Making Curry with Rice

An Optimizing Curry Compiler

Steven Libby

May 15, 2022

Contents

Chapter 0

INTRODUCTION

With all of the chaos in the world today, sometimes it is nice to just relax and make a nice Curry. But people today are impatient. They cannot wait; they want their Curry fast. This is a problem, because Curry has historically been considered slow. Some have considered it unusably slow, which is a shame, because Curry is actually a great languguage, and can solve many problems well. In this dissertation we aim to recitify the problem of Curry taking too long. We present the `RICE` Curry compiler, and show how it can deliver a fast, satisfying, Curry.

0.0.1 Why Curry?

Functional logic programming is a very powerful technique for expressing complicated ideas in a simple form. Curry implements these ideas with a clean, easy to read syntax, which is similar to Haskell, a well known functional programming language. It is also lazy, so evaluation of Curry programs is similar to Haskell as well. Curry extends Haskell with two new concepts. First, there are non-deterministic functions, such as "?". Semantically $a \mathbin{?} b$ will evaluate a and b and will return both answers to the user. Second, there are free, or logic, variables. A free variable is a variable that is not in the scope of the current function. The value of a free variable is not defined, but it may be constrained.

Consider the following Curry code for solving n-queens:

$$\begin{aligned}&\mathit{queens} \mid \mathit{isEmpty}\;(\mathit{set1}\;\mathit{unsafe}\;p) = p\\ &\quad \textbf{where}\; p = \mathit{permute}\;[1\mathinner{\ldotp\ldotp}n]\\ &\mathit{unsafe}\;(\mathit{xs} \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \mathit{ys} \mathbin{+\!\!+} [b] \mathbin{+\!\!+} \mathit{zs}) = \mathit{abs}\;(a-b) =:= \mathit{length}\;\mathit{ys}\end{aligned}$$

In the *unsafe* function the input list is broken into 5 pieces. Two of the pieces, a and b, are lists with a single element. The sublists, xs, ys, and zs are free to be as long as they want. However, We have constrained the total list $\mathit{xs} \mathbin{+\!\!+} [a] \mathbin{+\!\!+} \mathit{ys} \mathbin{+\!\!+} [b] \mathbin{+\!\!+} \mathit{zs}$ to be the same as the argument. The effect is that a and b are arbitrary elements in the list, and ys is the list of elements between a and b. If the difference between a and b is equal to the distance between the two element, then two queens could capture each other diagonally. So, we compute the set of all capturing queens, If the set is empty, then we return that permutation.

Free variables are given concrete values in Curry programs through narrowing. The semantics of narrowing and non-determinism in Curry are given by Antoy et al. [18] Whenever the value of a free variable is needed, then we select a possible value to fill it in. For example if xs is a list, we might fill it in with either $[\,]$ or $(a : as)$ **where** a, as **free**. We give a fuller account of narrowing in chapters 1 and 2,

0.0.2 Current Compilers

There are currently two mature Curry compilers, Pakcs [52] and Kics2 [26]. Pakcs compiles Curry to Prolog in an effort to leverage Prolog's non-determinism and free variables. Kics2 compiles Curry to Haskell in an effort to leverage Haskell's higher order functions and optimizing compiler. Both compilers have their advantages. Pakcs tends to perform better on non-deterministic expressions with free variables, where Kics2 tends to perform much better on deterministic expressions. Unfortunately neither of these compilers perform well in both circumstances.

Sprite [20], an experimental compiler, aims to fix these inefficiencies. The strategy is to compile to a virtual assembly language, known as LLVM. So far, Sprite has shown promising improvements over both Pakcs and Kics2 in performance, but it is not readily available for testing at the time of this writing.

Similarly Mcc [71] also worked to improve performance by compiling to C. While Mcc often ran faster than both Pakcs or Kics2, it could perform very slowly on common Curry examples. It is also no longer in active development.

One major disadvantage of all four compilers is that they all attempt to pass off optimization to another compiler. Pakcs attempts to have Prolog optimize the non-deterministic code; Kics2 attempts to use Haskell to optimize deterministic code; Sprite attempts to use LLVM to optimize the low level code; and Mcc simply did not optimize its code. Unfortunately none of these approaches works very well. While some implementations of Prolog can optimize non-deterministic expressions, they have no concept of higher order functions, so there are many optimizations that cannot be applied. Kics2 is in a similar situation. In order to incorporate non-deterministic computations in Haskell, a significant amount of code must be threaded through each computation. This means that any non-deterministic expression cannot be optimized in Kics2. Finally, since

LLVM does not know about either higher order functions or non-determinism, it loses many easy opportunities for optimization.

Curry programs have one last hope for efficient execution. Recently, many scientists [80, 88] have developed a strong theory of partial evaluation for functional logic programs. While these results are interesting, partial evaluation is not currently automatic in Curry. Guidance is required from the programmer to run the optimization. Furthermore, the optimization fails to optimize several common programs.

0.0.3 The Need for Optimizations

So far, none of these approaches have included the large body of work on program optimizations [2, 3, 5–7, 21, 22, 38, 40, 44–46, 60, 70, 83, 87, 95, 99, 100]. This leads to the inescapable conclusion that Curry needs an optimizer. We propose a new compiler environment for developing and testing optimizations, which we call the Reduction Inspired Compiler Environment (`RICE`) Curry compiler. This compiler is intended to make developing new optimizations for Curry as simple as possible. We test this idea by developing several common optimizations for the `RICE` compiler. Furthermore we implement three specific optimizations for Curry, Unboxing [58], Case Shortcutting [19], and Deforestation [40]. While Unboxing and Deforestation are well known in the function languages community, the techniques have not been applied in a function logic setting. Case Shortcutting is a unique optimization for functional logic programs. We chose these optimizations specifically because they focus on reducing the amount of memory consumed by programs, which is a common problem for Curry programs [68].

The rest of this dissertation is organized as follows. Chapter 1 presents the mathematical background of Term and Graph Rewriting. Notions from rewriting will be used throughout this dissertation, both because the operational semantics of Curry were first described using rewriting, and because our optimizing engine is based on constructing rewrite rules. Chapter 2 presents the Curry Language and its semantics. We introduce the Curry language and describe the IR FlatCurry as well as some conceptual hurdles with implementing a functional logic language. We also introduce two novel approaches to improving the performance of evaluation, case function and fast backtracking. Case functions can be applied to any evaluation model for Curry, while fast backtracking is specific to backtracking implementations. Chapter 3 discusses the target code for this compiler. We describe, by example, the generated code for simple functions, then we describe the changed needed to add additional features of Curry. Chapter 4 introduces the

GAS system for implementing optimizations. We describe the system, its implementation, and show how to construct optimizations with it. Chapter 5 overviews the compiler pipeline, and the translation to C. We show the compiler pipeline, and how GAS simplifies several of the transformations. Chapter 6 discusses the implementation of several common optimizations. We show several common optimizations including inlining, reduction, and case canceling. We also introduce A-Normal form, which is required for the correctness of these optimizations. Chapter 7 discusses the implementation of Unboxing, Shortcutting, and Deforestation. Chapter 8 shows the results of our optimizations. Finally, chapter 9 concludes and discusses future work.

Chapter 1

MATHEMATICAL BACKGROUND

When cooking, it is very important to follow the rules. You do not need to stick to an exact recipe, but you do need to know the how ingredients will react to temperature and how different combinations will taste. Otherwise you might get some unexpected reactions.

Similarly, there is not a single way to compile Curry programs, however we do need to know the rules of the game. Throughout this compiler, we will be transforming Curry programs in many different ways, and it is important to make sure that all of these transformations respect the rules of Curry. As we will see, if we break these rules, then we may get some unexpected results.

We introduce the concept of Rewriting, along with the more specific Term and Graph Rewriting. We give a basic intuition about how to apply these topics, and show several examples using a small, but not trivial, example of a rewrite system for Peano Arithmetic **??**.

1.1 REWRITING

In programming language terms, the rules of Curry are its semantics. The semantics of Curry are generally given in terms of rewriting. [13, 18, 48] While there are other semantics [4, 43, 96], rewriting is a common formalism for many functional languages, and the general theory of Curry grew out of this discipline [18], a good fit for Curry [51]. We will give a definition of rewrite systems, then we will look at two distinct types of rewrite systems: Term Rewrite Systems, which are used to implement transformations and optimizations on the Curry syntax trees; and Graph Rewrite Systems, which define the operational semantics for Curry programs. This mathematical foundation will help us justify the correctness of our transformations even in the presence of laziness, non-determinism, and free variables.

An Abstract Rewriting System (ARS) is a set A along with a relation $\to$. We use $a \to b$ as a shorthand for $(a, b) \in\to$, and we have several modifiers on our relation.

- $a \to^n b$ iff $a = x_0 \to x_1 \to \dots x_n = b$.
- $a \to^{\leq n} b$ b iff $a \to^i b$ and $i \leqslant n$.
- reflexive closure: $a \to^= b$ iff $a = b$ or $a \to b$.
- symmetric closure: $a \leftrightarrow b$ iff $a \to b$ or $b \to a$.
- transitive closure: $a \to^+ b$ iff $\exists n \in \mathbb{N}.a \to^{\leq n} b$.
- reflexive transitive closure: $a \to^* b$ iff $a \to^= b$ or $a \to^+ b$.
- rewrite derivation: a sequence of rewrite steps $a_0 \to a_1 \to \dots a_n$.
- a is in Normal Form (NF) if no rewrite rules can apply.

A rewrite system is meant to invoke the feeling of algebra. In fact, rewrite system are much more general, but they can still retain the feeling. If we have an expression $(x \cdot x + 1)(2 + x)$, we might reduce this with the reduction in figure **??**.

We can conclude that $(x \cdot x + 1)(x + 2) \to^+ x^3 + 2x^2 + x + 2$. This idea of rewriting invokes the feel of algebraic rules. The mechanical process of rewriting allows for a simple implementation on a computer.

It is worth understanding the properties and limitations of these rewrite systems. Traditionally there are two important questions to answer about any rewrite system. Is it *confluent*? Is it *terminating*?

A *confluent* system is a system where the order of the rewrites does not change the final result. For example, consider the distributive rule. When evaluating $3 \cdot (4 + 5)$ we could either evaluate the addition or multiplication first. Both of these reductions arrived at the same answer as can be seen in figure **??**.

In a *terminating* system every derivation is finite. That means that eventually there are no rules that can be applied. The distributive rule is terminating, whereas the commutative rule is not terminating. See figure **??**.

Confluence and termination are important topics in rewriting, but we will largely ignore them. After all, Curry programs are neither confluent nor terminating. However, there will be a few cases where these concepts will be important. For example, if our optimizer is not terminating, then we will never actually compile a program.

Now that we have a general notation for rewriting, we can introduce two important rewriting frameworks: term rewriting and graph rewriting, where we are transforming trees and graphs respectively.

1.2 TERM REWRITING

As mentioned previously, one application of term rewriting is to transform terms representing syntax trees. This will be useful in optimizing the Abstract Syntax Trees (ASTs) of Curry programs. Term rewriting is a special case of abstract rewriting. Therefore everything from abstract rewriting will apply to term rewriting.

A term is made up of signatures and variables. [78][Def 3.1.2] We let Σ and V be two arbitrary alphabets, but we require that V be countably infinite, and $\Sigma \cap V = \emptyset$ to avoid name conflicts. A *signature* $f^{(n)}$ consists of a name $f \in \Sigma$ and an arity $n \in \mathbb{N}$. A *variable* $v \in V$ is just a name. Finally a *term* is defined inductively. The term t is either a variable v, or it is a signature $f^{(n)}$ with children $t_1, t_2, \dots t_n$, where $t_1, t_2, \dots t_n$ are all terms. We write the set of terms all as $T(\Sigma, V)$. If $t \in T(\Sigma, V)$ then we write $Var(t)$ to denote the set of variables in t. By definition $Var(t) \subseteq V$. We say that a term is *linear* if no variable appears twice in the term [78][Def. 3.2.4].

This inductive definition gives us a tree structure for terms. As an example consider Peano arithmetic $\Sigma = \{+^2, *^2, -^2, <^2, 0^0, S^1, True^0, False^0\}$. We can define the term $*(+(0, S(0)), +(S(0), 0))$. This gives us the tree in figure **??**. Every term can be converted into a tree like this and vice versa. The symbol at the top of the tree is called the root of the term.

A *child* c of term $f(t_1, t_2, \dots t_n)$ is one of $t_1, t_2, \dots t_n$. A *subterm* s of t is either t itself, or it is a subterm of a child of t. We write $s = t|_p$ where $p = [i_1, i_2, \dots i_n]$ to denote that t has child t_{i_1} which has child t_{i_2} and so on until $t_{i_n} = s$. Note that we can define this recursively as $t|_{[i_1, i_2, \dots i_n]} = t_{i_1}|_{[i_2, \dots i_n]}$, which matches our definition for subterm. We call $[i_1, i_2, \dots i_n]$ the *path* from t to s [78][Def 3.1.5]. We write ϵ for the empty path, and $i{:}p$ for the path starting with the number i and followed by the path p, and $p \cdot q$ for concatenation of paths p and q.

In our previous term $S(0)$ is a subterm in two different places. One occurrence is at path $[0, 1]$, and the other is at path $[1, 0]$.

We write $t[p \rightarrow r]$ to denote replacing subterm $t|_p$ with r. We define the algorithm for this in figure **??**.

In our above example $t = *(+(0, S(0), +(S(0), 0)))$, We can compute the rewrite $t[[0, 1] \rightarrow *(S(0), S(0))]$, and we get the term $*(+(0, *(S(0), S(0))), +(S(0), 0))$, with the tree in figure **??**.

A substitution replaces variables with terms. Formally, a *substitution* is a mapping from $\sigma{:}V \rightarrow T(\Sigma, V)$, such that $\sigma(x) \neq x$ [78][Def. 3.1.7]. We write $\sigma = \{v_1 \mapsto t_1, \ldots v_n \mapsto t_n\}$ to denote the substitution where $s(v_i) = t_i$ for $i \in \{1 \ldots n\}$, and $s(v) = v$ otherwise. We can uniquely extend σ to a function on terms by figure **??**

Since this extension is unique, we will just write σ instead of σ'. Term t_1 *matches* term t_2 if there exists some substitution σ such that $t_1 = \sigma(t_2)$ [78][3.1.8],. We call σ a *matcher*. Two terms t_1 and t_2 *unify* if there exists some substitution σ such that $\sigma(t_1) = \sigma(t_2)$ [78][3.1.8],. In this case σ is called a *unifier* for t_1 and t_2.

We can order substitutions based on what variables they define. A substitution $\sigma \leqslant \tau$, iff, there is some substitution ν such that $\tau = \nu \circ \sigma$. The relation $\sigma \leqslant \tau$ should be read as σ is more general than τ, and it is a quasi-order on the set of substitutions. A unifier u for two terms is *most general* (or an mgu), iff, for all unifiers v, $v \leq u$. Mgus are unique up to renaming of variables. That is, if u_1 and u_2 are mgus for two terms, then $u_1 = \sigma_1 \circ u_2$ and $u_2 = \sigma_2 \circ u_1$. This can only happen if σ_1 and σ_2 just rename the variables in their terms.

As an example $+(x, y)$ matches $+(0, S(0))$ with $\sigma = \{x \mapsto 0, y \mapsto S(0)\}$. The term $+(x, S(0))$ unifies with term $+(0, y)$ with unifier $\sigma = \{x \mapsto 0, y \mapsto S(0)\}$. If $\tau = \{x \mapsto 0, y \mapsto S(z)\}$, then $\tau \leq \sigma$. We can define $\nu = \{z \mapsto 0\}$, and $\{\sigma = \nu \circ \tau\}$

Now that we have a definition for a term, we need to be able to rewrite it. A *rewrite rule* $l \rightarrow r$ is a pair of terms. However this time we require that $Var(r) \subseteq Var(l)$, and that $l \notin V$. A *Term Rewriting System (TRS)* is the pair $(T(\Sigma, V), R)$ where R is a set of rewrite rules.

Definition 1.2.1. Rewriting: Given terms t, s, path p, and rule $l \rightarrow r$, we say that t rewrites to s if, l matches $t|_p$ with matcher σ, and $t[p \rightarrow \sigma(r)] = s$. The term $\sigma(l)$ is the *redex*, and the term $\sigma(r)$ is the *contractum* of the rewrite.

There are a few important properties of rewrite rules $l \rightarrow r$. A rule is left or right linear if l or r is linear respectively [78][Def. 3.2.4]. A rule is *collapsing* if $r \in V$. A rule is *duplicating* if there is an $x \in V$ that occurs more often in r than in l [78][Def. 3.2.5].

