texttt{True}\}$, function symbols $\mathcal{F} = \{\texttt{id}, \texttt{not}\}$, and signature $\Sigma = \mathcal{C} \cup \mathcal{F}$. The set of rewrite rules can be written as follows:

$$\rho = \{(\texttt{id x} \to \texttt{x}),\ (\texttt{not True} \to \texttt{False}),\ (\texttt{not False} \to \texttt{True})\}$$

This induces the rewrite relation partially shown below[2]:

$$(\to_\rho)\ \supset \{(\texttt{id True}, \texttt{True}),\ (\texttt{id False}, \texttt{False}),$$
$$(\texttt{not True}, \texttt{False}),\ (\texttt{not False}, \texttt{True})\}$$

The rewrite system defined by this program is $\mathcal{R} = \langle \Sigma, \to_\rho \rangle$.

Given the expression $e = \texttt{not (id True)}$, the rewrite rule $\texttt{id x} \to \texttt{x}$ can be applied at $e|_s = \texttt{id True}$ with substitution $\sigma = \{\texttt{x} \mapsto \texttt{True}\}$. The substituted RHS is $\texttt{x}\sigma = \texttt{True}$ and the result is given by $e[s \leftarrow \texttt{True}] = \texttt{not True}$. Continuing in this way, we derive a value by the computation $e \to^* \texttt{False}$. – *end example.*

[2]Note that there are infinitely-many more instances of `id`. One of these, for instance, is `(id (not True), not True)`.

### 3.1.2 LOIS Systems

The class of rewrite systems accepted by the FS is crucial to its suitability for use in a Curry compiler, its relative simplicity, and its provable correctness and efficiency. Curry provides a variety of high-level features that ease programming. Many of these do not appear in the above definition of rewrite systems. For instance, functional patterns (see Sect. 2.3.4) do not meet the above definition of patterns. Similarly, other features such as list comprehensions, partial applications, lambda functions, and modules are conveniences for Curry programmers but obstacles for a compiler writer. Extending our notion of a rewrite system to accommodate every such feature would represent a significant and unnecessary complication. Instead, the FS is based on a relatively simple class of rewrite systems that Curry programs can be mapped into called Limited Overlapping Inductively Sequential (LOIS). This class, though simple, is powerful enough to accommodate all of the programs we are interested in.

Studies finding LOIS systems suitable for functional logic programming [45, 74] motivate our decision to use them as a core language for Curry. We recall the following results, which were previously summarized in the publication of the FS [62] and are reproduced verbatim here:

1. Any LOIS system admits a complete, sound and optimal evaluation strategy [43];

2. Any constructor-based conditional rewrite system is semantically equivalent to a LOIS system [75];

3. Any *narrowing* computation in a LOIS system is semantically equivalent to a *rewriting* computation in another similar LOIS system [41]; and

4. In a LOIS system, the order of execution of disjoint steps of an expression does not affect the values of the expression [43, 58].

LOIS systems are in other words general enough to perform any computation needed for Curry [75] and powerful enough to do so via simple rewriting [41, 76] without performing unnecessary steps [43] nor depending on the order of evaluation [43, Lemma 20].

The mapping to LOIS takes place via a series of transformations that includes lambda lifting [77], elimination of partial applications and high-order functions [78], removal of overlapping rules and rule conditions [75], and replacement of free variables with generator functions [41, 76]. This approach greatly simplifies the implementation of a Curry compiler by reducing the number of features the compiler must implement and cleanly separating source programs into deterministic and non-deterministic parts. The crucial aspect of LOIS, in the latter regard, is the way in which it isolates non-determinism.

All non-determinism in a LOIS system is represented as applications of the choice operator (see Sect. 2.3.1), whose definition we recall here:

```
1  x ? _ = x
2  _ ? y = y
```

Overlap is "limited" in the sense that the only overlapping rules allowed are these two. Part of the transformation from Curry into LOIS involves reformulating the non-determinism of a source program as choices. This greatly simplifies one of the most challenging aspects of a Curry compiler — namely, implementing the various forms of non-determinism — by reducing it to a single representation in the core language that can be dealt with uniformly.

Every other operation in a LOIS system is characterized by a *definitional tree* [79], which provides a hierarchical organization of its rewrite rules. Definitional trees guide pattern-matching and rewrite steps and, in doing so, completely characterize the deterministic parts of a Curry program. The following definition recalls this concept:

**Definition 3.1.4** (**Definitional Tree**)**.** $\mathcal{T}$ is a *partial definitional tree* (pdt) if and only if one of the following holds:

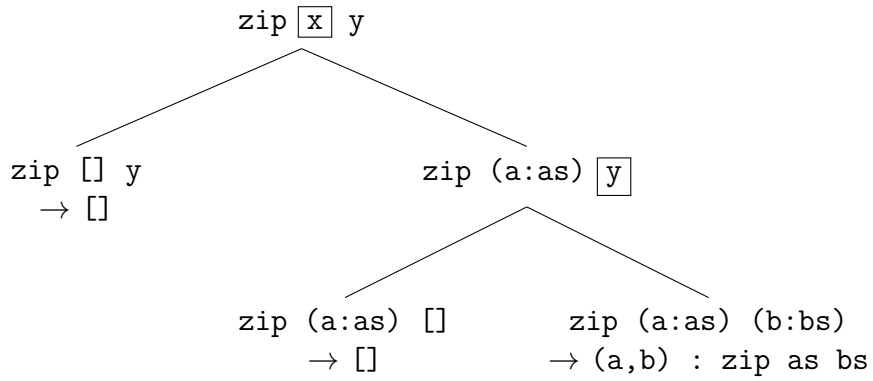
$\mathcal{T} =$ $branch(\pi, o, \overline{\mathcal{T}})$, where $\pi$ is a pattern; $o$ is a subexpression of $\pi$ labeled by a variable of type $T$ having $n$ constructors; and $\overline{\mathcal{T}}$ is a sequence of $n$ pdts such that the $i^{\text{th}}$ pdt has pattern $\pi[o \leftarrow c_i(\overline{x})]$ for fresh variables $\overline{x}$, where $c_i$ is the $i^{\text{th}}$ constructor of $T$;

$\mathcal{T} =$ $rule(\pi, l \to r)$, where $\pi$ is a pattern and $l \to r$ is a rewrite rule such that $l = \pi$ modulo a renaming of variables and nodes; or

$\mathcal{T} =$ $exempt(\pi)$, where $\pi$ is a pattern.

$\mathcal{T}$ is a *definitional tree* if and only if it is a pdt with `f` $\overline{x}$ as the pattern argument such that no $\overline{\mathcal{T}}$ of any branch in $\mathcal{T}$ consists entirely of exempt nodes. – *end definition.*

**Example 3.1.5.** A definitional tree for `zip` (see Sect. 2.3.1) is shown below:




We see in Def. 3.1.4 that every pdt node contains a pattern, so we label every node above with the pattern of the pdt it represents. For branches, the inductive position is indicated by a box and the elements of $\overline{\mathcal{T}}$ are its children. When a node occurs at an inductive position, it is referred to as an inductive node. For leafs, the right-hand

side is shown below the pattern. Left-hand sides are not shown because the patterns were chosen so that $\pi = l$. The root is a branch with pattern $\pi =$ `zip x y` and inductive position `x`. Of its children, the first (left) child is a rule node with pattern `zip [] y` and rewrite rule `zip [] y` $\rightarrow$ `[]`. – *end example.*

A function is called *inductively sequential* if there exists a definitional tree encompassing all and only its defining rules [79]. Aside from the choice operator, every function in a LOIS system is inductively sequential. To concisely distinguish them from choices, we shall refer to the inductively sequential functions of a source program as its *operations*, reserving the more general term "function" for operations and choices taken together. Just as all non-determinism in a LOIS system is embodied by the choice operator, all its deterministic meaning is expressed as inductively sequential operations. This leads us to the following definition of LOIS systems:

**Definition 3.1.6** (**LOIS System**)**.** A LOIS system is a constructor-based graph rewriting system in which every function in the signature is either inductively sequential or is the choice operator. – *end definition.*

## 3.2 Non-Determinism Strategy

A computation by the FS involves two types of steps. The first is the rewrite step, which was already discussed. This step applies the deterministic rules of inductively sequential operations via alternating phases of pattern matching and replacement

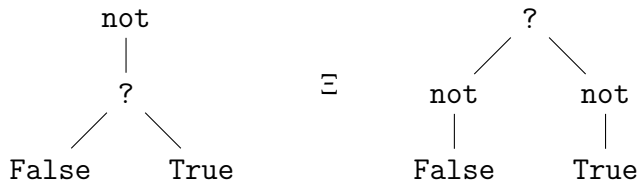
that follow their definitional trees. The second type of step, called a *pull-tab*, is used to process choices, whose overlapping rules are never applied. In Sect. 3.2.1, we define the pull-tab step and extend our concept of computation to accommodate it. Then, in Sect. 3.2.2 we discuss the issue of making consistent choices, which arises as a consequence of pull-tabbing. Finally, in Sect. 3.2.3 we discuss the handling of free variables.

### 3.2.1 Pull-Tabbing

Choices are the universal, structural representation of non-determinism in the FS. Pull-tab steps are a manifestation of the additive (see Sect. 2.2.4) property applied to that representation. Pull-tabbing allows computations to proceed in the presence of choices without ever applying overlapping rules[3]. A choice can prevent rewriting when it occurs at an inductive position relative to a node labeled by an inductively sequential function. Pull-tabbing is a value-preserving transformation that moves such choices out of the way so that rewriting may proceed. The effect is to transform a function applied to a superposition into a superposition of function applications. The intuition behind this name is that the step behaves like pulling on the tab of a zipper, as applying a pull-tab step "unzips" an expression.

A simple example will help demonstrate. A pull-tab transformation ($\Xi$) for the expression `not (False ? True)` is shown below:

[3]Recall from Sect. 2.2.4 that applying an overlapping rule alters the corresponding value state.




Prior to this (left), no rule of `not` can match because its argument contains a function symbol (the choice operator). The replacement (right) contains two redexes. On the left side, the node labeled `not` is called the *target* and the node labeled `?` is called the *source*. Pull-tabbing exchanges the positions of these and, in doing so, creates a node (compare the number of nodes before and after). To assist with our definition of pull-tabbing we define the shallow copy-replace relation shown in Def. 3.2.1.

**Definition 3.2.1** (**Shallow Copy-Replace**)**.** Let $e$ be an expression labeled by a symbol $f$ with arity $n$. Let path $p$ be a list of integers with construction $(i : j)$ and empty list $[\,]$. The shallow copy of $e$ with replacement $r$ at $p$ is given by $\mathrm{copyr}(e, p, r)$, where copyr is defined as:

$$\begin{aligned} &\mathrm{copyr}(e, [\,], r) && = r \\ &\mathrm{copyr}(e, (i : j), r) && = f \;\; e_1 \ldots e_{i-1} \;\; \mathrm{copyr}(e_i, j, r) \;\; e_{i+1} \ldots e_n \end{aligned}$$

– *end definition.*

The pull-tab transformation generates a binary relation over the expressions of the signature of a source program. In this way it is similar to a rewrite step, though it serves a different purpose. The previous example shows one instance, and the set of all

instances forms the relation. In the context of a rewrite system, this relation defines pull-tab steps. This is similar to the way that the rules of a rewrite system induce a rewrite relation (see Page 85). For an expression $e$ with a suitable subexpression, $u$, such that $u \; \Xi \; v$ is an instance of the pull-tab transformation, then a pull-tab step at $e|_u$ is defined as $e[u \leftarrow v]$. The next definition makes this precise.

**Definition 3.2.2** (**Pull-Tab Step**)**.** Let $e$ be an expression of a LOIS system with subexpression target $t$, such that a successor of $t$ is a choice-rooted source $t|_s = l \; ? \; r$. A pull-tab step at $e|_t$ is described by $e \; \Xi \; e[t \leftarrow L \; ? \; R]$ where the left- and right-side replacements are given by $L = \mathit{copyr}(t, s, l)$ and $R = \mathit{copyr}(t, s, r)$, respectively. – *end definition.*

Extending our definition of computation, we write $\xrightarrow{\Xi}$ to represent the application of either a pull-tab or rewrite step. A pull-tabbing computation (or derivation), written $e_0 \xrightarrow{\Xi}{}^* \; e_n$, is a sequence of any number of rewrite and pull-tab steps.

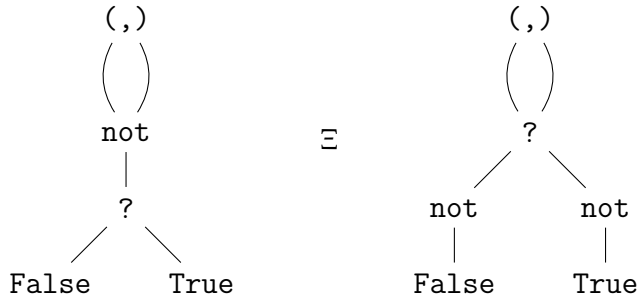
**Example 3.2.3.** Consider the expression `id (not (False ? True))` in which a pull-tab step can be found with context rooted by `id`, target by `not`, and source by `?`. The left-side replacement, $L =$ `not False`, is found by copying the target and replacing its argument with the left alternative of the choice, `False`. The right-side replacement is computed similarly. Applying the pull-tab step yields `id((not False) ? (not True))`. – *end example.*

We note in passing that the definition of a path given in Def. 3.2.1 is borrowed from term rewriting. The customary definition for term graphs, due to Echahed and Janodet [71], is not suitable for defining the pull-tab transformation. A path by their definition is a sequence of nodes, $s_1, \ldots s_n$, such that for all $i = 1, \ldots, n-1$, $s_{i+1}$ is a successor of $s_i$. That definition is suitable for rewriting. For instance, in a rewriting computation of `(x, x) where x = 0 ? 1`, we might be interested in a rewrite step that produces `(0, 0)`. The position of node `x` is not needed to describe that step. A pull-tabbing transformation, on the other hand, involves intermediate expressions such as `(0, x)` and `(1, x)` that are most easily described using indices.

In the preceding example function `id` was included in order to demonstrate the role the context plays as a place were replacement occurs. When the context is not important, we may omit it with the understanding that *some* context always exists for a step to occur. One may use this context-free notation to more simply write, for instance, `[a ? b, a]` $\overset{=}{\rightarrow}$ `[a, a] ? [b, a]`. Note that this twice copies a *cons* node but does not copy anything else. In particular, none of `a`, `b`, or the list tail `[a]` is duplicated by this step.

In Def. 3.2.2, the path from the root of the context to the target is not specified. This is intentional because the path is irrelevant — any one will do. When a context contains multiple occurrences of the target of a pull-tab step, that step is shared. Consider the following expression in which the target of a pull-tab appears twice: `let x=not (False ? True) in (x, x)`. A pull-tab step in this expression is shown

below:



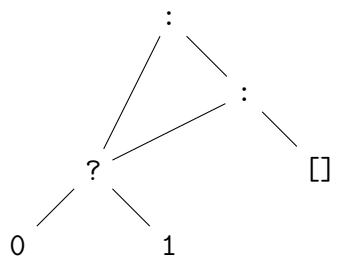

Ensuring this is done correctly contributes to the efficiency of an implementation of the FS. In contrast, when a choice has multiple predecessors, pull-tabbing duplicates the choice. This leads to a consistency issue that is the topic of the next section.

### 3.2.2 Consistency

To understand the hazard that comes from pull-tab steps duplicating choices, consider the following expression:

```
let d = (0 ? 1) in [d, d]
```

Since `d` is shared, this expression contains only one choice. This can be seen more clearly in its graphical representation:



Accordingly, the only values are `[0, 0]` and `[1, 1]`. Now, study the following pull-tabbing derivation (in which the source of each pull-tab is underlined):

```
[0 ? 1, 0 ? 1]   ⇉  [0, 0 ? 1] ? [1, 0 ? 1]
                 ⇉  [0, 0] ? [0, 1] ? [1, 0 ? 1]
                 ⇉  [0, 0] ? [0, 1] ? [1, 0] ? [1, 1]
```

The result of this contains three choice symbols and has a value set different from the original expression.

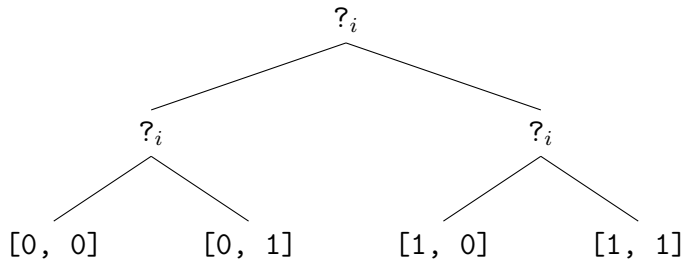
The above derivation leads to the spurious values `[0, 1]` and `[1, 0]` and is therefore unsound. This aspect of pull-tabbing has been noted previously [47, 52]. If, instead of pull-tabbing as above, we were to apply the rewrite rules of a choice — that is, arbitrarily replace the choice with one or the other of its subexpressions — the structure of this expression would ensure that that selection was reflected everywhere. When a pull-tab step duplicates a choice, however, the original and the copy can assume different values. This leads to the spurious values seen above. We say these are the result of an *inconsistent* handling of the choices, since they arise only when one mixes the left and right alternatives of the same choice. Fortunately, a simple countermeasure exists.

We shall annotate each choice with information called a *choice identifier* (cid) that uniquely identifies it. Identifiers are used to correlate all copies of the same choice, ensuring they all choose the same alternative. When a fresh choice is placed into service, it is labeled with a fresh identifier, and subsequent pull-tab steps preserve

the identifier when duplicating choices. With this in place, to ensure consistency one needs only to discard values that are reached via an inconsistent traversal of choices: i.e., ones that combine the left and right alternatives of the same choice. This is formalized in the next definition (*cf.* [52, Def. 4]).

**Definition 3.2.4** (**Consistent Computation**)**.** Let $C$ be a rewriting computation in a LOIS system. We say a step in $C$ is a *choice step* if the redex it replaces is rooted by the choice symbol. Let $\rho_{\mathrm{L}}$ and $\rho_{\mathrm{R}}$ designate the rules of choice that select the left and right alternatives, respectively. $C$ is a *consistent computation* iff for every choice $?_i$ reduced by a step in $C$ there exists an $\alpha \in \{\mathrm{L}, \mathrm{R}\}$ such that every choice step in $C$ that reduces a choice with cid $i$ applies rule $\rho_\alpha$. – *end definition.*

To illustrate, consider the following representation of the final expression derived above, now with cids included, and reformatted to more clearly show its structure.




All three choice nodes share one identifier since they originated from the same choice. Every traversal from root to leaf crosses two choices with the same annotation. Hence, the consistent values are those reachable by always following the left branches or

always following the right branches. These are `[0, 0]` and `[1, 1]`, the intended values. The spurious values can only be reached via inconsistent traversals. For instance, `[0, 1]` can only be reached by first taking the left branch of choice $i$ at the root and then the right branch of choice $i$ one rank down.

### 3.2.3 Elimination of Free Variables

The only computational steps we have defined are the rewrite and pull-tab, and our notion of value does not admit variables. How, then, should the FS approach the following expression?

```
(x::Bool) where x free
```

This is a normal form but not a value, and it contains no choices that might let us execute a pull-tab step. There is seemingly no way for the FS to make further progress, yet this expression has the values `False` and `True`. Similar problems arise whenever free variables appear. The FS deals with free variables by replacing them with *generator functions*, which were introduced in Sect. 2.3.1. The next definition makes this concrete.

**Definition 3.2.5** (**Generator Function**)**.** A generator function for type $T$ is a non-deterministic operation of arity zero that produces the superposition between all and only the values of $T$. – *end definition.*

A generator for the `Bool` type, called `boolGen`, was shown on Page 61. To compute the above expression, a Curry compiler based on the FS replaces `x` with `boolGen` at compile time. A pull-tabbing computation then obtains the correct values.

This replacement of free variables is justified by the results of Dios Castro and López-Fraguas [76], who show it does not change the values derived via rewriting, and Antoy and Hanus [41], who show that overlapping rules and free variables can each be expressed solely in terms of the other. These results are crucial to our approach, since they guarantee the transformation from Curry to LOIS is possible for any source program. Taken together, they show that any non-determinism arising from overlapping rules in the source program can be expressed as free variables and that all free variables can be eliminated by replacing them with generator functions utilizing choices. In this way, the non-determinism of any Curry program can be reduced solely to applications of the choice operator.

A drawback is that variable replacement can make certain finite computations infinite. For example, the expression `(x:[Bool]) where x free` produces a superposition over all Boolean lists. In a computational model that admits variables in values, all values can be written simply as `x`. With the FS, however, this becomes an infinite computation. While this does not impact correctness — for every pure state $|v\rangle$ in the original expression, a computation by the FS will eventually produce a value corresponding to $|v\rangle$ — it raises an obvious practical concern. Free variables will be readmitted in the next chapter to a variation of the FS that performs narrowing

computations.

### 3.3 Definition of the Fair Scheme

We are now ready to define the FS. The starting point is a LOIS system $\mathcal{S}$ called the *source program*. Except for the choice operator, each function is represented by its definitional tree. The program signature is extended to $\Sigma^* = \Sigma \cup \{\bot\}$, where $\bot$ is a reserved symbol representing failure, and this set of symbols is used to label nodes. The failure symbol does not appear in expressions of the source program but rather is introduced during evaluation to denote failed computations. A source program is compiled according to the FS into a *target program* consisting of the following three abstract procedures:

**D** (Dispatch): Processes the disjoint branches of a computation.

**N** (Normalize): Normalizes an expression.

**S** (Step): Applies a rewrite step.

The **D** and **N** procedures are fixed for all source programs. The rules of **S** are defined inductively on the definitional trees in $\mathcal{S}$. Generating this latter set of rules is one of the main activities that a Curry compiler based on the FS must perform. The procedures are mutually recursive and ordered: each procedure may invoke itself on one or more subexpressions or invoke a lower (in the order they are listed above) procedure.

A Curry program is run by applying **D** to an expression called the goal and ends when **D** completes. This arrangement makes creating an implementation based on the FS relatively simple. To build a Curry program, one needs only to place implementations of the **D** and **N** procedures into a static library (along with other elements of the runtime system), compile the program-specific rules of **S**, and then link these together.

The FS computes by repeatedly applying steps until a normal form is derived. If that normal form is a value, it is yielded to an outside observer such as a read-eval-print loop. Since functional-logic computations are multi-valued, producing one value does not terminate the computation. Further evaluation may yield additional values, terminate, or diverge into a fruitless non-terminating computation. It is therefore not practical to collect values; rather they should be acted on as they are produced. As such, a computation by the FS is best seen as a coroutine that, given compute resources, might eventually produce an additional value and might terminate, though neither is guaranteed.

The FS compiler is defined in Figure 3.1. Each target procedure is defined using pattern matching to select amongst its defining rules. The rules are ordered and matching occurs in order. For this reason, more general rules are listed after more specific ones. Rule **N**.3, for instance, which matches any expression rooted by constructor $c$, appears after rule **N**.2, which matches any expression rooted by $c$ that also has a choice-rooted successor.

The running program is stored in a queue, represented by $\bar{G}$, that manages a set of live computations. We write $(\bar{G}; g)$ and $(g; \bar{G})$ to construct and destruct queues. The empty queue is written *null*. For our purpose in this chapter, which is only to describe and prove properties of the FS, the queue elements are simply expressions. Later, we will see that an implementation might attach additional data structures to queue elements.

The queue contains roughly the expressions with the potential to produce values; more precisely, it contains those expressions not known for certain to be failures. These are called the *live* computations (or expressions) of the program. To evaluate a goal, one constructs an initial state in which $\bar{G}$ contains only the goal and then applies the **D** procedure. Target procedures apply rewrite and pull-tab steps to the expression at the head of the queue. Over time, pull-tabbing causes deep-occurring choices (if they are consequential) to migrate towards the root. Additional elements are created by splitting live expressions when they become choice-rooted. Due to this mechanism, distinct elements hold expressions that are not necessarily connected, but may share subexpressions; thus, the queue in general holds the roots of a many-rooted, disconnected graph.

Queue elements are destroyed once they become a value (after that value is yielded to the consumer) or their computation fails. Fairness is ensured by appending elements to the end of the queue after each step. This ensures that computational resources are distributed over the live expressions so that none is neglected forever.

We will discuss in Chapter 5 how an implementation may modify this policy to balance operational efficiency with completeness.

**D** manages the queue. It removes the first element, inspects it, and then takes the appropriate action. If it is the result of an inconsistent traversal of choices or a failure, it is discarded (rules **D**.1 and **D**.3, respectively). If it is a choice, then the computation forks and the two alternatives are appended to the queue as independent tasks (rule **D**.2). If it is a value, it is yielded to an outside consumer (rule **D**.4). Otherwise, the **N** procedure is applied to carry out a finite number of steps in the hope that one of the previous cases will match the next time this expression comes to the head of the queue (rule **D**.5). After each step, the expression at the head of the queue is rotated to the end so that another has the opportunity to proceed. If the queue is ever exhausted, the computation ends (rule **D**.6).

**N** attempts to normalize an expression. It recursively traverses constructor expressions, either applying steps that do not depend on the rules of the source program or invoking **S** to apply ones that do. **N** returns a Boolean value indicating whether its argument cannot be derived to a value and may therefore be discarded. Rule **N**.1 returns *false* for constructor expressions found to contain the failure symbol. Choices found in constructor expressions are subjected to a pull-tab step (rule **N**.2). Rule **N**.3 recurses through constructor expressions. Function-rooted subexpressions are in rule **N**.4 passed to **S** for program-specific processing.

**S** applies the rules of inductively sequential operations in $\mathcal{S}$. These are generated

from definitional trees with the help of the abstract procedure `compile`. A function `f` of $\mathcal{S}$ with definitional tree $\mathcal{T}_{\mathtt{f}}$ is compiled into one or more **S** rules specific to `f`, which we call $\mathbf{S}_{\mathtt{f}}$ rules, each of whose patterns begins with the symbol `f`. More specific rules are generated first and prevent the application of more general ones, when possible. If $\pi$ is the pattern of a pdt in $\mathcal{T}_{\mathtt{f}}$, then its children include only patterns that are instances of $\pi$. This ensures that the post-order traversal of $\mathcal{T}_{\mathtt{f}}$ performed by `compile` generates these rules in the desired order. Rule **S**.1 reduces the `f`-rooted redexes found in the leafs of $\mathcal{T}_{\mathtt{f}}$. Rule **S**.2 reduces pattern-matching failures to computational failures by rewriting to the failure symbol, $\bot$. Rule **S**.3 likewise reduces to failure expressions in which a failed computation appears at an inductive position. Rule **S**.4 performs a pull-tab step when a choice appears in such positions and rule **S**.5 recursively invokes **S** when a function-rooted expression does. Finally, rule **S**.6 applies when an `f`-rooted expression has already been reduced to constructor-rooted form. This can occur in the presence of shared subexpressions. For instance, if rule **N**.3 applies (indirectly) **S** to different occurrences of the same subexpression.

### 3.4 Properties of the Fair Scheme

We now turn to proving the correctness and optimality of the FS. We begin with a series of preliminary definitions and results needed to build up to that.

### 3.4.1 Preliminaries

The seminal concept of *needed redexes* [70] frames efficiency analyses of rewriting computations in orthogonal rewriting systems. In the context of term rewriting systems, a redex is a subexpression that can be reduced by a rewrite rule. A needed redex is a redex that must be reduced in order to reach a normal form, or a simplified version of an expression. Rewriting computations that only reduce needed redexes are considered optimal because they take the minimum number of steps to reach the normal form.

However, the concept of a needed redex is not applicable to LOIS systems, which are not orthogonal due to the presence of the choice operator. To analyze the computations dictated by the FS, the novel definition of need shown in Def. 3.4.1 was proposed by this author and Antoy [62].

**Definition 3.4.1** (**Need**)**.** Let $e$ be an expression of a LOIS system with subexpression $n$. The *need relation* is defined as follows:

1. Node $n$ is needed for $e$ if and only if in any derivation of $e$ to a constructor-rooted form, $e|_n$ is derived to a constructor-rooted form; and

2. Node $n$ is needed for $e$ if and only if it is needed for some maximal function-rooted subexpression of $e$.

A computation $e_0 \to e_1 \to \ldots$ is needed if and only if it reduces only needed redexes; that is, if and only if each redex is related to the root by the need relation. – *end definition.*

In contrast to the classic definition, Def. 3.4.1 is a relation between two expressions. The first case defines one needed expression in terms of another. If $e$ is needed, then whichever of its subexpressions are perforce reduced when reducing $e$ are needed for it. The second case provides the basis, establishing which nodes of a goal expression are needed in the first place. Any function-rooted expression is itself needed. In a constructor-rooted expression such as `(f, g h)`, the maximal function-rooted expressions — in this case, `f` and `g` — are needed. Note that `h` is not needed, and that for certain definitions of these symbols, such as `g _ = True`, evaluating it could represent wasted effort.

The need relation is reflexive, meaning that a function-rooted expression is needed for itself. This applies to both redexes and non-redexes. For example, in `head []`, the expression itself is considered to be needed, despite the fact it is not a redex. In general, any expression that is not a value contains at least one needed subexpression. The FS involves finding and reducing these needed subexpressions. It is important to establish the need relation for more than just redexes, as shown in the example above, because we may want to perform an action on a non-redex, such as replacing it with the failure symbol. The next definition formalizes the concept of failure.

**Definition 3.4.2** (**Failure**)**.** Let $e$ be an expression of a LOIS system. We say $e$ is a failure if and only if there exists no derivation of $e$ to a constructor-rooted form. – *end definition.*

Def. 3.4.2 is more permissive than the definition of failure presented in Sect. 2.3.2. For example, according to Def. 3.4.2, `[head []]` cannot be derived to a value, but is not considered a failure. In general, detecting failure in computations is undecidable. However, failure can occur in rewriting computations due to unsuccessful pattern matches, which are represented by *exempt* nodes in definitional trees. When it does, the FS marks the failing computation with the failure symbol, $\perp$.

The definition of need given in Def. 3.4.1 is transitive [62, Lemma 1]. This allows us to form chains from needed subexpressions to deeper needed subexpressions. Also, rewriting steps guided by definitional trees are applied only at needed nodes. This result is recalled in the next lemma.

**Lemma 3.4.3** (**Needed Inductive Positions**)**.** Let $\mathcal{S}$ be a source program, $e$ a function-rooted expression of $\mathcal{S}$, and $\mathcal{T}$ a definitional tree of that function. If there exists a branch node of $\mathcal{T}$ with pattern $\pi$ and inductive node $o$, and a match-making substitution $\sigma$ such that $\pi\sigma = e$ and $o\sigma = p$ for some function-rooted subexpression $e|_p$, then $p$ is needed for $e$.

*Proof.* See [62, Lemma 4]. □

To prove the properties of the FS we shall rely on *call trees*. A call tree is a tree whose branches represent calls to a target procedure and whose leafs represent steps of the source program. For a goal expression, $e$, evaluated by the FS, a left-to-right traversal of the call tree of $\mathbf{D}(e)$ is equivalent to the pull-tabbing computation of $e$ by the FS. We refer to this as the *simulated computation* of $e$. The next definitions formalizes these concepts (*cf.* [62, Def. 10]).

**Definition 3.4.4** (**Call Tree**)**.** Let $X$ be an invocation of **D**, **N**, or **S**. A call tree rooted by $X$, denoted $\Delta X$, is defined as follows:

| | |
|---|---|
| $\Delta X = X$ | if $X$ is a null action, rewrite step, or pull-tab step; |
| $\Delta X = X\ \overline{\Delta Y}$ | otherwise, if X executes a sequence of actions $\overline{Y}$. |

The notation $X\ \overline{\Delta Y}$ indicates a branch labeled $X$ with children $\Delta Y_1, \ldots, \Delta Y_n$. – *end definition.*

**Definition 3.4.5** (**Simulated Computation**)**.** Let $\mathcal{S}$ be a source program, **D** the dispatch procedure of the corresponding target program, and $e$ an expression of $\mathcal{S}$. The simulated computation of $e$ in $\mathcal{S}$, denoted $\omega(\mathbf{D}(e))$, is a left-to-right traversal of the leafs of $\Delta\mathbf{D}(e)$. – *end definition.*

**Example 3.4.6** (**Call Tree**)**.** Figure 3.2 shows the topmost portion of the call tree for the expression introduced in Sect. 3.2.2. Each node is labeled with the expression in $\bar{G}$ at the corresponding point in the computation. The graphical representation

emphasizes the relationship between shared subexpressions following the pull-tab step directed by **N**.2. Where relevant, edges are labeled with the corresponding step from Figure 3.1. – *end example.*

### 3.4.2 Correctness

Our evaluation strategy is designed to be efficient by computing only some of the steps possible for an expression. The way this is done is important, as it can affect the correctness of the values produced. A correct evaluation strategy is one that is both sound and complete. A sound strategy applied to an expression will only produce the values that are intended for that expression, while a complete strategy will produce all of the intended values for the expression.

These concepts are formalized in Theorem 3.4.7 which states that if a value can be obtained from an expression by pure rewriting, then that value can also be obtained by applying pull-tab steps to the expression and then rewriting. Additionally, if a value can be obtained by applying pull-tab steps to an expression and then rewriting, then that value can also be obtained by pure rewriting on the original expression. In other words, the interposition of pull-tab steps does not drop any values or create spurious ones.

**Theorem 3.4.7** (**Correctness of Pull-Tabbing**)**.** Let $g$ be an expression of a LOIS system. If $g \xrightarrow{\sqsupset}{}^{*} g'$ is a pull-tabbing computation with no choice steps, then:

(1) for any value $v$ such that $g \to^* v$ is a consistent computation (see Def. 3.2.4), there exists a value $v'$ such that $g' \to^* v'$ is a consistent computation and $v = v'$ (modulo a renaming of nodes); and

(2) for any value $v'$ such that $g' \to^* v'$ is a consistent computation, there exists a value $v$ such that $g \to^* v$ is a consistent computation and $v = v'$ (modulo a renaming of nodes).

*Proof.* See [52, Theorem 1]. □

Theorem 3.4.7 is useful because it shows that a correct computation need only apply rewrite steps according to the definitional trees of inductively sequential functions and apply pull-tab steps to choices without regard to the order and location of these steps.

With this result in hand, proving the correctness of the FS is a relatively straightforward matter as one needs only to show that the FS performs pull-tabbing computations. This was first done in [62, Lemma 6 and Corollary 2]. We shall next prove two preliminary results and then show the correctness of the FS.

**Lemma 3.4.8** (**Finiteness of $\Delta\mathbf{S}$**)**.** Let $\mathcal{S}$ be a source program, $\mathbf{S}$ the step procedure of the corresponding target program, and $e$ an expression of $\mathcal{S}$. The call tree $\Delta\mathbf{S}(e)$ is finite.

*Proof.* The proof is by inspection over the rules of **S** in Figure 3.1. The call tree has a single child that is either a step of $\mathcal{S}$ (rules **S**.1-4) or the recursive application $\mathbf{S}(e|_o)$, where $o \neq e$ (rule **S**.5). Each recursive invocation is applied to a strictly smaller argument and the original expression is finite and acyclic (see Sect. 3.1.1). □

The finiteness of $\Delta\mathbf{N}$ can be easily proved by a similar argument (see also [62, Corollary 1]).

**Lemma 3.4.9** (**Simulation**)**.** Let $\mathcal{S}$ be a source program, **D** the dispatch procedure of the corresponding target program, and $e$ an expression of $\mathcal{S}$. The simulated computation $\omega(\mathbf{D}(e))$ is a pull-tabbing computation of $e$.

*Proof.* By inspecting Figure 3.1 and noting that $\Delta\mathbf{N}$ and $\Delta\mathbf{S}$ are always finite (Lemma 3.4.8 and [62, Corollary 1]), one can readily see that the rightmost path in $\Delta(\mathbf{D}(e))$ contains all and only the applications of **D** because every invocation of **D** is the final action of a rule of **D** (see also [62, Lemma 5]). By [62, Lemma 6], if $\mathbf{D}(L_0)$, $\mathbf{D}(L_1)$, ... is the rightmost path of $\Delta(\mathbf{D}(e))$, then each element of every list $(L_i)$ is a subexpression of a state of the computation of $e$. By inspecting Figure 3.1, one sees that every step in $\omega(\mathbf{D}(e))$ is a rewrite or pull-tab step of $\mathcal{S}$, which, by the above argument, is applied to an expression in the state of the computation of $e$. This sequence of steps is therefore a pull-tabbing computation of $e$. □

**Theorem 3.4.10** (**Correctness of the Fair Scheme**)**.** Let $\mathcal{S}$ be a source program, $\mathbf{D}$ the dispatch procedure of the corresponding target program, $e$ an expression of $\mathcal{S}$, and $\omega(\mathbf{D}(e)) = t_0 \;\; \overset{=}{\rightarrow} \;\; t_1 \;\; \overset{=}{\rightarrow} \;\; \ldots$ the simulated computation of $e$. Modulo a renaming of nodes:

(1) if $e \rightarrow^* v$ in $\mathcal{S}$ for some value $v$, and $t_k$ is an element of $\omega(\mathbf{D}(e))$ for some $k \geq 0$, then $t_k \rightarrow^* v$ for some consistent computation in $\mathcal{S}$; and

(2) if $t_k$ is an element of $\omega(\mathbf{D}(e))$ for some $k \geq 0$ and $t_k \rightarrow^* v$ is a consistent computation in $\mathcal{S}$ for some value $v$ of $\mathcal{S}$, then $e \rightarrow^* v$ in $\mathcal{S}$.

*Proof.* By Lemma 3.4.9, $e \overset{=}{\rightarrow}^* t_k$ defines a pull-tabbing computation of $e$ in $\mathcal{S}$ that executes no choice steps. Both claims then follow directly from Theorem 3.4.7. □

### 3.4.3 Optimality

Our evaluation strategy aims to be efficient by computing the minimum number of steps necessary to produce the values of an expression. Needed steps are considered necessary because they are required to reduce an expression to constructor-rooted form, which means that skipping them could result in the inability to correctly reduce the expression. Therefore, a strategy that only performs needed steps can be considered optimal. In this section, we will demonstrate the optimality of the FS in this sense. The precise meaning of this will be explained in Theorem 3.4.12. The

following preliminary result shows that the **S** target procedure recurses only on operation symbols.

**Lemma 3.4.11** (**S-Recursion on Operation Symbols**)**.** Let $\mathcal{S}$ be a source program, **S** the step procedure of the corresponding target program, and $e$ an operation-rooted expression of $\mathcal{S}$. Let $s$ be an arbitrary invocation of **S**.5 in $\Delta\mathbf{S}(e)$, if any exists. Then in the recursive invocation, $\mathbf{S}(e|_o)$, that occurs in the action of $s$, the subexpression $e|_o$ is labeled by an operation symbol.

*Proof.* For the purposes of this proof, we will assume that $s$ is the closest invocation of **S**.5 to the root of $\Delta\mathbf{S}(e)$. We can then use induction to show that the claim holds for any suitable invocation. We know that every call tree rooted by **S** is linear, because each case of **S** either leads to a leaf in the corresponding call tree (**S**.1-4,6) or is a recursive call (**S**.5). Thus, the recursive calls are ordered. Suppose the first recursive call occurs at subexpression $e|_p$, and we can show that $e|_p$ is operation-rooted. Then, we can apply the same argument to $\Delta\mathbf{S}(e|_p)$ to prove that the second recursive call is also operation-rooted, and we can continue this process for every invocation of **S**.5 in $\Delta\mathbf{S}(e)$.

In order to prove that the subexpression at $e|_o$ is labeled by an operation, we will eliminate the other possibilities: constructor, failure, or choice symbol. Let $\mathcal{T}_{\mathtt{f}}$ be the definitional tree of the operation at the root of $e$. By examining Figure 3.1, we see that **S**.5 is output by compiling a branch of $\mathcal{T}_{\mathtt{f}}$, which we denote $\mathcal{T} = \mathit{branch}(\pi, o', \overline{\mathcal{T}})$.

The inductive position ($o'$) is labeled with a variable (according to Def. 3.1.4) of some type, $T$. For the invocation ($s$) to exist, there must be a match-making substitution ($\sigma$) such that $\pi\sigma = e$, and this means $\sigma$ contains a mapping from the variable at $o'$ such that $o'\sigma = o$. Therefore, we can conclude that $e|_o$ also has type $T$. For every constructor ($c$) in $T$, there is a child in $\overline{\mathcal{T}}$ whose pattern is $\pi[o' \leftarrow c\,\overline{x}]$ (again, according to Def. 3.1.4). If $e|_o$ were labeled by a constructor, then one of these more specific patterns would match, because the cases of $\mathbf{S}$ are generated through a post-order traversal of $\mathcal{T}_{\mathtt{f}}$ and the constructor cases ($\mathbf{S}$.1) are generated before the recursive one ($\mathbf{S}$.5), as shown in Figure 3.1. Thus, $e|_o$ is not labeled by a constructor symbol. It is also not labeled by the failure ($\bot$) symbol, because in that case a preceding instance of $\mathbf{S}$.3 with the pattern $\pi[o' \leftarrow \bot]$ would match. A similar argument excludes $e|_o$ being labeled by the choice (?) symbol, because a preceding instance of $\mathbf{S}$.4 would match. Therefore, through process of elimination, we can conclude that $e|_o$ is labeled by an operation. □

**Theorem 3.4.12** (**Optimality of the Fair Scheme**)**.** Let $\mathcal{S}$ be a source program and $\mathbf{S}$ the step procedure of the corresponding target program. If $e$ is an operation-rooted expression of $\mathcal{S}$ then:

1. $\mathbf{S}(e)$ executes a replacement at some subexpression $e|_n$;

2. node $n$ is needed (Def. 3.4.1) for $e$; and

3. if the step at $n$ is a reduction to $\perp$ then $e$ is a failure (Def. 3.4.2).

*Proof.* Each claim is proved by structural induction on the call tree $\Delta\mathbf{S}(e)$, which is finite by Lemma 3.4.8. The base case occurs when a rewrite step (rule **S**.1), replacement with failure (rules **S**.2-3), or pull-tab step (rule **S**.4) is performed. The inductive case occurs when **S** is applied recursively (rule **S**.5). Rule **S**.6 is excluded by assumption, since $e$ is operation-rooted.

1. The call tree $\Delta\mathbf{S}(e)$ terminates with a single leaf labeled by a rewrite (**S**.1-3) or pull tab (**S**.4) step. The subexpression at which this step occurs is taken to be $e|_n$;

2. If $n = e$ the claim is trivially true, since the need relation is reflexive and $e$ is operation-rooted. Otherwise, the child of $\mathbf{S}(e)$ in its call tree is $\mathbf{S}(e|_o)$ for some node $o$, and $e|_o$ is operation-rooted by Lemma 3.4.11. By Lemma 3.4.3, $o$ is needed for $e$, and by the induction hypothesis, there exists a node $n$ such that $\mathbf{S}(e|_o)$ executes a step at $n$. As the *need* relation is transitive ([62, Lemma 1]), $n$ is needed for $e$.

3. Suppose that $\mathbf{S}(e)$ results in the step $e|_n \to \perp$. Then $e|_n$ is rooted by an operation and is matched by the pattern of some *exempt* node in the corresponding definitional tree. Hence, there are no rules that can reduce $e|_n$ and, since it is

not already constructor-rooted, $e|_n$ is a failure (Def. 3.4.2). By point 2 of this theorem, $n$ is needed for $e$, hence $e$ is a failure.

Since each claim holds separately, the theorem is proved. □

Theorem 3.4.12 shows that the FS is optimal in the sense that every step it takes is needed. However, taking only needed steps does not guarantee the shortest derivation. It is possible to imagine rewrite strategies that omit certain needed steps while still being sound and complete. For example, consider the following function:

```
f = 0 ? f
```

The derivation `f` $\to$ `f` $\to$ `0` performs only needed steps, but `f` $\to$ `0` arrives at the same value in fewer steps. This means that the step `f` $\to$ `f` is unnecessary in some sense. By analyzing the function, we can determine that the only possible value of `f` is `0`. A clever optimizer, therefore, could correctly replace the original definition with this one:

```
f = 0
```

This apparent weakness in the definition of optimality can be seen as an exception that proves the rule. The example relies on determining the values of `f` at compile time, which is generally undecidable. Therefore, the example does not provide a principle that a compiler can consistently apply. In other words, while this type of simplification may be useful in special cases for practical purposes, it should not

be used as the standard against which we compare our general compilation procedure. Instead, we should compare the FS to other strategies that only use decidable properties of a program. We can do this by limiting ourselves to strategies that only consider the left-hand sides of rules. This prevents us from drawing conclusions about the values of a function at compile time, which avoids unhelpful arguments like the one above. With this restriction, we can say that needed steps are unavoidable in the sense that no other admissible strategy could omit them to produce a shorter derivation.

With this, we have defined the FS and established its soundness, completeness, and optimality. In the next chapter, we will describe extensions that change its operational principles to enhance the handling of free variables.

$\mathbf{D}(g;\bar{G}) =$
  `if` $g$ `is inconsistent` $\mathbf{D}(\bar{G})$ `else` — $\mathbf{D}.1$
  `case` $g$ `of`
    `when` $a$ `?` $b$`:` $\mathbf{D}(\bar{G};a;b);$ — $\mathbf{D}.2$
    `when` $\bot$`:` $\mathbf{D}(\bar{G});$ — $\mathbf{D}.3$
    `when` $g$ is a value`:` $\mathbf{D}(\bar{G});$ -- *yield* $g$ — $\mathbf{D}.4$
    `default:` `if` $\mathbf{N}(g)$ `then` $\mathbf{D}(\bar{G};g);$ `else` $\mathbf{D}(\bar{G});$ — $\mathbf{D}.5$
$\mathbf{D}(null) = null;$ -- *program ends* — $\mathbf{D}.6$

---

$\mathbf{N}(c(\ldots,\bot,\ldots)) = \mathit{false}$ — $\mathbf{N}.1$
$\mathbf{N}(c(\ldots,p_{(?)},\ldots)) = \textsc{pull}(p);\ \mathit{true}$ — $\mathbf{N}.2$
$\mathbf{N}(c(\overline{x}) = \mathbf{N}(x_1) \wedge \ldots \wedge \mathbf{N}(x_k)$ — $\mathbf{N}.3$
$\mathbf{N}(n) = \mathbf{S}(n);\ \mathit{true}$ — $\mathbf{N}.4$

---

`compile` $\mathcal{T}$
  `case` $\mathcal{T}$ `of`
    `when` $\mathit{rule}(\pi, l \to r)$`:`
      `output` $\mathbf{S}(l) = \textsc{rewr}(r);$ — $\mathbf{S}.1$
    `when` $\mathit{exempt}(\pi)$`:`
      `output` $\mathbf{S}(\pi) = \textsc{rewr}(\bot);$ — $\mathbf{S}.2$
    `when` $\mathit{branch}(\pi, o, \bar{\mathcal{T}})$`:`
      $\forall\ \mathcal{T}' \in \bar{\mathcal{T}}$ `compile` $\mathcal{T}'$
      `output` $\mathbf{S}(\pi[o \leftarrow \bot]) = \textsc{rewr}(\bot);$ — $\mathbf{S}.3$
      `output` $\mathbf{S}(\pi[o \leftarrow p_{(?)}]) = \textsc{pull}(p);$ — $\mathbf{S}.4$
      `output` $\mathbf{S}(\pi) = \mathbf{S}(\pi|_o);$ — $\mathbf{S}.5$
$\mathbf{S}(c(\ldots)) = null$ — $\mathbf{S}.6$

FIGURE 3.1: Compilation of a source program, $\mathcal{S}$, by the FS. The rules of **D** and **N** are fixed for every $\mathcal{S}$. The rules of **S** are obtained from the definitional trees of the operations of $\mathcal{S}$ with the help of procedure `compile`. The notation $p_{(?)}$ indicates a node, $p$, labeled by the choice symbol. A sequence of variables is written $\overline{x}$. **N** evaluates to a Boolean indicating whether its argument might still be derived to a value. **D** and **S** return nothing. The symbol $c$ stands for a generic constructor. Line comments (following "--") indicate meta-actions.

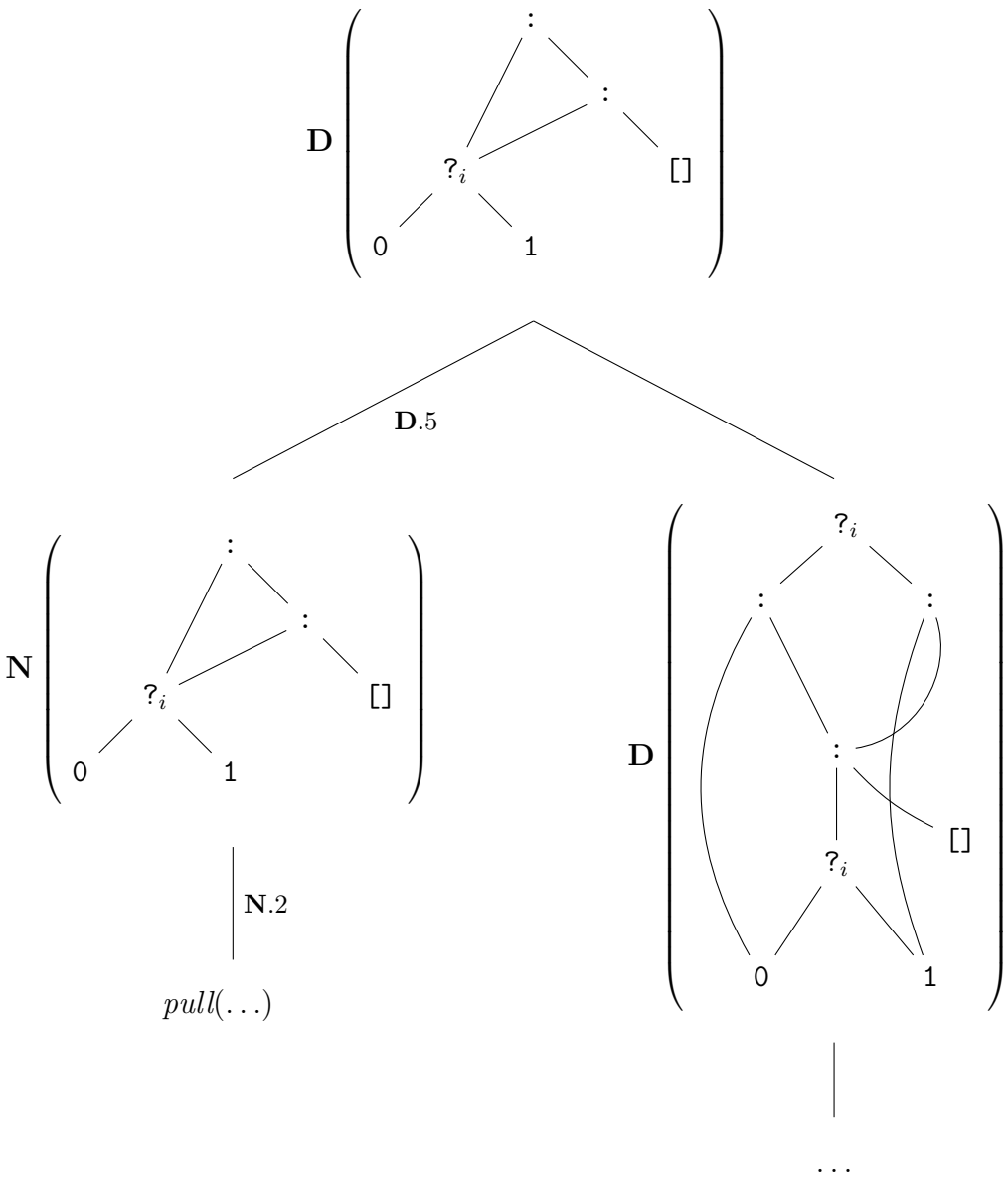



FIGURE 3.2: The topmost portion of the call tree for the expression `let d = (0 ? 1) in [d, d]`. Edge labels indicate which case of a target procedure is applied to obtain the next node of the call tree.

## Chapter 4

## Extensions to the Fair Scheme

The Fair Scheme (FS) is a correct and optimal evaluation strategy: provided sufficient time and other resources, it eventually computes every value of a source program without performing unnecessary steps. Even so, when using the FS to implement a Curry compiler one encounters two practical problems. First, the operational principles used do not necessarily yield the shortest derivations or the most compact value sets, and this can negatively impact performance. In particular, the eager elimination of free variables prevents their appearance in values, where they could concisely represent many results simultaneously. To take an extreme example, consider the expression `x::[Bool] where x free`, which under the FS gives an infinite computation that evaluates to all Boolean lists: `[]`, `[False]`, `[True]`, `[False, False]`, and so on. A more compact representation is simply `x`. Second, Curry defines functions with non-standard semantics that that do not map directly to LOIS or the FS. These include equational constraint operators and set functions.

Because of this, we will extend the FS before discussing its implementation in the

next chapter. While it would be possible to extend the proofs of Chapter 3 with a new formalization besides LOIS (see Sect. 3.1.2), this would be a significant undertaking that is beyond the scope of this work. Instead, we will support each extension with an informal argument. In Sect. 4.1 , we will present an extension called FS-x that allows us to perform computations in the presence of free variables. In Sect. 4.2 , we will present a further extension called FS-$\beta$ that implements strict and non-strict equational constraints. In Sect. 4.3, we will present another extension, FS-$\mathcal{S}$, for set functions.

## 4.1 Extensions for Narrowing (FS-x)

In a LOIS system, a rewrite rule consists of a left-hand side (LHS) and a right-hand side (RHS). As discussed in Sect. 3.1.1, every variable reference in the RHS of a rewrite rule must be introduced in the LHS of that same rule. This ensures that a successful pattern match always produces a binding for every variable. Because of this, an implementation of the FS does not require an explicit representation of variables in expressions.