Two terms s and t are *overlapping* if t unifies with a subterm of s, or s unifies with a subterm of t at a non-variable position [78][Def. 4.3.3]. Two rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ if l_1 and l_2 overlap. A rewrite system is *overlapping* if, and only if, any two rules overlap. Otherwise it is non-overlapping. Any non-overlapping left linear system is *orthogonal* [78][Def.4.3.4]. Orthogonal systems have several nice properties, such as the following theorem [78][Thm. 4.3.11].

Theorem 1. *Every orthogonal TRS is confluent.*

As an example, in figure **??** examples (b) and (c) both overlap. It is clear that these systems are not confluent, but non-confluence can arise in more subtle ways. The converse to theorem 2.1 is not true. There can be overlapping systems which are confluent.

When defining rewrite systems we usually follow the constructor discipline; we separate the set $\Sigma = C \uplus F$. C is the set of *constructors*, and F is the set of *function symbols*. Furthermore, for every rule $l \to r$, the root of l is a function symbol, and every other symbol is a constructor or variable. We call such systems *constructor systems*. As an example, the rewrite system for Peano arithmetic is a constructor system.

The two sets are $C = \{0, S, True, False\}$ and $F = \{+, *, -, \leq\}$, and the root of the left hand side of each rule is a function symbol. In contrast, the SKI system is not a constructor system. While S, K, I can all be constructors, the Ap symbol appears in both root and non-root positions of the left hand side of rules. This example will become important for us in Curry. We will do something similar to implement higher order functions. This means that Curry programs will not directly follow the constructor discipline. Therefore, we must be careful when specifying the semantics of function application.

Constructor systems have several nice properties. They are usually easy to analyze for confluence and termination. For example, if the left hand side of two rules do not unify, then they cannot overlap. We do not need to check if subterms overlap. Furthermore, any term that consists entirely of constructors and variables is in normal form. For this reason, it is not surprising that most functional languages are based on constructor systems.

1.3 NARROWING

Narrowing was originally developed to solve the problem of semantic unification. The goal was, given a set of equations $E = \{a_1 = b_2, a_2 = b_2, \ldots a_n = b_n\}$, to solve the equation $t_1 = t_2$ for arbitrary terms t_1 and t_2. Here a solution to $t_1 = t_2$ is a substitution σ such that $\sigma(t_1)$ can be transformed into $\sigma(t_2)$ by the equations in E.

As an example let $E = \{*(x + (y, z)) = +(*(x, y), *(x, z))\}$ Then the equation $*(1, +(x, 3)) = +(+(*(1, 4), *(y, 5)), *(z, 3))$ is solved by $\sigma = \{x \mapsto +(4, 5), y \mapsto 1, z \mapsto 1\}$. The derivation is in figure **??**.

Unsurprisingly, there is a lot of overlap with rewriting. One of the earlier solutions to this problem was to convert the equations into a confluent, terminating rewrite system. [63] Unfortunately, this only works for ground terms, that is, terms without variables. However, this idea still has merit. So we want to extend it to terms with variables.

Before, when we rewrote a term t with rule $l \to r$, we assumed it was a ground term, then we could find a substitution σ that would match a subterm $t|_p$ with l, so that $\sigma(l) = t|_p$. To extend this idea to terms with variables in them, we look for a unifier σ that unifies $t|_p$ with l. This is really the only change we need to make [78]. However, now we record σ, because it is part of our solution.

Definition 1.3.1. Narrowing: Given terms t, s, path p, and rule $l \to r$, we say that t narrows to s if, l unifies with $t|_p$ with unifier σ, and $t[p \to \sigma(r)] = s$. We write $t \leadsto_{p,l\to r,\sigma} s$. We may write $t \leadsto_\sigma s$ if p and $l \to r$ are clear.

Notice that this is almost identical to the definition of rewriting. The only difference is that σ is a unifier instead of a matcher.

Narrowing was first developed to solve equations for automated theorem provers [93]. However, for our purposes it is more important that narrowing allows us to rewrite terms with free variables. [47]

At this point, rewrite systems are a nice curiosity, but they are completely impractical. This is because we do not have a plan for solving equations in them. In the definition for both rewriting and narrowing, we did not specify how to find σ the correct rule to apply, or even what subterm to apply the rule.

In confluent terminating rewrite systems, we could simply try every possible rule at every possible position with every possible substitution. Since the system is confluent, we could choose the first rule that could be successfully applied, and since the system is terminating, we would be sure to find a normal form. In a narrowing system, this is still not guaranteed to halt, because there could be an infinite number of substitutions. This is the best possible case for rewrite systems, and we still cannot ensure that our algorithm will finish. We need a systematic method for deciding what rule should be applied, what subterm to apply it to, and what substitution to use. This is the role of a strategy.

1.4 REWRITING STRATEGIES

Our goal with a rewriting strategy is to be able to find a normal form for any term. Similarly our goal for narrowing will be to find a normal form and substitution. However, we want to be efficient when rewriting. We would like to use only local information when deciding what rule to select. We would also like to avoid unnecessary rewrites. Consider the following term from the SKI system defined in figure **??** $Ap(Ap(K,I), Ap(Ap(S, Ap(I,I)), Ap(S, Ap(I,I))))$. It would be pointless to reduce $Ap(Ap(S, Ap(I,I)), Ap(S, Ap(I,I))))$ since $Ap(Ap(K,I,z)$ rewrites to I no matter what z is. In this particular case, since $Ap(Ap(S, Ap(I,I)), Ap(S, Ap(I,I))))$ reduces to itself, we have turned a potentially non-terminating reduction to a terminating one.

A *Rewriting Strategy* $\mathcal{S}{:}T(\Sigma, V) \to Pos \times R$ is a function that takes a term, and returns a position to rewrite, and a rule to rewrite with [62]. Furthermore we require that if $(p, l \to r) = \mathcal{S}(t)$, then $t|_p$ is a redex that matches l. The idea is that $S(t)$ should give us a position to rewrite, and the rule to rewrite with.

For orthogonal rewriting systems, there are two common rewriting strategies that do not run in parallel,[1] innermost and outermost rewriting [62, 72]. Innermost rewriting corresponds to eager evaluation in functional programming. We rewrite the term that matches a rule that is the furthest down the tree. Outermost rewriting correspond roughly to lazy evaluation. We rewrite the highest possible term that matches a rewrite rule.

A strategy is *normalizing* if, when a term t has a normal form, then the strategy will eventually find it. While outermost rewriting is not normalizing in general, it is for left-normal systems, which is a large subclass of orthogonal rewrite systems [62]. This matches the intuition from programming languages. Lazy languages can perform computations that would run forever with an eager language.

While both of these strategies are well understood, we can actually make a stronger guarantee. We want to reduce only the redexes that are necessary to find a normal form. To formalize this we need to understand what can happen when we rewrite a term. Specifically for a redex s that is a subterm of t, how can s change as we rewrite t. If we were rewriting at position p with rule $l \to r$, then there are 3 cases to consider.

Case 1: we are rewriting s itself. That is, s is the subterm $t|_p$. Then s disappears entirely.

[1] we avoid discussing parallel strategies, because our work is focused on sequential execution of Curry programs. That has been a lot of work done on parallel execution of Curry programs elsewhere [49, 50].

Case 2: s is either above $t|_p$, or they are completely disjoint. In this case s does not change.
Case 3: s is a subterm of $t|_p$. In this case s may be duplicated, or erased, moved, or left unchanged. It depends on whether the rule is duplicating, erasing, or right linear.
These cases can be seen in figure **??** We can formalize this with the notion of descendants with the following definition from [78][Def. 4.3.6].

Definition 1.4.1. Descendant: Let $s = t|_v$, and $A = l \rightarrow_{p,\sigma,R} r$ be a rewrite step in t. The set of descendants of s is given by $Des(s, A)$

$$Des(s,A) = \begin{cases} \emptyset & \text{if } v = u \\ \{s\} & \text{if } p \not\leq v \\ \{t|_{u \cdot w \cdot q} : r|_w = x\} & \text{if } p = u \cdot v \cdot q \text{ and } t|_v = x \text{ and } x \in V \end{cases}$$

This definition extends to derivation $t \rightarrow_{A_1} t_1 \rightarrow_{A_2} t_2 \rightarrow_{A_2} \ldots \rightarrow_{A_n}, t_{n+1}$. $Des(s, A_1, A_2 \ldots A_n) = \bigcup_{s' \in Des(s,A_1)} Des(s', A_2, \ldots A_n)$.

The first part of the definition is formalizing the notion of descendant. The second part is extending it to a rewrite derivation. The extension is straightforward. Calculate the descendants for the first rewrite, then for each descendant, calculate the descendants for the rest of the rewrites. With the idea of a descendant, we can talk about what happens to a term in the future. This is necessary to describing our rewriting strategy. Now we can formally define what it means for a redex to be necessary for computing a normal form.

Definition 1.4.2. Needed: A redex s that is a subterm of t is *needed* in t if, for every derivation of t to a normal form, a descendant of s is the root of a rewrite.

This definition is good because it is immediately clear that, if we were going to rewrite a term to normal form, we need to rewrite all of the needed redexes. In fact, we can guarantee more than that with the following theorem [59].

Theorem 2. *For an orthogonal TRS, any term that is not in normal form contains a needed redex. Furthermore, a rewrite strategy that rewrites only needed redexes is normalizing.*

This is a very powerful result. We can compute normal forms by rewriting needed redexes. This is also, in some sense, the best possible strategy. Every needed redex needs to be rewritten. Now we just need to make sure our strategy only rewrites needed redexes. There is only one problem with this plan. Determining if a redex is needed is undecidable in general. However, with some restrictions, there are rewrite systems where this is possible [62][def. 3.3.7].[2]

Definition 1.4.3. Sequential A rewrite system is *sequential* if, given a term t with n variables $v_1, v_2 \dots v_n$, such that t is in normal form, then there is an i such that for every substitution σ from variables to redexes, $\sigma(v_i)$ is needed in $\sigma(t)$.

If we have a sequential rewrite system, then this leads to an efficient algorithm for reducing terms to normal form. Unfortunately, sequential is also an undecidable property. There is still hope. As we will see in the next section, with certain restrictions we can ensure the our rewrite systems are sequential. Actually we can make a stronger guarantee. The rewrite system will admit a narrowing strategy that only narrows needed redexes.

1.5 NARROWING STRATEGIES

Similar to rewriting strategies, narrowing strategies attempt to compute a normal form for a term using narrowing steps. However, a narrowing strategy must also compute a substitution for that term. There have been many narrowing strategies including basic [56], innermost [39], outermost [101], standard [36], and lazy [74]. Unfortunately, each of these strategies are too restrictive on the rewrite systems they allow.

Fortunately there exists a narrowing strategy that is defined on a large class of rewrite systems, only narrows needed expressions, and is sound and complete. However this strategy requires a new construct called a definitional tree.

The idea is that since we are working with constructor rewrite systems, we can group all of the rules defined for the same function symbol together. We will put them together in a tree structure defined below, and then we can compute a narrowing step by traversing the tree for the defined symbol.

Definition 1.5.1. T is a *partial definitional tree* if T is one of the following.
$T = exempt(\pi)$ where π is a pattern.
$T = leaf(\pi \to r)$ where π is a pattern, and $\pi \to r$ is a rewrite rule.
$T = branch(\pi, o, T_1, \dots T_k)$, where π is a pattern, o is a path, $\pi|_o$ is a variable, $c_1, \dots c_k$ are constructors, and T_i is a pdt with pattern $\pi[c_i(X_1, \dots X_n)]_o$ where n is the arity of c_i, and $X_1, \dots X_n$

[2] The original definition used the notion of a context in normal form.

are fresh variables.

Given a constructor rewrite system R, T is a *definitional tree* for function symbol f if T is a partial definitional tree, and each leaf in T corresponds to exactly one rule rooted by f. A rewrite system is *inductively sequential* if there exists a definitional tree for every function symbol.

The name "inductively sequential" is justified because there is a narrowing strategy that only reduces needed redexes for any of these systems. We show an example to clarify the definition. In figure **??** we show the definitional tree for the $+, \leq$, and $=$ rules. The idea is that, at each branch, we decide which variable to inspect. Then we decide what child to follow based on the constructor of that branch. This gives us a simple algorithm for outermost rewriting with definitional trees. However, we need to extend this to narrowing.

In order to extend the strategy from rewriting to narrowing we need to figure out how to compute a substitution, and we need to define what it means for a narrowing step to be needed. The earliest definition involved finding a most general unifier for the substitution. This has some nice properties. There is a well known algorithm for computing mgus, which are unique up to renaming of variables. However, this turned out to be the wrong approach. Computing mgus is too restrictive. Consider the step $x \leq y + z \rightsquigarrow_{2 \cdot \epsilon, R_1, \{y \mapsto 0\}} x \leq z$. Without further substitutions $x \leq z$ is a normal form, and $\{y \mapsto 0\}$ is an mgu. Therefore this should be a needed step. But if we were to instead narrow x, we have $x \leq y + z \rightsquigarrow_{\epsilon, R_8, \{x \mapsto 0\}} True$. This step never needs to compute a substitution for y. Therefore we need a definition that is not dependent on substitutions that might be computed later.

Definition 1.5.2. A narrowing step $t \rightsquigarrow_{p,R,\sigma} s$ is needed, iff, for every $\eta \geq \sigma$, there is a needed redex at p in $\eta(t)$.

Here we do not require that σ be an mgu, but, for any less general substitution, it must be the case that we were rewriting a needed redex. So our example, $x \leq y + z \rightsquigarrow_{2\dot{\epsilon}, R_1, \{y \mapsto 0\}} x \leq z$, is not a needed narrowing step because $x \leq y + z \rightsquigarrow_{2\dot{\epsilon}, R_1, \{x \mapsto 0, y \mapsto 0\}} 0 \leq z$, Is not a needed rewriting step.

Unfortunately, this definition raises a new problem. Since we are no longer using mgus for our unifiers, we may not have a unique step for an expression. For example, $x < y \rightsquigarrow_{\epsilon, R_8, \{x \mapsto 0\}} True$, and $x < y \rightsquigarrow_{\epsilon, R_9, \{x \rightarrow S(u), t \mapsto S(v)\}} u \leq v$ are both possible needed narrowing steps.

Therefore we define a *Narrowing Strategy* $\mathcal{S}$ as a function from terms to a set of triples of a position, rule, and substitution, such that if $(p, R, \sigma) \in \mathcal{S}(t)$, then $\sigma(t)|_p$ is a redex for rule R.

At this point we have everything we need to define a needed narrowing strategy.

Definition 1.5.3. Let t be a term rooted by function symbol f, T be the definitional tree for f, and "?" be a distinguished symbol to denote that no rule could be found.

$$\lambda(t, T) \in \begin{cases} (\epsilon, R, mgu(t, \pi)) & \text{if } T = rule(\pi, R) \\ (\epsilon, ?, mgu(t, \pi)) & \text{if } T = exempt(\pi) \\ (p, R, \sigma) & \text{if } T = branch(\pi, o, T_1, \ldots T_n) \\ & t \text{ unifies with } T_i \\ & (p, R, \sigma) \in \lambda(t, T_i) \\ (o{:}p, R, \sigma \circ \tau) & \text{if } T = branch(\pi, o, T_1, \ldots T_n) \\ & t \text{ does not unify with any } T_i \\ & \tau = mgu(t, \pi) \\ & T' \text{ is the definitional tree for } t|_o \\ & (p, R, \sigma) \in \lambda(t|_o, T') \end{cases}$$

The function λ is a narrowing strategy. It takes an expression rooted by f, and the definition tree for f, and it returns a position, rule and substitution for a narrowing step. If we reach a rule node, then we can just rewrite; if we reach an exempt node, then there is no possible rewrite; if we reach a branch node, then we match a constructor; but if the subterm we were looking at is not a constructor, then we need to narrow that subterm first.

Theorem 3. *λ is a needed narrowing strategy. Furthermore, λ is sound and complete.*

It should be noted that while λ is complete with respect to finding substitutions and selecting rewrite rules [18], this says nothing about the underlying completeness of the rewrite system we were narrowing. We may still have non-terminating derivations.

This needed narrowing strategy is important in developing the evaluation strategy for Curry programs. In fact, one of the early stages of a Curry compiler is to construct definitional trees for each function defined. However, if we were to implement our compiler using terms, it would be needlessly inefficient. We solve this problem with graph rewriting.