We are now changing the rules of the game. We depart from LOIS systems with the introduction of *free variables* in expressions. A free variable is a variable that only appears on the RHS of a rule and is therefore not bound to anything. This is interpreted as a placeholder for an unknown value. To convert a Curry program into the LOIS representation, variables introduced with the `free` keyword on the RHS of

a rule can be replaced with generator functions to eliminate them. In this section, we extend the FS to retain free variables and compute with them through the principle of *narrowing*. During a narrowing computation, a free variable may or may not become bound to a specific value. As discussed in Sect. 2.3.1, free variables are a different but equivalent way to represent non-determinism, in addition to using choices. In our extension to the FS, when a free variable requires a binding, its non-determinism is converted into choices so that pull-tabbing can take over from that point.

Introduced in the context of automated theorem proving, narrowing has a long and successful track record in the functional logic domain [80]; indeed, narrowing has historically been the most popular basis for proposed functional-logic evaluation strategies (e.g., see [40, 81–86]). The role of narrowing in early implementations of Curry was discussed briefly in Sect. 2.3.1. Sound, complete, and optimal narrowing strategies, such as [54], are known for functional logic programs. There also exist functional logic languages, such as Life [87] and Escher [88], that do not provide narrowing but use residuation [89] for a similar purpose. Narrowing and residuation are not in conflict, and may coexist in an elegant model [90]. See [2] for a survey of functional logic languages that includes a discussion of both narrowing and residuation.

Narrowing is a computational step that is similar in purpose to rewriting and is used to evaluate expressions containing free variables. Rewriting is sufficient when there are no free variables present, and in these cases, narrowing degenerates to rewriting. Narrowing is a binary relation over terms that simultaneously instantiates a

free variable and applies a rewrite step. The effect is to replace certain subexpressions with simpler ones that have fewer free variables. The definition and example below serve to introduce this concept. Note that the following definition has been striped of certain technical details, as it is only intended to support informal arguments.

**Definition 4.1.1** (**Narrowing Step**)**.** Let $\mathcal{R} = (\Sigma, \rightarrow)$ be a rewrite system, $R$ a rule of $\mathcal{R}$, and $\xrightarrow[R]{}$ a rewrite step that applies $R$. Then for expressions $u$, $v$ and substitution $\sigma$, where $u\sigma \xrightarrow[R]{} v$, we say $u \overset{\sim}{\underset{\sigma,R}{\rightarrow}} v$ is a *narrowing step* with substitution $\sigma$ and rule $R$. We say this step *instantiates* a variable `x` if `x` appears in the domain of $\sigma$. – *end definition.*

**Example 4.1.2.** Consider the expression `not x` where `x` is a free variable and `not` is defined by the following rules:

```
1  not False = True        (rule R1)
2  not True  = False       (rule R2)
```

Two narrowing steps are possible:

$$\texttt{not x} \overset{\sim}{\underset{\sigma,R_1}{\rightarrow}} \texttt{True} \qquad \text{with } \sigma = \{\texttt{x} \mapsto \texttt{False}\}$$
$$\texttt{not x} \overset{\sim}{\underset{\delta,R_2}{\rightarrow}} \texttt{False} \qquad \text{with } \delta = \{\texttt{x} \mapsto \texttt{True}\}$$

The first performs in concerted fashion the substitution `(not x)`$\sigma =$ `not False` and a rewrite step according to rule $R_1$. The second case is similar. Both steps instantiate `x`. – *end example.*

Our interest in narrowing stems from a desire to perform computations in the presence of variables. Our ultimate aim is to avoid unnecessarily binding values to variables, so we shall delay doing so until it becomes unavoidable. As a result, variables are present in expressions as a computation progresses, which means that rewriting cannot be used as the main operational principle. A finite or infinite sequence of narrowing steps, $e_0 \leadsto e_1 \leadsto^* ...$, is called a *narrowing computation*, a *narrowing derivation* or, more simply, just a *computation* or *derivation*.

One challenge of using narrowing strategies, such as [54], in practice is the need to keep track of all the narrowing steps taken (e.g., through backtracking) to ensure that they are all eventually considered. Narrowing steps are typically selected non-deterministically and may be incomplete, as they can change the values of an expression in a way similar to what is shown in Example 4.1.2. The FS avoids the need for bookkeeping by using pull-tabbing to distribute computational resources. Therefore, it may not be suitable to directly apply narrowing steps in an extension of the FS. Instead, we will simulate their effects by combining variable instantiation with pull-tab and rewrite steps. We will replace free variables with generator functions based on the need relation, and then use pull-tabbing computations to evaluate the resulting non-deterministic expressions.

Our approach involves the following changes:

1. Allow variables in goals and permit their introduction through rewrite steps, rather than replacing them at compile time.

2. Define an *instantiation* step that replaces a variable with a suitable generator function in a specific context in the following way (see `boolGen` on Page 61):

$$\texttt{not x} \rightsquigarrow \texttt{not boolGen}$$

3. Only bind values to free variables as needed, using the instantiation step.

4. Extend the concept of consistency to accommodate instantiation.

As an example, consider the following derivation of `not x` using our approach:

$$\begin{array}{lcl} \texttt{not x} & \rightsquigarrow & \texttt{not boolGen} \\ & \not\Rightarrow & \texttt{not (False ? True)} \\ & \not\Rightarrow & \texttt{not False ? not True} \\ & \not\Rightarrow & \texttt{True ? not True} \\ & \not\Rightarrow & \texttt{True ? False} \end{array}$$

Variable instantiation refers to the two-step process of creating a variable and binding a value to it. In the computation process of our FS extension, these two aspects are separated. Variable creation occurs when a goal expression is created or through the application of rewrite steps. Variable binding is achieved through a combination of variable replacement (with a generator) and pull-tabbing. We refer to $\rightsquigarrow$ as instantiation rather than replacement to clearly distinguish it from the replacement

mechanism defined in Def. 3.1.1. While perhaps not ideal, we consider this a reasonable compromise because the instantiation step commits an occurrence of a variable to having a value bound to it, thereby completing its instantiation.

### 4.1.1 Extensions to the Need Relation

In the FS, free variables are eliminated by replacing them wherever they appear in the goal or the RHS of a rule. These transformations usually take place at compile time, but they could also be interspersed among computational steps. As an example, recall the function `unknown` (see Page 34) and consider the following derivation, which follows the FS:

$$\texttt{not unknown} \;\rightarrow\; \texttt{not boolGen} \;\rightarrow\; \ldots$$

The replacement of the free variable in the rule of `unknown` was done at compile time but it can also be included in a computation by using the following steps:

$$\texttt{not unknown} \;\rightarrow\; \texttt{not x} \;\rightsquigarrow\; \texttt{not boolGen} \;\rightarrow\; \ldots$$

The rewrite step now produces free variable `x` and the subsequent instantiation step replaces it. If instantiation unconditionally followed rewriting in this manner, it would be difficult to distinguish an implementation of the FS that replaced free variables at compile time from one that did so during a computation. In particular, such an implementation would have all the properties shown in Sect. 3.4.

To produce non-ground values, however, we cannot blindly replace free variables as they appear. In our extension of the FS, variable replacement is guided by an extension of the need relation to narrowing computations. The revised definition below considers free variables as constructor-rooted forms, recognizing that they are simply placeholders for unknown constructor expressions.

**Definition 4.1.3** (**Need, for Narrowing**)**.** Let $e$ be an expression of a LOIS system with subexpression $n$. The *narrowing need relation* is defined as follows (with differences between this and Def. 3.4.1 emphasized):

1. Node $n$ is needed for $e$ if and only if in any **narrowing** derivation of $e$ to a **variable or** constructor-rooted form, $e|_n$ **is instantiated or** is derived to a constructor-rooted form

2. Node $n$ is needed for $e$ if and only if it is needed for some maximal function-rooted subexpression of $e$.

A **narrowing** computation $\mathbf{e_0} \leadsto \mathbf{e_1} \leadsto \ldots$ is needed if and only if it reduces only needed subexpressions; that is, if and only if each replaced subexpression is related to the root by the need relation. – *end definition.*

In point 1 of Def. 4.1.3, the ability to derive the expression $e|_n$ to a constructor-rooted form or instantiate it depends on whether $e|_n$ is a free variable or not. If $e|_n$ is a free variable, it may be possible to instantiate it through a narrowing step, but

it cannot be rewritten into a constructor-rooted form. On the other hand, if $e|_n$ is not a free variable, then it may be possible to rewrite it to a constructor-rooted form, but it cannot be instantiated

The definition of need given in Def. 4.1.3 is not reflexive but it is transitive. As the examples below illustrate, it allows variables to appear unneeded in values. In each example, we ask whether the free variable `x` is needed.

- `x` is not needed in $e =$ `x`, since a derivation of zero steps reduces $e$ to a variable without instantiating `x`. Note that no narrowing steps can be applied to `x`, so there is no derivation of `x` to a constructor-rooted form. By contrast, `True` is needed in $e =$ `True` because every possible derivation (there is only one) derives `True` to the constructor-rooted expression `True`, in zero steps.

- `x` is not needed in $e =$ `[x]`, for reasons similar to the previous case, except that $e$ is in this case derived in zero steps to a constructor-rooted form. In general, if every node of every path from $e$ to `x` is constructor-labeled, then `x` is not needed for $e$.

- `x` is needed in $e =$ `not x` because $e$ cannot be derived to a non-function-rooted form without instantiating `x`. More specifically, by a narrowing step rooted at $e$ that instantiates `x`.

- `x` is needed in $e =$ `(not x, x)` because `not x` is a maximal function-rooted subexpression of $e$ and `x` is needed for it.

### 4.1.2 Free Variable Replacement Rules

Free variables, like choices, are a structural representation of non-determinism that blocks pattern matching. Instantiation steps are used to make progress in cases where a free variable, like `x` in the expression `not x`, prevents a rule from matching, and these are only performed for needed variables.

This suggests a very simple way to extend the FS. In Figure 3.1, we see that procedure **S**.4 removes choices from the inductive positions of needed operations. We can handle free variables similarly by adding a target procedure to instantiate needed free variables. When compiling **S**-rules, the following additional case is added after the case that generates rule **S**.4:

$$\texttt{output}\ \mathbf{S}(\pi[o \leftarrow p_{(x)}]) \quad = \textsc{inst}(p); \qquad \mathbf{S}.\texttt{x}$$

Here, $p_{(x)}$ represents a node labeled by a variable and INST is an action that performs an instantiation step for the variable at $p$. This additional target procedure does not override subsequent ones, except when a variable appears at an inductive position of the operation being compiled.

In addition to this, we must also extend the **N** target procedure for free variables. The rule shown below is added after **N**.3. This ignores free variables, since they are always considered constructor expressions. The helper function *bound*, which indicates whether its argument has a binding in the current context, will be discussed in Sect. 4.1.4 (see Page 138).

$$\mathbf{N}(p_{(x)}) \quad = \texttt{if } \mathit{bound}(p) \texttt{ then } \textsc{inst}(p);\ \mathit{true} \qquad \mathbf{N}.\texttt{x}$$

Our extended version of the FS that includes **N**.x and **S**.x is called FS-x. In a computation performed according to FS-x, the steps taken are similar to those taken in a computation performed according to the FS (using unconditional, interspersed instantiation steps, as discussed in Sect. 4.1.1). The order of steps may differ, and unnecessary actions might be omitted. An action is considered unnecessary if it relates to the handling of a choice that arises from a free variable that has not yet been instantiated in any previous step of the computation. While these differences may affect the order and efficiency of computations, we believe they do not affect their correctness. We note that the correctness of evaluation strategies often does not depend on the order of steps, as seen in the correctness of pull-tabbing computations (Theorem 3.4.7) and needed narrowing [54].

After a rewrite step introduces a free variable (which in the FS would be instantiated right away), the computation in FS-x proceeds as in the FS unless and until **S**.x is invoked, except that:

1. invocations of **D**.2 might instead become invocations of **D**.4; and

2. invocations of **N**.2 might be omitted.

However, neither of these omits a necessary action, as the actions omitted both relate to the processing of choices that would arise from a free variable for which an

instantiation step has not yet occurred. If that step is performed later, the corresponding actions will occur. If **S**.`x` is invoked, it performs an instantiation step for some variable-labeled node, $p_{(x)}$, that would have occurred earlier in the FS (immediately following the introduction of $p$). After this, the computation by FS-`x` proceeds in a similar way to that of the FS. Procedure **S**.`x` is only invoked when **S**.5 would have been invoked in the FS, as in that case the node $p$ would be labeled by a generator function. In other words, the introduction of **S**.`x` effectively moves an instantiation step from just after a variable is introduced to just before the generator function that takes its place would have been reduced, if that occurs at all. An efficient implementation of **S**.`x` could combine these instantiation and rewrite steps, and perhaps even the ensuing pull-tab step (by **S**.4).

### 4.1.3 Replacement Mechanism

Instantiation is more complex than simply replacing a variable globally. A variable replacement in one context should not be observed in a separate context because the variable being replaced may not be needed in that other context. For example, in the computation of the expression $e =$ `not x ? x where x free`, there is a needed instantiation step at `not x` that leads to the values `False` and `True`. However, instantiation of the occurrence of `x` on the right side of the expression is not needed so the right side should simply produce `x`. This suggests that instantiation should be applied to occurrences of free variables rather than to the variables themselves,

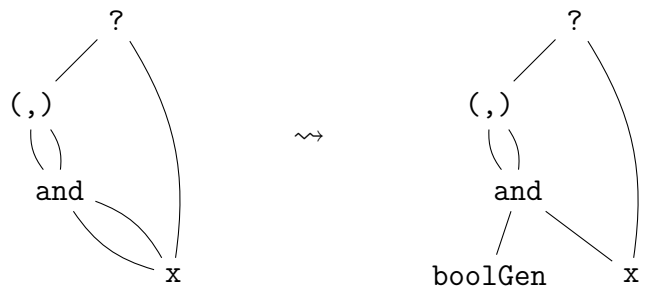
meaning that the most straightforward approach to performing instantiation, such as $e[\texttt{x} \leftarrow \texttt{boolGen}]$, may not be the best option.

Instead of replacing the free variable directly, we take a more conservative approach by replacing the function-rooted node ($f$) whose definitional tree directly made instantiation necessary. We consider this approach to be more conservative because the effects of the replacement can be observed in smaller and fewer contexts. While this may involve repeating some instantiation steps, it ensures that no instantiation is observed in a context where it is not needed. This differs from rewriting, which relies on replacing shared subexpressions to avoid repeating steps, but it is similar to pull-tabbing in that it involves duplicating a shared representation of non-determinism by replacing the nearest function-labeled node.

**Definition 4.1.4** (**Variable Replacement**)**.** Let `x` be a free variable in a function-rooted expression ($f$), $\gamma$ a suitable generator function that can be substituted for `x` according to the FS, and $e$ a context where $f$ appears. Recalling the definitions of copyr and path from Def. 3.2.1, let $p$ be a path obtained according to the definitional tree of $f$ that leads from the root of $f$ to a needed subexpression where `x` appears. The instantiation of `x` in $f$ at $p$ is given by $e[f \leftarrow \text{copyr}(f, p, \gamma)]$. – *end definition.*

**Example 4.1.5.** An instantiation step is shown below for the following expression:

$$e = \texttt{let f = and x x in (f,f) ? x where x free}:$$




This corresponds to $e[f \leftarrow u]$ with $f =$ `and x x` and $u = \text{copyr}(f, [0], \texttt{boolGen})]$. – *end example.*

It is important to note a few details about Exam. 4.1.5. The step shown involves replacing an element in a pair with a new value, while maintaining the shared connection between the elements of the pair. This is achieved by performing the replacement at node $f$, which preserves the connection. In contrast, if a copy-replacement were performed at a different node, such as the pair itself (with path $[0, 0]$), the connection between the elements of the pair would be broken. The linear form of the expression $e$ contains three occurrences of the variable (`x`), but only one of these is affected by the replacement. An instantiation according to Def. 4.1.4 always replaces exactly one occurrence of a variable. Also, the generator function (`boolGen`) introduced is dominated by the node that replaces $f$. This is a crucial property because it ensures that the generator function is only observed where it is certainly needed, since we know that $f$ is needed. As a result, the right choice alternative following the replacement still refers to the variable (`x`). While it would be ideal to replace both occurrences of `x` under $f$, this may not be practical because it depends on how the call is constructed

rather than the definition of `and`. A potential optimization might be to perform a concurrent replacement of all occurrences of the variable being instantiated that appear directly under the node being replaced ($f$), but we do not explore this option.

We can argue that our instantiation mechanism is generally correct by considering every context in which the replaced function ($f$) might appear. If $f$ is needed in a particular context, then the instantiation step is also needed by an extension of Lemma 1 in [62] and our amended definition of need. There must be at least one such context where $f$ is needed in order for the instantiation step directed by $f$ to be performed. In any context where $f$ is not needed, replacing it has no effect because the function will be discarded in any value produced, so changes to its structure will not be observed. Additionally, the replacement of a subexpression of $f$ cannot have any effect on pattern matching (for $f$) and cannot lead to a choice rising beyond $f$ through pull-tabbing unless $f$ were needed (Lemma 3.4.11), which it is not, by assumption.

### 4.1.4 Consistency

To ensure correctness, it is important that choices arising from variable replacements are annotated consistently. Recall that a choice identifier (cid; see Sect. 3.2.2) is used to annotate choice symbols. Since the mechanism in Def. 4.1.4 only replaces a single occurrence of a variable, it is possible that one variable is subjected to multiple instantiation steps. For example, consider the following derivation:

```
(not x, not x)  ⇝*  (not boolGen, not boolGen)
                →*  (not (False ?_i True), not (False ?_j True))
```

For the reasons discussed in Sect. 3.2.2, the above derivation is only correct if $i = j$. However, the current generator function, `boolGen`, introduces fresh cids each time it is rewritten, which means it is not adequate. Additionally, when a rewrite step is applied to the corresponding generator function for a type with a non-nullary constructor, it introduces free variables that must be used consistently. In order to properly handle these cases, the generator function needs to be refined.

To achieve this goal, we will add a runtime action that dynamically creates generator functions. The generator functions we have used previously (such as `boolGen`) will now be referred to as *generator function templates* because their cids and free variables are not specified. An instance of such a template, called a *generator function instance*, has the same structure but uses specific cids and free variables. When an instantiation step is first applied to a free variable (`x`), the generator function template corresponding to the type of `x` is instantiated, producing a generator function instance containing fresh cids and free variables unique to `x`. For example, if `x` and `y` are Boolean variables, then applying an instantiation step to each might produce the following two generator function instances:

```
1  boolGen_x = False ?_i True
2  boolGen_y = False ?_j True
```

We can now specify the behavior of the INST($p$) target action that appears in **S**.`x`:

1. Get the generator function instance ($\gamma$) corresponding to the variable $p$, creating it if necessary;

2. Perform the transformation described in Def. 4.1.4 using replacement $\gamma$.

To make the presentation simpler, we will omit $\gamma$ by combining the subsequent rewrite step that replaces $\gamma$ into our depiction of instantiation steps. An implementation can also perform this optimization to save a rewrite step for each instantiation.

We call the RHS of a generator function instance a *generator expression* and write, for example, $\Gamma_{\texttt{x}} = \texttt{False ?}_i\ \texttt{True}$ to indicate the generator expression for free variable `x`. Note that a generator expression and fingerprint together can tell us whether a free variable is bound in a particular context and also gives us the binding if one does exist. For example, the fingerprint $\phi = \{i \mapsto \text{L}\}$ corresponds to the binding of `x` to `False`. If $\Gamma_{\texttt{x}}$ has not been created or if a particular expression does not contain a choice outcome for $i$ in its fingerprint, then `x` is free in that expression.

The use of the above mechanism avoids inconsistency problems arising from duplicate free variable replacements because all instantiations of a given variable use the same cids and free variables. However, the presence of variables also introduces the possibility of a new consistency issue. To see this, consider the following step:

$$\texttt{(not x, x)} \rightsquigarrow \texttt{(not (False ?}_i\ \texttt{True), x)}$$

Without some care, a computation proceeding from this point may derive the incorrect values `(True, x)` and `(False, x)`. These are the results of inconsistent computations because they are derived using a binding of `x` that was not applied to every occurrence of `x`. For instance, `(True, x)` arises when binding `x` to the value `False` to obtain the first part of the pair, but fails to replace the occurrence of `x` in the second part of the pair. Compare the values above with the correct values produced by narrowing:

$$\begin{array}{llll} \texttt{(not x, x)} & \overset{\sigma,R_1}{\rightsquigarrow} & \texttt{(True, False)} & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{False}\} \\ \texttt{(not x, x)} & \overset{\delta,R_2}{\rightsquigarrow} & \texttt{(False, True)} & \text{with } \delta = \{\texttt{x} \mapsto \texttt{True}\} \end{array}$$

To obtain these values, we must ensure that all occurrences of instantiated variables are eventually replaced. This is the reason INST appears in rule **N**.x. The helper function *bound* shown on Page 130 identifies free variables that, like `x` in the above example, have a binding. When that is the case, the occurrence encountered by **N**.x must also be instantiated.

As an optimization, an implementation could define INST to replace a variable directly with a more specific value found by traversing the choices of that variable's generator expression, using the current fingerprint as a guide. If this is done, it would not be safe to to replace the variable directly. For example, in the computation of `(not x, [x]) where x free`, we obtain two pairs `(True, [x])` and `(False, [x])` whose second elements are shared. Replacing the shared subexpression with a specific value such as `[False]` would lead to inconsistent results. To avoid this, **N**.x would

need to be defined using copyr as in Def. 4.1.4, so that neither replacement affects the other.

## 4.2 Extensions for Constraints (FS-$\beta$)

Despite the fact that FS-`x` only instantiates free variables that are needed according to Def. 4.1.3, it may still perform certain variable instantiations that seem excessive. For example, consider the following goal over Boolean variables:

```
y=:=x &> (x,y) where x,y free
```

Since both variables are needed in the equational constraint, FS-`x` would instantiate them both. This leads to the following outcomes:

- Value `(False,False)`, with bindings $\{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{False}\}$.
- Value `(True,True)`, with bindings $\{\texttt{x} \mapsto \texttt{True}, \texttt{y} \mapsto \texttt{True}\}$.
- Failure, with inconsistent bindings $\{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{True}\}$.
- Failure, with inconsistent bindings $\{\texttt{x} \mapsto \texttt{True}, \texttt{y} \mapsto \texttt{False}\}$.

This is not ideal because more compact representations, such as `(x,x)`, exist and can perhaps be derived in fewer steps. In this section, we will present a further enhancement to FS-`x` called FS-$\beta$ that prevents unnecessary instantiations such as this that arise from applications of the equational constraint operator. We will also describe

how the mechanism for this can be used to define a non-strict equational constraint operator (`=:<=`), which plays a central role in evaluating functional patterns.

### 4.2.1 Strict Equational Constraints

Our enhanced evaluation strategy is based on the observation that an expression involving a needed application of the equational constraint to two free variables, `x` and `y`, can be evaluated by assuming the constraint succeeds and then discarding computations in which `x` and `y` ultimately become bound to different values. Such detection can occur as the computation proceeds to discard inconsistent computations as soon as they are discovered. When neither variable is bound to a specific value, we end up in a relatively straightforward situation like the one discussed above, where at most a substitution of equivalent variables is needed to express values concisely. However, when a more specific binding for `x` or `y` arises through narrowing, the situation becomes more complicated.

The following example illustrates. Let $\mathcal{V}(e)$ denote the value set of expression $e$ and consider the expression $u =$ `(not x, y)`. The values of $u$ are shown below:

$$\mathcal{V}(u) = \{\texttt{(True,False)}, \texttt{(True,True)}, \texttt{(False,False)}, \texttt{(False,True)}\}$$

We can define a subset of $\mathcal{V}(u)$ that excludes results arising when `x` and `y` are bound to different values. We write $\mathcal{V}|_{\texttt{x=y}}(e)$ to denote the values of $e$ subject to the constraint that `x` and `y` have identical bindings. Therefore, we have the following:

$$\mathcal{V}|_{\texttt{x=y}}(u) = \{\texttt{(True,False)}, \texttt{(False,True)}\}$$

A value such as `(True,True)` does not appear in this because it arises from a computation in which `x` and `y` are bound to different values. Adding an equational constraint between free variables, as we do in going from $\mathcal{V}$ to $\mathcal{V}|_{\texttt{x=y}}$, can never introduce additional values. It can only remove values that are inconsistent under the additional constraint. Because of this, we can devise an evaluation strategy that lazily instantiates variables in equational constraints, keeps track of which computations are subject to which equational constraints, and discards computations that are found to be inconsistent.

FS-$\beta$ uses a reserved unary symbol $\beta$ to determine which expressions are subject to which constraints. This symbol is annotated with two free variables and indicates that they are to be constrained equal. The effect is similar to binding one free variable to the other, though it is important to note that the variables retain their separate identities and can become bound to more specific values through instantiation and pull-tab steps as a computation proceeds. We write $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$ to indicate a $\beta$ node constraining `x` and `y` equal. $\beta$-labeled nodes are processed by FS-$\beta$ similar to the way that nodes labeled by the choice symbol are processed by the FS. Overall, the strategy of FS-$\beta$ is to replace certain needed applications of `=:=` with $\beta$-rooted expressions, apply pull-tab steps to those, and then remove $\beta$-labeled nodes when they appear at

the root of an expression in the queue ($\bar{G}$) of expressions being evaluated by FS-$\beta$. The removal of a $\beta$-labeled node provides information that might lead a computation to be found inconsistent. This is similar to the way that target procedures **D**.1 and **D**.2 work together to discard computations with inconsistent choice outcomes.

To construct FS-$\beta$, the following changes are made to FS-`x`:

1. The definition of the equational constraint is changed so that an equational constraint between two free variables, `x` and `y`, is replaced with the expression $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$ `True`.

2. The PULL target action is extended to support pull-tabbing of $\beta$-labeled nodes, where the pull-tab relation is extended to $\beta$ nodes as indicated below:

$$\texttt{f u } (\beta_{\texttt{x}\leftrightarrow\texttt{y}} \texttt{ a) v} \quad \Xi \quad \beta_{\texttt{x}\leftrightarrow\texttt{y}} \texttt{ (f u a v)}$$

3. Three additional target procedure rules are defined to handle the pull-tabbing and removal of $\beta$-labeled nodes. First, when the **D** procedure is applied to a $\beta$-rooted expression, the $\beta$ node is removed by the following rule added as an additional case just after rule **D**.2:

`when` $\beta$ $a$: $\qquad\qquad$ $\mathbf{D}(\bar{G}; a);$ $\qquad\qquad$ $\mathbf{D}.\beta$

Second, the **N** procedure is extended by adding the following just after rule **N**.2:

$$\mathbf{N}(c(\ldots, p_{(\beta)}, \ldots)) \qquad = \textsc{pull}(p);\ \ \mathit{true} \qquad\qquad \mathbf{N}.\beta$$

Third, when compiling **S**-rules, the following additional case is added after the case that generates rule **S**.4:

$$\texttt{output}\ \ \mathbf{S}(\pi[o \leftarrow p_{(\beta)}]) \quad = \textsc{pull}(p); \qquad\qquad \mathbf{S}.\beta$$

It can be shown, by an argument similar to the one on Page 135 arguing for the correctness of instantiation steps, that:

1. If an expression $e$ contains a needed subexpression $s$ whose root is labeled by $\beta_{\mathtt{x}\leftrightarrow\mathtt{y}}$, then in a computation of $e$ by FS-$\beta$, $e$ will eventually be replaced with an expression whose root is $\beta_{\mathtt{x}\leftrightarrow\mathtt{y}}$.