1.6 GRAPH REWRITING

As mentioned above term rewriting is too inefficient to implement Curry. Consider the rule $double(x) = x + x$. Term rewriting requires this rule to make a copy of x, no matter how large it is, whereas we can share the variable if we use a graph. In programming languages, this distinction moves the evaluation strategy from "call by name" to "call by need", and it is what we mean when we refer to "lazy evaluation".

As a brief review of relevant graph theory: A *graph* $G = (V, E)$ is a pair of vertices V and edges $E \subseteq V \times V$. We will only deal with directed graphs, so the order of the edge matters. A *rooted graph* is a graph with a specific vertex r designated as the *root*. The *neighborhood* of v, written $N(v)$ is the set of vertices adjacent to v. That is, $N(v) = \{u \mid (v, u) \in E\}$. A *path* p from vertex u to vertex v is a sequence $u = p_1, p_2 \ldots p_n = v$ where $(p_i, p_{i+1}) \in E$. A rooted graph is *connected* if there is a path from the root to every other vertex in the graph. A graph is *strongly connected* if, for each pair of vertices (u, v), there is a path from u to v and a path form v to u. A path p is a cycle [3] if its endpoints are the same. A graph is acyclic if it contains no cycles. Such graphs are referred to as Directed Acyclic Graphs, or DAGs. A graph H is a *subgraph* of G, $H \subseteq G$ if, and only if, $V_H \subseteq V_G$ and $E_H \subseteq E_G$. A strongly connected component S of G is a subgraph that is strongly connected. We will use the well-known facts that strongly connected components partition a graph. The component graph, which is obtained by shrinking the strongly connected components to a single vertex, is a DAG. To avoid confusion with variables, we will refer to vertices of graphs as nodes.

We define term graphs in a similar way to terms. Let $\Sigma = C \uplus F$ be an alphabet of constructor and function names respectively, and V be a countably infinite set of variables. A *term graph* is a rooted graph G with nodes in N where each node n has a label in $\Sigma \cup V$. We will write $L(n)$ to refer to the label of a node. If $(n, s) \in E$ is an edge, then s is a successor of n. In most applications the order of the outgoing edges does not matter, however it is very important in term graphs. So, we will refer to the first successor, second successor and so on. We denote this the same way we did with terms n_i is the ith successor of n. The arity of a node is the number of successors. Finally, no two nodes can be labeled by the same variable.

While the nodes in a term graph are abstract, in reality, they connected using pointers in the

[3] Some authors will use walk and tour and reserve path and cycle for the cases where there are no repeated vertices. This distinction is not relevant for our work.

implementation. It can be helpful to keep this in mind. As we define more operations on our term graphs, there exists a natural implementation using pointers.

We will often use a linear notation to represent graphs. This has two advantages. The first is that it is exact. There are many different ways to draw the same graph, but there is only one way to write it out a linear representation [35] The second is that this representation corresponds closely to the representation in a computer. The notation these graphs is given by the following grammar, where the set of nodes and the set of labels are disjoint.

$$\begin{aligned} \mathit{Graph} &\rightarrow \mathit{Node} \\ \mathit{Node} &\rightarrow n : L\ (\mathit{Node}, \ldots \mathit{Node}) \\ &\mid\ n \end{aligned}$$

We start with the root node, and for each node in the graph, If we have not encountered it yet, then we write down the node, the label, and the list of successors. If we have seen it, then we just write down the node. If a node does not have any successors, then we will omit the parentheses entirely, and just write down the label.

A few examples are shown in figure **??**. Example 1 shows an expression where a single variable is shared several times. Example 2 shows how a rewrite can introduce sharing. Example 3 shows an example of an expression with a loop. These examples would require an infinitely large term, so they cannot be represented in term rewrite systems. Example 4 shows how reduction changes from terms to graphs. In a term rewrite system, if a node is in the pattern of a redex, then it can safely be discarded. However, in graph rewriting this is no longer true.

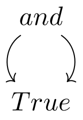
Definition 1.6.1. Let p be a node in G, then the *subgraph* $G|_p$ is a new graph rooted by p. The nodes are restricted to only those reachable from p.

Notice that we do not define subgraphs by paths like we did with subterms. This is because there may be more than one path to the node p. It may be the case that $G|_p$ and G have the same nodes, such as if the root of G is in a loop.

Definition 1.6.2. A *replacement* of node p by graph u in g (written $g[p \rightarrow u]$) is given by the following procedure. For each edge $(n, p) \in E_g$ replace it with an edge $(n, root_u)$. Add all other edges from E_g and E_u. If p is the root of g, then $root_u$ is now the root.

It should be noted that when implementing Curry, we do not actually change any of the pointers when doing a replacement. Traversing the graph to find all of the pointers to p would be horribly inefficient. Instead we change the contents of p to be the contents of u.

We can define matching in a similar way to terms, but we need to be more careful. When matching terms the structure of the term must to be the same. That is, both terms must have exactly the same tree. However, when matching graphs the structure can be wildly different. Consider the following graph.



Here the graph should match the rule $and(True, True) \to True$. But $and(True, True)$ is a term, so they no longer have the same structure. Therefore we must be more careful about what we mean by matching. We define matching inductively on the structure of the term.

Definition 1.6.3. A graph K *matches* a term T if, and only if, T is a variable, or $T = l(T_1, T_2 \dots T_n)$, the root of K is labeled with l, and for each $i \in \{1 \dots n\}$, K_i matches T_i.

Now, it may be the case that we have multiple successors pointing to the same node when checking if a graph matches a pattern, but this is OK. As long as the node matches each sub pattern, then the graph will match. We extend substitutions to graphs in the obvious way. A substitution σ maps variables to Nodes. In this definition for matching σ may have multiple variables map to the same node, but this does not cause a problem.

Definition 1.6.4. A *rewrite rule* is a pair $L \to R$ where L is a term, and R is a term graph. A graph G matches the rule if there exists subgraph K where K matches L with matcher σ. A *rewrite* is a triple $(K, L \to R, \sigma)$, and we apply the rewrite with $G[K \to \sigma(R)]$.

From here we can define narrowing similarly to how we did for terms. We do not give the definitions here, because they are similar to the definitions in term rewriting. At this point we have discussed the difference between graphs and terms, and how a replacement can be done in a graph. For our purposes in this compiler, that is all that is needed, but the definition of narrowing and properties about inductively sequential GRSs can be found in Echaned and Janodet [35]. They also show that the needed narrowing strategy is still valid for graph rewriting systems.

1.7 PREVIOUS WORK

This was not meant to be an exhaustive examination of rewriting, but rather an introduction to the concepts, since they form this theoretical basis of the Curry language. Most work on term rewriting up through 1990 has been summarized by Klop [62], and Baader and Nipkow [23]. The notation and ideas in this section largely come from Ohlebusch [78], although they are very similar to the previous two summaries. The foundations of term rewriting were laid by Church, Rosser, Curry, Feys, Newman. [31, 33, 77] Most of the work on rewriting has centered on confluence and termination. [62] Narrowing has been developed by Slagle [93]. Sequential strategies were developed by Huet and Levy [54], who gave a decidable criteria for a subset of sequential systems. This led to the work of Antoy on inductively sequential systems [17]. The needed narrowing strategy came from Hanus, Antoy, and Echahed [18]. Graph rewriting is a bit more disconnected. Currently there is not a consensus on how to represent graphs mathematically. We went with the presentation in [35], but there are also alternatives in [23, 62, 78]

Here we saw how we can rewrite terms and graphs. We will use this idea in the next chapter to rewrite entire programs. This will become the semantics for our language. Now that we have some tools, It is time to find out how to make Curry!

Chapter 2

THE CURRY LANGUAGE

The Curry language grew out of the efforts to combine the functional and logic programming paradigms [47]. Originally there were two approaches to combine these paradigms, adding functional features to logic languages, and adding logic features to functional languages. The former approach was very popular and spawned several new languages including Ciao-Prolog [53], Mercury [94], HAL [34], and Oz [91]. The extension of functional languages lead to fewer new languages, but it did lead to libraries like the logict monad in Haskell [61].

Ultimately the solution came from the work on automated theorem proving [93]. Instead of adding features from one paradigm to another, it was discovered that narrowing was a good abstraction for combining the features from both paradigms. This spawned the Curry [48] and Toy [29] languages.

In this chapter we explore the Curry language syntax and semintics. We give example programs to show how programming in Curry differs from Prolog and Haskell. Then we discuss the choices we made in our implementation compared to previous implementations. Finally we give an example of generated code to demonstrate how we compile Curry programs.

2.1 THE CURRY LANGUAGE

In order to write a compiler for Curry, we need to understand how Curry works. We will start by looking at some examples of Curry programs. We will see how Curry programs differ from Haskell and Prolog programs. Then we will move on to defining a small interpreter for Curry. Finally we will use this interpreter to define equivalent C code.

Curry combines the two most popular paradigms of declarative programming: Functional languages and logic languages. Curry programs are composed of defining equations like Haskell or ML, but we are allowed to have non-deterministic expressions and free variables like Prolog. This will not be an introduction to modern declarative programming languages. The reader is expected to be familiar with functional languages such as Haskell or ML, and logic languages such as Prolog. For an introduction to programming in Curry see [14]. For an exhaustive explanation of the syntax and semantics of Curry see [51].

To demonstrate the features of Curry, we will examine a small Haskell program to permute a list. Then we will simplify the program by adding features of Curry. This will demonstrate the features of Curry that we need to handle in the compiler, and also give a good basis for how we can write the compiler.

First, let us consider an example of a permutation function. This is not the only way to permute a list in Haskell, and you could easily argue that it is not the most elegant way, but we chose it for three reasons. There is no syntactic sugar, and the only two library functions are *concat* and *map*, both very common functions, and the algorithm for permuting a list is similar to the algorithm we will use in Curry.

```
perms          :: [a] → [[a]]
perms []       = [[]]
perms (x : xs) = concat (map (insert x) (perms xs))
  where
    insert x []       = [[x]]
    insert x (y : ys) = (x : y : ys) : map (y:) (insert x ys)
```

The algorithm itself is broken into two parts. The *insert* function will return a list of lists, where x is inserted into ys at every possible position. For example: *insert* 1 $[2, 3]$ returns $[[1, 2, 3], [2, 1, 3], [2, 3, 1]]$. The *perms* function splits the list into a head x and tail xs. First, it computes all permutations of xs, then it will insert x into every possible position of every permutation.

While this algorithm is not terribly complex, it is really more complex than it needs to be. The problem is that we need to keep track of all of the permutations we generate. This does not seem like a big problem here. We just put each permutation in a list, and return the whole list of permutations. However, now every part of the program has to deal with the entire list of results. As our programs grow, we will need more data structures for this plumbing, and this problem will grow too. This is not new. Many languages have spent a lot of time trying to resolve this issue. In fact, several of Haskell's most successful concepts, such as monads, arrows, and lenses, are designed strictly to reduce this sort of plumbing.

We take a different approach in Curry. Instead of generating every possible permutation, and searching for the right one, we will non-deterministically generate a single permutation. This seems like a trivial difference, but its really quite substantial. We offload generating all of the possibilities onto the language itself.

We can simplify our code with the non-deterministic *choice* operator ?. Choice is defined by the rules:

```
x ? y = x
x ? y = y
```

Now our permutation example becomes a little easier. We only generate a single permutation, and when we insert *x* into *ys*, we only insert into a single arbitrary position.

```
perm          :: [a] → [a]
perm []       = []
perm (x : xs) = insert x (perm xs)
  where
    insert x []       = [x]
    insert x (y : ys) = x : y : ys ? y : insert x ys
```

In many cases functions that return multiple results can lead to much simpler code. Curry has another feature that is just as useful. We can declare a *free variable* in Curry. This is a variable that has not been assigned a value. We can then constrain the value of a variable later in the program. In the following example *begin*, *x*, and *end* are all free variables, but they are constrained by the guard so that *begin* ++ [*x*] ++ *end* is equal to *xs*. Our algorithm then becomes: pick an arbitrary *x* in the list, move it to the front, and permute the rest of the list.

```
perm    :: [a] → [a]
perm [] = []
perm xs
   | xs==(begin ++ [x] ++ end) = x : perm (begin ++ end)
  where begin, x, end free
```

Look at that. We have reduced the number of lines of code by 25%. In fact, this pattern of declaring free variables, and then immediately constraining them is used so often in Curry that we have syntactic sugar for it. A *functional pattern* is any pattern that contains a function that is not at the root.[1] We can use functional patterns to simplify our *perm* function even further.

```
perm                         :: [a] → [a]
perm []                      = []
perm (begin ++ [x] ++ end) = x : perm (begin ++ end)
```

Now the real work of our algorithm is a single line. Even better, it is easy to read what this line means. Decompose the list into *begin*, x, and *end*, then put x at the front, and permute *begin* and *end*. This is almost exactly how we would describe the algorithm in English.

There is one more important feature of Curry. We can let expressions fail. In fact we have already seen it, but a more explicit example would be helpful. We have shown how we can generate all permutations of a list by generating an arbitrary permutation, and letting the language take care of the exhaustive search. However, we usually do not need, or even want, every permutation. So, how do we filter out the permutations we do not want? The answer is surprisingly simple. We just let expressions fail. An expression fails if it cannot be reduced to a constructor form. The common example here is *head* [ ], but a more useful example might be sorting a list. We can build a sorting algorithm by permuting a list, and only keeping the permutation that is sorted.

```
sort :: (Ord a) ⇒ [a] → [a]
sort xs | sorted ys = ys
  where
    ys = perm xs
    sorted []           = True
    sorted [x]          = True
    sorted (x : y : ys) = x ⩽ y ∧ sorted (y : ys)
```

In this example every permutation of *xs* that is not sorted will fail in the guard. Once an expression has failed, computation on it stops, and other alternatives are tried. As we will see later on, this ability to conditionally execute a function will become crucial when developing optimizations.

These are some of the useful programming constructs in Curry. While they are convenient

[1] This is not completely correct. While the above code would fully evaluate the list, a functional pattern is allowed to be more lazy. Since the elements do not need to be checked for equality, they can be left unevaluated.

for programming, we need to understand how they work if we are going to implement them in a compiler.

2.2 SEMANTICS

As we have seen, the syntax of Curry is very similar to Haskell. Functions are declared by defining equations, and new data types are declared as algebraic data types. Function application is represented by juxtaposition, so $f\ x$ represents the function f applied to the variable x. Curry also allows for declaring new infix operators. In fact, Curry really only adds two new pieces of syntax to Haskell, **fcase** and **free**. However, the main difference between Curry and Haskell is not immediately clear from the syntax. Curry allows for overlapping rules and free variables. Specifically Curry programs are represented as *Limited Overlapping Inductively Sequential (LOIS)* Rewrite systems. These are is indicatively sequential systems with a single overlapping rule. On the other hand, Haskell programs are transformed into non-overlapping systems.

To see the difference consider the usual definition of factorial.

$$\mathit{fac} :: \mathit{Int} \to \mathit{Int}$$
$$\mathit{fac}\ 0 = 1$$
$$\mathit{fac}\ n = n * \mathit{fac}\ (n - 1)$$

This seems like an innocuous Haskell program, however It is non-terminating for every possible input for Curry. The reason is that *fac* 0 could match either rule. In Haskell all defining equations are ordered sequentially, which results in control flow similar to the following C implementation.

```
int fac(int n)
{
    if(n == 0)
    {
        return 1;
    }
    else
    {
        return n * fac(n-1);
    }
}
```

In fact, every rule with multiple defining equations follows this pattern. In the following equations let p_i be a pattern and E_i be an expression.

$$\begin{aligned}
&f\ p_1 = E_1\\
&f\ p_2 = E_2\\
&\dots\\
&f\ p_n = E_n
\end{aligned}$$

Then this is semantically equivalent to the following.

$$\begin{aligned}
&f\ p_1 &&= E_1\\
&f\ \mathit{not}\ p_1 \wedge p_2 &&= E_2\\
&\dots\\
&f\ \mathit{not}\ p_1 \wedge \mathit{not}\ p_2 \wedge \ldots \wedge p_n &&= E_n
\end{aligned}$$

Here *not* p_i means that we do not match pattern i. This ensures that we will only ever reduce to a single expression. Specifically we reduce to the first expression where we match the pattern.