2. If an expression $e'$ contains an unneeded subexpression $s'$, then replacing $s'$ with a $\beta$-rooted expression does not change the values of $e'$.

Rules **D**.$\beta$, **N**.$\beta$, and **S**.$\beta$ define the mechanical changes that constitute FS-$\beta$. $\beta$-labeled nodes are created, subjected to pull-tab steps, and ultimately removed. These changes alone do not have the intended effect. We must also extend our notion of a consistent computation to exclude values that arise from inconsistent bindings. When $\beta$-labeled nodes are taken into account, we are only interested in values whose bindings satisfy the constraints indicated by any $\beta$-labeled nodes that were removed in their computation. Recall that every value produced by the FS and, by extension,

FS-$\beta$ is associated with a fingerprint. We extend our definition of consistency as stated in Def. 4.2.1

**Definition 4.2.1. Consistent Computations (FS-$\beta$).** A value $v$ yielded in rule **D**.4 of FS-$\beta$ with fingerprint $\phi$ is considered inconsistent if the computation of $v$ included an application of **D**.$\beta$ with head symbol $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$ and the bindings for `x` and `y` in $\phi$ are different. – *end definition.*

The next example shows how fingerprints, generator expressions, and $\beta$-labeled nodes work together to identify computations with inconsistent bindings.

**Example 4.2.2.** Consider $e(u)$ = `(`$u$`, x=:=y) where x,y free`, and expression $u$ = `not x && y`. The computation of the expression with substitution $\sigma = \{\texttt{x} \mapsto \texttt{False}, \texttt{y} \mapsto \texttt{True}\}$ by FS-$\beta$ is as follows[1]:

$$
\begin{array}{lll}
\texttt{(not x \&\& y, x=:=y)} & \rightrightarrows & \texttt{(not (False ?}_i\texttt{ True) \&\& y, x=:=y)} \\
 & \rightrightarrows^* & \texttt{(not False \&\& y, x=:=y) ?}_i\texttt{ ...} \\
 & \rightrightarrows & \texttt{(True \&\& y, x=:=y)} \\
 & \rightrightarrows & \texttt{(True \&\& (False ?}_j\texttt{ True), x=:=y)} \\
 & \rightrightarrows^* & \texttt{... ?}_j\texttt{ (True \&\& True, x=:=y)} \\
 & \rightrightarrows & \texttt{(True, x=:=y)} \\
 & \rightrightarrows & \texttt{(True, } \beta_{\texttt{x}\leftrightarrow\texttt{y}}\texttt{ True)} \\
 & \rightrightarrows & \beta_{\texttt{x}\leftrightarrow\texttt{y}}\texttt{ (True, True)}
\end{array}
$$

This applies rule **D**.$\beta$ to an expression rooted by $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$, so `x` and `y` are constrained equal. The generator expressions for these variables are $\Gamma_{\texttt{x}}$ = `False ?`$_i$ `True` and

[1]To simplify the presentation, `&&` here performs Boolean AND without short-circuiting.

$\Gamma_{\texttt{y}} = \texttt{False ?}_j\ \texttt{True}$, respectively. The constraint implies that corresponding choice identifiers — $i$ and $j$ in this case — must either both be present or both absent in the fingerprint, and if they are present, must both be bound to identical values. But since the fingerprint $\phi = \{i \mapsto \text{L}, j \mapsto \text{R}\}$ contains different values for $i$ and $j$, this computation is inconsistent. – *end example.*

A $\beta$-labeled node may occur after the free variables involved have been considered by a rule of FS-$\beta$. Therefore, one might be concerned that it appears "too late" to be handled properly. For example, consider the computation of an expression of the form $e(u) =$ `(`$u$`, x=:=y) where x,y free`. The first pair element, $u$, is inspected by FS-$\beta$ before $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$ is created. More specifically, in a left-to-right, depth-first traversal of the call tree $\Delta\mathbf{D}(e)$, the subtree $\Delta\mathbf{N}(u)$ occurs before $\Delta\mathbf{N}(\texttt{x=:=y})$ does.

This does not lead to any problems. On the one hand, if $\mathbf{N}(u)$ does not instantiate `x` or `y`, then a constraint between these variables has no effect. On the other hand, if $\mathbf{N}(u)$ instantiates `x`, then subsequent steps have the effect of narrowing, as described in Sect. 4.1. The eventual result is some number of invocations of the form $\Delta\mathbf{D}(e(u'))$, where $u'$ is the result of a narrowing step $e(u) \rightsquigarrow e(u')$ that instantiates `x`. In each of these, the fingerprint of $e(u')$ contains information about the binding of `x`. The same argument applies equally to instantiations of `y`. Therefore, the information needed to detect inconsistent bindings will be available.

**Example 4.2.3.** Consider $e(u) =$ `(`$u$`, x=:=y) where x,y free`, and expression $u =$

`not x`. The computation of the expression with substitution $\sigma = \{\texttt{x} \mapsto \texttt{False}\}$ by FS-$\beta$ is as follows:

```
(not x, x=:=y)  -=->   (not (False ? True), x=:=y)
                -=->*  (not False, x=:=y) ? ...
                -=->   (True, x=:=y)
```

The subsequent computation is $\Delta\mathbf{D}(e(u'))$ where $u' =$ `True` is the result of the narrowing step $u \overset{\sigma,R}{\leadsto} u'$ that instantiates `x` with $R$ being the rule `not False → True`. – *end example.*

### 4.2.2 Non-Strict Equational Constraints

The non-strict equational constraint operator (`=:<=`) is used to implement functional patterns, which were discussed in Sect. 2.3.4. This operator is involved in the dynamic generation of patterns and pattern matches that arise when evaluating functional patterns. To implement a full-featured Curry system, we will need to implement this operator. A complete definition of this operator is given in [42], but we do not need to discuss every detail here. What is important for us is to know that this operator includes an action binding a free variable to an arbitrary expression. Following that, references to the variable in any context where the binding step was performed should be treated as references to the expression it is bound to.

Fortunately, the mechanisms and methods we have already used for narrowing and to evaluate strict equational constraints are well-suited to the non-strict equational

constraint. We now allow $\beta$ nodes of the form $\beta_{\mathtt{x}\mapsto e}$, where `x` is a free variable and $e$ is an arbitrary expression. The rules of FS-$\beta$ are not changed for these $\beta$ nodes. They are still subjected to pull-tab and removal steps. We provide a definition of `=:<=` such that applications in which a binding is called for are replaced as in the following step: `x =:<= e` $\rightarrow$ $\beta_{\mathtt{x}\mapsto e}$ `True`.

To make good on the requirement that references to `x` are replaced with the bound expression, we update the helper function *bound* and the target action INST so that **N**.`x` (see Page 130) also replaces free variables bound through an application of `=:<=`. This mechanism is like that of the instantiation step, which is to say using *copyr*, except that a variable occurrence is replaced with the expression it is bound to (in the relevant context) rather than its corresponding generator expression. When rule **D**.$\beta$ removes a node labeled $\beta_{\mathtt{x}\mapsto e}$, the binding between `x` and $e$ is stored, similar to the way a strict equational constraint between free variables is stored.

### 4.3 Extensions for Set Functions (FS-$\mathcal{S}$)

In this section, we will present a modification to the FS that enables it to evaluate set functions. As mentioned in Section 2.3.5, set functions allow for the encapsulation of search in logic programming languages. Curry offers a representation of value sets and provides functions for manipulating them. The evaluation of set functions presents specific challenges that we will address by further extending the types of objects that appear in our computational model of programs.

The main challenge is how to distinguish the non-determinism of a function from that of its arguments. To explore this idea, we shall reconsider the integer adjacency function, `adj` (Page 71) and binary digit function, `binDigit` (Sect. 2.2.1). Our first goal will be to evaluate $\texttt{adj}_S$ `binDigit`. To do this, we will examine different methods and introduce the concept of a *set capsule*, which is used to enclose computations whose values are collected into sets. We initially use curly braces to indicate the presence of a set capsule. Currently, we are using this notation in a general way to show that something needs to be modified and to explain the issues that need to be addressed. As our evaluation strategy becomes clearer, we will more precisely define set capsules and related structures and introduce a different notation.

### 4.3.1 Set Capsules

A set capsule is used to capture non-determinism. If the computation within a capsule generates multiple values, those values form a set. In our evaluation strategy, non-determinism is eventually represented as choice-labeled nodes that undergo pull-tab steps. This suggests a straightforward method for evaluating set functions: simply replacing a set function-labeled node with the application of the corresponding non-set function inside a set capsule and prohibiting pull-tab steps that would move a choice outside of a set capsule. This idea is demonstrated in the derivation shown below:

```
adjS binDigit  ⇉  adjS (0 ? 1)
               ⇉  adjS 0 ? adjS 1
               ⇉*  {adj 0} ? {adj 1}
               ⇉*  {(0-1) ? (0+1)} ? {(1-1) ? (1+1)}
               ⇉*  {(-1) ? 1} ? {0 ? 2}
```

Here, each application of $\texttt{adj}_S$ is replaced with an application of `adj` to the same argument inside a set capsule. When a choice reaches the root of a set capsule, it is not allowed to undergo a pull-tab step because that would remove it from the capsule. Although this approach appears promising because each set capsule holds the correct values, it is not always successful. In fact, simply changing the evaluation order in this scheme can lead to a different result, as shown in the following derivation:

```
adjS binDigit  ⇉  {adj binDigit}
               ⇉  {adj (0 ? 1)}
               ⇉*  {adj 0 ? adj 1}
               ⇉*  {(0-1) ? (0+1) ? (1-1) ? (1+1)}
               ⇉*  {(-1) ? 1 ? 0 ? 2}
```

With this derivation, the set capsule is created before applying a rewrite step to `binDigit`, which results in the set capsule incorrectly containing a choice that originated from an argument the set function was applied to.

The problem of order-dependence in the steps used to evaluate a set function has been previously identified [58, 61, 91]. A simple, though approximate, way to address this is to eagerly evaluate set function arguments, which removes any non-determinism present before the value collection process starts. This approach, although used in

at least one popular implementation of Curry (PAKCS), is not ideal because it is not lazy. If implemented in an extension of the FS, it would likely compromise the correctness and optimality properties discussed in Sect. 3.4.

We will present an alternative approach that is better suited to our pull-tabbing evaluation strategy. This method is lazy, meaning it generates the correct values while only performing needed steps. The idea is to create a "leaky" set capsule that selectively captures choices related to the applied function while allowing other choices (i.e., those arising from non-determinism in function arguments) to escape. From now on, each set capsule will have a set of choice identifiers (cids) called the escape set. The elements of the escape set are denoted by subscripts after the closing curly brace of the set capsule. For example, $\{\ldots\}_{\{i\}}$ indicates a set capsule that allows choices with cid $i$ to escape. When the expression within a set capsule becomes choice-rooted, a further pull-tab step is allowed only if the cid of the root choice appears in the escape set of the enclosing set capsule. If a pull-tab step is allowed, it will split the set capsule into two and divide the alternatives of the root choice between the two resulting capsules.

To demonstrate, let us assume there is a function called `setmin` that produces the minimum element of a value set. Then the following pull-tab step is allowed:

$$\texttt{setmin \{a ?}_i\texttt{ b\}}_{\{i\}} \quad \rightarrow \quad \texttt{setmin \{a\}}_{\{i\}}\ \texttt{?}_i\ \texttt{setmin \{b\}}_{\{i\}}$$

This step is allowed because the root choice is annotated with a cid ($i$) that appears in the escape set. On the other hand, no pull-tab step is available in $\texttt{setmin \{a ?}_i\ \texttt{b\}}_{\emptyset}$ because $i$ does not appear in the escape set of the enclosing set capsule. If the non-determinism arising from set function arguments can be accurately determined and used to populate escape sets, then this scheme seems to provide an evaluation strategy for set functions that is compatible with the FS. To simplify the presentation, we may omit a function, such as `setmin`, that is applied to a set capsule when writing pull-tab steps.

### 4.3.2 Monitoring Subexpressions

Our scheme involves monitoring the arguments a set function is applied to. We will show this by drawing boxes around monitored subexpressions. The rule for relating boxes to set capsules is straightforward: the cid of every choice that appears inside a box, and only those cids, should be included in the escape set of the enclosing set capsule.

Boxes do not interfere with rewrite or pull-tab steps and are propagated in a way that ensures all and only the subexpressions whose non-determinism should escape the set capsule are boxed. The presence of a box does not affect pattern-matching, but it is taken into account during variable binding. After a successful pattern match in which a variable is bound to a boxed subexpression, any reference to that variable in the RHS of a rule is placed inside a box in the replacement. The reason for this is

simple: if we are monitoring an expression for non-determinism, then we must track its subexpressions through rewrite steps. For example, recall the function `fst` from Page 63. The following rewrite step is possible: `fst` $\boxed{\texttt{(a,b)}} \rightarrow \boxed{\texttt{a}}$. The replacement is boxed because the rule of `fst` references a variable bound to a boxed subexpression.

The above scenario only occurs when a box appears between a function-rooted symbol and a variable-labeled subexpression. In addition to this, other steps occur normally inside boxes. For example, compare the previous step with this one: $\boxed{\texttt{fst (a,b)}} \rightarrow \boxed{\texttt{a}}$. The result is boxed as before, but for a different reason. Here, the step simply takes place inside a box and there is no reason to doubly box the reference to `a`.

These rules effectively keep the non-determinism arising from functions and function arguments separated. Notice what happens when non-determinism arises in the rules of a function symbol during rewriting: `adj` $\boxed{\texttt{a}} \rightarrow \boxed{\texttt{a}} - 1$ `?` $\boxed{\texttt{a}} + 1$. Since the overall expression rooted by `adj` is unboxed, any non-determinism arising from the rules of `adj` should also be unboxed, while the box around its argument should be maintained through the step. This is exactly what happens in the step shown above. On the other hand, if the step occurs inside a box, as in $\boxed{\texttt{adj a}} \rightarrow \boxed{\texttt{a - 1 ? a + 1}}$, then the choice created by the rule of `adj` is boxed and can be attributed to an argument of a set function application.

When higher-order arguments are involved, we must be careful to regard partial application expressions as data: e.g., the normal form of `adj` is `adj`. The nondeterminism in the rule of `adj` is not involved in the computation of this value. When the final argument is applied to a boxed partial application expression, the resulting application expression is not boxed because it is not a reference to a boxed expression. Therefore, given the function `apply f a = f a`, the evaluation of $\texttt{apply}_S$ `adj a` proceeds as `{apply` $\boxed{\texttt{adj}}$ $\boxed{\texttt{a}}$`}` $\rightarrow$ `{adj` $\boxed{\texttt{a}}$`}`.

The FS does not apply the rules of the choice operator, relying instead on pull-tabbing. However, to determine how a pull-tab step should behave in the presence of boxes, it is helpful to apply the rules of choice within a boxed expression. An example is shown below, where the two rules of choice are applied to a boxed function argument:

$$\texttt{f u } \boxed{\texttt{a ? b}} \texttt{ v} \quad \rightarrow \quad \texttt{f u } \boxed{\texttt{a}} \texttt{ v}$$
$$\texttt{f u } \boxed{\texttt{a ? b}} \texttt{ v} \quad \rightarrow \quad \texttt{f u } \boxed{\texttt{b}} \texttt{ v}$$

Since we will not be applying these rules, we must be careful in how we define a pull-tab step applied to a choice-rooted, boxed expression in order to propagate boxes as shown above. As the example above shows, references to the left and right alternatives of the pull-tab source should be boxed. Therefore, a corresponding pull-tab step should be constructed according to the following relation:

$$\texttt{f u } \boxed{\texttt{a ? b}} \texttt{ v} \quad \Xi \quad \texttt{(f u } \boxed{\texttt{a}} \texttt{ v) ? (f u } \boxed{\texttt{b}} \texttt{ v)}$$

A pull-tab step such as this one, in which a boxed choice gives rise to an unboxed one presents a convenient opportunity to update the escape set of the enclosing set capsule because it is at this point that the choice and box are in closest proximity to one another. This allows for a local rule that can be implemented efficiently. Therefore, we will add a rule that pull-tab steps are monitored for boxes between the source and target. When one occurs, the escape set of the enclosing set capsule will be updated. We do not need to worry that boxed choices are not added to an escape set immediately after they are created. If some boxed choice is not needed for the expression at the root of a set capsule, then omitting it from the escape set does no harm, because only needed choices reach the root of an expression through pull-tab steps. And, if a boxed choice ever does arrive unboxed at the root of an expression through a sequence of pull-tab steps, then one of those steps will have updated the escape set by the rule above prior to the unboxed choice-rooted expression being inspected.

Whenever a choice attributed to a set function argument occurs at the root of the expression inside a set capsule, either a previous pull-tab step removed it from its box or the expression is itself still boxed. To handle the latter case, steps like the following one must be permitted:

$$\{\boxed{\texttt{a } ?_i \texttt{ b}}\} \quad \Xi \quad \{\boxed{\texttt{a}}\}_{\{i\}} \; ?_i \; \{\boxed{\texttt{b}}\}_{\{i\}}$$

The overall effect of the rules for propagating boxes is to always and only maintain boxes around subexpressions related to the evaluation of set function arguments. These rules are local, which makes an efficient implementation more likely. Specifically, the rule for placing boxes around variable references in the RHS of a rule depends only on whether a box was encountered while forming a pattern match for the LHS of that rule and not on any other contextual information. Similarly, the rules for updating escape sets only depend on whether a box was crossed during a pull-tab step, or whether the expression in a set capsule is a boxed choice.

We are now ready to return the original example. A derivation of $\texttt{adj}_S$ `binDigit` using the scheme we have described is shown in Figure 4.1. The first step (1) replaces the set function application with an application of `adj` inside a set capsule. The argument to `adj` inside the capsule is boxed to monitor its non-determinism. As the argument is evaluated, a choice arising from within it appears at the root of the box (2). A pull-tab step is applied with that choice as source, and when that occurs the cid ($i$) is added to the escape set of the enclosing set capsule (3). The choice escapes the set through another pull-tab step (4). Two other choices, $?_j$ and $?_k$, that arise from evaluating the `adj` function do not escape their respective set capsules.

$$
\begin{array}{lll}
\texttt{adj}_{\mathcal{S}}\ \texttt{binDigit} & \Rightarrow\!\!\!\!\!=\ & \texttt{\{adj } \boxed{\texttt{binDigit}}\texttt{\}} \qquad (1)\\
& \Rightarrow\!\!\!\!\!=\ & \texttt{\{adj } \boxed{\texttt{0 ?}_i\texttt{ 1}}\texttt{\}} \qquad (2)\\
& \Rightarrow\!\!\!\!\!=\ & \texttt{\{(adj } \boxed{\texttt{0}}\texttt{) ?}_i\texttt{ (adj } \boxed{\texttt{1}}\texttt{)\}}_{\{i\}} \qquad (3)\\
& \Rightarrow\!\!\!\!\!=\ & \texttt{\{adj } \boxed{\texttt{0}}\texttt{\}}_{\{i\}}\texttt{ ?}_i\texttt{ \{adj } \boxed{\texttt{1}}\texttt{\}}_{\{i\}} \qquad (4)\\
& \Rightarrow\!\!\!\!\!=^*\ & \texttt{\{(}\boxed{\texttt{0}}\texttt{-1) ?}_j\texttt{ (}\boxed{\texttt{0}}\texttt{+1)\}}_{\{i\}}\texttt{ ?}_i\texttt{ \{(}\boxed{\texttt{1}}\texttt{-1) ?}_k\texttt{ (}\boxed{\texttt{1}}\texttt{+1)\}}_{\{i\}} \qquad (5)\\
& \Rightarrow\!\!\!\!\!=^*\ & \texttt{\{(-1) ?}_j\texttt{ 1\}}_{\{i\}}\texttt{ ?}_i\texttt{ \{0 ?}_k\texttt{ 2\}}_{\{i\}} \qquad (6)
\end{array}
$$

FIGURE 4.1: Set function evaluation with boxed subexpressions. Boxes are introduced around arguments during set capsule creation (1) and enclose subexpressions to monitor for non-determinism. Boxes do not interfere with pattern matching and replacement occurs inside them (2). When pull-tabbing moves a choice out of a box, its cid is added to the escape set of the enclosing set capsule (3). If the expression in a set capsule is choice-rooted, then a pull-tab step (which creates a new set capsule) is performed only if the cid of that choice appears in the escape set of the capsule (4, but not after 6). If a variable in the RHS of a rule is bound to a boxed value, then references to that variable are boxed in the replacement (5). Ordinarily, the expressions in the right-hand sides of rules are unboxed (6).

### 4.3.3 Encapsulated Computations

In the FS, choice-rooted expressions are separated into individual elements in a queue. It is not difficult to see how an encapsulated computation is similar to a regular computation in the FS, except that certain choice-rooted expressions should not be processed by **D**.2, but rather by a different process that allows the choice to escape the set capsule. This suggests that the evaluation of a set function can be achieved through a recursive application of the **D** procedure with some modifications.

To accomplish this, we will now describe our scheme for nesting invocations of an extended version of the FS, which we call FS-$\mathcal{S}$. The extensions described here can

be used in combination with those described in Sect. 4.1 and Sect. 4.2. Our implementation ensures that computational resources are distributed between invocations in a way that prevents unproductive nonterminating computations within a capsule from blocking the production of values outside that capsule. The extended scheme FS-$\mathcal{S}$ is created as follows:

1. We introduce a representation of set capsules, which is an opaque data object with specific attributes that will be described later.

2. We introduce a new type of symbol called a *set monitor*, which represents boxed subexpressions. A set monitor has one successor, which is the expression it monitors.

3. We extend the REWR and PULL actions, as well as the definitions of pattern matching and variable binding, to account for set monitors and set capsules as described in Sect. 4.3.2.

4. We introduce a reserved function symbol (`evalS`) to perform encapsulated computations, and extend the rules of the **S** target procedure to evaluate `evalS`.

Items 2 and 3 above have already been discussed, so in the remainder of this section we will focus on set capsules and the evaluation of the expressions they encapsulate.

In Sect. 3.3 we saw that an invocation of the FS behaves like a coroutine that pauses to allow the values it produces to be consumed. In FS-$\mathcal{S}$, we achieve a similar

effect by defining the rules applied to `evalS` to make it behave like a suspendible invocation of the **D** target procedure. Every time a step is applied to `evalS`, at most one step is performed within the capsule, and then control is returned to the outer computation. The expression $\left[\overline{G}\right]_E$ denotes a set capsule with escape set $E$ whose live computations are those found in the sequence $\overline{G}$. The escape set will be omitted except where needed for the discussion.

To evaluate a set function, we apply `evalS` to a set capsule to reduce it to a value set. To begin this evaluation, we perform a rewrite step like the one shown below:

$$\mathtt{f_S}\ a_0\ \ldots\ a_n \quad \rightarrow \quad \mathtt{evalS}\left[\mathtt{f}\ \boxed{a_0}\ \ldots\ \boxed{a_n}\right]_\emptyset$$

This applies `evalS` to a new set capsule whose goal is an application of the ordinary function (`f`) corresponding to the applied set function ($\mathtt{f_S}$). The function arguments inside the capsule are the same arguments supplied to the set function, but each one is placed under a set monitor (indicated by a box). The escape set associated with a new capsule is initially empty.

Now, we will define additional rules of **S** to handle `evalS`. In Figure 3.1, each rule of **D** ends with a recursive call (**D**.1-5) or terminates the computation (**D**.6). To help us describe the behavior of `evalS`, we will now define a target procedure $\mathbf{D}^*$ that applies only a single step. This target procedure is identical to **D** except that wherever **D** would invoke itself recursively, $\mathbf{D}^*$ returns the argument to **D** in that

recursive call. Rule $\mathbf{D}^*.6$ returns an empty sequence. It is straightforward to show that applying $\mathbf{D}^*$ iteratively until the result is an empty sequence performs the same computation as applying $\mathbf{D}$.

The rules of $\mathbf{S}$ used to evaluate `evalS` are shown in Figure 4.2. These rules inspect the first live expression within the set capsule, and may then perform a pull-tab step (rule $\mathbf{SF}.1$) or produce a value (rule $\mathbf{SF}.3$). Between steps, control is returned to the outer context. Whenever the first of the live expressions within the capsule becomes choice-rooted, rule $\mathbf{SF}.1$ has an opportunity to intervene so that the intended choices escape the capsule through a pull-tab step. Otherwise, rule $\mathbf{SF}.2$ invokes $\mathbf{D}^*$ with that expression in its current position, resulting in the firing of $\mathbf{D}^*.2$ and the creation

$\mathbf{S}(\texttt{evalS}\left[g;\bar{G}\right]_E) =$

`case` $g$ `of`

| | | |
|---|---|---|
| `when` $a$ `?`$_i$ $b$`:` | `if` $i \in E$ `then` PULL$(g)$ | |
| | `else` REWR$\Big(\texttt{evalS}\left[\mathbf{D}^*(g;\bar{G})\right]_{E^+}\Big)$; | $\mathbf{SF}.2$ |
| `when` $g$ is a value`:` | REWR$\Big(g$`:evalS`$\left[\bar{G}\right]_E\Big)$; | $\mathbf{SF}.3$ |
| `default:` | REWR$\Big(\texttt{evalS}\left[\mathbf{D}^*(g;\bar{G})\right]_{E^+}\Big)$; | $\mathbf{SF}.4$ |

$\mathbf{S}(\texttt{evalS}\,[null]) =$ REWR$(\texttt{[]})$ $\mathbf{SF}.5$

FIGURE 4.2: Evaluation of set capsules in FS-$\mathcal{S}$. The reserved function `evalS` is evaluated by the rules of $\mathbf{S}$ shown in this figure. Rule $\mathbf{SF}.1$ executes a pull-tab step when a choice is in the escape set, $E$. Rules $\mathbf{SF}.2$ and $\mathbf{SF}.4$ apply one step of the encapsulated computation and then reapply `evalS` to the result. If either step identifies an escaping cid, that cid is added to $E^+$. When the encapsulated computation produces a value, rule $\mathbf{SF}.3$ makes it available outside of the capsule. Rule $\mathbf{SF}.5$ terminates the encapsulated computation.

of a fork in the computation within the capsule.

Similarly, rule **SF**.3 intervenes when a value appears in the capsule and produces that value in the context where the set capsule is situated. Therefore, rule $\mathbf{D}^*$.4 is not used and can be omitted in an implementation. The production of set elements in Figure 4.2 uses lists, but this is not essential. An alternative representation may be preferred because lists define an ordering between elements that should not exist in a value set. An implementation that provides an abstract interface to sets can choose a different representation. In general, the list constructor in **SF**.3 can be replaced with any symbol that constructs a value set from the element on its left-hand side and the value set on its right-hand side. The empty list in **SF**.5 can be replaced with a corresponding representation of an empty value set. An implementation may also make the value set data type opaque, requiring a function like `setmin` to be applied to access any useful information about an object of that type.