Curry rules, on the other hand, are unordered. If we could match multiple patterns, such as in the case of *fac*, then we non-deterministically return both expressions. This means that *fac* 0 reduces to both 1 and *fac* (-1). Exactly how Curry reduces an expression non-deterministically will be discussed throughout this dissertation, but for now we can think in terms of sets. If the expression $e \to e_1$ and $e \to e_2$, $e_1 \to^* v_1$ and $e_2 \to^* v_2$, then $e \to^* \{v_1, v_2\}$.[2]

This addition of non-determinism can lead to problems if we we are not careful. Consider the following example:

$$\begin{aligned}
&\mathit{coin} = 0 \mathbin{?} 1\\
&\mathit{double}\ x = x + x
\end{aligned}$$

We would expect that for any x, *double* x should be an even number. However, if we were to rewrite *double coin* using ordinary term rewriting, then we could have the derivation.

$$\mathit{double}\ \mathit{coin} \Rightarrow \mathit{coin} + \mathit{coin} \Rightarrow (0 \mathbin{?} 1) + (0 \mathbin{?} 1) \Rightarrow 0 + (0 \mathbin{?} 1) \Rightarrow 0 + 1 \Rightarrow 1$$

[2] This should really be thought of as a multiset, since it is possible for v_1 and v_2 to be the same value.

This is clearly not the derivation we want. The problem here is that when we reduced *double coin*, we made a copy of the non-deterministic expression *coin*. This ability to clone non-deterministic expressions to get different answers is known as run-time choice semantics. [57].

The alternative to this is call-time choice semantics. When a non-deterministic expression is reduced, all instances of the expression take the same value. One way to enforce this is to represent expressions as graphs instead of terms. Since no expressions are ever duplicated, all instances of *coin* will reduce the same way. This issue of run-time choice semantics will appear throughout the compiler.

2.2.1 FlatCurry

The first step in the compiler pipeline is to parse a Curry program into FlatCurry. The definition is given in figure **??**. The FlatCurry language is the standard for representing Curry programs in compilers [20, 25, 26, 52], and has been used to define the semantics of Curry programs [4].

The semantics of Curry have already been studied extensively [4], so we informally recall some of the more important points. A FlatCurry program consists of datatype and function definitions. For simplicity we assume that all programs are self contained, because the module system is not relevant to our work. However, the Rice compiler does support modules. A FlatCurry function contains a single rule, which is responsible for pattern matching and rewriting an expression. Pattern matching is converted into case and choice expressions as defined in [4]. A function returns a new expression graph constructed out of **let** , **free**, f_k, C_k, ?, l, v expressions.

Our presentation of FlatCurry differs from [4] in three notable ways. First, function and constructor applications contain a count of the arguments they still need in order to be fully applied. The application $f_k\ e_1\ e_2 \dots e_n$ means that f is applied to n arguments, but it needs k more to be fully applied, so the arity of f is $n + k$. Second, we include **let** $\{v\}$ **free** to represent free variables. This was not needed in [4, 26] because free variables we translated to non-deterministic generators. Since we narrow free variables instead of doing this transformation, we must represent free variables in FlatCurry. Finally, we add an explicit failure expression $\perp$ to represent a branch that is not present in the definitional tree. While this is meant to simply represent a failing computation, we have also occasionally found it useful in optimization.

2.2.2 Evaluation

Each program contains a special function *main* that takes no arguments. The program executes by reducing the expression *main* to a *Constructor Normal Form*[3] as defined in figure **??**. Similar to Kics2, Pakcs, and Sprite, [20, 26, 52] we compute constructor normal form by first reducing the *main* to *Head Constructor Form*. That is where the expression is rooted by a constructor. Then each child of the root is reduced to constructor normal form.

Most of the work of evaluation is reducing an expression to head constructor form. Kics2 and Pakcs are able to transform FlatCurry programs into an equivalent rewrite system, and reduce expressions using graph rewriting [26, 52]. The transformation simply created a new function for every nested case expression. This created a series of tail calls for larger functions.

To see this transformation in action, we can examine the FlatCurry function == on lists **??**. This function is inductively sequential, however both Pakcs and Kics2 will transform it into a series of flat function calls with a single case at the root. Since this would drastically increase the number of function calls, we avoid this transformation. It would also defeat much of the purpose of an optimizing compiler if we were not allowed to inline functions.

2.2.3 Non-determinism

Currently there are three approaches to evaluating non-deterministic expression in Curry: *backtracking*, *Pull-Tabbing* [8], and *Bubbling* [9]. At this time there are no complete strategies for evaluating Curry programs, so we have elected to use backtracking. It is the simplest to implement, and it is well understood.

In our system, backtracking is implemented in the usual way. When an expression rooted by a node n with label by f is rewritten to an expression rooted by e, we push the rewrite $(n, n_f, \mathit{Continue})$ onto a backtracking stack, where n_f is a copy of the original node labeled by f. If the expression is labeled by a choice $e_1 \mathbin{?} e_2$, and it is rewritten to the left hand side e_1, then we push $(n, n_?, \mathit{Stop})$ onto the backtracking stack to denote that this was an alternative, and we should stop backtracking.

Unfortunately, while backtracking is well defined for rewriting systems, our representation of FlatCurry programs is not a graph rewrite system. This is because we do not flatten our

[3] This is constructor normal form, and not simply a normal form, because a failing expression, like *head* [], is a normal form, since it can not be rewritten, but it contains a function at the root.

FlatCurry functions like Pakcs and Kics2. As an example of why FlatCurry programs are not a graph rewriting system, consider the FlatCurry function *weird* **??**. This function defines a local variable x which is used in a case expression. If this were a rewrite system, then we would be able to translate the **case** expression into pattern matching, but a rule can not pattern match on a locally defined variable. We show the reduction of *wierd* in figure **??**.

We have entered an infinite loop of computing the same rewrite. The problem is that when we were backtracking, and replacing nodes with their original versions, we were going too far back in the computation. In this example, when backtracking *weird*, we want to backtrack to a point where x has been created, and we just want to evaluate the case again.

We solve this problem by creating a new function for each case expression in our original function. Figure **??** show an example for *weird* and == which were defined above. This is actually very similar to how Pakcs and Kics2 transformed their programs into rewrite systems by flattening them. The difference is that we do not need to make any extra function calls unless we are already backtracking. There is no efficiency cost in either time or space with our solution. The only cost is a little more complexity in the code generator, and an increase in the generated code size. This seems like an acceptable trade off, since our programs are still similar in size to equivalent programs compiled with GHC.

As far as we are aware, this is a novel approach for improving the efficiency of backtracking in rewriting systems. The correctness of this method follows from the redex contraction theorem, which is proved later.

2.2.4 Free Variables

Free variables are similar to non-deterministic expressions. In fact, in both Kics2 and Sprite [20, 26] they are replaced by non-deterministic generators of the appropriate type [11]. However, in Rice, free variables are instantiated by narrowing. If a free variable is the scrutinee of a case expression, then we push copies of the remaining patterns onto the stack along with another copy of the variable. If the free variable is replaced by a constructor with arguments, such as *Just*, then we instantiate the arguments with free variables.

This is easier to see with an example. Consider the traffic light function in figure **??**. The *change* function moves the light from *Red* to *Green* to *Yellow*. When calling this function with a free variable, we have the derivation below in figure **??**.

2.2.5 Higher Order Functions

Now that we have a plan for the logic features of Curry, we move on to higher order functions. This subject has been extensively studied by the function languages community, and we take the approach of [81]. Higher order functions are represented using defunctionalization [90]. Recall that in FlatCurry, an expression f_k represents a partial application that is missing k arguments. We introduce an *apply* function that has an unspecified arity, where *apply* $f_k\ e_1\ e_2 \dots e_n$ applies f_k to the arguments $e_1\ e_2 \dots e_n$.

The behavior of *apply* is specified below.

```
$apply\ f_k\ x_1 \dots x_n$
    $\mid k > n\ \ = f_{k-n}\ x_1 \dots x_n$
    $\mid k{=}{=}n = f\ x_1 \dots x_n$
    $\mid k < n\ \ = apply\ (f\ x_1 \dots x_k)\ x_{k+1} \dots x_n$
```

If the first argument f of *apply* is not partially applied, then evaluate f until it is, and proceed as above. In the case that f is a free variable, then we return $\bot$, because we do not support higher order narrowing.

2.2.6 Backtracking Performance

Now that we have established a method for implementing non-determinism, we would like to improve the performance. Currently we push nodes on the backtracking stack for every rewrite. Often, we do not need to push most of these rewrites. Consider the following code for computing Fibonacci numbers:

```
fib n = case n < 2 of
                    True  → n
                    False → fib (n − 1) + fib (n − 2)
main = case fib 20==(1 ? 6765) of
                    True → putStrLn "found answer"
```

This program will compute *fib* 20, pushing all of those rewrites onto the stack as it does, and then, when it discoverers that *fib* $20 \not\equiv 1$, it will undo all of those computations, only to redo them immediately afterwards! This is clearly not what we want. Since *fib* is a deterministic function, we would like to avoid pushing these rewrites onto the stack. Unfortunately, this is not as simple as it would first seems for two reasons. First, determining if a function is non-deterministic in general is undecidable, so any algorithm we developed would push rewrites for some deterministic computations. Second, a function may have a non-deterministic argument. For example, we could easily change the above program to:

$$\begin{aligned} &\mathit{main} = \textbf{case}\ \mathit{fib}\ (1\ ?\ 20){=}{=}6765\ \textbf{of} \\ &\qquad\qquad \mathit{True} \to \mathit{putStrLn}\ \texttt{"found answer"} \end{aligned}$$

Now the expression with *fib* is no longer deterministic. We sidestep the whole issue by noticing that while it is impossible to tell if an expression is non-deterministic at compile time, it is very easy to tell if it is at run time.

As far as we are aware, this is another novel solution. Each expression contains a Boolean flag that marks if it is non-deterministic. We called these *nondet* flags, and we refer to an expression whose root node is marked with a nondet flag as nondet. The rules for determining if an expression e is nondet are: if e is labeled by a choice, then e is nondet; if e is labeled by a function that has a case who is scrutinee is nondet, or is a forward to a nondet, then e is nondet; if $e' \to^* e$ and e' is nondet, then e is nondet.

Any node not marked as nondet does not need to be pushed on the stack because it is not part of a choice, all of its case statements scrutinized deterministic nodes, and it is not forwarding to a non-deterministic node. However proving this is a more substantial problem.

We prove this for the class of limited overlapping inductively sequential graph rewriting systems, with the understanding our system is equivalent. This proof is based on a corresponding proof for set functions in Curry [12][Lemma 2]. The original proof was concerned with a deterministic derivation from an expression to a value. While the idea is similar, we do not want to necessarily derive an expression to a value. Instead we define a deterministic redex, and deterministic step below, and show that there is an analogous theorem for a derivation of deterministic steps, even if it does not compute a value.

Definition 2.2.1. Given a rewrite system R with fixed strategy ϕ, a *computation space* [12] of expression e, $C(e)$ is finitely branching tree defined inductively the rule $C(e) = \langle e, C(e_1), C(e_2) \dots C(e_n)\rangle$.

We now need the notions of a deterministic redex and a deterministic rewrite. Ultimately we want to show that if we have a deterministic reduction, then we can perform that computation at any point without affecting the results. One implication of this would be that performing a deterministic computation before a non-deterministic choice was made would be the same as performing the computation after the choice. This would justify our fast backtracking scheme, because it would be equivalent to performing the computation before the choice was made.

Definition 2.2.2. A redex n in expression e is deterministic if there is at most one rewrite rule that could apply to $e|_n$. A rewrite $e \rightarrow_n e'$ is deterministic if n is a deterministic redex.

Next we rephrase our notion of nondet for a LOIS system.

Definition 2.2.3. let $e \rightarrow e_1 \rightarrow \ldots v$ be a derivation for e to v. A node n in e_i is *nondet* iff

1. n is labeled by a choice.
2. A node in the redex pattern [10] of n is *nondet.*
3. There exists some $j < i$ where n is a subexpression of e_j and n is *nondet.*

The first property is that all choice nodes are nondet. The second property is equivalent to the condition that any node that scrutinizes a nondet node should be nondet. Finally, the third property is that nondet should be a persistent attribute. This corresponds to the definition we gave for nondet nodes above.

If n is a redex that is not marked as nondet, then n ca not be labeled by a choice. Since choice is the only rule in a LOIS system that is non-deterministic, n must be a deterministic redex. We recall a theorem used to prove the correctness of set function. [12][Def. 1, Lemma 1]

Lemma 4. *Given an expression e where $e \rightarrow_{n_1} e_1$ and $e \rightarrow_{n_2} e_2$, if $n_1 \neq n_2$, then there exists a u_1 and u_2 where $t_1 \rightarrow^= u_1$ and $t_2 \rightarrow^= u_2$ and $u_1 = u_2$ up to renaming of nodes.*

This leads directly to our first important theorem. If n is a deterministic redex in a derivation, then we can move it earlier in the derivation.

Theorem 5 (Redex Compression Theorem)**.** *if n is a deterministic redex of e where $n \rightarrow n'$, and $e \rightarrow e_1 \rightarrow_n e_2$. Then there exists a derivation $e[n \rightarrow n'] \rightarrow^= e'$ where $e_2 = e'$ up to renaming of nodes.*

Proof. By definition of rewriting $e \rightarrow_n e[n \rightarrow n']$. Since n is a deterministic redex, it must be the case that the redex in $e \rightarrow e_1$ was not n. So by the previous lemma, we can swap the order of the rewrites. □

Finally we show that if a is a subexpression of e and $a \rightarrow^* b$ using only deterministic redexes, then $e[a \rightarrow b]$ rewrites to the same values.

Theorem 6 (Path Compression Theorem)**.** *if a is a subexpression of e and $a \to^* b$ using only deterministic rewrites, and $e \to e_1 \to \dots e_n$ is a derivation where b is a subexpression of e_n, then there is a derivation $e[a \to b] \to^* e_n$.*

Proof. This follows by induction on the length of the derivation. In the base case $a = b$, and there is nothing to prove. In the inductive case $a \to_p a' \to^* b$. Since $a \to_p a'$ is deterministic by assumption, we can apply the path compression theorem and say that $e[a \to a'] \to^* e_n$. By the inductive hypotheses we can say that $e[a \to a'][a' \to b] \to^* e_n$. Therefore $e[a \to b] \to^* e_n$. This establishes our result. □

2.2.7 Collapsing Functions

While this result is great, and it allows us to avoid creating a large number of stack frames, there is a subtle aspect of graph rewriting that gets in the way. If a node n_1 labeled by function f is rewritten to n_2, then the definition of applying a rewrite rule [35][Def. 8, Def. 10, Def. 19] would require us to traverse the graph, and find every node that has n_1 as a child, and redirect that pointer to n_2. This is clearly inefficient, so this is not done in practice. A much faster method is to simply replace the contents, the label and children, of n_1 with the contents of n_2. This works most of the time, but we run into a problem when a function can rewrite to a single variable, such as the *id* function. We call these functions *collapsing functions*. One option to solve this problem is to evaluate the contractum to head constructor form, and copy the constructor to the root [65]. This is commonly used in lazy functional languages, however it does not work for Curry programs. Consider the expression following expression.

```
f = let x = True ? False
      y = id x
      in not y
```

When y is evaluated, then it will evaluate x, and x will evaluate to *True*. If we then copy the *True* constructor to y, then we have two copies of *True*. But, since y is deterministic, we do not need to undo y when backtracking. So, y will remain *True* after backtracking, instead of returning to *id x*. While constructor copying is definitely invalid with fast backtracking, it is unclear if it would be valid with a normal backtracking algorithm.

We can solve this problem by using forwarding nodes, sometimes called indirection nodes [86]. The idea is that when we rewrite an expression rooted by a collapsing function, instead of copying the constructor, we just replace the root with a special forwarding node, $FORWARD(x)$, where x is the variable that the function collapses to.

There is one more possibility to address before we move on. One performance optimization with forwarding nodes is *path compression*. If we have a chain of forwarding nodes $FORWARD(FORWARD(FORWARD($ we want to collapse this to simply $FORWARD(x)$. This is unequivocally invalid in non-deterministic backtracking systems. Consider the following function.

```
f = let x = True ? False
        y = id x
    in  case y of
          False → case x of
                    False → ()
```

When reducing this function, we create two forwarding nodes that are represented by the variables x and y. We refer to these nodes as $FORWARD_x$ and $FORWARD_y$ respectively. So x is reduced to $FORWARD_x(True)$, and y is reduced to $FORWARD_y(FORWARD_x(True))$. If we contract y to $FORWARD_y(True)$, then when we backtrack we replace x with $FORWARD_x(False)$, and y is replaced with $id(True)$. The reason that y does not change to $id(False)$ is because y has lost its reference to x. Now, not only do we fail to find a solution for f, we have ended up in a state where x and y have different values.

In this chapter we have discussed the Curry language, and overviewed the semantics of Curry programs. We have shown different approaches to implementing a system for running Curry programs, and we have discussed the choices that we made. When a decision needed to be made, we prioritized correctness, then efficient execution, and then ease of implementation. In the next chapter we discuss the implementation at a low level. This will give us an idea of what the code we want to generate should look like.

Chapter 3

THE GENERATED CODE

Now that we have examined all of the different choices to make in constructing a compiler, we can start to design the generated code and runtime system for the compiler. In this chapter we give examples of generated code to implement Curry functions, and discuss the low level details of the Rice runtime. We start with a first order deterministic subset of Curry, then we add higher order function, finally we add non-determinism and free variables. Throughout this section we will use `teletype font` to represent generated C code to distinguish it from Curry or FlatCurry code.

We will introduce the generated code by looking at the *not* function defined below. We choose this function, because it is small enough to be understandable, but it still demonstrates most of the decisions in designing the generated code and runtime system.

$$\begin{aligned} &\mathit{not}\ x = \textbf{case}\ x\ \textbf{of} \\ &\qquad\qquad \mathit{False} \rightarrow \mathit{True} \\ &\qquad\qquad \mathit{True}\ \ \rightarrow \mathit{False} \end{aligned}$$

Before we discuss generated code, we need to discuss expressions and the runtime system for programs.

When a FlatCurry module is compiled, it is translated into a C program. Every function f defined in the FlatCurry module is compiled into a C function that can reduce an expression, rooted by a node labeled with f, into head constructor form. These functions are called `f_hnf` for historical reasons [52].

An expression in our compiled code is a rooted labeled graph. nodes in the graph are given the definition in figure **??**.

A `field` is a union of a `Node*` and the representations of the primitive types `Int`, `Float`, and `Char`, as well as a `field*` to be described shortly. The use of fields instead of nodes for the children will be justified when we discuss primitive values and unboxing in chapter 7 The `children` field contains an array of children for this node. If a node could have more than three children, such as a node representing the (, , , , , ,) constructor, then `children[3]` holds a pointer to a variable length array that holds the rest of the children. This leads to non-uniform indexing into nodes. For example `n->children[1]` returns the second child of the node, but the sixth child must be retrieved with `n->children[3].a[2]`. We use a `child_at` macro to simplify the code, so `child_at(n,5)` returns the sixth child. The `symbol` field is a pointer to the static information of the node. This includes the name, arity, and `tag` for the node, as well as a function pointer responsible for reducing the node to head constructor form. We include a `TAG` macro to access the tag of a node. This is purely for convenience. For a node labeled by function *f*, this is a pointer to `f_hnf`. Because the calling convention is complicated, we hide this detail with an `HNF` macro, so `HNF(f)` evaluated the node labeled by *f* to head constructor form. The `missing` field represents a partial function application. If `missing` is greater than 0, then `f` is partially applied. The `nondet` field represents the nondet marker described in the fast backtracking algorithm.

Each function and constructor generates a `set` and `make` function. For the *not* function, we would generate

```
void set_not(field root, field x);
field make_not(field x);
```

The `set_not` function sets the `root` parameter to be a *not* node. This is accomplished by changing the symbol and children for `root`. The `make_not` function allocates memory for a new *not* node.

Each program in our language defines an expression *main*, and runs until *main* is evaluated to constructor normal form. This evaluation is broken up into two pieces. The primary driver of a program is the `nf` function, which is responsible for evaluating the main expression to constructor normal form. The `nf` function computes this form by first evaluating an expression to head constructor form. When an expression is in head constructor form, `nf` evaluates each subexpression to constructor normal form, producing the loop in figure **??**.

All that is missing here is the `hnf` functions. We give a simplified version of the `not_hnf` function in **??**, and we will fill in details as we progress.

We can see that the main driver of this function is the `while(true)` loop. The loop looks up the tag of `x`, and if it is a function tag, when we evaluate it to head constructor form. If the tag for `x` is `FAIL`, which represents an exempt node, then we set the root to `FAIL` and return. If the tag is `Prelude_True` or `Prelude_False`, we set the root to the corresponding expression, and return from the loop. Finally, in order to implement collapsing functions, we introduce a `FORWARD` tag. If the tag is `FORWARD`, then we traverse the forwarding chain, and continue evaluating the `x`.

Finally, while we are evaluating the node stored in the local variable `x`, we introduce a new variable `scrutenee`. This is because if `x` evaluates to a forwarding node, we need to evaluate the child of `x`. If we were to update `x`, and then return an expression containing `x` later, then we would have compressed the forwarding path. As mentioned previously, this is not valid.

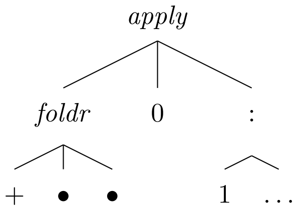
At this point we have a strategy for how to compile first order deterministic Curry functions. Next we show how we handle partial application and higher order functions.

3.0.1 Higher Order Functions

Earlier we gave an interpretation of how to handle *apply* nodes, but there are still a few details to work out. Recall the semantics we gave for apply nodes:

$$\begin{array}{l}
\mathit{apply}\ f_k\ [x_1, \ldots x_n] \\
\quad \mid k > n \ \ = f_{k-n}\ x_1 \ldots x_n \\
\quad \mid k{=}{=}n = f\ \ x_1 \ldots x_n \\
\quad \mid k < n \ \ = \mathit{apply}\ (f\ \ x_1 \ldots x_k)\ [x_{k+1}, \ldots x_n]
\end{array}$$

If f is missing any arguments, then we call f a partial application. Let us look at a concrete example. In the expression $foldr_2\ (+_2)$, *foldr* is a partial application that is missing 2 arguments. We will write this as *foldr* $(+_2) \bullet \bullet$ where $\bullet$ denotes a missing argument. Now, suppose that we want to apply the following expression.



Remember that each node represents either a function or a constructor, and each node has a fixed arity. For example, + has an arity of 2, and *foldr* has an arity of 3. This is true for every + or *foldr* node we encounter. However, it is not true for *apply* nodes. In fact, an *apply* node may have any positive arity. Furthermore, by definition, an *apply* node can not be missing any arguments. For this reason, we use the `missing` field to hold the number of arguments the node is applied to.[1]

The algorithm for reducing apply nodes is straightforward, but brittle. There are several easy mistakes to make here. The major problem with function application is getting the arguments in the correct positions. To help alleviate this problem we make a non-obvious change to the structure of nodes. We store the arguments in reverse order. To see why this is helpful, let us consider the *foldr* example above. But this time, decompose it into 3 apply nodes, so we have $\mathit{apply}\ (\mathit{apply}\ (\mathit{apply}\ foldr_3\ (+_2))\ 0)\ [1,2,3]$. In our innermost apply node, which will be evaluated first, we apply $foldr_3$ to $+_2$ to get $foldr_2\ (+_2) \bullet\bullet$. This is straightforward. We simply put + as the first child. However, when we apply $foldr_2\ (+_2) \bullet \bullet$ to 0, we need to put 0 in the second child slot. In general, when we apply an arbitrary partial application f to x, what child do we put x in? Well, if we are storing the arguments in reverse order, then we get a really handy result. Given function f_k that is missing k arguments, then $\mathit{apply}\ f_k\ x$ reduces to $f_{k-1}\ x$ where x is the $k-1$ child. The missing value for a function tells us exactly where to put the arguments. This is completely independent of the arity of the function.

$$\begin{aligned}
&\mathit{apply}\ (\mathit{apply}\ (\mathit{apply}\ (foldr_3 \bullet \bullet\bullet)\ (+_2))\ 0)\ [1,2,3] \\
&\Rightarrow \mathit{apply}\ (\mathit{apply}\ (foldr_2 \bullet \bullet (+_2))\ 0)\ [1,2,3] \\
&\Rightarrow \mathit{apply}\ (foldr_1 \bullet 0\ (+_2))\ [1,2,3] \\
&\Rightarrow foldr_0\ [1,2,3]\ 0\ (+_2))
\end{aligned}$$

The algorithm is given in figure **??**. There are a few more complications to point out. To avoid complications, we assume arguments that a function is being applied to are stored in the array at `children[3]` of the apply node. That gives us the structure $\mathit{apply}\ f \bullet\bullet a_n \ldots a_1$. This is not done in the runtime system because it would be inefficient, but it simplifies the code for this presentation. We also make use of the `set_child_at` macro, which simplifies setting child nodes and is similar to `child_at`. Finally, the loop to put the partial function in head constructor form uses `while(f.n->missing <= 0)` instead of `while(true)`. This is because our normal form is a partial application, which does not have its own tag.

We reduce an apply node in two steps. First get the function `f`, which is the first child of an apply node. Then, reduce it to a partial application. If `f` came from a non-deterministic expression, the save the apply node on the stack. We split the second step into two cases. If `f`

[1] In reality we set missing to the negative value of the arity to distinguish an apply node from a partial application.

is under applied, or has exactly the right number of arguments, then copy the contents of f into the root, and move the arguments over and reduce. If f is over applied, then make a new copy of f, and copy arguments into it until it is fully applied. Finally we reduce the fully applied copy of f and apply the rest of the arguments.

3.0.2 Implementing Non-determinism

Now, we that we can reduce a higher order functional language, we would like to extend our implementation to handle features from logic languages.

The implementation does not change too much. First we add two new tags CHOICE and FREE to represent non-deterministic choice and free variable nodes respectively. The choice nodes are treated in a similar manner to a function. We call the choose function to reduce a choice to HCF, and push the alternative on the stack.

The choose function in ?? reduces a choice node to head constructor form. Since choice is a collapsing rule, we return a forwarding node. The function is also responsible for keeping track of which branch of the choice we should reduce, and pushing the alternative on the backtracking stack. We accomplish this by keeping a marker in the second child of a choice node. This marker is 0 if we should reduce to the left hand side, and 1 if we should reduce to the right hand side.

Free variables are more interesting. To narrow a free variable we pick a possible constructor, and replace the scrutinee node with that constructor. All arguments to the constructor are instantiated with free variables. Then, we push a rewrite on the stack to replace scrutinee with a free variable using the push_frame function. This is because after each possible choice has been exhausted, we want to reset this node back to a free variable in case it is used in another non-deterministic branch of the computation. Finally, for every other constructor, we push an alternative on the backtracking stack using the push_choice function.

The only other necessary change is to push a rewrite onto the backtracking stack when we reach either a fail, or constructor case. The Prelude_not_1 function is a function at a case expression discussed in section 2.2.3. The changes to the *not* function are give in figure ??. Due to space constraints not all sections are show. The pieces of code that do not changed are omitted and replaced with ....

3.0.3 Fast Backtracking

Finally we show how we implement the fast backtracking technique described earlier. The implementation actually does not change much, we simply make use of the `nondet` flag in each node. While we are evaluating `scrutinee`, we keep track of whether or not we have seen a non-deterministic node in a local variable, and if we have, we push the root on the backtracking stack. If we have not seen a non-deterministic node, then we can simply avoid pushing this rewrite. The generated code for *not* is given in figure **??**.

In the last two chapters we have discussed the choices we have made with our generated code, and given an idea with what the generated code should look like. In some sense, we have given a recipe of how to translate Curry into C. In the next chapter we introduce the tools to make this recipe. We introduce a system for implementing transformations as rewrite rules. We then show how this system can simplify the construction of a compiler, and use it to transform FlatCurry programs into a form that is easier to optimize and compile to C.

Chapter 4

GENERATING AND ALTERING SUBEXPRESSIONS

In this chapter we introduce our engine for Generating and Altering Subexpressions, of the GAS system. This system proves to be incredibly versatile and is the main workhorse of the compiler and optimizer. We show how to construct, combine, and improve the efficiency of transformations, as well as how the system in implemented.

4.1 BUILDING OPTIMIZATIONS

Throughout this dissertation we look at the process of developing compiler optimizations. For our purposes we are concerned with *compile time optimizations*. These are transformations on a program, performed at compile time, that are intended to produce more efficient code. Most research in the Curry community has been done on *run time optimizations* , which are improvements to the evaluation of Curry programs. This can include the development of new rewriting strategies, or improvements to pull-tabbing and bubbling [16,20]. These improvements are important, but they are not our concern for this compiler.

Developing compile time optimizations is usually considered the most difficult aspect of writing a modern compiler. It is easy to see why. There are dozens of small optimizations to make, and each one needs to be written, shown correct, and tested.

Furthermore, there are several levels where an optimization can be applied. Some optimizations apply to a programs AST, some to another intermediate representation, some to the generated code, and even some to the runtime system. There are even optimizations that are applied during transformations between representations. For this chapter, we will be describing a system to apply optimizations to FlatCurry programs. While this is not the only area of the compiler where we applied optimizations, it is by far the most extensive, so it is worth understanding how our optimization engine works.

Generally speaking, most optimizations have the same structure. Find an area in the AST where the optimization applies, and then replace it with the optimized version. As an example, consider the code for the absolute value function defined below.

$$\begin{aligned}
&\mathit{abs}\ x\\
&\quad \mid x < 0 \qquad\quad = -x\\
&\quad \mid \mathit{otherwise} = x
\end{aligned}$$

This will be translated into FlatCurry as

$$\begin{aligned}
\mathit{abs}\ x = \textbf{case}\ (x < 0)\ \textbf{of}&\\
&\mathit{True}\ \ \to -x\\
&\mathit{False} \to \textbf{case}\ \mathit{otherwise}\ \textbf{of}\\
&\qquad\qquad\quad \mathit{True}\ \ \to x\\
&\qquad\qquad\quad \mathit{False} \to \bot
\end{aligned}$$

While this transformation is obviously inefficient, it is general and has a straightforward implementation. A good optimizer should be able to recognize that *otherwise* reduces to *True*, and reduce the case-expression. So for this one example, we have two different optimizations we need to implement. We need to reduce *otherwise* to *True*, then we can reduce the second case expression to x.

There are two common approaches to solving this problem. The first is to make a separate function for each optimization. Each function will traverse the AST and try to apply its optimization. The second option is to make a few large functions that attempt to apply several optimizations at once. There are trade-offs for each.

The first option has the advantage that each optimization is easy to write and understand. However, is suffers from a lot of code duplication, and it is not very efficient. We must traverse the entire AST every time we want to apply an optimization. Both LLVM and the JVM fall into this category [66, 79]. The second option is more efficient, and there is less code duplication, but it leads to large functions that are difficult to maintain or extend.

Using these two options generally leads to optimizers that are difficult to maintain. To combat this problem, many compilers will provide a language to describe optimization transformation. Then the compiler writer can use this domain specific language to develop their optimizations. With the optimization descriptions, the compiler can search the AST of a program to find any places where optimizations apply. However, it is difficult or impossible to write many common optimizations in this style [85].

The aim of our solution is to try to get the best of all three worlds. We have developed an approach to simplify Generating and Altering Subexpressions (GAS) . Our approach was to do optimization entirely by rewriting. This has several advantages, and might be the most useful result of this work. First, developing new optimizations is simple. We can write down new optimizations in this system within minutes. It was often easier to write down the optimization and test it, than it was to try to describe the optimization in English. Second, any performance improvement we made to the optimization engine would apply to every optimization. Third, optimizations were easy to maintain and extend. If one optimization did not work, we could look at it and test it in isolation. Fourth, this code is much smaller than a traditional optimizer. This is not really a fair comparison given the relative immaturity of our compiler, but we were able to implement 16 optimizations and code transformations in under 150 lines of code. This gives a sense of scale of how much easier it is to implement optimizations in this system. Fifth, Since We are optimizing by rewrite rules, the compiler can easily output what rule was used, and the position where it was used. This is enough information to entirely reconstruct the optimization derivation. We found this very helpful in debugging. Finally, optimizations are written in Curry. We did not need to develop a DSL to describe the optimizations, and there are no new ideas for programmers to learn if they want to extend the compiler.

We should note that there are some potential disadvantages to the GAS system as well. The first disadvantage is that there are some optimizations and transformations that are not easily described by rewriting. The second is that, while we have improved the efficiency of the algorithm considerably, it still takes longer to optimize programs then we would like.

The first problem is not really a problem at all. If there is an optimization that does not lend itself well to rewriting, we can always write it as a traditional optimization. Furthermore, as we will see later, we do not have to stay strictly in the bounds of rewriting. The second problem is actually more fundamental to Curry. Our implementation relies on finding a single value from a set generated by a non-deterministic function. Current implementations are inefficient, but there are new implementations being developed [1]. We also believe that an optimizing compiler would help with this problem [69].