If none of the above cases apply, then one step is performed within the capsule (**SF**.4) or the encapsulated computation terminates (**SF**.5).

### 4.3.4 Escape Set Handling

As mentioned in Sect. 4.3.2, the $\mathbf{D}^*$ rule may identify a cid that needs to be added to the enclosing escape set. Therefore, we can consider $\mathbf{D}^*$ as having two outputs, even though only one is shown in Figure 4.2 for brevity. During recursive invocations of `evalS` in **SF**.2 and **SF**.4, the set capsules created should contain the appropriate

escape set, formed by adding any cid identified in a call to $\mathbf{D}^*$ to the escape set, $E$, that was passed as an argument. This "augmented" version of $E$ is written $E^+$ in the figure.

It is worth noting that an implementation could potentially improve efficiency by utilizing mutable set capsules. There is a bijection between nodes labeled `evalS` and the set capsules such nodes are applied to, as these are always created and destroyed together and no operation ever separates them or provides direct access to the set capsule. As a result, an implementation could use the same object for both the incoming and outgoing set capsules in rules **SF**.2-4 for efficiency.

Rule **SF**.1 performs a pull-tab step that splits one set capsule into two. This step is worth examining in more detail. An example of such a step is shown below:

$$\texttt{evalS}\ \left[a\ \texttt{?}\ b;\ \overline{G}\right]_E \quad \multimap\!\!\!\!\!\rightarrow \quad \texttt{evalS}\ \left[\overline{G};\ a\right]_E\ \texttt{?}\ \texttt{evalS}\ \left[\overline{G};\ b\right]_E$$

The two set capsules produced in this step have different sequences of expressions, but may contain many shared expressions. This is beneficial in terms of efficiency, as steps applied to shared expressions will also be shared between the capsules.

Both capsules also contain the same elements in their respective escape sets, but it is unclear whether each should each have its own copy or whether they should share a single escape set. On the one hand, it is possible that a pull-tab step needed for both capsules might cause one copy of an escape set to be updated without updating the

other. But since the step is needed for both capsules, the choice it applies to should escape both set capsules. On the other hand, it may seem possible that a choice could arise that should escape one set but not the other. Fortunately, it can be shown that the latter scenario cannot occur, so these capsules should share a single escape set.

We argue that all choices arising during the evaluation of a set function can be divided into two groups: those stemming from the set function itself, and those resulting from the input it is applied to. These groups determine whether a choice can appear in the escape set of an enclosing capsule. A pull-tab step, like the one shown above, that splits a set capsule does not affect this property. Therefore, a choice can either escape all capsules it appears in, or none of them.

Let $C = \left[\texttt{f}\ \boxed{a_0}\ \ldots\ \boxed{a_n}\right]$ be the set capsule produced by a rewrite step applied to $\texttt{f}_S\ a_0\ \ldots\ a_n$. We consider every way that a choice might be introduced. Choices that arise from a rewrite rule of `f` are unboxed, and this property is transitive. If `f` is rewritten to produce another function symbol $g$, then $g$ is also unboxed. Thus, if $g$ introduces a choice through rewriting, that choice is also unboxed. Choices that appear in an argument $\boxed{a_i}$ are boxed, and choices introduced by a rewrite step within a box are also boxed. If an expression is rewritten and references a boxed subexpression, the reference is also boxed. Therefore, if a rewrite step produces a choice by rewriting a subexpression of a boxed expression, the resulting choice is also boxed.

A pull-tab step performed by **SF**.1 produces two set capsules, $C_{\mathrm{L}}$ and $C_{\mathrm{R}}$. This

step does not add or remove any boxes, except that if the source of that step is boxed, the references to its left and right alternatives in the result are also boxed. Therefore, it does not affect the property that all choices can be divided into the two groups discussed. This means it is not possible for any choice to belong to both groups, so no choice can arise that should be added to the escape set of $C_{\mathrm{L}}$ but not the escape set of $C_{\mathrm{R}}$. It can occur that a choice is needed in one capsule but not the other, but in this case, adding its cid to the shared escape set does not cause any problems.

A complication arises when dealing with nested set function evaluations. This occurs when a boxed argument is used in a further set function expression. For example, consider the case where we have `f = g`$_{\mathcal{S}}$, which allows us to derive the following:

$$\texttt{f}_{\mathcal{S}}\ \texttt{a} \ \rightarrow\ \texttt{evalS}\left[\texttt{f}\ \boxed{\texttt{a}}\right] \ \rightarrow\ \texttt{evalS}\left[\texttt{g}_{\mathcal{S}}\ \boxed{\texttt{a}}\right] \ \rightarrow\ \texttt{evalS}\left[\texttt{evalS}\left[\texttt{g}\ \boxed{\boxed{\texttt{a}}}\right]\right]$$

In this example, the argument `a` is nested within two capsules and is also double-boxed. Any choice arising from this argument must escape both capsules. In order to properly handle this situation, we need to keep track of which capsule a box is associated with. Fortunately, this is easy to do by annotating each set monitor with an identifier or reference to the escape set that must be modified when a choice escapes it. When propagating set monitors, this information is preserved. When a pull-tab step moves a choice past a set monitor, the associated escape set is then modified.

## Chapter 5

## The Sprite Curry System

This chapter discusses our implementation of the Fair Scheme (FS) in the Sprite Curry system. A Curry program compiled by Sprite consists of a fixed portion and a variable portion. The fixed portion, which includes the implementations of the **D** and **N** target procedures, is integrated into a runtime library that is installed along with the compiler. The variable component is dependent on the program being compiled and encompasses program-specific **S**-rules and details of function and constructor symbols. During program construction, the variable component is compiled and linked with the runtime library.

In Sect. 5.1, we will examine the compilation process in detail, including the intermediate representations used by Sprite and the series of transformations that convert a Curry source program into its final executable form.

In Sect. 5.2, we shift focus to the runtime library. We will discuss the fundamental data structures and procedures that underlie the evaluation process, including the structure of expressions and the mechanisms for selecting rules and performing

replacements.

In Sect. 5.3, we take a closer look at expression evaluation and will consider how to efficiently apply the target procedures of the FS. Instead of performing recursive function calls as Figure 3.1 might suggest we scan expressions while looping applications of the target procedures. This performs essentially the same steps as the FS but with greater efficiency, resulting in faster program execution.

## 5.1 Compilation

The Sprite compilation pipeline is illustrated in Figure 5.1. It is the process by which Sprite converts Curry source programs into an executable form through a series of transformations. The pipeline is divided into two parts: the front-end and the back-end.

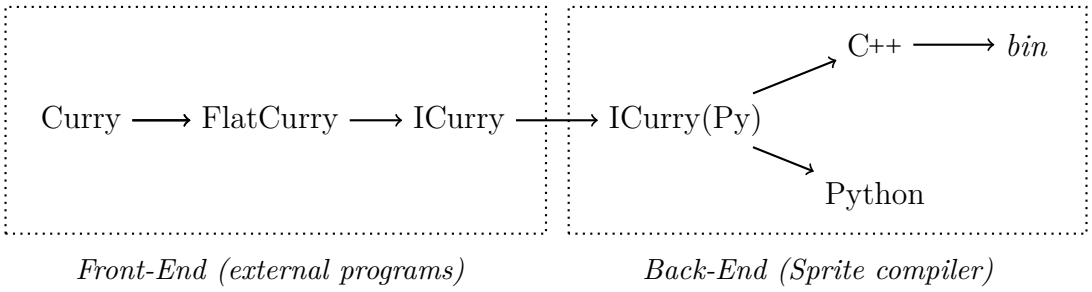



FIGURE 5.1: The Sprite compilation pipeline. The front-end converts Curry source code to ICurry via FlatCurry. The back-end reads ICurry into a Python-based representation and then converts that to either C++ or Python code. C++ is converted into machine-specific binary code, indicated as *bin* in the diagram.

The front-end uses two external programs provided by Hanus et al. to convert Curry into FlatCurry and then into ICurry. We will take a closer look at these representations in short order. Standard versions of both programs are available through the Curry Package Manager [92]. These preparatory steps are uniform across all forms of the final executable. The Sprite compiler proper is found in the back-end, which loads ICurry and then converts it into an executable form through either C++ or Python code. Here, the steps performed depend on the executable form being targeted.

Our strategy for compiling does not currently include any optimizations specific to Curry, but there is potential for adding them in the future. Our pipeline can handle any optimizer that improves programs represented as FlatCurry or ICurry. ICurry has been shown to be a good choice for optimizations [68]. Additionally, the C++ target benefits from optimizations offered by modern C++ compilers. Furthermore, it is possible to create Curry-specific optimizations by converting C++ to LLVM [93] and applying custom optimizations, and work in this area has been published [67]. Although the focus of the Python target is not on efficiency, its performance could potentially be improved through methods such as just-in-time compilation [94] or by cross-compiling to a more efficient target [95].

$$
\begin{array}{lcll}
D & ::= & f(\overline{x}) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & c(\overline{e}) & \text{(constructor call)} \\
 & | & f(\overline{e}) & \text{(function call)} \\
 & | & case\ e\ of\ \{\ \overline{p \rightarrow e;}\ \} & \text{(case expression)} \\
 & | & e_1\ or\ e_2 & \text{(disjunction)} \\
 & | & let\ \{\ \overline{x = e;}\ \}\ in\ e & \text{(let binding)} \\
 & | & let\ \overline{x}\ free\ in\ e & \text{(free variables)} \\
p & ::= & c(\overline{x}) & \text{(pattern)}
\end{array}
$$

FIGURE 5.2: The abstract syntax of a FlatCurry function [98, Figure 3].

### 5.1.1 FlatCurry

FlatCurry is an intermediate representation used by the PAKCS and KiCS2 Curry systems to translate Curry programs into either Prolog or Haskell [65, 66]. FlatCurry has also been used to develop an operational semantics of Curry [44], construct analysis tools [96], and to verify properties of Curry programs [4, 97]. FlatCurry has constructs for defining modules, types, and functions, however, for our purpose of compiling the **S**-rules of the FS, we only need to focus on function definitions. The abstract syntax for a FlatCurry function is shown in Figure 5.2.

FlatCurry simplifies and removes unnecessary features from Curry functions. It transforms each function into a single rule with a linear left-hand side that implements pattern matching as nested cases on the right-hand side, using variables from the left-hand side, let bindings, or free variable declarations to build replacements from variables, constructor calls, function calls, and disjunctions. This results in simpler

function definitions that are easier to compile than Curry. The FlatCurry form of the `zip` function is shown in the next example.

**Example 5.1.1.** A FlatCurry representation of the `zip` function is shown below:

```
1 zip x y = case x of
2     []     -> []
3     (a:as) -> case y of
4         []     -> []
5         (b:bs) -> (a, b) : zip as bs
```

We borrow Curry's syntax here to minimize the need for creating a new syntax. The multiple defining rules of the `zip` function (shown in Figure 2.2) have been combined into a single rule whose arguments are distinct variables. Pattern matching is implemented in the function body through nested case distinctions, with the cases structured in a manner that aligns with the definitional tree shown in Example 3.1.5. – *end example.*

Although FlatCurry simplifies Curry programs considerably, it is not an ideal representation for implementing the FS. Two constructs in particular pose difficulties. The first is a cyclical graph, such as the following:

```
1 inflist = let a = 0 : b
2               b = 1 : a in a
```

To translate this to an imperative language requires serializing its construction, but FlatCurry merely provides a definition of this expression, not instructions on how to build it.

The second is nested case expressions, an example of which is seen in the following definition of a function to compute the reciprocal of a number:

```
1 recip x = 1 /
2       case x == 0 of True -> failed; False -> x
```

Evaluating this expression in an imperative language requires pausing evaluation of the division operation while handling the case, then returning to the division. Similarly, evaluating the case involves first evaluating the subexpression `x == 0` and then returning to the case. Such nested subexpressions can occur to any depth and be arbitrarily complex, which makes an implementation more complex than necessary. Our implementation simplifies the generation of target code by converting FlatCurry to ICurry.

### 5.1.2 ICurry

ICurry is a procedural representation of Curry programs that is intended to simplify translations to imperative languages. It has been used for such targets as C, C++, Python, JavaScript, and Go [67, 98]. This step eliminates problems in FlatCurry by giving clear instructions for building cyclic expressions and evaluating nested expressions, making the compiler back-end simpler.

We will again focus on function definitions since the main activity of our compiler is to generate code for **S**-rules. The structure of an ICurry function is depicted in Figure 5.3. It consists of a block divided into three parts: variable declarations,

variable assignments, and a statement defining the replacement. This separation of variable declaration and assignment, which is exclusive to the imperative style, serializes the process of creating cyclic expressions and makes it easier to translate ICurry to the target language. For instance, to build the right-hand side of `inflist`, ICurry says to first allocate memory for variables `a` and `b`, then assign values to `a` and `b`, and finally return `a`. Each of these steps is readily expressed in an imperative target. The simplicity of the process hinges on using the address of `b` in creating the value for `a` before its contents are assigned (or vice versa).

| | | | |
|---|---|---|---|
| $D$ | ::= | $f = blck$ | (function definition) |
| $blck$ | ::= | $decl_1 \ldots\ decl_k\ asgn_1 \ldots asgn_n\ stm$ | (block) |
| $decl$ | ::= | `DECLARE` $x$ | (local variable declaration) |
| | \| | `FREE` $x$ | (free variable declaration) |
| $asgn$ | ::= | $v = exp$ | (variable assignment) |
| $stm$ | ::= | `RETURN` $exp$ | (return statement) |
| | \| | `EXEMPT` | (failure statement) |
| | \| | $case\ x\ of\ \{c_1 \to blck_1; \ldots; c_n \to blck_n\}$ | (case statement) |
| $exp$ | ::= | $v$ | (variable) |
| | \| | `NODE`$(l, exp_1, \ldots, exp_n)$ | (node construction) |
| | \| | $exp_1$ `OR` $exp_2$ | (disjunction) |
| $v$ | ::= | $x$ | (local variable) |
| | \| | $v[i]$ | (node access) |
| | \| | `ROOT` | (root of function call) |
| $l$ | ::= | $c$ | (constructor symbol) |
| | \| | $f$ | (function symbol) |
| | \| | `LABEL`$(v)$ | (node label symbol) |

FIGURE 5.3: The abstract syntax of an ICurry function [98, Figure 3]. A function consists of a symbol ($f$) and a block ($blck$) comprising variable declarations ($decl$), assignments ($asgn$), and a statement ($stm$). A statment defines a replacement for an application of this function and may involve pattern-matching through case statements and additional blocks. Elements expected to be provided by an implementation are represented in capital letters (e.g., `NODE`).

As with FlatCurry, pattern matching is carried out in the right-hand sides of functions. A statement can either return an expression (the replacement) or be a case statement that performs a distinction and continues to another block that ultimately returns a replacement. Exemptions are treated as a type of return statement (returning an expression labeled by $\perp$), in alignment with how the FS implements failure.

ICurry eliminates the issues associated with nested applications by utilizing auxiliary functions. For instance, it transforms the function `recip` into one that unconditionally rewrites to an expression using an auxiliary function as shown below:

```
1 recip x = 1 / aux x (x == 0)
2 aux x c = case c of True -> failed; False -> x
```

This approach simplifies the back-end as the evaluation of the case expression and subsequent division are handled by the rules of the FS.

This example highlights a key difference between the case expressions in FlatCurry and the case statements in ICurry. In ICurry, the discriminator in a case statement must be a variable rather than an arbitrary expression. This restriction eliminates the possibility of nested expressions, forcing the use of auxiliary functions that simplify the implementation.

Following the transformation of a source program into ICurry, the Sprite back-end is invoked to produce the parts of the program that depend on its function definitions.

### 5.1.3 Generating Target Code

The Sprite back-end converts ICurry into either Python or C++. The conversion process is relatively straightforward, as it only involves defining a translation for each ICurry construct shown in Figure 5.3. The real challenge is creating the runtime library, which must provide definitions for key objects (such as nodes and symbols), routines for node allocation and assignment, a mechanism for case distinction, access to the designated `ROOT` expression, access to successors, and implementations for replacement (REWR), pull-tab (PULL), and instantiation steps (INST) among other things. These details will be discussed in the next section.

The target languages we have selected are complementary. Python produces code that is easier to inspect, modify, and debug, but is less efficient. Its runtime system is simpler and memory is managed by Python itself. This means that one can start a Python debugger or interactive prompt without recompiling, and also inspect the results of compilation programmatically. It is also not difficult to modify the generated code by hand. These features can make the development process much smoother. A drawback of the Python back-end is that is uses recursion when invoking the FS target procedures, and so can reach Python's recursion limit for large computations. Therefore, it is not suitable for practical use with many real programs. On the other hand, the C++ target is designed for maximum performance and evaluates Curry programs as quickly as possible. This results in a more complex runtime system and

generated code, making it harder to debug, inspect, and develop, but greatly improves practicality.

When generating standalone Python, Sprite creates a file for each compiled Curry module. Loading one of these files in Python imports necessary Curry modules, creates a Python object for the Curry module, defines and loads its constructor and function symbols, and registers the module with the Sprite runtime. Running the file as a Python script also evaluates a goal, which can be built into the file (often as `main`) or specified on the command line. When generating Curry code dynamically, the Python code is executed immediately in the Python interpreter.

The process is similar when generating C++ code. In standalone mode, each Curry module is output as a C++ file that defines and registers the module. To build a Curry program, the necessary modules are compiled with a C++ compiler and linked with the Sprite runtime library. In dynamic mode, the C++ code is compiled into an Executable and Linkable Format (ELF) shared object and loaded with `dlopen` (on Linux). The runtime system coordinates these activities so that users can simply interact with Sprite through its Python interface. Usage details will be discussed in the next chapter.

## 5.2 The Runtime Library

Our focus now shifts to the specifics of the code generated by Sprite, with a particular emphasis on the C++ target. We will not delve into the Python target, as the general

approach is largely similar despite differences in programming techniques and other details.

In this section, we will examine the fundamental elements present in the Sprite runtime library. This includes the representation of expressions and the core methods utilized for selecting rules of the FS and performing replacements. These building blocks serve as the foundation of the implementation and are used to realize both the fixed components found in the **D** and **N** target procedures and the variable parts generated by the compiler back-end.

### 5.2.1 Expressions

Expressions in Sprite are represented as graphs, which are the central objects in our implementation. Unlike other representations where vertices and edges are stored in aggregate structures, graphs in Sprite consist of individual nodes. Each node represents a single vertex and directly stores the specific data associated with that vertex, including references to its successors. This approach is well-suited to the incremental changes that occur during stepwise computations.

Our implementation utilizes garbage collection, rather than reference counting, for memory management. Nodes are allocated from a contiguous pool of memory, and during the replacement process, some nodes may become unreachable and be considered as "garbage." The garbage collector runs periodically to reclaim this memory. Our implementation follows well-established practices in garbage collection and does

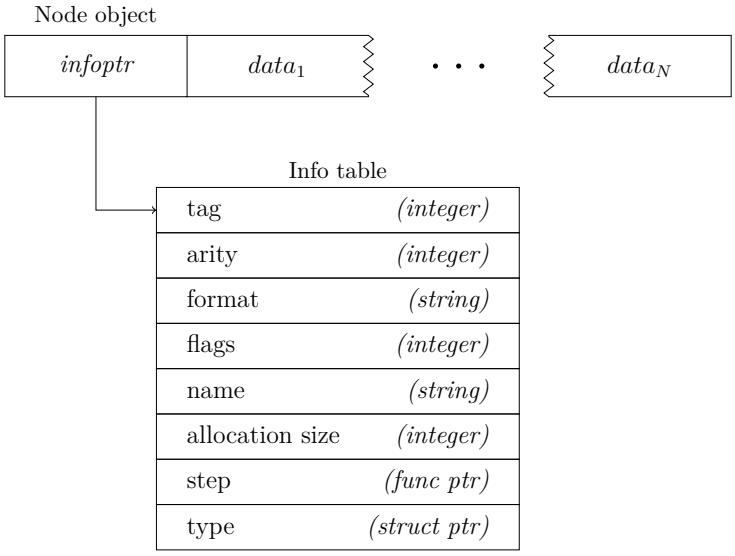



FIGURE 5.4: The node object memory layout. Each node consists of a pointer to a static info table followed by one or more data words. An info table describes the symbol labeling a node. This includes information about the name, arity, and kind of symbol, the format of the data section, and the function used to reduce expressions if this kind (when the symbol is a function).

not bring any new innovations to the field, so we will not elaborate further on this aspect of Sprite.

A diagram of the node memory layout is shown in Figure 5.4. Each node consists of two parts: an *info pointer* and a sequence of instance-specific data words. The info pointer refers to a read-only, compiler-generated table called the *info table*, which contains information about the symbol such as its name, arity, memory size, and other details. The data words contain the actual data, such as the integer value in an `Int` node or pointers to the successors of the node. The size of each node varies depending on its contents. For example, choice nodes contain both an integer (choice

identifier) and two pointers (to the left and right alternatives), while an `Int` node contains only a single word of data. The *format* field of an info table is a string of characters indicating the type of each data word, allowing graphs to be easily traversed without additional information. The info table also includes information specific to the symbol category, such as the constructor ordinal (for constructor symbols) and a target function (for function symbols) that performs a single step according to the compiled **S**-rules of that function.

Our runtime library defines objects and procedures to support the ICurry elements expected to be provided by an implementation, as indicated in Figure 5.3. For example, a construction procedure implements the `NODE` action, which allocates a node and sets its info pointer and data words. An ICurry variable (i.e., introduced by `DECLARE`) is similar to a pointer to a node, but also includes additional information to track set monitors encountered on the path to the expression it is bound to. This is used to box variable references in replacement expressions (as described in Sect. 4.3). An assignment procedure associates a variable with a node. Procedures corresponding to `FREE` and `OR` allocate a fresh free variable or choice identifier, respectively, during construction of a specific type of node. Another procedure returns the function-labeled node, `ROOT`, currently being evaluated.

This is sufficient to implement everything depicted in Figure 5.3, except for the case distinctions that select from among the **S**-rules of a function. This will be our next topic.

### 5.2.2 Rule Selection

The selection of **S**-rules in the FS is done by performing a case distinction on tags found in the info tables, as described in [67]. Case statements in ICurry identify the nodes to be examined, which are the inductive positions in definitional trees. These statements also specify the actions to be taken when a constructor is encountered, which align with rules **S**.1 and **S**.5 of the FS (see Figure 3.1). The actions for non-constructor nodes, defined in other **S**-rules, are combined with these actions to create the full set of actions to be executed when any node is found. The selection of rules in the other target procedures, **D** and **N**, is performed in a similar manner, except the positions to be inspected are predetermined rather than being dependent on the program being evaluated.

The possible tag values are chosen carefully to always differentiate between nodes that are handled by different rules in the FS. This process can involve several cases. For example, the rules **S**.3-5 of the FS inspect a node ($o$) and perform a rewrite step, pull-tab step, or recursive invocation of **S** when $o$ is labeled by a failure, choice, or function symbol, respectively. Sprite defines separate tags for failures, choices, and functions so that these can be easily distinguished. Additional rules such as **S**.x (Page 130) and **S**.$\beta$ (Page 143) define additional cases but do not alter this general pattern. The **D** and **N** target procedures similarly select from among several rules at once.

Figure 5.5 shows a selection of the tags used in Sprite. Each constructor of a

type is assigned a unique tag, counting up from zero. Since programs are well-typed, there is no need to compare constructors from different types, so constructor tags are not unique globally. Other nodes are assigned negative values. A complete match of an $f$-rooted expression, where $f$ is a function symbol, involves applying rule **S**.5 once for each branch encountered in a traversal of $f$'s definitional tree. Each step of the matching process performs a distinction between the constructor symbols of a certain type (and additional non-constructor symbols besides that). When the match succeeds, this culminates in the completion of a rewrite step (by the REWR action) when the rule **S**.1 finally matches, and in other cases results in a different action when a needed node labeled by another symbol, such as a failure, choice, free variable, or $\beta$, is encountered.

Our implementation compiles all of the **S** rules for $f$ into a single target procedure called its *step function*, whose address is stored in $f$'s info table. Unless a needed

| Name | Value | Description |
|---|---|---|
| `T_SETMON` | $-7$ | A set monitor. |
| `T_FAIL` | $-6$ | A failure. |
| `T_CONSTR` | $-5$ | A constraint ($\beta$). |
| `T_FREE` | $-4$ | A free variable. |
| `T_FWD` | $-3$ | A forwarding node. |
| `T_CHOICE` | $-2$ | A choice. |
| `T_FUNC` | $-1$ | A function. |
| `T_CTOR` | $\geq 0$ | A constructor. |

FIGURE 5.5: A selection of tag values used in Sprite. The constructors of each type are numbered $0, 1, \ldots$ in their declared order and other tags take negative values. Taken together, all tag values occupy a contiguous range of integers.

function-labeled node is encountered during the matching process, an invocation of a step function results in a single rewrite, pull-tab, or instantiation step being performed without invoking other step functions. This may involve several case distinctions, depending on the depth of $f$'s definitional tree. When a needed function-rooted subexpression is encountered, a data structure tracking the current position is updated and the step function returns, which causes processing of the nested subexpression to begin. The details of this procedure will be discussed later, but it is important to note that Sprite explicitly manages a heap-based stack rather than using the system stack. This avoids a great deal of function-call overhead and ensures that deep recursion does not lead to stack overflow.

To efficiently handle multi-way branches, we implement case distinction using C++ switch statements. The cases must cover all potential actions required to process any node at a particular point in the FS and rely on the specific arrangement of tag values shown in Figure 5.5 to ensure a contiguous range of integer indices. This is necessary for an optimizing compiler to be able to transform these switches into jump tables. However, instead of placing all the cases into a single switch, we use a two-step process because many actions are common to all of these switches. For example, the rule for a failure (i.e., tag value `T_FAIL`) always results in rewriting a node with the failure symbol. Duplicating the code for these cases everywhere would bloat compiled programs, increase their size, cause more unique instruction-containing memory addresses to be loaded, and potentially increase their execution

time. Therefore, our runtime provides a function to head-normalize needed nodes. This function handles non-constructor nodes with standard actions and otherwise defers so that a second switch, unique to the specific function being compiled, can provide function-specific cases. Although this approach roughly doubles the number of case distinctions, it can significantly reduce the amount of duplication in programs.

### 5.2.3 Replacement Actions

Implementing the REWR, PULL and INST target actions involves constructing expressions and performing replacements. We have already discussed construction, so we will now discuss replacements in our implementation.

To replace a node means to change its label, embedded data, and successors. It is important to change the node itself rather than simply redirect a particular reference to it, since the point of using term graphs is to share steps performed at common subexpressions. In the graph implementation we have described, this can be accomplished by overwriting the info pointer and data words. But a complication arises from the fact that nodes have varying sizes based on their contents. For example, the unary function `positives`, defined on Page 33, has a size of two words (one info pointer and one successor), while the non-null list constructor has a size of three words (one info pointer and two successors). Because of this, the step `positives [1]` $\rightarrow$ `1 : positives []` involves replacing a smaller node with a larger one, so a straightforward approach is not possible.