4.1.1 The Structure of an Optimization

The goal with GAS is to make optimizations simple to implement and easily readable. While this is a challenging problem, we can actually leverage Curry here. Remember that the semantics of Curry are already non-deterministic rewriting.

Each optimization is going to be a function from a FlatCurry expression to another FlatCurry expression.

$$\textbf{type } \mathit{Opt} = \mathit{Expr} \to \mathit{Expr}$$

For readability we use the FlatCurry syntax defined in figure **??**, While this version of FlatCurry is easier to read, we will need the actual representation of FlatCurry programs to implement the compiler. This representation is given in figure **??**, and the transformation from the FlatCurry syntax to the FlatCurry representation is given in figure **??**. We can describe an optimization by simply describing what it does to each expression. As an example consider the definition for floating let-expressions:

$$\mathit{float}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [\mathit{Let}\ \mathit{vs}\ e] +\!\!+ \mathit{bs})) = \mathit{Let}\ \mathit{vs}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [e] +\!\!+ \mathit{bs}))$$

This optimization tells us that, if an argument to a function application is a **let** expression, then we can move the let-expression outside. This works for let-expressions, but what if there is a free variable declaration inside of a function? We can just define that case with another rule.

$$\begin{aligned}
&\mathit{float}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [\mathit{Let}\ \mathit{vs}\ e] +\!\!+ \mathit{bs})) &&= \mathit{Let}\ \mathit{vs}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [e] +\!\!+ \mathit{bs}))\\
&\mathit{float}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [\mathit{Free}\ \mathit{vs}\ e] +\!\!+ \mathit{bs})) &&= \mathit{Free}\ \mathit{vs}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [e] +\!\!+ \mathit{bs}))
\end{aligned}$$

This is where the non-determinism comes in. Suppose we have an expression:

$$f\ (\textbf{let}\ x = 1\ \textbf{in}\ x)\ (\textbf{let}\ r\ \textbf{free in}\ 2)$$

This could be matched by either rule. The trick is that we do not care which rule matches, as long as they both do eventually. This will be transformed into one of the following:

$$\begin{aligned}
&\textbf{let}\ r\ \textbf{free in let}\ x = 1\ \textbf{in}\ f\ x\ 2\\
&\textbf{let}\ x = 1\ \textbf{in let}\ r\ \textbf{free in}\ f\ x\ 2
\end{aligned}$$

Either of these options is acceptable. In fact, we could remove the ambiguity by making our rules a confluent system, as shown by the code below. However, we will not worry about confluence for most optimizations.

$$\begin{aligned}
&\mathit{float}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [\mathit{Let}\ \mathit{vs}\ e] +\!\!+ \mathit{bs})) &&= \mathit{Let}\ \mathit{vs}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [e] +\!\!+ \mathit{bs}))\\
&\mathit{float}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [\mathit{Free}\ \mathit{vs}\ e] +\!\!+ \mathit{bs})) &&= \mathit{Free}\ \mathit{vs}\ (\mathit{Comb}\ \mathit{ct}\ f\ (\mathit{as} +\!\!+ [e] +\!\!+ \mathit{bs}))\\
&\mathit{float}\ (\mathit{Let}\ \mathit{vs}\ (\mathit{Free}\ \mathit{ws}\ e)) &&= \mathit{Free}\ \mathit{ws}\ (\mathit{Let}\ \mathit{vs}\ e)
\end{aligned}$$

Great, now we can make an optimization. It was easy to write, but it is not a very complex optimization. In fact, most optimizations we write will not be very complex. The power of optimization comes from making small improvements several times.

Now that we can do simple examples, let us look at a more substantial transformation. Let-expressions are deceptively complicated. They allow us to make arbitrarily complex, mutually recursive, definitions. However, most of the time a large let expression could be broken down into several small let expressions. Consider the definition below:

```
let a = b
    b = c
    c = d + e
    d = b
    e = 1
in  a
```

This is a perfectly valid definition, but we could also break it up into the three nested let expressions below.

```
let e = 1
in  let b = c
        c = d + e
        d = b
    in  let a = b
        in  a
```

It is debatable which version is better coding style, but the second version is inarguably more useful for the compiler. There are several optimizations that can be safely performed on a single let bound variable. Unfortunately, splitting the let expression into blocks is not trivial. The algorithm involves making a graph out of all references in the let block, then finding the strongly connected components of that reference graph, and, finally, rebuilding the let expression from the component graph. The full algorithm is given below in figure **??**

While this optimization is significantly more complicated then the *float* example, We can still implement it in our system. Furthermore, we are able to factor out the code for building the graph and finding the strongly connected components. This is the advantage of using Curry functions as opposed to strict rewrite rules. We have much more freedom in constructing the right-hand side of our rules.

Now that we can create optimizations, what if we want both *blocks* and *float* to be able to run? This is an important part of the compilation process to get expressions into a canonical form. It turns out that combining two optimizations is simple. We just make a non-deterministic choice between them.

$$floatBlocks = float \mathbin{?} blocks$$

This is a new optimization that will apply either *float* or *blocks*. The ability to compose optimizations with ? is the heart of the GAS system. Each optimization can be developed and tested in isolation, then they can be combined for efficiency.

4.1.2 An Initial Attempt

Our first attempt is quite simple, really. We pick an arbitrary subexpression with *subExpr* and apply an optimization. We can then use a non-deterministic fix point operator to find all transformations that can be applied to the current expression. We can define the non-deterministic fix point operator using either the Findall library, or Set Function [12, 27]. The full code is given in figure **??**.

While this attempt is short and readable, there is a problem with it. It is unusably slow. While looking at the code, it is pretty clear to see what the problem is. Every time we traverse the expression, we can only apply a single transformation. This means that if we need to apply 100 transformations, which is not uncommon, then we need to traverse the expression 100 times.

4.1.3 A Second Attempt: Multiple Transformations Per Pass

Our second attempt runs much faster. Instead of picking an arbitrary subexpression, we choose to traverse the expression manually. Now, we can check at each node if an optimization applies. We only need to make two changes. The biggest is that we eliminate *subExpr* and change *reduce* to traverse the entire expression. Now *reduce* can apply an optimization at every step. We have also made *reduce* completely deterministic. The second change is that since *reduce* is deterministic, we can change *fix* to be a more traditional implementation of a fix point operator. The new implementation is given in figure **??**

This approach is significantly better. Aside from applying multiple rules in one pass, we also limit our search space when applying our optimizations. While there is still more we can do, the new approach makes the GAS library usable on larger Curry programs, like the standard Prelude.

4.1.4 Adding More Information

Rather surprisingly our current approach is actually sufficient for compiling FlatCurry. However, to optimize Curry we are going to need more information when we apply a transformation. Specifically, we will be able to create new variables. To simplify optimizations, we will require that each variable name can only be used once. Regardless, we need a way to know what is a safe variable name that we are allowed to use. We may also need to know if we are rewriting the root of an expression. Fortunately, all we need to change is to define *Opt* to accept more parameters. For each optimization, we will pass in an $n :: Int$ that represents the next variable v_n that is guaranteed to be fresh. We will also pass in a $top :: Bool$ that tells us if we are at root of a function we are optimizing. We also return a pair of $(Expr, Int)$ to denote the optimized expression, and the number of new variables we used.

$$\textbf{type}\ Opt = (Int, Bool) \rightarrow Expr \rightarrow (Expr, Int)$$

If we later decide that we want to add more information, then we just update the first parameter. The only problem is, how do we make sure we are calling each optimization with the correct n and *top*? We just need to update *reduce* and *runOpt*. In order to keep track of the next available free variable we use the *State* monad. We do need to make minor changes to *fix* and *simplify*, but this is just to make them compatible with *State*. The full implementation is in figure **??**.

4.1.5 Reconstruction

Right now we have everything we need to write all of our optimizations. However, we've found it useful to be able to track which optimizations were applied and where they were applied. This helps with testing, debugging, and designing optimizations, as well as generating optimization derivations that we will see later in this dissertation. It is difficult to overstate just how helpful this addition was in building this compiler.

If we want to add this, then we need to make a few changes. First, we need to decide on a representation for a rewrite derivation. Traditionally a rewrite derivation is a sequence of rewrite steps, where each step contains the rule and position of the rewrite. We describe paths in figure **??**. To make reconstruction easier, we also include the expression that is the result of the rewrite. This gives us the type:

```
type Path = [Int]
type Step = (String, Path, Expr)
type Derivation = [Step]
```

This leads to the last change we need to make to our *Opt* type. We need each optimization to also tell us its name. This is good practice in general, because it forces us to come up with unique names for each optimization.

```
type Opt = (Int, Bool) → Expr → (Expr, String, Int)
```

We only need to make a few change to the algorithm. Instead of using the *State* monad, we use a combination of the *State* and *Writer* monads, so we can keep track of the derivation. We have elected to call this the *ReWriter* monad. We add a function *update* :: *Expr* → *Step* → *Int* → *ReWriter Expr* that is similar to *put* from *State*. This updates the state variable, and creates a single step. The *reduce* function requires few changes. We change the Boolean variable *top* to a more general *Path*. Because of this change, we need to add the correct subexpression position, instead of just changing *top* to *False*. The *RunOpts* function is similar. We just change *top* to a *Path*, and check if it is null. Finally *fix* and *simplify* are modified to remember the rewrite steps we have already computed. We change the return type of *simplify* so that we have the list of steps. The full implementation is in figure **??**

Now that we have computed the rewrite steps, it is a simple process to reconstruct them into a string. The *pPrint* function comes from the FlatCurry Pretty Printing Library.

```
reconstruct :: Expr → [Step] → String
reconstruct _ [] = ""
reconstruct e ((rule, p, rhs) : steps) = let e' = e[p → rhs]
                                         in "=>_" ++ rule ++ "  " ++ (show p) ++ "\n" ++
                                            pPrint e' ++ "\n" ++
                                            reconstruct e' steps
```

Now that our optimization engine is running and printing out optimization derivations, there are a few tricks we can use to improve the efficiency. Remember that our optimizing engine is going to run for every optimization, so it is worth taking the time to tune it to be as efficient as possible. The first trick is really simple. We add a Boolean variable *seen* to the ReWriter monad. This variable starts as *False*, and we set it to *True* if we apply any optimization. This avoids the linear time check for every call of *fix* to see if we actually ran any optimizations. The second quick optimization is to notice that variables, literals, and type expressions are never going to run an optimization, so we can immediately return in each of those cases without calling *runOpt*. This is actually a much bigger deal than it might first appear. All of the leaves are going to either be variables, literals, or constructors applied to no arguments. For expression trees the leaves are often the majority of the nodes in the tree. Finally, we can put a limit on the number of optimizations to apply. If we ever reach that number, then we can immediately return. This can stop our optimizer from taking too much time.

Now that the GAS system is in place, we can move onto compiling FlatCurry programs. In This chapter we have introduced the GAS system that allows us to represent transformations In a simple form that is easy to extend and test. We have seen how we can represent an optimization as a function from expressions to expressions. Then we showed that we can extend this idea to create more powerful optimizations, and automatically generate optimization derivations. In the next chapter we put this system to work. We use the GAS system to implement several transformations to turn FlatCurry code in to a form that can be more easily compiled. Then we show how to generate efficient C code for FlatCurry programs.

Chapter 5

THE COMPILER PIPELINE

In the last chapter we developed the GAS system for representing transformation. In this chapter we show an extended example of using the GAS system to transform FlatCurry programs into a canonical form. We then show how to translate these canonical progras to the ICurry intermediate representation. Finally, we compile the ICurry progrms to C code, as discussed in chapter 3.

This compiler, unsurprisingly, follows a traditional compiler pipeline. While we start with an AST, there are still five phases left before we can generate C code. First, we normalize FlatCurry to a canonical form. Second, we optimize the FlatCurry. Third, we sanitize the FlatCurry to simplify the process of generating C code. Fourth, we compile the FlatCurry to ICurry, an intermediate representation that is closer to C. Finally, we compile the ICurry to C. These steps are referred to as pre-process, optimize, post-process, toICurry, and toC within the compiler [67].

We give an example of a function as it passes through each of the stages of the compiler in figure **??**. After pre-processing, the let expression has been floated to the top, and the missing branch has been filled in. After optimization, code is organized into blocks, and functions have been reduced. After post processing, let bound variables with a case expression have been factored out into their own functions. At this point the code is ready to be translated into ICurry and then C.

While there are several small details that are important to constructing a working Curry compiler, we will concern ourselves with the big picture here.

5.1 CANONICAL FLATCURRY

The pre-process and post-process steps of the compiler make heavy use the of GAS system, and transform the FlatCurry program in to a form that is more amenable to C, including removing case and let expression from inside function applications. We will discuss the optimization phase in the next section, but for now we can see how transformations work.

Let us start with an example:

$1 + \textbf{let } x = 3 \textbf{ in } x$

This is a perfectly fine Curry program, but C does not allow variable declarations in an expression, so we need to rewrite this Curry expression to:

$\textbf{let } x = 3 \textbf{ in } 1 + x$

We do not reduce **let** $x = 3$ **in** x yet, because that would be an optimization. However, this will be reduced later. We can translate the new expression to C in a direct manner. This is the purpose of the pre-process and post-process steps. We rewrite a Curry expression that does not make sense in C to an equivalent Curry expression that we can translate directly to C. Most of the transformations consist of disallowing certain syntactic constructs. Canonical FlatCurry is defined in figure **??**.

Examples of the pre-processing transformations are presented in figures **??** and **??**. We use the symbol $\Rrightarrow$ for the optimization relation. The implementation is presented in figure **??**. We only show the initial implementation of an optimization that excludes the name and path, but it can be extended to the full optimization in a straightforward manner. The full implementation can be found in the `src/Optimize/Preprocess.curry` file at [67].

In practice several of these rules are generalized and optimized. For example let-expressions may have many mutually recursive variables, and when floating a let bound variable inward, we may want to recursively traverse the expression to find the innermost declaration possible. However, these extensions to the rules are also included in the repository [67].

While most of these transformations are simple, a few require some explanation. The **blocks** transformation takes a let block with multiple variable definitions, and rewrites it to a series of let blocks where all variables are split into strongly connected components. This is not strictly necessary, but it removes the need to check for mutual recursion during the optimization phase. It will often transform a block of mutually defined variables into a cascading series of let expressions with a single variable, which will allow more optimizations to run throughout the compiler.

The **alias** transformation will remove any aliased variables. If one variables is aliased to another, then it will do the substitution, but if a variable is aliased to itself, then it cannot be reduced to a normal form, so we can replace it with an infinite loop.

Finally the **Fill cases** transformation completes the definitional tree. If we have a case with branches for constructors $C_1, C_2 \dots C_k$, then we look up the type T that all of these constructor belong to. Next we get the list Ctrs of all constructors that belonging to T. This list will contain $C_1, C_2, \ldots C_n$, but it may contain more. For each constructor not represented in the case-expression, we create a new branch $C_i \to \bot$.

After running all of these transformations, our program is in canonical form, and we may choose to optimize it, or we may skip straight to the post-processing phase. At this point we only need two transformations for post-processing however, we will need to add more to support some of the optimizations. If we ever have an expression of the form **let** $x =$ **case**..., then we need to transform the case-expression into a function call. We do not do this transformation in pre-processing because we do not want to split functions apart during optimizations. The **Let-Case** transformation has a single rule given in figure **??**.

Every let with a case-expression creates a new function $f\#n$ where n is incremented every time.

Finally, in our post-processing phase we simply factor out the scrutinee of a case-expression into a variable. The transformation is straightforward. An example of a pre-process derivation is given in **??**. At this point we are ready to transform the canonicalized FlatCurry into ICurry.

5.2 COMPILING TO ICURRY

ICurry is meant to be a bridge between Curry code and imperative languages like C, Python, and Assembly. The let and case-expressions have been transformed into statements, and variables have been explicitly declared. All mutually recursive declarations are broken here into two steps: Declare memory for each node, then fill in the pointers. This allows us to create expression graphs with loops in them. Each function is organized into a sequence of blocks, and each block is broken up into declarations, assignments, and a single statement. A statement can either fail, return a new expression graph, or inspect a single variable to choose a case.

After we have finished transforming the FlatCurry, the transformation to ICurry is much easier to implement. The algorithm from [15], given in figure **??**, can be applied directly to the translated program. We show an example of translating the function f from figure **??** into ICurry in figure **??**.

The algorithm itself is broken up into 6 pieces. First $\mathcal{F}$ Compiles a FlatCurry function into an ICurry function. Then $\mathcal{B}$ takes the function arguments, the expression, and the root, and compiles it into a block. We factor out $\mathcal{B}$ instead of leaving it a part of $\mathcal{F}$ because we will be able to recursively call it to construct nested blocks. This is also why we pass in a root parameter. In subsequent calls, the scrutinee of a case expression. While this is not explicit in the algorithm here, in our implementation, the root of any block under a case expression is always v_1. This will become the variable `scrutenee` from the C code in chapter 3. Next we declare variables with the $\mathcal{D}$ function. Each variables bound by a **let** or **free** expression must be declared. We also declare a variable for the scrutinee of the case statement, if this block has one. Then, $\mathcal{R}$ generates code for the return value. If the expression is a case, then examine the case variable and generate code for the associated blocks, otherwise we return the expression. Finally $\mathcal{E}$ generates code for constructing a piece of the expression graph. If the expression contains choices, function calls, or constructor calls, then the corresponding nodes are generated. If the expression is a variable, then it is returned. If the expression is a let or a free expression, then the principal expression is generated.

5.3 GENERATING C CODE

Now that we have a program in ICurry, we can translate this to C. We already have a good idea of what the C code should look like, and our ICurry structure fits closely with this. The difference is that we need to be sure to declare and allocate memory for all variables, which leads to a split in the structure of the generated code. The code responsible for creating expression graphs and declaring memory will go in the *.h file, and the code for executing the hnf function will go in the *.c file. This is a common pattern for structuring C and C++ code, so it is not surprising that we take the same approach.

For each Data type D, we generate both a `make_D` function and a `set_D`. The difference is that `make_D` will allocate memory for a new node, while `set_D` takes an existing node as a parameter, and transforms it to the given type of node. We do the same thing for every ICurry function f, and produce a `make_f` and `set_f` function in C. Each node contains a `symbol`, that denotes the type of node, and holds information such as the name, arity, and hnf function of the node. Along with setting the `symbol` from chapter 3, the make and set functions reset the nondet flag to `false`, and set any children that were passed into the node.

The code to generate the C source file is given in figures **??**, **??**, and **??**. This is a standard syntax directed translation. We hold of on showing the generated code for literal cases until Chapter 7 where we discuss our implementation of unboxing. We also skip over the generation of the functions for case expressions discussed in section 2.2.3. The code for this is largely the same. We just begin generating code at each block inside the function, after the declarations and assignments.

The translation is similar to how we translated ICurry programs. Figure **??** is the main entry point. We translate the function, blocks, declarations, and assignments. $\mathcal{F}$ translates an ICurry function to a C function. $\mathcal{B}$ translates an ICurry block. Along with the block to translate, we also pass in the function name, and current path to the block. This allows us to generate unique names for each of the functions for case expressions. We will use this information in the call to `save`, which pushes a rewrite onto the backtracking stack. The $\mathcal{D}$ function translates a variable declaration, and $\mathcal{A}$ translates an assignment.

Figure **??** generates code for translating statements. The $\mathcal{S}$ function translates an ICurry statement. Both *return* and $\perp$ just set the root of the expression to the appropriate value, but case statements require us to generate the switch case loop from figure **??**. Most of the loop is largely identical to the example, but to simplify the code generation process, we introduce a function `save`, which takes the root node, and a copy of the current function at this particular case, and pushes it on the backtracking stack. The notation $f|_p$ is read as the function with symbol f at the position p, and is just a unique identifier for this particular case statement. We also use a helper function FV to find all of the free variables in the rest of the body, since those will be needed to construct $f|_p$.

Finally figure **??** translates free variables, case branches, and expressions to C. The $\mathcal{V}$ function generates code to translate free variables. The final free variable, and the constructors containing free variables are pushed on the stack in reverse order. Then we set the root to be the first constructor. The $\mathcal{C}$ function translates a case branch to a C case statement. We insert the check and call to the `save` function, and generate code for the block. We split the generation of expressions into two functions. The $\mathcal{E}_\mathcal{S}$ function sets the root to an expression. The $\mathcal{E}_\mathcal{M}$ function creates nodes for a new expression.

In this chapter we used this library to transform FlatCurry programs into a canonical form that we could then translate to ICurry. We also showed how to translate ICurry program to C. In short we wrote the back end of a compiler in a simple, clear, and short implementation. This shows the power of the GAS system for applying simple transformations to Curry programs. In the next chapter we will see how we can use it to write an Optimizer. Now we're cooking with GAS!

Chapter 6

BASIC OPTIMIZATIONS

In the last chapter we saw how the GAS tool let us write transformation rules as rewrite rules in Curry. The power of this tool came from two aspects. The first is that it is easy to write rules syntactically. The second is that the rules are written in Curry, so we are not limited by our rewriting system. We will put this second part to use in optimizating Curry expressions.

In this chapter we outline a number of optimizations that were necessary to implement in order for unboxing, deforestation, and shortcutting to be effective. We start by introducing a new restriction on FlatCurry expressions called Administrative Normal Form, or A-Normal Form. This is a common form for functional program optimizers to take, and it provides several benefits to Curry too. We describe the transformation, and why it is useful, then we detail a few smaller optimizations that move let-expressions around. The goal is to move the let-expression to a position just before the variable is used in the expression. Finally we discuss four optimizations that will do most of the work in the compiler: Case canceling, dead code elimination, inlining, and reduction. These optimization are an important part of any optimizing compiler, but they are often tricky to get right. In fact, with the exception of dead code elimination, It is not clear at all that they are even valid for Curry. We show an effective method to implement them in a way that they remain valid for Curry expressions.

In this chapter we discuss one of the major hurderls to optimizing FlatCurry programs, we then present a solution in A-Normal form, We do on to develop some standard optimizations for FlatCurry includeing dead code elimination, case cancelling, and inlinig. Finally, we show these optimiztions at work optimizing the implementation the implementation $\leqslant$ for the *Bool* type.

6.1 A-NORMAL FORM

Before we discuss any substantial optimizations, we need to deal with a significant roadblock to optimizing Curry. Equational reasoning, in the sense of replacing expressions with their derived values, is not valid when optimizing FlatCurry programs. The reason is that expressions in

FlatCurry are not referentially transparent [55]. The evaluation of Curry programs is graph rewriting, which maintains referential transparency, but since FlatCurry is composed of terms, and not graph, we can not substitute expressions with their values.

While there have been graph intermediate representation proposed for languages [24, 42] FlatCurry is not one of these. We do think that incorporating the graph based IR might improve the optimization process, and we believe it is a promising area of future work.

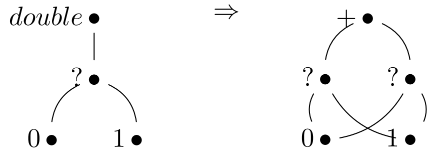
To see an example of why this an issue, let us consider the following program.

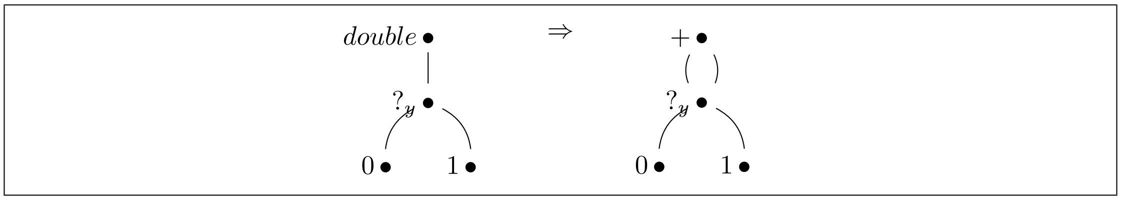
$$\begin{aligned}&\mathit{double}\ x = x + x\\&\mathit{main} \qquad = \mathit{double}\ (0\ ?\ 1)\end{aligned}$$

In pure lazy functional languages, it is always safe to replace a function with its definition. So we should be able to rewrite *main* to $(0\ ?\ 1) + (0\ ?\ 1)$, but this expression will produce a different set of answers. This is the primary problem with optimizing functional logic languages, but exactly why this happens is a bit tricky to pin down. The non-determinism is not the only problem, for example evaluating *id* $(0\ ?\ 1)$ at compile time is fine. We can even duplicate non-deterministic expressions with the following example.

$$\begin{aligned}&\mathit{double}\ x = x + x\\&\mathit{main} \qquad = \textbf{let}\ y = (0\ ?\ 1)\\&\qquad\qquad\quad \textbf{in}\ \mathit{double}\ y\end{aligned}$$

Here y is a non-deterministic expression, because it produces two answers when evaluated, but the expression **let** $y = (0\ ?\ 1)$ **in** $y + y$ is still equivalent to our example. The real problem with our first example is a bit more subtle, and we have to step back into the world of graph rewriting. If we construct the graph for the first expression we see:





Now the real issue comes to light. In the second example, while we copied a non-deterministic expression in the code, we did not copy the non-deterministic expression in the graph. This gives us a powerful tool when reasoning about Curry expressions. Even if a variable is duplicated in the source code, it is not copied in the graph. Since this duplication of non-deterministic expressions was the main concern for correctness, the solution is pretty straightforward. If we copy an expression in FlatCurry, then we should instead store that expression in a variable and copy the variable.

We can enforce this restriction by disallowing any compound expressions. Specifically, all function calls, constructor calls, choices, and case expression must either be applied to literal values or variables. Fortunately we are not the first to come up with this idea. In fact this restricted form is used in many functional compilers, and is known as Administrative Normal Form (ANF) [38]. The idea originally was to take CPS, another well known intermediate representation for functional languages, and remove common "administrative redexes". After removing the administrative redexes, we can remove the continuations, and rewrite the program using let-expressions. Flanagan et al. showed that these transformations can be reduced into a single A-Normal form transformation. We give the definition of A-Normal Form for Curry programs in figure **??** and we implement the transformation using GAS in figure **??** with the Curry implementation in figure **??**.

As long as we enforce this A-Normal Form structure, we restore equational reasoning for Curry programs. We do not even need to enforce A-Normal Form strictly here. During optimization, it is often useful to be able to replace variable bound to constructors and partial applications with their definitions. Since these nodes have no rewrite rules that can apply at the root, we can do this replacement without fear of problems with non-deterministic expressions. This will be referred to as limited A-Normal Form.

In fact, this is exactly how the operational semantics were defined for FlatCurry. In [4] FlatCurry programs are translated into a normalized form before evaluation begins. We choose to flatten these expressions as well because it produces more uniform programs, and more optimizing transformations become valid. Some examples of programs in ANF are given in figure **??**.

6.2 CASE CANCELING

Finally, we come to our first example of an optimization. In fact, this is arguably our most important optimization. It is a very simple optimization, but it proves to be very powerful. Consider the following code:

```
notTrue = case True of
                True → False
                False → True
```

Expressions like this come up frequently during optimization. This is fantastic, because it is clear what we should do here. We know that the *True* branch will be taken, so we might as well evaluate the case expression right now.

```
notTrue = False
```

This transformation is called Case Canceling, and it is the workhorse of all of our other optimizations. The transformation is given and **??** and examples of the transformation are given in **??**. If the scrutinee of a case is labeled by a constructor, then we find the appropriate branch, and reduce to that branch. The only real complication is that we need to keep the expression in A-Normal form. However, we can simply add let-expressions for every variable that the constructor binds.

We also include two other optimizations. These optimizations are really about cleaning up after Case Canceling runs. The first is Case Variable elimination. Consider the expression from the optimization of *compare* for *Bool* in figure **??**. The use of Case Variable elimination allows us to set up a situation where a case can cancel later. This occurs a lot in practice, but this optimization may raise red flags for some. In general it is not valid to replace a variable with an expression in FlatCurry. That variable could be shared, and it could represent a non-deterministic expression. Fortunately, this is still viable in Curry.

We give a short sketch of why Case Variable elimination is viable in Curry with the following example. Suppose I have the following FlatCurry definition for *notHead*. This function will look at the first element of a list, and return *not True* if the head of the list evaluates to True.

```
notHead xs = case xs of
                 x:_ → case x of
                           True → not x
```

We use a key fact from Brassels work [25][Lemma 4.1.10]. Lifting a case into it is own function Does not change the set of values an expression evaluates to. We can use this to lift the inner case into it is own function.

$$
\begin{array}{ll}
\mathit{notHead}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of} & \\
 & x{:}\_ \to \mathit{notHead}_1\ x \\
\mathit{notHead}_1\ x = \textbf{case}\ x\ \textbf{of} & \\
 & \mathit{True} \to \mathit{not}\ x
\end{array}
$$

Since uniform programs can be viewed as inductively sequential rewrite systems. The function $\mathit{notHead}_1$ should be equivalent to the following Curry program.

$$
\begin{array}{l}
\mathit{notHead}_1\ \mathit{True} = \mathit{not}\ \mathit{True} \\
\mathit{notHead}_1\ \mathit{False} = \bot
\end{array}
$$

Now this program could be compiled into the following semantically equivalent FlatCurry program.

$$
\begin{array}{ll}
\mathit{notHead}_1\ x = \textbf{case}\ x\ \textbf{of} & \\
 & \mathit{True}\ \to \mathit{not}\ \mathit{True} \\
 & \mathit{False} \to \bot
\end{array}
$$

Finally, by the path compression theorem we can reduce the call in *notHead* to get the following result.

$$
\begin{array}{lll}
\mathit{notHead}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of} & & \\
 & x{:}\_ \to \textbf{case}\ x\ \textbf{of} & \\
 & & \mathit{True} \to \mathit{not}\ \mathit{True}
\end{array}
$$

This gives us a general procedure for converting FlatCurry programs to the same program after performing Case Variable elimination. While we do not perform these steps in practice, each one has already been shown to be valid on their own, so our transformation is also valid.

Finally we have Dead Code Elimination. This is a standard optimization. In short, if we have an empty **let** or **free** expression, then we can remove them. This may happen due to the aliasing rule from last chapter. Furthermore if a variable is never used, then it can also be removed. Finally, if we have **let** $x = e$ **in** x, then we do not need to create the variable x. These are correct as long as we are careful to make sure that our variable definitions are not recursive.

Now that we have finally created an optimization, we can get back to moving code around in convoluted patterns. In the next section we look at how we can inline functions. Unlike Case Canceling, It is harder to determine the correctness of Inlining. In fact, we have to do a lot of work to inline functions in Curry.

6.3 INLINING

As mentioned at the start of this chapter, inlining is not generally valid in Curry. So, we need to establish cases when inlining is valid, determine when it is a good idea to inline, and ensure that our inlining algorithm is correct. This work is largely based on [28, 83].

Similarly to [83], we need to make a distinction between inlining and reduction. When we use the term *inlining* we are referring to replacing a let bound variable with it is definition. For example **let** $x = \mathit{True}$ **in** $\mathit{not}\ x$ could inline to $\mathit{not}\ \mathit{True}$. When we use the term *reduction*, we are referring to replacing a function call with the body of the function where the parameters of the function are replaced with the arguments of the call. Again, as an example **let** $x = \mathit{True}$ **in** $\mathit{not}\ x$ could reduce to:

$$
\begin{array}{l}
\textbf{let}\ x = \mathit{True} \\
\textbf{in}\ \ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \rightarrow \mathit{False} \\
\qquad\qquad \mathit{False} \rightarrow \mathit{True}
\end{array}
$$

The first problem with inlining and reduction we encounter is recursion. Consider the expression:

$$\textbf{let}\ \mathit{loop} = \mathit{loop}\ \textbf{in} \ldots$$

If we were to inline this variable, we could potentially send the optimizer into an infinite loop. So, we need to somehow mark all recursive variables and functions. The next problem follows immediately after that. So far we have done transformations with local information, but reduction is going to require global information. In fact, for reduction to be effective, it will require information from different modules. Consider the function:

$$\mathit{sumPrimes} = \mathit{foldr}\ (+)\ 0 \circ \mathit{filter}\ \mathit{isPrime} \circ \mathit{enumFromTo}\ 1$$

Aside from the fact that *sumPrimes* contains mostly recursive functions, we would not be able to optimize it anyway, because $\circ$ is defined in the standard Prelude. If we can not reduce the definition of $\circ$, then we are fighting a losing battle.

This brings us to our third problem with inlining. The *sumPrimes* function is actually partially applied. Its type should be $\mathit{sumPrimes} :: \mathit{Int} \rightarrow \mathit{Int}$, but *sumPrimes* is defined in a point-free style. Point-free programming causes a lot of problems, specifically because FlatCurry is a combinator language. In IRs like Haskell's Core, we could solve this problem by inlining a lambda expression, but it is not clear at all that inlining a lambda expression is valid in Curry. Instead, to solve this problem, we convert functions to be fully applied.