Instead, we implement replacement using special objects called *forwarding nodes*, denoted $\hookrightarrow$. A forwarding node holds a reference to another node called the target, and when one of these is encountered, the implementation simply excises it and continues as though it was never there. With this, the rewrite step above produces the expression `1` $\hookrightarrow$`: positives []`, where the node previously labeled `positives` has been relabeled $\hookrightarrow$ and its successor set to point to the replacement, a list constructor. This approach partially resolves the original issue, but nullary functions pose a problem because their nodes are smaller than a forwarding node. To avoid this, we set a minimum size of two words for function-labeled nodes to ensure that every redex provides enough space for its replacement.

With this, we have everything needed to create expressions and perform the runtime actions associated with the FS. Next, we will describe how Sprite puts these pieces together into a complete evaluation system for Curry expressions.

## 5.3 Expression Evaluation

With the fundamental components of the runtime library now in place, we turn our attention to evaluating expressions. Our implementation adheres to the actions specified by the FS but modifies certain details to enhance efficiency. In Chapters 3-4 we described the FS as three mutually recursive procedures. Although this method was suitable for defining its behavior, directly translating it would lead to excessive function calls, inefficient use of the runtime's call stack, and suboptimal performance.

To see this, consider the list reversal function:

```
1 reverse []     = []
2 reverse (a:as) = reverse as ++ [a]
```

A computation reversing the list `[1,2,3]` using the FS would involve a step, given by `((`[`[] ++ [3]`]`) ++ [2]) ++ [1]` $\rightarrow$ `([3] ++ [2]) ++ [1]`, whose redex is found by three invocations of the **S** target procedure. In general, reversing a list of length $n$ would require searching for redexes at a depth of up to $n$ nested invocations of **S**, which, if implemented naïvely, would involve $n$ function calls and require stack space proportional to $n$. This runs the risk of overflowing the stack for long lists and involves many more function calls than necessary.

Moreover, a naïve implementation would also be inefficient in the way it searches for redexes. An application of **D** scans an expression by repeated invocations of **N**, then performs one action and moves the expression at the front of the computation queue to the end. As the computation continues, that expression moves through the queue and eventually arrives again at the head and is reconsidered. At that point, **D** is applied once again and the scan is repeated from the beginning. Restarting in this way is not only inefficient but also makes the evaluation complexity dependent on seemingly irrelevant details, such as the location of a redex in a list. For example, reducing an element at the front of a list would be more efficient than reducing one at the back.

We aim to do much better. The scanning procedure in Sprite continues from a nearby location after each step, enabling it to complete in a single pass what would require multiple passes with the naïve approach. This changes the order in which steps are taken. In Sect. 5.3.1 we present the details involved and discuss the potential issues that arise. We discuss how target procedures can be looped to improve efficiency while preserving important properties of the FS. In Sect. 5.3.2 we present the core data structures used to orchestrate computations in Sprite.

### 5.3.1 Looping Invocations

To increase the ratio of useful steps to useless graph traversal we will loop invocations of the **N** and **S** target procedures (see Figure 3.1). This means that when **N**.4 receives control back from its invocation of **S** or when **S**.1 completes a rewrite step, neither will automatically return control to their respective callers but may instead iterate on the updated expression.

To illustrate, consider a list $L = [a_1, \ldots, a_n]$ in which a certain element ($a_i$) is a function-rooted expression that can be derived to a value by deterministic rewrite steps as $a_i \rightarrow \ldots \rightarrow v$ and every other element is an integer (or other irreducible expression). A naïve implementation of the FS would traverse the list up to $a_i$, apply a single step and then start over, traversing the list again before applying the next step. Our objective is to complete this computation in a single pass by traversing the

list up to $a_i$, applying as many steps as necessary to obtain $v$, and then proceeding through the remaining list elements.

This involves changes to our evaluation strategy such as the following:

1. Loop invocations of **S** until the argument is not function-rooted.

2. Loop invocations of **N**.4 unconditionally, applying **N**.

3. Extend **N** to accept any expression.

The first change causes **S** to iterate on the node $a_i$ until it is no longer function-rooted. The second change causes **N** to be reapplied once **S** returns, which allows the evaluation to continue at the $i^{\text{th}}$ list element without traversing the list again. Although $a_i$ is constructor-rooted at this point, the application of rule **N**.3 may lead to additional steps being applied in a subexpression of $a_i$. The third change is necessary to extend the rules of **N** to handle additional possibilities, since control does not return to **D**. Ordinarily, the return to **D** and the existence of rule **D**.3 means that **N** does not need to consider failures. But since we are looping invocations of **N** this case must now be handled. Determining the actions of these additional rules is straightforward and amounts only to considering which target procedures would be invoked in the absence of looping and duplicating their effects.

The list of changes provided is not comprehensive, but serves as an illustration of the required changes. In general, whenever a target procedure would return control,

we must consider how the next invocation of **D** would proceed and loop it in a way that eliminates unnecessary target procedure calls. For example, apart from the aforementioned modifications, we must also modify the **N** target procedure to ensure that it follows pull-tab steps with a loop of **N** at the target of the pull-tab in order to execute a series pull-tab steps efficiently. This adjustment allows the following steps to be performed without rescanning the list:

```
[a, b, c ? d]  ⇉  a : ([b, c] ? [b, d])
               ⇉  [a, b, c] ? [a, b, d]
```

Looping target procedures improves the complexity of computing list $L$ by a factor of $i$ but raises two concerns. First, an invocation must be looped only when doing so does not cause unneeded steps to be performed. Second, the evaluation process must never become stuck on a non-terminating computation such that it fails to ever switch contexts, since in order to preserve completeness we must ensure that computational resources are shared among all expressions that could potentially reduce to a value.

To allay the first concern, we can note that if applying **S** to a function-rooted expression ($e$) results in a function-rooted expression ($e'$), then the call may safely be looped as $\mathbf{S}(e')$: since $e$ was needed, we can be certain that $e'$ is also needed. Similarly, we can loop any invocation of $\mathbf{N}.4$, such as $s = \mathbf{N}.4(e)$, without introducing unnecessary steps. By examining Figure 3.1, we can see that the call sequence for $s$ involves invoking $\mathbf{D}.5$ followed by zero or more invocations of $\mathbf{N}.3$, and finally $s$ itself.

If control is returned to **D**, the expression currently at the front of the computation queue will eventually return there. When it does, if rule **D**.5 matches, the same sequence of **D**.5 and **N**.3 invocations leading to $s$ will occur again because only a subexpression of $e$ was replaced by $s$. If a rule prior to **D**.5 matches, a replica of that rule was added to **N** so the same action occurs when looping. Thus, looping the call to **N** in rule **N**.4 simply shortcuts the process by which the element at the front of the queue is rotated to the back, works its way to the front, and is reconsidered. This does not introduce unneeded steps.

The second consideration requires us to add something. The danger in looping invocations is that it skips the context switch that allows other live expressions to receive resources so that they might produce values before the current one completes (which might never occur). A context switch need not occur after every step, but must occur every so often to retain the completeness property. We do this by means of a step counter. Our implementation keeps track of the number of steps performed in each invocation of **D** and if after a prescribed number of steps the invocation has not completed (for example, by processing a choice in **D**.2, by failing in **D**.3, or by producing a value in **D**.4), then a context switch is forced.

These changes allow us to construct a procedure that efficiently scans expressions and applies computational steps without restarting the scan unnecessarily and without becoming stuck on non-terminating computations. The implementation of this will be discussed in the next section along with other elements of the runtime

orchestration.

### 5.3.2 Runtime Orchestration

Each time Sprite begins evaluation of a Curry expression, it creates data structures for managing the computation state and invokes the **D** target procedure. As values are produced, control is returned to an external caller and when additional values are requested the computation continues. When the **D** procedure terminates, the computation state is destroyed. In this section we will examine these data structures and the evaluation process in detail.

Each activation of the evaluation system is managed by a data structure called a *runtime state* (RTS), which coordinates the evaluation of one goal. The principal component of this is a computation queue, made up of *configurations* that combine an expression, fingerprint, constraint store, binding store, scan state, and other data. The RTS keeps only configurations with live expressions, meaning expressions that are not known to have no values. Evaluation starts by creating a new RTS object with a single configuration, whose expression is the goal being evaluated. The evaluation process involves inspecting the configuration at the head of the queue (called the "current" configuration) and applying steps, which may reduce the expression to a value, divide it into new configurations, cause the constraint store or binding store to be updated, or lead to the conclusion that it is inconsistent. Any number of RTS objects may exist simultaneously and, thanks to the structure of expressions, steps

are shared between them. Certain data structures, such as the pool of available choice identifiers, are shared so that these interoperate properly. The RTS also tracks the number of steps taken in order to periodically rotate its queue.

A fingerprint is a record of the choices made during the evaluation process and is initially empty. The evaluation process splits configurations when choice-rooted expressions are encountered, leading to the creation of new configurations referred to as children. For example, if the parent configuration contained the expression $\texttt{a}\ ?_i\ \texttt{b}$ with fingerprint $\phi$, the first child would contain the expression `a` with fingerprint $\phi \cup \{i \mapsto \textsc{l}\}$, and the second child would contain the expression `b` with fingerprint $\phi \cup \{i \mapsto \textsc{r}\}$. This process is only performed if it results in a consistent computation. If $\phi$ already includes the decision outcome $i \mapsto \textsc{l}$, for example, the second child would be inconsistent and would not be created.

Constraints arise when the strict equational constraint is applied between two free variables, as discussed in Sect. 4.2. This leads to the creation of nodes labeled by the $\beta$ symbol, which are subjected to pull-tab steps. When a configuration rooted by $\beta_{\texttt{x}\leftrightarrow\texttt{y}}$ is examined, that node is removed and the constraint store is updated to note that `x` and `y` must have identical choice outcomes. At this point, the fingerprint is inspected to determine whether the current configuration has become inconsistent and if this configuration is forked in the future, the constraints will be considered to avoid the creation of inconsistent configurations.

Bindings arise from applications of the non-strict equational constraint and always bind a free variable to an expression. Like information about strict equational constraints, information about bindings is carried to the roots of expressions by $\beta$ nodes. When a node labeled $\beta_{\mathtt{x}\mapsto e}$ is examined, it is removed and the bindings are updated. The INST target action consults this data structure when determining how to replace a free variable occurrence.

Each configuration includes a scan state that manages a depth-first exploration of the expression in a configuration by moving a cursor through it. The scan can progress by either descending into the expression under the cursor or moving the cursor to the next sibling under a constructor-rooted expression. These actions correspond to two steps in **N**.3: applying **N** to a term $x_i$ and moving to the next term $x_{i+1}$. In addition to the cursor location, the scan state also keeps track of the path to the cursor and any remaining subexpressions to be explored at each node along that path. In this way, its structure is analogous to a stack of loops over node successors, nested to arbitrary depth. This serves a similar purpose as the system stack would in a straightforward implementation involving recursive function calls, but has the advantage that each configuration maintains its own scan, rather than using a single shared, global stack.

Evaluation proceeds by repeatedly examining the node under the cursor and performing a control transfer as described in Sect. 5.2.2. When the next action is simply to invoke a target procedure, the cursor position can be updated accordingly. However, if the action involves a target action (REWR, PULL, or INST), a replacement is

also made at the cursor. Recursive invocations of **S** by rule **S**.5 occur in step functions and are achieved by updating the scan state to reposition the cursor at the new potential redex and returning control. This ensures that step functions always return, rather than recurse and consume system stack space.

Many of the RTS components can be implemented using well-known data structures. For example, the computation queue in Sprite is simply a `std::deque` from the C++ standard library. The constraint store is an instance of a disjoint set union, so we use a straightforward implementation of the UNION-FIND data structure and algorithm [99]. Other components require careful balancing of time efficiency and space efficiency. For example, our fingerprint implementation uses a segmented, hierarchical design with copy-on-write principles, which is efficient for both small and large fingerprints. A fingerprint is represented as a tree with branches and leafs, where leafs store choice outcomes in a compact, bitwise fashion, and branches contain multiple subtrees and are reference-counted for sharing between fingerprints. When a configuration forks, only the root pointer is copied and its reference count is incremented. Inserting a choice outcome involves modifying unique nodes directly or copying shared nodes first. This approach prevents computations from progressively slowing down as fingerprints increase in size, which occurs if a standard implementation such as `std::map` is used, since that would copy all choice outcomes.

To evaluate set functions, we require a representation of set capsules, which were introduced in Sect. 4.3. A set capsule is implemented as a node with two data words

that hold a reference to the escape set and a computation queue. To enable control transfers into set functions, the RTS object maintains a stack of computation queues instead of a single queue. During the evaluation of `evalS`, its queue is pushed onto the stack and **D** is invoked recursively. After the nested invocation returns a value or when a context switch is required, the stack is popped.

This concludes our description of the Sprite implementation. We have explained how the FS is put into action, including adjustments such as looping target procedures. These changes significantly enhance the efficiency of our implementation. However, they are necessary because our definition of the FS was intended to make its properties easily provable rather than to create an implementation that maximizes efficiency via direct translation. In the next chapter, we will shift our focus to the practical use of Sprite. We will provide an in-depth examination of the Python interface to demonstrate how Curry code can be compiled and loaded into Python, how Curry expressions can be built and evaluated, and how these distinct programming domains can be integrated. To cap it off, we will develop a simple application to demonstrate how Curry can be embedded into Python.

# Chapter 6

## Using Sprite

In the previous chapters, we presented our evaluation strategy for Curry programs and described our implementation, Sprite. This chapter focuses on the practical aspects of using Sprite, particularly its high-level Python interface. We will discuss why and how one might use Sprite, and what sets it apart from other implementations. In the next chapter, we will present benchmarking data and efficiency analysis.

Sprite's close integration with imperative programming through its Python interface allows for a simplified programming process. Dynamic definition, compilation, and loading of Curry code are possible, as well as interoperability between programming paradigms. This allows for easy embedding of Curry in Python and extending Curry to use Python. By lowering the barrier between paradigms, programmers have more freedom in choosing an implementation approach for each task.

In Sect. 6.1, we discuss the high-level features of the Python interface, while Sect. 6.2 presents an example of embedding Curry in an imperative program.

## 6.1 The Python Interface

The Sprite Python interface offers several options for transforming and executing Curry code. It includes a static compiler that converts Curry files into executable formats. Additionally, the interface provides functions to load or define Curry code dynamically, compile it on-demand, and execute it interactively.

Qualitatively speaking, the capabilities for compiling Curry into native programs do not add much over other Curry systems, and there is not much to say about their use. Therefore, in the following discussion, we will focus on the unique ways that the Python interface of Sprite can be used to interact with Curry through a Python interpreter. Using Python in conjunction with Sprite can simplify the integration of Curry into a project, especially for developers who are unfamiliar with functional logic programming. Integrating Sprite into a Python project does not incur performance or other penalties to parts of the project that do not use it. In other words, it adds extra capabilities without subtracting from existing ones.

Sprite seeks to enable Curry to be used as transparently as possible. The interface installs import hooks, which means that Curry code can be loaded using the standard Python import mechanism, and Curry modules appear in Python as regular modules. Sprite also provides ways to convert between Python and Curry for simple data types. These conversions preserve semantics when crossing the language boundary and, in particular, Curry evaluations are represented in Python using a mechanism

that preserves their delayed semantics and allows them to be discarded when not all values of a computation are required.

Python's popularity, active community, and support for a wide range of applications make it an attractive language for integrating with Curry. Aside from this, Python serves as a gateway to other programming domains, since it is possible to extend Python with modules built in other languages, such as C and C++. Python is used for machine learning [12–14], numerical and scientific computing [15–17], data analysis [18, 19], computer vision [20], graph analysis [21], HTTP servicing [22–24], web application development [25], web scraping [26], parser generation [27], and much more.

Python's functional programming features, such as lambda functions and list comprehensions, simplify integration with Curry. *Generators*, a lazy representation of computations, are particularly helpful in representing Curry computations. Generators are coroutines constructed by using the `yield` keyword inside a function. They can be used to represent computations lazily. When a value is taken from a generator by applying the `next` function, control returns to the caller and the generator is retained. If `next` is applied again the computation continues where it left off. This can be used to represent arbitrary computations. For example, the following generator produces an infinite sequence of Fibonacci numbers:

```
1 def fibs():
2   a,b = 0,1
```

```
3     while True:
4       yield a
5       a,b = b,a+b
```

To print the first few elements, one can write:

```
1  fib = fibs()
2  for _ in range(8):
3    print(next(fib))
```

This prints the sequence $0, 1, 1, 2, 3, 5, 8, 13$.

Sequences can be manipulated as well by functional techniques. Two functions `take` and `drop`, to take or drop a specified number of sequence elements, can be defined as follows:

```
1  def take(n, gen):
2    for _ in range(n):
3      yield next(gen)
4
5  def drop(n, gen):
6    for _ in range(n):
7      next(gen)
8    return gen
```

Using these, the expression `take(3, drop(4, fibs()))` produces the fifth, sixth, and seventh Fibonacci numbers, i.e., 3, 5, and 8. Using the higher-order function `map`, one can transform these elements into their squares as shown here:

```
map(lambda x: x*x, take(3, drop(4, fibs())))  # [9,25,64]
```

Equivalently, a list containing the squares can be defined using a comprehension:

```
[x*x for x in take(3, drop(4, fibs()))]
```

Capabilities such as these make Python an ideal choice for integrating with Curry. Since Curry expressions are multi-valued, a Curry evaluation must be represented in the host language as an object with delayed evaluation. Python generators provide an excellent language feature for this purpose and other functional capabilities in Python fit naturally with Curry.

Next, we demonstrate some basic features of the Sprite Python interface. We do not provide an exhaustive coverage of Sprite features here but aim to give the reader a general sense of how Curry and Python interact. The Sprite software comes with a user guide and other reference materials that document it in detail. In the examples, Python commands are prefixed with the default Python prompt (`>>>`) and any output is shown below that.

### 6.1.1 Importing Curry Code

To use Sprite in Python, one first imports it with the following statement:

```
>>> import curry
```

This uses the standard Python syntax for importing code modules. Curry modules are imported in similar fashion, but to avoid ambiguities between Curry and Python modules, they are loaded relative to the virtual module `curry.lib`, which identifies

them as coming from the Curry library. A Curry module named `Peano` could be loaded into Python using the following command:

```
>>> from curry.lib import Peano
```

The non-relative import statement `import Peano` would be ambiguous if we allowed it to import Curry code, since Python would not know whether to look for the Python module `Peano.py` or the Curry module `Peano.curry`.

Sprite searches for Curry modules using the paths listed in `curry.path`. This is initialized with a system path that allows built-in modules to be found. Users can add additional paths by setting the environment variable `CURRYPATH` or by modifying `curry.path` once Python is running. For instance, to search for Curry code in the current directory, the following command to insert `'.'` at the path head could be issued:

```
>>> curry.path.insert(0, '.')
```

With this, if the current directory contains a file named `Peano.curry`, then it can be loaded with the previous statement.

From Python's perspective, `Peano` is an ordinary Python module. It contains Python objects representing the constructor and function symbols defined in the Curry module, `Peano.curry`, which are compiled by Sprite during the import process.

Suppose `Peano.curry` contains the following code:

```
data Peano = S Peano | Z

toNum :: Peano -> Int
toNum Z = 0
toNum (S n) = 1 + toNum n
```

Then the `Peano` module would contain the symbols `S`, `Z`, and `toNum`:

```
>>> Peano.S
<curry constructor 'S'>
>>> Peano.Z
<curry constructor 'Z'>
>>> Peano.toNum
<curry function 'toNum'>
```

Python provides excellent tools for introspection, which allow us to inspect these symbols further. For example, here are some of the properties of `Peano.S` (the full list of attributes can be found with Python's `dir` command):

```
>>> Peano.S.fullname
'Peano.S'
>>> Peano.S.name
'S'
>>> Peano.S.typename
'Peano.Peano'
>>> Peano.S.info.arity
1
```

Function symbols contain similar information, corresponding to what is stored in the info tables (see 5.2.1) produced by Sprite, and also provide access to the generated code. For example, to display the code produced for `toNum`, the following command can be issued:

```
>>> print(Peano.toNum.getimpl())
```

Having the generated code available programmatically means it can be accessed by Python's debugger, so one can step through it to learn about its structure or to aid in debugging. We will not go through the generated code in detail, though interested readers may find it instructive to review that in detail.

Curry code can also be compiled dynamically with `curry.compile`. The following creates a function to add Peano numbers:

```
1 Add = curry.compile(
2   '''
3   import Peano
4   add :: Peano -> Peano -> Peano
5   add Z     m = m
6   add (S n) m = S (add n m)
7   '''
8   )
```

Note that in Python triple-quotes deliminate multi-line string literals. In this example, `curry.compile` returns a new module that contains the function `add`.

### 6.1.2 Constructing Curry Expressions

Curry expressions can be built in two ways. One option is to construct them in Python using the `curry.expr` function, which converts a description of a Curry expression into that expression. A sequence starting with a symbol specifies a node, whose label is given by the first element, and whose successors are defined by converting the

remaining elements recursively. Subexpressions can be specified using Python lists. For instance, the following statement constructs the Peano number `S (S Z)`:

```
>>> two = curry.expr(Peano.S, [Peano.S, Peano.Z])
```

`curry.expr` automatically converts built-in types including lists, tuples, numbers, strings, and Booleans. For instance, the following constructs a list of pairs of different types:

```
curry.expr([(True, 1), ("Curry", 3.14)])
```

At first, list conversion might seem ambiguous, but no ambiguity exists. A list argument specifies a node construction if and only if its first element is a Curry symbol. There is no representation of Curry symbols in Curry, so every list passed to `curry.expr` unambiguously specifies either a node or a Curry list. Accordingly, the expression `Z = curry.expr([Peano.Z])` constructs a node representing zero rather than a list because the first element is a symbol. On the other hand, `curry.expr([Z])` constructs a Curry list because `Z` is a node rather than a symbol.

Python provides a rich syntax, and `curry.expr` can be used to build arbitrary complex expressions. Curry expressions are passed through untouched, so calls to `curry.expr` can be nested. Cyclic expressions can be built using named arguments and the helper function `curry.ref`, which specifies a reference to a named subexpression that should be passed as a keyword argument to `curry.expr`. For instance, consider the following example that constructs the Curry expression `let a = S a in a`:

```
inf = curry.expr(
    curry.ref('a')                # ... in a
  , a=[Peano.S, curry.ref('a')]   # let a = S a ...
  )
```

This call (to `curry.expr`) involves two arguments, one of which is positional and the other a keyword with name `a`. Each keyword argument supplies a description that is converted as if `curry.expr` were applied to it. Keyword names can be chosen freely by the caller so long as there are no unresolved references. The expression being constructed is formed only from the positional argument, with the keyword(s) providing context.

We can use the built-in function `id`, which gives a node's memory address, and subexpression access through square brackets to confirm that `inf` is indeed cyclical:

```
>>> id(inf) == id(inf[0]) == id(inf[0][0])
True
```

The second way to construct expressions is by calling `curry.compile` with the argument `mode='expr'`. This method is safer but slower. Unlike expressions built with `curry.expr`, which bypass the Curry front-end, expressions built with `curry.compile` undergo type-checking. This eliminates the possibility of introducing type errors, which can lead to undefined behavior. Using the second method, we can build an expression specifying the addition $1 + 2$ in the Peano system as follows:

```
onePlusTwo = curry.compile(
    'add (S Z) (S (S Z))'
```

```
3     , mode='expr', imports=[Add, Peano]
4     )
```

To resolve the symbols, we must provide a list of imported modules. In this we include the dynamically-compiled module `Add`, defined earlier, to provide the definition of `add`.

### 6.1.3 Evaluating Curry Expressions

Sprite provides the `curry.eval` function to evaluate Curry expressions. This function produces a generator that yields all the values of the expression. To begin evaluation of the Peano addition represented by `onePlusTwo`, we can write the following:

```
>>> comp = curry.eval(onePlusTwo)
```

`comp` represents a lazy computation in Curry. Since this is a generator, we can apply `next` to obtain the value:

```
1 >>> print(next(comp))
2 S (S (S Z))
```

The result of this evaluation is a Curry expression of type Peano. If we want to convert it to an integer, we can apply `toNum` as shown below:

```
1 >>> goal = curry.expr([Peano.toNum, onePlusTwo])
2 >>> comp = curry.eval(goal, converter='topython')
3 >>> print(next(comp))
4 3
```

In this example, a converter named `'topython'`, provided by Sprite, is used to convert a Curry integer into a Python integer. Without it, the result would be a Curry expression rather than a Python object of type `int`.

For multi-valued expressions, the generator produces each element of the expression as a separate value. In addition to using `next`, all values of a generator can be taken by iterating over it with a `for` loop. As an example, consider the following program that finds solutions that satisfy a small Boolean expression:

```
>>> Sat = curry.compile(
  '''
  sat :: Bool -> Bool -> Bool -> (Bool, Bool, Bool)
  sat a b c | ((a&&b) || (b&&c)) && not (a&&c) = (a, b, c)
  goal :: (Bool, Bool, Bool)
  goal = let x,y,z free in sat x y z
  '''
  )
>>> for value in curry.eval(Sat.goal):
...   print(value)
(False, True, True)
(True, True, False)
```

In this example, the expression over three Boolean variables ($a$, $b$, and $c$) is satisfied when $a \wedge b$ or $b \wedge c$ is true, as long as $a \wedge c$ is not. The Curry evaluation produces two values as separate elements, each representing one solution computed by Sprite. Though it is not apparent from the representations shown, the values printed are Curry expressions because the `'topython'` conversion was not specified as in the previous example. Depending on how the result of a Curry computation is used, it might be more appropriate to use either representation.

In addition to representing delayed Curry computations in Python, Sprite also allows data to flow lazily in the opposite direction. Therefore, it is possible to pass a Python generator to Curry without eagerly evaluating it. Whenever `curry.expr` encounters a generator, it treats it as a computation that produces a list. To see how this works, we return to the `fibs` function, a Python implementation of the infinite sequence of Fibonacci numbers. The following example uses the Curry `take` function to take the first ten Fibonacci numbers from a sequence generated by Python:

```
1 >>> from curry.lib import Prelude
2 >>> fib10 = curry.expr([Prelude.take, 10, fibs()])
3 >>> comp = curry.eval(fib10, converter='topython')
4 >>> print(next(comp))
5 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

This bidirectional interoperability is a powerful mechanism for constructing hybrid programs. With this, users are free to cross the boundary between Python and Curry whenever doing so is convenient, and so hybrid programs using Sprite can more easily select whichever programming paradigm is best suited to the problem at hand. In the next section, we present an application that demonstrates the usefulness of this approach.

## 6.2 Embedding Curry: Blocks World

In order to demonstrate a practical use of the Python interface, we shall develop a small application that makes good use of Curry and Python together. This is a

web application that solves a classical problem in artificial intelligence called "Blocks World." Users can enter an instance of the Blocks World problem into a form in their internet browser and then have our program display a solution. This approach could also be used to create a service that solves Blocks World problem instances for remote clients. To create the web portion of this application, we will use a Python micro web framework called *Flask* [25]. This makes creating the application straightforward. The Blocks World problem itself is well suited to Curry. This is a well-known problem in the automated planning domain, where the concern is to compute strategies or action sequences that achieve specific goals within a formally modeled domain [100]. When a problem domain can be determined ahead of time, this approach allows one to derive and evaluate solutions for an autonomous actor (such as a robot or self-driving vehicle) to carry out at a later time.

The value of this exercise lies in its demonstration of how Sprite allows one to easily integrate Curry into an imperative programming domain. The Blocks World problem is known to be NP-hard [101, 102], so a reasonable strategy is to non-deterministically try all possible solution paths. Implementing such a solution in Python would be more complex and involve more fine details not directly related to the problem, since it would need to manage the search space. Using Curry to implement the web portion of the application is also not ideal, particularly since Flask makes this part so simple. Although Curry web libraries exist, the greater prevalence of Python, the size of its programming community, and the availability of training materials and professional

support make it a sound choice for web programming. These factors motivate us to build a hybrid application in which Curry is used only to solve instances of the Blocks World problem, while all other aspects of the implementation are done in other ways.

Blocks World is a generalization of the Tower of Hanoi problem discussed on Page 13. It involves a "world" made up of several stacks of labeled blocks, and a solution is a sequence of moves that transforms an initial configuration into a desired configuration. This can be described through a list of moves or a sequence of configurations called a trace. In each move, the top block of a stack is removed and placed on top of a different stack. In our simplified version of the problem, there are only five blocks labeled $A$, $B$, $C$, $D$, and $E$, three stacks, and no restrictions on which blocks can be stacked on top of others. We will not elaborate on variations or address input validation for the demonstration. An instance of the Blocks World problem is given in the following example.

**Example 6.2.1.** An instance of the Blocks World problem is given below:

*Initial configuration*: $A$ sits atop $B$ in the first stack.

*Final configuration*: $A$ sits atop $B$ in the third stack.

A solution to this problem is the following sequence of moves:

1. Move block $A$ from the first stack to the second stack.

2. Move block $B$ from the first stack to the third stack.

3. Move block $A$ from the second stack to the third stack.

*– end example.*

Our application serves a single web page that presents a form used to specify an initial and a final configuration. When the form is submitted, a solution is computed and the display is updated to show it. A picture of the interface is shown in Figure 6.1. The project consists of Python, Curry, and HTML code. The main part of the application is defined with Flask, using an HTML template to define the graphical interface. The Blocks World solver is written in Curry. Additional Python code uses the Sprite interface to connect this to the main application. The files that make up this project are described below:

- `main.py`: The web application definition (code listed in Figure 6.2).
- `form.html`: The graphical user interface definition (code not listed).
- `solver.curry`: The Blocks World solver (code listed in Figure 6.3).
- `logic.py`: The application logic (code listed in Figure 6.4).

Our main interest is in `logic.py`, since that represents the unique contribution of our work. We will briefly comment on the other files before looking at that in detail.

The Flask Python code is shown in Figure 6.2. This creates an application to serve HTML code and routes user actions to the appropriate handler functions. The first few lines import code from the Flask library (line 1) and from the logic portion of

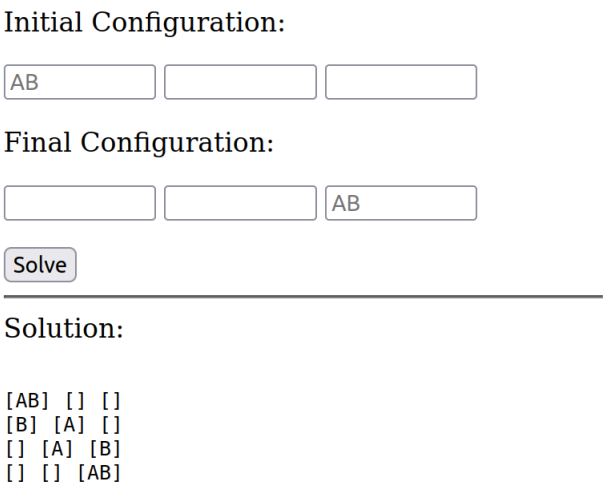



FIGURE 6.1: The graphical interface of the Blocks World web application.

our application (line 2) and then creates a Flask object representing the application (line 4). Statements beginning with the "at" (`@`) symbol use Python decorators to connect the function defined below them to the application. This is used to define two handlers for HTTP GET and POST methods.

Our application serves a single form produced by rendering a template found in the file `form.html`. The rendering process implements certain safeguards, such as protecting against HTML injection attacks, that could be important for real applications. Such security considerations are a reason developers may prefer a widely-deployed web framework such as Flask over a less widely-used solution in Curry. This form contains an optional section for displaying a solution, which is rendered only when the parameter `solution` is passed to `render_template` in line 13. In response to an

```
1  from flask import Flask, render_template, request
2  from . import logic
3
4  app = Flask(__name__)
5
6  @app.route('/', methods=['GET'])
7  def display():
8    return render_template('form.html')
9
10 @app.route('/', methods=['POST'])
11 def respond():
12   solution = logic.get_solution(request.form)
13   return render_template('form.html', solution=solution)
14
15 if __name__ == '__main__':
16   app.run()
```

FIGURE 6.2: Contents of `main.py`. A Flask application is created (line 4) and run (line 16). This application routes HTTP GET requests to the function `display` (line 7) and HTTP POST requests to the function `respond` (line 11).

HTTP GET request, the form is displayed with no solution (line 8). In response to an HTTP POST request, which occurs when the form is submitted, it is displayed with the computed solution (lines 12-13), as shown in Figure 6.1.

The HTML code found in `form.html` is straightforward but lengthy and is not listed. It defines three input fields (`i1`, `i2`, `i3`) for the initial configuration and three input fields (`f1`, `f2`, `f3`) for the final configuration.

The Blocks World solver is shown in Figure 6.3. A block is represented simply as `A`, `B`, `C`, `D`, or `E`, and a configuration (type `World`) is a triple of lists of blocks. A solution is a list of configurations (type `Trace`). The problem instance described in Exam. 6.2.1 can be solved by evaluating the following expression:

```
1  data Block = A | B | C | D | E deriving Eq
2  type World = ([Block], [Block], [Block])
3  type Trace = [World]
4
5  solve :: World -> World -> Trace
6  solve initial final = extend [initial]
7    where
8      move :: World -> World
9      move (x:xs, ys, zs) = (xs, x:ys, zs) ? (xs, ys, x:zs)
10     move (xs, y:ys, zs) = (y:xs, ys, zs) ? (xs, ys, y:zs)
11     move (xs, ys, z:zs) = (z:xs, ys, zs) ? (xs, z:ys, zs)
12
13     extend :: Trace -> Trace
14     extend (t:ts)
15       | t == final = reverse (t:ts)
16       | elem t ts  = failed
17       | otherwise  = extend (move t : t : ts)
```

FIGURE 6.3: Contents of `solver.curry`, a solver for the Blocks World problem. A configuration is a triple of lists of blocks (type `World`) and a solution is a trace (type `Trace`). Function `solve` reduces non-deterministically to all solutions for a given problem instance. Function `move` generates all legal moves available from a given configuration. Function `extend` extends a trace by one step. The phrase `deriving Eq` (line 1) makes the `Block` type comparable by operator `==`. From the Curry standard prelude, function `reverse` (line 15) reverses a list, `elem` (line 16) indicates whether a particular value appears in a list, and `failed` (line 16) represents a failed computation.

```
solve ([A,B],[],[]) ([],[],[A,B])
```

This produces several solutions, among which we find the following one that corresponds to the solution described in the example (and also shown in Figure 6.1):

```
[([A,B],[],[]),([B],[A],[]),([],[A],[B]),([],[],[A,B])]
```

The logic portion of our application connects the other components together. Its source code is listed in Figure 6.4. This module accesses the Blocks World solver

by importing its Curry implementation. The import relative to `curry.lib` (line 2) instructs Sprite to load the Curry file `solver.curry` as a Python module.

When it is time to compute a solution and update the graphical display the following actions are performed: (1) a problem description is extracted from the

```
1  import curry
2  from curry.lib import solver
3
4  FORM = ('i1','i2','i3'),('f1','f2','f3')
5
6  def get_solution(data):
7    configs = (
8        tuple(build_stack(data[fld]) for fld in fields)
9            for fields in FORM
10     )
11   goal = curry.expr(solver.solve, *configs)
12   trace = next(curry.eval(goal))
13   return show_trace(trace)
14
15 def build_stack(stk):
16   # E.g., "AB" -> [A, B] where A, B are Curry exprs.
17   return [curry.expr(getattr(solver, c)) for c in stk]
18
19 def show_trace(trace):
20   return '\n'.join(show_config(cfg) for cfg in trace)
21
22 def show_config(cfg):
23   return ' '.join(show_stack(stk) for stk in cfg)
24
25 def show_stack(stk):
26   return '[%s]' % ''.join(str(blk) for blk in stk)
```

FIGURE 6.4: Contents of `logic.py`. The built-in function `getattr` accesses object attributes, so `getattr(solver, c)` is equivalent to `solver.A` when `c` is bound to the string `"A"`. The embedded use of the `for` keyword (lines 8, 9, 17, 20, 23, and 26) produces sequence comprehensions. The use of square brackets (`[` and `]`) in line 16 constructs a list (using a list comprehension).

form data, (2) the problem description is converted into a Curry goal, (3) the goal is evaluated, (4) the solution is transformed into a string suitable for display in the updated form.

These actions are carried out by the function `get_solution` (lines 6-13). This function accepts a single argument (`data`) containing the form data and returns a text description of the solution to the given Blocks World problem. The field names listed in the global variable `FORM` are used to extract descriptions of the initial and final configurations from the form data (lines 7-10). For example, when `fld` is bound to `'i1'`, the expression `data[fld]` returns the string `"AB"` for the problem instance we have been using.

A configuration is a triple of lists of blocks and each list is constructed by a call to function `build_stack`, which converts a string into a (Python) list of Curry `Block` expressions. For example, the expression `build_stack("AB")` returns the Python list `[A, B]`, where `A` and `B` are Curry expressions constructed by `curry.expr` in line 17.

The Curry goal is constructed (line 11) by applying function `solve` (defined in Figure 6.3) to the tuples stored in `configs`, which holds descriptions of the initial and final configurations. `curry.expr` converts the Python tuples and lists into Curry tuples and lists and applies the function, producing a Curry expression. The call to `curry.eval` (line 12) invokes Sprite to evaluate the goal. The use of Python's `next` function to retrieve a single result means that only one value is computed. Variations are possible. A modified version of this application could, for example, report all

solutions or choose a shortest one. In the latter case, the selection of a shortest solution could be done in Curry through the use of set functions or in Python by iterating over all of the results of a complete computation.

To display the results, we convert each configuration in the trace into a string using `show_trace` (line 19) and the related functions below it. Each stack is shown as a sequence of characters enclosed in square brackets (as seen in Figure 6.1). The expression `str(blk)` converts a Curry object of type `Block` into the Python string matching its label (line 26). For example, if `blk` is bound to the Curry expression `A`, then `str(blk)` produces the Python string `"A"`.

The Blocks World application showcases the benefits of integrating Curry into an imperative programming environment using Sprite. Our solver, written in Curry, is both efficient and concise. A significant factor contributing to its simplicity is that the Curry code does not require input or output operations, as those are handled in Python. This feature may significantly increase the appeal of this approach for users coming from an imperative programming background who might not be familiar with Curry or Haskell.

Our web application's code is brief, comprising only a few lines of Python code. The "glue" code found in `logic.py` demonstrates the simplicity of crossing the boundary between Curry and Python. A short comprehension expression using the one-line function `build_stack` handles the encoding of a Blocks World configuration provided by the user. Additionally, a few one-line functions convert the solution from Curry

into text for display.

In the next chapter, we will explore the runtime performance and other attributes of programs Sprite generates by comparing these with program generated by two other Curry systems for a variety of benchmark programs.

# Chapter 7

# Benchmarks

In previous chapters, we outlined our compilation strategy for Curry programs, the Fair Scheme (FS), and introduced our Curry system, Sprite. In this chapter, our focus is on benchmarking Sprite to compare its performance against other popular implementations. This analysis not only allows us to determine Sprite's relative standing but also identify potential optimizations for both Sprite and related systems.

We will be comparing Sprite with two other Curry systems: The Portland Aachen Kiel Curry System (PAKCS) Version 3.4.1-b6 using SWI Prolog Version 7.6, and The Kiel Curry System (KICS) Version 3.0.0-b5 using GHC Version 8.6.5. Previous comparisons have been made between these systems, and we base our benchmarking suite on the programs that were originally used [47], with the addition of a few new ones. We omit programs that rely on non-standard Curry modules not implemented in Sprite, and resize inputs in certain cases. All measurements were carried out on an Intel(R) Core(TM) i7-1075H CPU operating at 2.60 GHz, and all reported times are in seconds. Programs were generated and compiled with optimizations enabled as

determined by the respective Curry system used, and the C++ emitted by Sprite was compiled using the GNU C++ compiler Version 9.4.0 with the `-O3` optimization flag.

We will begin by demonstrating the completeness property of Sprite in Sect. 7.1, which sets it apart from other Curry systems. Next, we will divide the remaining benchmarks into two groups. In Sect. 7.2 we will discuss the performance of deterministic functional programs, and in Sect. 7.3, we will examine the broader category of non-deterministic functional logic programs. Finally, in Sect. 7.4, we will analyze the benchmark results and discuss possible areas for improvement.

## 7.1 Completeness

The completeness property shown in Figure 3.4.10 is a unique feature of Sprite, and as far as we know, no other Curry system can claim to have it. It is possible to create a program that can be computed by Sprite but not by PAKCS or KiCS2, such as the following:

```
1 loop = loop
2 main = loop ? 1 ? loop
```

This program contains an endless loop, causing the computation of `main` to never terminate. Despite this, Sprite instantly produces the value `1`, while PAKCS and KiCS2 do not produce this value at all. Additionally, Sprite produces all values where the others do not in the following case:

```
1 loop = loop
2 main = 1 ? loop ? 2
```

Both PAKCS and KiCS2 produce the value `1` but not `2`. This reveals the left-bias in those systems, which process choices from left-to-right.

The symbol `loop` cannot be reduced to a value in any number of steps. A different issue arises when a symbol produces an infinite number of choices, as is the case with the next program:

```
1 branch = branch ? branch
2 main = branch ? 1
```

Sprite produces the value `1`, which the other system do not find. Here, it is crucial for Sprite that new choice alternatives are placed at the end of the computation queue, as done by rule **D**.2 in Figure 3.1. If that rule instead put the alternatives at the head of the queue, the endless proliferation of new computation paths ahead of the fruitful one could prevent the value from being produced.

Despite the theoretical completeness of the FS, our implementation still has practical limitations, so it is not always possible to guarantee that every value is produced. For example, consider the following:

```
1 bgen n = bgen (2 * n + 1) ? bgen (2 * n) ? n
2 main | bgen 1 == 1000000 = True
```

This program generates natural numbers using an exponential process, and although it has a value, the memory on our system is exhausted long before Sprite computes it.

If the target value is small enough, however, such as 10, Sprite produces the intended value. Alternatively, if we consider time as a resource, we can define a program whose value is not produced by Sprite due to an impossibly long computation, such as attempting to compute Ackerman's function for anything but tiny inputs. Such practical limits are unavoidable and do not detract from our completeness claim.

## 7.2 Functional Benchmarks

The Curry systems that we examine each handle deterministic programs differently, and these differences significantly impact their efficiency. In PAKCS, function applications are evaluated using the principle of resolution, which is relatively inefficient. In fact, as we discussed in Sect. 1.2, a primary motivation for bringing functional features into logic programming was to improve the relatively poor efficiency of logic programming as compared to functional programming. By translating Curry into the logic programming language Prolog, we should expect PAKCS to sacrifice efficiency relative to other approaches.

Both KiCS2 and Sprite translate deterministic programs into graph reduction systems. Although their approaches are similar, the Haskell programs generated by KiCS2 are compiled by GHC, which is far more mature than Sprite and handles most aspects of program evaluation more efficiently. In addition to having a more efficient runtime system, including a sophisticated garbage collector [103] and other

| | PAKCS | KICS | SPRITE |
|---|---|---|---|
| Half | 10.48 | 0.29 | 0.81 |
| Hamming | 10.68 | 0.07 | 1.31 |
| Fib | 9.15 | 0.05 | 0.60 |
| Palindrome | 9.00 | 0.04 | 1.06 |
| Peano | 1.24 | 0.04 | 0.29 |
| Primes | 89.27 | 0.18 | 8.67 |
| Psort | 282.90 | 0.69 | 43.80 |
| Qsortlet | 22.25 | 0.03 | 2.23 |
| Queens(10) | 83.37 | 0.14 | 6.94 |
| Quicksort | 108.01 | 0.11 | 13.55 |
| Reverse | 5.29 | 0.13 | 1.62 |
| ReverseBuiltin | 10.42 | 0.15 | 1.20 |
| ReverseGroups | 20.97 | 0.02 | 1.53 |
| ReverseHO | 10.27 | 0.11 | 1.16 |
| ReverseUser | 5.34 | 0.13 | 1.62 |
| SearchMAC | 2.14 | 7.42 | 7.96 |
| Tak | 124.00 | 0.26 | 6.15 |
| TakPeano | 42.86 | 0.13 | 6.72 |

FIGURE 7.1: Execution times (s) of deterministic benchmarks programs.

such details, Haskell programs are subject to sophisticated optimizations that can significantly reduce execution time. Since deterministic Curry programs are essentially Haskell programs, we should not expect Sprite to match the performance of KiCS2 in such cases. We will discuss the implications of this in Sect. 7.4.

The results for our deterministic benchmarking programs are shown in Figure 7.1. Without exception, the execution times are longest for PAKCS and shortest for KiCS2, with Sprite falling in between. Most of these programs were taken from the KiCS2 benchmark suite and are discussed in detail in [47]. We include both first-order programs (such as `Tak` and `TakPeano`) and higher-order programs

(such as `Primes` and `Queens`). We also include standard benchmarks for computing Fibonacci numbers (`Fib`), reversing lists (`Reverse*`), and sorting lists (`Psort`, `Qsortlet`, and `Quicksort`). Certain programs compute with the Peano representation of natural numbers (`Half`, `Peano`, and `TakPeano`) while others work with native integers (`Hamming`, `Primes`, and `Tak`).

We will use `Queens` for comparison with non-deterministic approaches to the same problem in the next section, so we include its source code here:

```
queens nq = length (gen nq)

gen n = if n==0
    then [[]]
    else [(q:b) | b <- gen (n-1), q <- [1..nq], safe q 1 b]

safe _ _ [] = True
safe x d (q:l) = x/=q && x/=q+d && x/=q-d && safe x (d+1) l

main = queens 10
```

We include the following program (`Hamming`) for computing Hamming numbers:

```
ordMerge (x:xs) (y:ys) | x==y = x:ordMerge xs ys
                       | x<y  = x:ordMerge xs (y:ys)
                       | x>y  = y:ordMerge (x:xs) ys

hamming = 1:ordMerge (map (*2) hamming)
                     (ordMerge (map (*3) hamming)
                               (map (*5) hamming))

main = take 200 hamming
```

Additionally, we include a program (`ReverseGroups`) to reverse sections of a list:

```
kreverse k xs = let (h,t) = (take k xs, drop k xs) in
    if length h == k then reverse h ++ kreverse k t else h

main = [kreverse i [1..5000] | i <- [1..50]]
```

`SearchMAC` solves the classic "Missionaries and Cannibals" problem.

Our goal is not to stress test Sprite. We therefore do not characterize how it scales to large problem sizes. However, we should note that we did not observe any indications of poor performance that could suggest problems with high algorithmic complexity for longer or larger programs. A basic demonstration of this can be made by running a program for various input sizes. The `Tak` program is defined as follows:

```
tak :: Int -> Int -> Int -> Int
tak x y z = if x <= y then z
            else tak (tak (x-1) y z)
                     (tak (y-1) z x)
                     (tak (z-1) x y)
```

Figure 7.2 reports the execution time for different input sizes. The results show a relatively constant ratio between the time needed by Sprite and the time required by either PAKCS or KiCS2. For example, the ratio between Sprite and PAKCS remains approximately constant at a factor of $20.0 \pm 0.5$, suggesting these implementations have similar computational complexity for this range of input sizes.

| | PAKCS | KICS | SPRITE |
|---|---|---|---|
| `tak 24 16 8` | 34.48 | 0.08 | 1.68 |
| `tak 27 16 8` | 124.00 | 0.26 | 6.15 |
| `tak 33 17 8` | 3405.46 | 6.80 | 174.87 |

FIGURE 7.2: Execution times (s) of `Tak` for a variety of input sizes.

### 7.3 Functional Logic Benchmarks

Curry is a non-deterministic programming language, and so benchmark results are particularly relevant when studying non-deterministic programs. These results are shown in Figure 7.3. PAKCS again tends to produce the longest execution times, but for these programs, KiCS2 and Sprite are more evenly matched. In 58% (7/12) of the cases, Sprite is faster, while KiCS2 is faster in 33% (4/12). In one case, the results are a tie. The difference in results between KiCS2 and Sprite is generally within a factor of about 2, except in three cases: for `PokerChoice`, Sprite is over 47 times faster;

| | | PAKCS | KICS | SPRITE |
|---|---|---|---|---|
| | ColormapChoice | 1.12 | 1.02 | 0.51 |
| | ColormapFree | 1.09 | 0.59 | 0.46 |
| | Horseman | 11.21 | 1.09 | 0.75 |
| | Last | 6.54 | 0.78 | 0.83 |
| | PaliFunPats | 18.94 | 0.17 | 0.17 |
| | PermSort | 28.38 | 8.07 | 4.99 |
| | PermSortPeano | 54.67 | 8.39 | 10.55 |
| $\mathcal{S}$ | PokerChoice | 1377.95 | 0.95 | 0.02 |
| $\mathcal{S}$ | PokerFree | sus | 0.37 | 0.02 |
| $\mathcal{S}$ | QueensSet | 30.26 | 11.93 | 0.67 |
| | RegExp | 35.81 | 1.72 | 3.04 |
| | SearchQueens | 35.48 | 3.71 | 7.57 |

FIGURE 7.3: Execution times (s) of non-deterministic benchmark programs. The result *sus* indicates execution suspended without completing. Programs that utilize set functions are indicated by the symbol $\mathcal{S}$.

for `PokerFree`, Sprite is over 18 times faster; and for `QueensSet`, Sprite is nearly 18 times faster. The largest advantage shown by KiCS2 is for `SearchQueens`, where it is just over 2 times faster than Sprite.

In two cases, PAKCS exhibits aberrant behavior, suspending the computation in `PokerFree` before it is complete and running four to five orders of magnitude slower than KiCS2 or Sprite, respectively, in the case of `PokerChoice`. We will not discuss these results in detail.

As in the previous section, many of these tests are taken from the KiCS2 benchmark suite, and were discussed in [47]. To these we have added several additional tests, which we discuss below. The `Colormap*` tests perform map coloring. Their implementation is as suggested by the following (where some repetitive code is omitted):

```
1  data Color = Red | Green | Yellow | Blue
2  aColor = Red ? Green ? Yellow ? Blue
3
4  correct l1 l2 l3 l4 l5 l6 l7 l8 l9 l10 l11 l12
5     | diff l1 l2 & diff l1 l3 & ...
6     = [l1,l2,l3,l4,l5,l6,l7,l8,l9,l10,l11,l12]
7     where diff x y = (x == y) =:= False
8
9  main = correct aColor aColor aColor aColor ...
```

The difference between `ColormapChoice` and `ColormapFree` lies in the definition of `aColor`. The code listing above shows the definition for `ColormapChoice`. For `ColormapFree`, the definition is changed as follows:

```
aColor = x where x free
```

`Horseman` solves the classic "Horses and Men" puzzle as follows:

```
1  data Nat = O | S Nat
2  add O     n = n
3  add (S m) n = S (add m n)
4
5  horseman m h heads feet =
6      heads =:= add m h &
7      feet  =:= add (add m m) (add (add h h) (add h h))
8
9  goal m h = horseman m h (int2nat 460) (int2nat 1160)
10 main = goal m h &> (m, h) where m,h free
```

This answers the question "How many horses and men together have 460 heads and 1160 feet." The function `int2nat` (not shown) builds the natural number corresponding to a given integer. This implementation uses unification to assign suitable values to the free variables, `m` and `h`, representing the number of men and horses, respectively.

`Last` computes the last element of a long list by unification. `PaliFunPats` defines a palindromic constraint using functional patterns. `PermSort` is a permutation sort in which permutations are defined non-deterministically. We should note that this program is larger (n=14) than the deterministic permutation sort represented by `Psort` (n=10) in Figure 7.1. `PermSortPeano` is identical to `PermSort` except that it works with natural numbers rather than native integers. `RegExp` implements pattern-matching with regular expressions.

The `Poker*` programs are adapted from the Curry tutorial and use functional patterns and default rules to determine whether a poker hand scores a four-of-a-kind. The source code of `PokerChoice` is shown below:

```
1  data Card = Card Rank Suit
2  data Rank = Ace | King | Queen | Jack | Ten | Nine | Eight
3            | Seven | Six | Five | Four | Three | Two
4  data Suit = Club | Spade | Heart | Diamond
5
6  rank (Card r _) = r
7  anyRank = Ace ? King ? Queen ? Jack ? Ten ? Nine ? Eight
8                ? Seven ? Six ? Five ? Four ? Three ? Two
9
10 isFour (x++[_]++z)
11     | map rank (x++z) == [r,r,r,r] = Just r where r free
12 isFour'default _ = Nothing
13
14 hands = [(Card anyRank Club),(Card anyRank Spade),
15    (Card anyRank Heart),(Card anyRank Diamond),
16    (Card anyRank Diamond)]
17
18 goal = isFour hands
19 main = minValueBy (\_ _->EQ) (set0 goal)
```

The two versions of these programs differ in how `anyRank` is defined in a similar way to how the `Colormap*` programs differ. In `PokerFree`, card ranks are generated as follows:

```
anyRank = x where x free
```

The inclusion of a set function applied to the goal and its reduction by `minValueBy`, which is defined in the set functions module, serves to avoid printing a large number

of results. We do this because we are interested in comparing computation times rather than formatting or printing times. Including this reduces the KiCS2 execution time appreciably. Similar techniques are used in certain other benchmark programs to reduce the output size.

`QueensSet` and `SearchQueens` provide two additional solutions to the $N$-Queens puzzle. Both use a generate-and-test approach where all permutations are generated on an 8-by-8 chess board. `QueensSet` is defined as follows:

```
1  perm [] = []
2  perm (x:xs) = ndinsert (perm xs)
3    where
4      ndinsert ys     = x : ys
5      ndinsert (y:ys) = y : ndinsert ys
6
7  queens n | isEmpty ((set1 unsafe) p) = p
8    where
9      p = perm [1..n]
10     unsafe (_++[x]++y++[z]++_) = abs (x-z) =:= length y + 1
11
12 main = queens 8
```

`SearchQueens` is similar but does not use set functions or functional patterns to filter out unsafe placements. Its source code is shown below:

```
1  permute :: Prelude.Data a => [a] -> [a]
2  permute [] = []
3  permute (x:xs) | u++v =:= permute xs = u++(x:v)
4      where u,v free
5
6  allSafe :: [Int] -> Bool
7  allSafe qs = allSafe' $ zip qs [1..] where
8    allSafe' :: [(Int,Int)] -> Bool
```

```
9     allSafe' [] = True
10    allSafe' (xy:xys) = all (safe xy) xys && allSafe' xys
11
12  safe :: (Int,Int) -> (Int,Int) -> Bool
13  safe (a,b) (c,d) = abs (a-c) /= abs (b-d)
14
15  abs :: Int -> Int
16  abs x | x < 0     = -x
17        | otherwise = x
18
19  queens :: Int -> [Int]
20  queens n | allSafe qs = qs where qs = permute [1..n]
21
22  main = queens 8
```

The significant differences in the time required to compute the various solutions to the $N$-Queens problem are worth noting. In the deterministic version (`Queens`), KiCS2 runs almost 50 times faster than Sprite. In one non-deterministic version (`SearchQueens`), KiCS2 is approximately 2 times faster. However, in the alternative non-deterministic version that uses set functions (`QueensSet`), Sprite is faster by a factor of almost 18. Generally speaking, we observed that programs involving set functions show the most significant efficiency advantage for Sprite, and by a wide margin.

## 7.4 Discussion

The benchmarking results presented in this study support several interesting conclusions and suggest future work. First, our findings indicate that the performance of Sprite typically falls between that of PAKCS and KiCS2, with Sprite being closer

in speed to the faster of the two, KiCS2. Since one of the main goals of functional logic programming is to improve upon the performance of logic programming, it is noteworthy that Sprite achieves this relative to PAKCS, particularly for deterministic programs. The significant differences between KiCS2 and Sprite can be partly attributed to the greater maturity of GHC. Therefore, improving the implementation quality of Sprite directly could improve its performance. This could involve optimizing parts of the Sprite runtime system or adding passes to optimize Curry programs compiled by Sprite. A factor that affects Sprite's execution times is its relatively simple memory system. Sprite's garbage collector is synchronous, non-parallel, and non-generational, and has not been characterized or optimized in any significant way. Future work to improve the garbage collector could reduce Sprite's execution times, bringing them closer to those of KiCS2.

It is likely that the differences in execution time for numeric benchmarks like `Tak` are largely due to unboxing optimizations performed by GHC. Numeric values are boxed by placing them inside another data structure, allowing the runtime system to treat them like any other object. Unboxing enables computations to use native representations of numbers, reducing the time required for creating and destroying nodes and reducing the number of memory read and write operations.

Although Sprite does not perform unboxing, an implementation based on the Fair Scheme (FS) could do so. One way to accomplish this would be to extend ICurry with a representation of unboxed data and then perform unboxing for deterministic,

numeric computations. To preserve the completeness property, it would be necessary to interrupt unboxed computations periodically to allow for context switching. Without boxes, there are no steps, so an implementation might utilize additional threads of execution and use signal interrupts for preemption. GHC performs additional optimizations, such as deforestation, that have been well-studied and documented. Libby has investigated several of these optimizations in the context of Curry and has demonstrated their potential to improve efficiency [68]. These optimizations can be applied to FlatCurry or ICurry, and could be integrated into Sprite with minimal disruption.

The benchmarking data clearly demonstrate that KiCS2 has a significant advantage over Sprite for deterministic programs, as shown in Figure 7.1. However, these programs do not make use of the most salient features of Curry, namely the non-deterministic ones. When non-determinism is involved, KiCS2 no longer exhibits a clear advantage over Sprite. One reason for this may be that non-determinism interferes with certain optimizations, including unboxing. To support non-determinism, KiCS2 extends certain types with additional constructors. For instance, the integer type is extended with constructors to represent failure and choices, among other possibilities. While GHC can replace a built-in `Int` with an unboxed integer, it cannot do so for an extended integer. KiCS2 performs analysis to avoid extending types unnecessarily in deterministic contexts, but it is unavoidable for non-deterministic

computations. Therefore, the presence of non-determinism hinders GHC from applying certain optimizations, which could account for the less pronounced differences in execution times shown in Figure 7.3.

While KiCS2 automatically separates the non-deterministic portions of each program to improve efficiency, Sprite provides alternative methods to achieve similar results. In particular, Sprite offers a hybrid programming environment that simplifies the process of defining deterministic functions outside of Curry. For instance, the *Cython* package [95] provides extensions to Python for compiling Python-like code to low-level C code. Using this, the `tak` function can be defined as follows:

```
1 cpdef long long tak(long long x, long long y, long long z):
2   if x <= y:
3     return z
4   else:
5     return tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y))
```

By adopting this approach, we can perform an efficient computation of `tak` outside of Curry while still utilizing non-deterministic features in other parts of an application with Curry. Figure 7.4 shows the execution times for this implementation, suggesting that this approach can be very efficient.

| | PAKCS | KICS | SPRITE | CYTHON |
|---|---|---|---|---|
| `tak 24 16 8` | 34.48 | 0.08 | 1.68 | 0.01 |
| `tak 27 16 8` | 124.00 | 0.26 | 6.15 | 0.01 |
| `tak 33 17 8` | 3405.46 | 6.80 | 174.87 | 0.30 |

FIGURE 7.4: Execution times (s) of `Tak`, including Cython.

The benchmarking data suggest Sprite is particularly efficient when evaluating set functions. In Figure 7.3, the tests utilizing set functions (indicated by $\mathcal{S}$) are considerably faster than PAKCS and KiCS2. This suggests our novel approach to set functions holds promise. Further work is needed to characterize our approach relative to others, determine why such a large difference exists, and see if this approach is suitable for use in other Curry systems, such as KiCS2. One possible explanation for this difference is the way our approach shares subexpressions between disjoint sets. In the `QueensSet` implementation of the $N$-Queens problem, any pair of unsafe queens eliminates a potential solution, even though the remaining queens may be placed in any permutation. With our implementation, computing a single unsafe pairing causes many sets to become non-empty, potentially without performing many additional steps. Since these computations are shared, many steps can be avoided in this way. This theory is supported by the data shown in Figure 7.5, where `QueensSet` involves fewer steps than other implementations.

Although choices and free variables are equivalent terms of their expressive power,

| | KiCS (s) | Sprite (s) | # of steps |
|---|---|---|---|
| Queens | 0.00 | 0.25 | 2167580 |
| QueensSet | 11.93 | 0.67 | 1691204 |
| SearchQueens | 3.71 | 7.57 | 32795975 |

Figure 7.5: Execution times (s) and number of steps taken by Sprite for solutions to the 8-Queens problem.

the results in Figure 7.3 suggest they differ in their efficiency. Comparing the program `ColormapChoice` with `ColormapFree` and `PokerChoice` with `PokerFree`, we observe that free variables may perform slightly better than choices in certain cases, and that this is true for both Sprite and KiCS2. For Sprite, we speculate that this may due to the higher cost of constructing large "choice trees" as compared to constructing generator expressions, which use a built-in, hand-optimized routine. This suggests several potential improvements. For instance, we could optimize programs by replacing certain choice expressions with free variables at compile time. Even in cases where most, but not all alternatives are covered, it may still be beneficial to use use a free variable while filtering out unwanted alternatives after the fact. Another possible improvement would be to optimize the construction of large choice trees, such as the ones found in `aColor` (`ColormapChoice`) and `anyRank` (`PokerChoice`), to more closely resemble the construction of generator expressions.

Overall, our benchmark results indicate that Sprite is a viable alternative to existing Curry systems. Although its efficiency does not compete with KiCS2 for deterministic programs, Sprite is competitive for non-deterministic programs and overall superior to PAKCS. In particular, our novel approach to set functions shows promise. Sprite's lower efficiency for deterministic programs is mitigated by the fact that it provides ways to integrate deterministic code from other programming environments with Curry.

# Chapter 8

# Conclusion

In this dissertation, we have described functional logic programming and the Curry programming language. We introduced an original evaluation strategy for Curry programs, called the Fair Scheme (FS). We provided definitions of soundness, completeness, and optimality and proved that the FS satisfies these properties. We extended the FS to support narrowing computations, constraint programming with equational constraints, and set functions, which enables the implementation of a full-featured Curry system.

We presented our implementation of the Sprite Curry system, which includes optimizations of the FS to improve efficiency. We demonstrated how embedding Curry in Python simplifies the creation of hybrid programs and provided an example application to illustrate this. Additionally, we evaluated the runtime performance of our implementation. Our results show that Sprite is more complete than its competitors, as it can compute correct solutions for certain programs that other implementations cannot. Furthermore, we found that Sprite is competitive with other Curry systems

in terms of speed, particularly for non-deterministic programs. Based on our results, our novel approach to set functions appears to be particularly effective.

Although Sprite is less mature than it competitors and its libraries are less fully developed, it meets our goal of promoting the practical use of Curry and can serve as a suitable starting point for further research into functional logic programming.

Before concluding, we shall discuss possible directions for future work building on this research. We are particularly interested in exploring practical applications of functional logic programming, and one area that appears poised for explosive growth is that of artificial intelligence. For instance, in November 2022, the American research laboratory OpenAI released a web interface to their Generative Pre-trained Transformer model GPT-3. This interface, called ChatGPT, reportedly reached one million users in just five days, making it the fastest web service to reach that number of users so quickly [104]. We see a bounty of opportunities for functional logic programming in this area.

A functional logic language like Curry is an ideal tool for working with natural language models such as GPT-3. These models predict the next word or symbol in a composition, generating multiple possibilities for each step. For example, given the starting phrase “I am coming” the model might predict the next word as “home“, “from”, “to”, “now“, “later”, “after” or other possibilities. Imperative programs might use loops and data containers to manage these possibilities, but this approach could become unwieldy, as the number of possibilities grows exponentially. The principle

of superposition offers a superior way of managing this. Potential completions can be combined and computed without explicitly representing aggregates such as "sets of words" in application code.

To effectively use Curry in this way, we need to enhance its search strategy, which is the set of rules and procedures that organize and execute exploration of the solution space defined by a logic program. A simple calculation shows that even if only a few alternatives are considered for each next word, a text composition of any significant length involves more possibilities than can be practically computed. This means that an application using a language model like GPT-3 cannot consider all compositions. A suitable objective is to compose one or a few compositions that meet certain criteria. This would likely require much more sophisticated control over the search strategy than Sprite offers.

Fortunately, our representation of computations as a queue of expressions is easily extendible in this direction. Changes to the search strategy correspond to changes in the way the computation queue of the Fair Scheme is managed. The Fair Scheme, as presented in Figure 3.1, implements a breadth-first search since each time a step is performed, the expression under consideration is moved to the end of the queue. Similarly, when a choice-rooted expression is split, the alternatives are placed at the end of the queue. These details simplify proving the completeness property but do not necessarily lead to the most practical solution.

In our implementation of Sprite, we relaxed the breadth-first rule by looping

applications of target procedures to improve efficiency. This can be viewed as a rule to place expressions back at the front of the queue after a step is performed. We used a step counter to force a rotation of the queue every so often. Without that, Sprite would implement a depth-first search. With the step counter, Sprite implements something in between breadth-first and depth-first search.

There are various other ways to manipulate the queue for different applications. In some cases, we might have information about the quality of potential values. For example, a language model may assign a weight to each possible next word, indicating its likelihood or quality. The overall quality of a string of words can then be estimated in part by taking the product of the weights. This suggests a search strategy that prioritizes better prospects, as measured by their cumulative weight, by placing them closer to the front of the queue.

In other applications, different metrics can be used to improve search performance. Experience shows that search performance can often be improved by introducing a heuristic, such as the distance heuristic in the A* algorithm. In that case, as long as a distance-like measure conforms to certain conditions, programmers have considerable flexibility in defining it. It is also relatively straightforward to incorporate user-defined heuristics in the FS. For example, a puzzle-solving program could include a heuristic that estimates how quickly a solution can be reached from a certain configuration. Sprite could then use this to prioritize queue elements by placing the most promising configurations near the front of the queue, in the hope of finding a solution sooner.

The user-defined portion of this could be implemented in Curry, Python, or another language, such as C++. A metric of this type that has been shown to improve search for the Blocks World problem from Sect. 6.2 estimates the distance of a configuration from the goal as the number of inserting (rather than stacking) moves required to reach the goal state.

If multiple heuristics are available, a potentially interesting strategy would be to choose the heuristic non-deterministically. In Curry, this might be challenging, as the search strategy is a hyperparameter of the evaluation. However, it may be easier to achieve in Sprite, where Curry computations can be nested more readily, as we did when implementing set functions. If heuristics only affect the order of queue elements, it should be possible to share computational steps between different evaluations. Initially, it may seem that this approach would increase computation time to the first solution by a factor of $n$, where $n$ is the number of heuristics. However, if any heuristic dramatically improves evaluation time, this approach could prove beneficial when it is not known *a priori* which heuristic is most effective.

Queue manipulations can be used to manage resource consumption in addition to improving search performance. If queue elements are ordered based on quality, then a program's memory usage can be bounded by dropping elements from the end of the queue as needed. Although this would result in an incomplete search, it may still be useful to conduct for certain applications. In some cases, the objective might not be to find a globally optimal solution but to expend a certain amount of effort,

such as compute time and memory, to improve an existing solution. For example, when considering a manufacturing or other business process, quickly finding an easy-to-implement change that affords a competitive advantage may be more important than ensuring the improvement reaches a global optimum.

Bounding the resources required in search may be more important than maximizing solution quality when systems need to adapt to new information in real-time. Consider the hypothetical case of a robot providing domestic help in someone's home, tasked with preparing a healthy, delicious meal on time using only the ingredients available. In such a scenario, an adequate solution found in time is better than an ideal solution that cannot be found before dinner is over. These factors may come into play whenever resources are limited and conditions cannot be fully anticipated.

As we consider adaptive software systems that can make incremental improvements while budgeting computational resources, it is hard not to be struck by the similarities to natural reasoning. To give a relatable example, let us continue the task of preparing a recipe for dinner. The order in which ingredients are measured can reduce the number of measuring spoons needed or the number of times a single spoon must be cleaned. For instance, if a dinner recipe requires a tablespoon each of sugar, ghee (butter), and milk, after measuring the ghee, the spoon becomes dirty and cannot be used to scoop sugar but can be used to pour milk (which similarly dirties the spoon). The problem of optimizing this seems simple enough for a clever software system to pose to itself and search for an improvement over the default,

such as choosing the order randomly. This search might be conducted within a time budget allotted proportionally to how much time could be saved by a potential optimization, limited by how much time can be spent without risking dinner being late. To this author, this process bears a striking similarity to what we all go through when learning and planning in everyday life.

These considerations give us cause for optimism regarding the future of functional logic languages such as Curry. As we progress towards a world in which software systems are more autonomous and adaptable, we should expect greater acceptance of logic programming principles. Curry appears to be in a favorable position in this respect. Additionally, as software systems become more integrated and interconnected, we envision a promising future for hybrid techniques that enable programmers to draw on features of one programming paradigm from others. We hope that the Sprite example will inspire further research in this direction.

## References


[1] Marc Levinson. *The Box: How the Shipping Container Made the World Smaller and the World Economy Bigger*. Princeton University Press, 2008.

[2] Michael Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, 19:583–628, 1994.

[3] Sergio Antoy and Michael Hanus. Functional Logic Programming. *Commun. ACM*, 53(4):74–85, Apr. 2010.

[4] Michael Hanus. Verifying Fail-Free Declarative Programs. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA, 2018. ACM.

[5] Sergio Antoy, Michael Hanus, and Steven Libby. Proving Non-Deterministic Computations in Agda. *Proceedings of WLP'15/'16/WFLP'16*, pages 180–195, 2016.

[6] Sergio Antoy and Michael Hanus. Contracts and Specifications for Functional Logic Programming. In *Proc. of the 14th International Symposium on Practical*

*Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.

[7] More Details About the October 4 Outage. Available at `https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/`, 2021.

[8] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS*, volume 95, pages 95–107, 1995.

[9] Michael Hanus (ed.). The Curry Homepage. Available at `https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/`, 2023.

[10] Python. Available at `https://www.python.org/`.

[11] PyPI. Available at `https://pypi.org/`, .

[12] TensorFlow. Available at `https://www.tensorflow.org/`.

[13] PyTorch. Available at `https://pypi.org/project/pytorch/`, .

[14] Keras. Available at `https://pypi.org/project/Keras/`.

[15] NumPy. Available at `https://pypi.org/project/numpy/`.

[16] SciPy. Available at `https://pypi.org/project/scipy/`.

[17] Theano. Available at `https://pypi.org/project/Theano/`.

[18] Pandas. Available at `https://pypi.org/project/pandas/`.

[19] MatPlotLib. Available at `https://pypi.org/project/matplotlib/`.

[20] OpenCV. Available at `https://pypi.org/project/opencv-python/`.

[21] GraphTool. Available at `https://pypi.org/project/graph-tool/`.

[22] Requests. Available at `https://pypi.org/project/requests/`.

[23] Django. Available at `https://pypi.org/project/django/`.

[24] Jinja. Available at `https://pypi.org/project/jinja/`.

[25] Flask. Available at `https://pypi.org/project/flask/`.

[26] Beautiful Soup. Available at `https://pypi.org/project/beautifulsoup4/`.

[27] Lark. Available at `https://pypi.org/project/lark/`.

[28] Tesseract. Available at `https://github.com/tesseract-ocr/tesseract`.

[29] PyTesseract. Available at `https://pypi.org/project/pytesseract/`, .

[30] John Tromp. Binary Lambda Calculus and Combinatory Logic. In *Randomness and Complexity, from Leibniz to Chaitin*, pages 237–260. World Scientific, 2007.

[31] Peyton Jones, Simon L, and Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, July 1992.

[32] John McCarthy. History of LISP. In *History of Programming Languages*, pages 173–185. Academic Press, 1978.

[33] Preface to the Haskell 98 Report. Available at `https://www.haskell.org/onlinereport/preface-jfp.html`, 1998.

[34] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[35] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[36] Simon Marlow and Simon L. Peyton Jones. The Glasgow Haskell Compiler. 2012.

[37] Robert Kowalski. *History of Logic Programming*, pages 523–569. Jan. 2014.

[38] Cordell Green. Application of Theorem Proving to Problem Solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier, 1981.

[39] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. 7(3), 1960.

[40] Uday S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proceedings of the 1985 Symposium on Logic Programming, Boston, Massachusetts, USA, July 15-18, 1985*, pages 138–151. IEEE-CS, 1985.

[41] Sergio Antoy and Michael 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.

[42] Sergio Antoy and Michael Hanus. Declarative Programming with Function Patterns. In *15th Int'nl Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR 2005)*, pages 6–22, London, UK, Sept. 2005. Springer LNCS 3901.

[43] Sergio Antoy. Optimal Non-Deterministic Functional Logic Computations. In *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298, pages 16–30, Southampton, UK, Sept. 1997. Springer LNCS.

[44] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005. Reduction Strategies in Rewriting and Programming special issue.

[45] Sergio Antoy. Evaluation Strategies for Functional Logic Programming. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.

[46] Juan Carlos González-Moreno, Maria Teresa Hortala-Gonzalez, Francisco Javier Lopez-Fraguas, and Mario Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *The Journal of Logic Programming*, 40(1):47–87, 1999.

[47] Bernd Braßel. Implementing Functional Logic Programs by Translation into Purely Functional Programs. 2010.

[48] George W Mackey. Quantum Mechanics and Hilbert Space. *The American Mathematical Monthly*, 64(8P2):45–57, 1957.

[49] Robert B Griffiths. Hilbert Space Quantum Mechanics.

[50] George W Mackey. *Mathematical Foundations of Quantum Mechanics*. Courier Corporation, 2013.

[51] Abdulla Alqaddoumi, Sergio Antoy, Sebastian Fischer, and Fabian Reck. The Pull-Tab Transformation. In *Proc. of the Third International Workshop on Graph Computation Models*, pages 127–132, 2010.

[52] Sergio Antoy. On the Correctness of Pull-Tabbing. *TPLP*, 11(4-5):713–730, 2011.

[53] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[54] Sergio Antoy, Rachid Echahed, and Michael Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[55] Sergio Antoy and Michael Hanus. Curry Without Success. In *WLP/WFLP*, pages 140–154, 2014.

[56] Joxan Jaffar and J-L Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[57] Sergio Antoy and Michael Hanus. New Functional Logic Design Patterns. In *20th International Workshop on Functional and (constraint) Logic Programming (WFLP 2011)*, pages 19–34, Odense, Denmark, July 2011. Springer LNCS 6816.

[58] Sergio Antoy and Michael Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 73–82, Lisbon, Portugal, Sept. 2009.

[59] Sergio Antoy and Michael Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017.

[60] Bernd Braßel Michael Hanus Frank Huch. Encapsulating Non-Determinism in Functional Logic Computations. In *WFLP'04 13th International Workshop on Functional and (Constraint) Logic Programming*, page 74, 2004.

[61] Sergio Antoy, Michael Hanus, and Finn Teegen. Synthesizing Set Functions. In *International Workshop on Functional and Constraint Logic Programming*, pages 93–111. Springer, 2018.

[62] Sergio Antoy and Andy Jost. Compiling a Functional Logic Language: The Fair Scheme. In *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers*, pages 202–219. Springer, 2014.

[63] Saumya K Debray and Prateek Mishra. Denotational and Operational Semantics for Prolog. *The Journal of Logic Programming*, 5(1):61–91, 1988.

[64] Krzysztof R. Apt and M. H. van Emden. Contributions to the Theory of Logic Programming. *J. ACM*, 29(3):841–862, July 1982.

[65] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Skrlac. KiCS2: A New Compiler from Curry to Haskell. pages 1–18, July 2011.

[66] Michael Hanus, Sergio Antoy, Bosak El, Martin Engelke, Michael Thoennessen, Johannes Koj, Philipp Niederau, Ramin Sadre, and Frøydis Steine. PAKCS: The Portland Aachen Kiel Curry System. 2000.

[67] Sergio Antoy and A. Jost. A Target Implementation for High-Performance Functional Programs. In *Presentation at the 14th International Symposium Trends in Functional Programming (TFP 2013)*, Provo, Utah, 2013.

[68] Steven Libby. RICE: An Optimizing Curry Compiler. In Michael Hanus and Daniela Inclezan, editors, *Practical Aspects of Declarative Languages*, pages 3–19, Cham, 2023. Springer Nature Switzerland.

[69] Jan Willem Klop. Term Rewriting Systems, Handbook of Logic in Computer Science. *Oxford Science Publications*, 3, 1992.

[70] Gérard P. Huet and Jean-Jacques Lévy. Computations in Orthogonal Rewriting Systems. In *Computational Logic - Essays in Honor of Alan Robinson*, 1991.

[71] Rachid Echahed and Jean-Christophe Janodet. On Constructor-based Graph Rewriting Systems. 1997.

[72] Detlef Plump. Term Graph Rewriting. *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*, pages 3–61, 1999.

[73] Sergio Antoy and Arthur Peters. Compiling a Functional Logic Language: The Basic Scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31. Springer LNCS 7294, 2012.

[74] Sergio Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.

[75] Sergio Antoy. Constructor-Based Conditional Narrowing. In *Proc. of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 199–206. ACM Press, 2001.

[76] Javier de Dios Castro and Francisco J. López-Fraguas. Extra Variables Can Be Eliminated from Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, 188:3–19, 2007. Proceedings of the Sixth Spanish Conference on Programming and Languages (PROLE 2006).

[77] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Conference on Functional Programming Languages and Computer Architecture*, pages 190–203. Springer, 1985.

[78] David HD Warren. *Higher-Order Extensions to Prolog – Are They Needed?* University of Edinburgh Department of Artificial Intelligence, 1990.

[79] Sergio Antoy. Definitional Trees. In *Algebraic and Logic Programming: Third International Conference Volterra, Italy, September 2–4, 1992 Proceedings 3*, pages 143–157. Springer, 1992.

[80] James R. Slagle. Automated Theorem-Proving for Theories with Simplifiers Commutativity, and Associativity. *J. ACM*, 21(4):622–642, Oct. 1974.

[81] Juan Carlos González Moreno, Maria Teresa Hortalá-González, and Mario Rodríguez-Artalejo. On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming. In *CSL*, 1992.

[82] Didier Bert and Rachid Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP 86*, pages 119–132, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[83] Laurent Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *SLP*, 1985.

[84] Joseph A. Goguen and José Meseguer. EQLOG: Equality, Types, and Generic Modules For Logic Programming. In *Logic Programming: Functions, Relations, and Equations*, 1986.

[85] Michael Hanus. Compiling Logic Programs with Equality. In Pierre Deransart and Jan Maluszyński, editors, *Programming Language Implementation and Logic Programming*, pages 387–401, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[86] Juan Jose Moreno-Navarro and Mario Rodriguez-Artalejo. Logic Programming with Functions and Predicates: The Language Babel. *The Journal of Logic Programming*, 12(3):191–223, 1992.

[87] Hassan Ait-Kaci. An Overview of LIFE. In *International East/West Database Workshop*, pages 42–58. Springer, 1990.

[88] John W Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and logic Programming*, 3(1-49):6–8, 1999.

[89] Hassan Aït-Kaci, Patrick Lincoln, and Roger Nasr. Le Fun: Logic, Equations, and Functions. In *In Proc. 4th IEEE Internat. Symposium on Logic Programming*, 1987.

[90] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–93, 1997.

[91] Jan Christiansen, Michael Hanus, Fabian Reck, and Daniel Seidel. A Semantics for Weakly Encapsulated Search in Functional Logic Programs. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, page 49–60, New York, NY, USA, 2013. ACM.

[92] Michael Hanus (ed.). CPM: The Curry Package Manager. Available at `https://www-ps.informatik.uni-kiel.de/~cpm/`, 2023.

[93] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International*

*Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[94] PyPy. Available at `https://pypy.org/`, .

[95] Cython. Available at `https://cython.org/`.

[96] Michael Hanus and Fabian Skrlac. A Modular and Generic Analysis Server System for Functional Logic Programs. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, page 181–188, New York, NY, USA, 2014. ACM.

[97] Michael Hanus. Combining Static and Dynamic Contract Checking for Curry. *Fundamenta Informaticae*, 173(4):285–314, 2020.

[98] Sergio Antoy, Michael Hanus, Andy Jost, and Steven Libby. ICurry. In *Declarative Programming and Knowledge Management: Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9–12, 2019, Revised Selected Papers 22*, pages 286–307. Springer, 2020.

[99] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, Apr. 1975.

[100] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[101] Stephen V Chenoweth. On the NP-Hardness of Blocks World. In *AAAI*, pages 623–628, 1991.

[102] Naresh Gupta and Dana S Nau. On the Complexity of Blocks-World Planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

[103] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 11–20, New York, NY, USA, 2008. ACM.

[104] ChatGPT sets record for fastest-growing user base. Available at `https://www.reuters.com/technology/`, 2022.

## Appendix: Supplemental Files

The source code for the Sprite Curry Compiler is provided with this document in a file named "`Sprite.FINAL.4.24.2023.tgz`" (2,497 kB). To unpack this archive, you can use the Linux `tar` program or similar software.