In order to solve these problems, we keep a map from function names to several attributes about the function. This includes: if the function is defined externally; if the function is known to be deterministic; if the function contains cases; the parameters of the function; the current number of variables in a function; the size of the function; and the function definition. This map is updated every time we optimize a new function, so we can reduce all functions that we have already optimized. We will use this map to determine when it is safe and effective to reduce a function.

6.3.1 Partial Applications

Dealing with partial applications is a bit more tricky. In fact, we can not use the GAS system to solve this problem because we may not know if a function is a partial application until we have optimized it. Consider the *sumPrimes* function again. It does not look like a partial application because the root function, $\circ$, is fully applied. Let us look at the definition for $\circ$. In Curry it is defined using a lambda expression.

$$f \circ g = \lambda x \rightarrow f\ (g\ x)$$

However, when translated to FlatCurry, this lambda expression is turned into a combinator.

$$f \circ g = \mathit{compLambda}_1\ f\ g$$
$$\mathit{compLambda}\ f\ g\ x = f\ (g\ x)$$

So, when we try to optimize *sumPrimes* we end up with the following derivation.

let $v_1 = p_2$
in let $v_2 = foldr_1\ v_1\ 0$
in let $v_3 = \mathit{isPrime\_1}$
in let $v_4 = \mathit{filter\_1}\ v_3$
in let $v_5 = \mathit{enumFromTo\_1}\ 1$

in let $v_6 = v_4 \circ v_5$

in $\boxed{v_2 \circ v_6}$

Reduce Base $\Rrightarrow [-1, -1, -1, -1, -1, -1]$

let $v_1 = p_2$

in let $v_2 = foldr_1\ v_1\ 0$

in let $v_3 = \mathit{isPrime\_1}$

in let $v_4 = \mathit{filter\_1}\ v_3$

in let $v_5 = \mathit{enumFromTo\_1}\ 1$

in let $v_6 = \boxed{v_4 \circ v_5}$

in $\mathit{compLambda}_1\ v_2\ v_6$

Reduce Let $\Rrightarrow [-1, -1, -1, -1, -1]$

let $v_1 = p_2$

in let $v_2 = foldr_1\ v_1\ 0$

in let $v_3 = \mathit{isPrime\_1}$

in let $v_4 = \mathit{filter\_1}\ v_3$

in let $v_5 = \mathit{enumFromTo\_1}\ 1$

in let $v_6 = \mathit{compLambda}_1\ v_4\ v_5$

in $\mathit{compLambda}_1\ v_2\ v_6$

The **Reduce Base** and **Reduce Let** transformations will be described later. At this point there is no more optimization that can be done, because everything is a partial function. But this is not a great result. We have created a pipeline, and when we pass it a variable, then everything will be fully applied. So, how do we solve the problem?

The key is to notice that if the root of the body of a function is a partial application, then we can rewrite our definition. We simply add enough variables to the function definition so the body of the function is fully applied. The transformation

Add Missing Variables

$$f\ \overline{v} = \mathit{g\_k}\ \overline{e} \qquad \Rrightarrow \qquad f\ \overline{v}\ \overline{x} = \mathit{apply}\ (\mathit{g\_k}\ \overline{e})\ \overline{x}$$

The *sumPrimes* functions is transformed with the derivation in **??** and we can continue to optimize the function.

6.3.2 The Function Table

In order to keep track of all of the functions we have optimized we create a function lookup table called $\mathbf{F}_{\mathcal{F}}$. The function table is just a map from function names to information about the function. We use the following definitions for lookups into the function table. $I_{\mathcal{F}}\ f$ returns true if we believe that f is a good candidate for reduction. We have designed the compiler so that whatever heuristic we use to decide if a function can be inlined, it is easy to tweak, but at the very least f should not be external, nor too big, and inlining f should not lead to an infinite derivation. $U_{\mathcal{F}}\ x\ f\ e$ attempts to determine if reducing the function f in the expression **let** $x = f \ldots$ **in** e would be useful. Again this heuristic is easily tweakable, but currently, a function is useful if x is returned from the function, it is used as the scrutinee of a case expression, or it is used in a function that is likely to be reduced. $S_{\mathcal{F}}\ f$ returns True if f is a simple reduction with no case expressions. It is always useful to reduce these functions. $C_{\mathcal{F}}\ f\ [e_1, \ldots e_n]$ returns true if reducing f with $e_1 \ldots e_n$ will likely cause Case Canceling.

6.3.3 Function Ordering

The problem of function ordering seems like it should be pretty inconsequential, but it turns out to be very important. However, this problem has already been well studied [28, 83], and the solutions for other languages apply equally well to Curry.

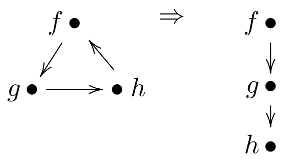
The problem seems very complicated at the start. We want to know what is the best order to optimize functions. Fortunately there is a very natural solution. If possible we should optimize a function before we optimize any function that calls it. This turns out to be an exercise in Graph Theory.

We define the Call Graph of a set of functions $\mathbf{F} = \{f_1, f_2, \ldots f_n\}$ to be the graph $G_{\mathbf{F}} = (\mathbf{F}, \{f_i \to f_j | f_i \ \textit{calls} \ f_j\})$. This problem reduces to finding the topological ordering of $G_{\mathbf{F}}$. Unfortunately, if $\mathbf{F}$ contains any recursion, then the topological ordering is not defined. So, instead, we split $G_{\mathbf{F}}$ into strongly connected components, and find the topological ordering of those components. Within each component, we pick an arbitrary function, called the *loop breaker*, which is removed from the graph. We then attempt to find the topological order of each component again. This process repeats until our graph is acyclic.

These loop breakers are marked in $\mathbf{F}_{\mathcal{F}}$, and they are never allowed to be reduced. Every other function can be reduced, because all functions that it calls, except for possibly the loop breakers, have been optimized.

Consider the program:

$$
\begin{aligned}
&f\ x = g\ x\\
&g\ x = h\ x\\
&h\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad 0 \to 0\\
&\quad \_ \to 1 + f\ x
\end{aligned}
$$



The graph for this function is a triangle, because f calls g which calls h which calls f. However, if we mark h as a loop breaker, then suddenly this problem is easy. When we optimize h, we are free to reduce f and g.

$$
\begin{aligned}
&h\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad 0 \to 0\\
&\quad \_ \to \textbf{let}\ y = \boxed{f\ x}\\
&\qquad\quad \textbf{in}\ 1 + y\\
&\Rrightarrow \textbf{Reduce Let}\\
&h\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad 0 \to 0\\
&\quad \_ \to \textbf{let}\ y = \boxed{g\ x}\\
&\qquad\quad \textbf{in}\ 1 + y\\
&\Rrightarrow \textbf{Reduce Let}\\
&h\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad 0 \to 0\\
&\quad \_ \to \textbf{let}\ y = \boxed{h\ x}\\
&\qquad\quad \textbf{in}\ 1 + y
\end{aligned}
$$

6.3.4 Inlining

Now that we have everything in order, we can start developing the inlining transformation. As mentioned before, we need to be careful with inlining. In general, unrestricted inlining is not valid in Curry. This is a large change from lazy languages like Haskell, where it is valid, but not always a good idea. The other major distinction is that FlatCurry is a combinator language. This means that we have no lambda expressions, which limits what we can even do with inlining.

Fortunately for us, these problems actually end up canceling each other out. In Peyton-Jones work [83] most of the focus was on inlining let bound variables, because this is where duplication of computation could occur. However, we have two things working for us. The first is that we can not inline a lambda since they do not exist. The second is that we have translated FlatCurry to A-Normal Form. While Haskell programs are put into A-Normal Form when translating to STG code [87], this is not the case for Core. Certain constraints are enforced, such as the trivial constructor argument invariant, but in general Core is less restricted.

Translating to A-Normal form gives us an important result. If we inline a constructor then we do not affect the computed results. This same result holds for literal values, but we will discuss how we handle literals in Curry in the next chapter.

Theorem 7. *If* **let** $x = e_1$ **in** e *is a Curry expression in limited A-Normal Form, and* e_1 *is rooted by a constructor application, or partial application, then* $e[x \to e_1]$ *computes the same results.*

Proof. First note that given our semantics for partial application, a partially applied function is a normal form. There are no rules for evaluating a partial application, only for examining one while evaluating an apply node.

If e_1 is a constructor, or partial application, then it is a normal form. Therefore it is a deterministic expression by definition 2.2.6. Since e_1 is deterministic by the path compression theorem, e evaluates to the same values as $e[x \to e_1]$ □

Now we have enough information to inline variables as long as we restrict inlining to literals, constructors, variables, and partial applications, although the case for variables is already subsumed by the **Alias** rule **??**. We add two new rules. **Let Folding** allows us to move variable definitions closer to where they are actually used, and **Unapply** allows us to simplify expressions involving *apply*. Both of these are useful for inlining and reduction. The GAS rules are given in figure **??**. Note that the **Unapply** rule corresponds exactly to the evaluation step for application nodes in our semantics. The inlining rules correspond to the cases discussed above. We add one more rule. We inline a variable bound to a case expression, if that expression occurs once in a needed position. Since we can not determine if the variable occurs in a needed position at compile time, we can use check if it occurs in a strict position [75]. This is usually good enough. The implementation of Inlining in the Gas system is given in figure **??**. The combinator (x, e) @> σ is used to build up substitutions. It means that we add we add $x \rightarrow e$ to the substitution σ. The *idSub* substitution is just the identity. In the *letFold* rule, *hasVar* checks is expression e contains variable v. In the fist Inlining rule, the *strict* and *uses* functions are just to ensure that x is a in a strict position, and that it is only at one position in e. These are not required for correctness, we have found that these restrictions generate better code.

6.3.5 Reduce

Finally we come to reduction. While this was a simpler task than inlining in GHC, it becomes a very tricky prospect in Curry. Fortunately, we have already done the hard work. At this point, in any given function definition, the only place a function symbol is allowed to appear in our expressions is as the root of the body, as the root of a branch in a **case** expression, as the root the result of a **let** expression, or as a variable assignment in a let-expression. Furthermore our functions only contain trivial arguments, so it is now valid to reduce any function we come across.

Theorem 8 (reduction)**.** *Let e be an expression in limited A-Normal Form, let $e|_p = f\ \overline{e}$, where f is a function symbol with definition $f\ \overline{v} = b$, and let $\sigma = \{\overline{v} \mapsto \overline{e}\}$. Then $e[p \rightarrow \sigma\ b]$ has the same values as e.*

Proof. First note that There is only one way to replace an expression where the root has symbol f, with the body of the definition for f. Therefore This is a deterministic step, and by the path compression theorem e and $e[p \rightarrow \sigma\ b]$ have the same values. □

We give the GAS rules for reduction in figure **??**. These rule make use of the function table We make sure that $B_{\mathcal{F}}\ f$ replaces the definition with fresh variables. Therefore, we avoid any need to deal with shadowing and name capture. This strategy was taken from [83] and it works very well. Although, since FlatCurry uses numbers exclusively to represent variables, we do not get the same readable code.

We end by giving a couple of examples of reductions to see how they work in practice. The first example returns from the start of this chapter. We see that *double* $(0\ ?\ 1)$ is reduced so we do not make a needless call to *double*, but we have avoided the problem of run time choice semantics.

Our next function comes from a possible implementation of $\leqslant$ for Boolean values. In fact, this is the implementation we chose for the instance of the *Ord* class for *Bool*. The example is a bit long, but it shows how many of these optimizations work together to produce efficient code.

In the next chapter we discuss three more optimizations, Unboxing, Case Shortcutting, and Deforestation. While Unboxing and Deforestation are in common use in lazy function compilers, they have not been used for functional-logic languages before. Case Shortcutting is a new optimization to Curry.

Chapter 7

MEMORY OPTIMIZATIONS

In this chapter we develop three new optimizations for Curry. First, Unboxing is an attempt to remove boxed values from our language. We discuss our implementation of primitive values and operations, and how explicitly representing the boxes around these values leads to optimizations. Second, we adapt the shortcutting optimization, which was designed for rewrite systems, to work in our compilation scheme for Curry. The techniques used in shortcutting allow us to skip the construction of the scrutenee of a case expression. Finally, Shortcut Deforestation is a optimization for removing intermediate lists. This has been studied extensively in functional languages, but it has not been shown to be valid in the presence of non-determinism. We prove its validity in Curry, and give a formulation that can apply to combinator languages.

7.1 UNBOXING

So far we have avoided talking about operations in Curry for primitive data types *Int*, *Char*, and *Float*. This is primarily because in all current implementations of Curry, primitive values are boxed. A *box* for a primitive value is a node in the expression graph that holds the primitive value. This is done primarily to give a uniform representation of nodes in our expression graph. There are many reasons why we would want to box primitive values. It makes the implementations of run-time systems, garbage collectors, and debugging software much easier. The choice of how we represent boxes has a pervasive effect on the compiler. Since we knew how we intended to implement Unboxing, we decided to use our representation from the beginning.

We chose to follow the style of Unboxing from Launchbury et al. [58] and represent all boxes explicitly in FlatCurry, as opposed to other system which may represent the boxes at run-time, but do not mention the boxes at compile time. This has several advantages, but one of the most important is that we can apply optimizations to the boxes themselves.

7.1.1 Boxed Values

Before we get into the process of unboxing values, we need to look at how we represent boxed values. The idea of boxing primitive values is common in higher level languages, since it allows us to simplify the run-time system. This is especially true in lazy languages where expressions such as $3 + 5$ are represented by an expression graph that will eventually hold the value 8 after it is evaluated. It is important that every node that points to the expression graph of $3 + 5$ at run-time will then point to the expression graph of 8 after it is evaluated. This update is difficult if 8 is the literal C integer `8`. However, if 8 is instead a constructor node containing the value 8, then this is fine. We just replace the contents of the node labeled by $+$ by a node labeled by *Int* with one child, which is the C integer 8.

The purpose of unboxing is not to remove boxes entirely. Instead we try to find cases where we replace the creation of new nodes in the expression graph with primitive arithmetic operations. The idea from Launchbury [58] is that we can find these cases where we can remove boxes more easily if the boxes are explicitly represented in the intermediate representation. In order to represent boxes we need to make three changes to our FlatCurry programs.

The first is that every primitive value, a literal value of type *Int*, *Float*, or *Char* is replaced with a constructor of the appropriate type. For example $5 + 6$ is transformed into the expression $(\mathit{Int}\ 5) + (\mathit{Int}\ 6)$.

Second, we need to wrap cases of literal values with cases that remove the box. This can best be demonstrated with the example in figure **??**. The value x is evaluated down to an *Int* node, then we extract the unboxed integer x_{prim} and proceed with the primitive case statement. We also add one dummy branch to the unboxing case for each branch in the primitive case. These branches are there to instruct the code generator on what values a free variable could take on.

Finally we need to give new definitions for primitive operations such as $+$ and $\leqslant$. All of the operators fit the same pattern, so we only give the definitions for $+$ and $\leqslant$ for integers in figure **??**. In order to evaluate a $+$ node, we evaluate the first argument to its box, then we unbox it with the case statement. We do not have any dummy branches for free variables. This represents the fact that $+$ is a rigid operation. We proceed to evaluate and unbox the second argument. Finally, we return the result inside of a new box. The $+_{\text{prim}}$ operation performs the addition, and is translated to an add expression in C. The $\leqslant_{\text{prim}}$ operation performs a comparison between two integers, and returns either *True* or *False* based on the result.

Before we even look at trying to remove these boxes, it is worth taking a second to see if we can optimize literal values. There are actually a couple of significant improvements we can make that apply more broadly.

The first is that for any constructor with no arguments, such as *True* or *Nothing*, we can create a single static node to represent that constructor. This eliminates the need to allocate memory for each instance of *True*. While this is great, we might expect to go further. For example, if we could turn case statements of Boolean expressions into simple if statements in C. We could compare the scrutinee to *True*, and if it is, then we evaluate the true branch, otherwise we evaluate the false branch. Unfortunately, this does not work for two reasons. First, not all instances of *True* can be the static *True* nodes. As an example, at run-time *not False* will evaluate to *True*, but the node is going to be in the same location as the original *not* node. Second, even if we have a Boolean expression that has been evaluated to a value, it could still be a `FAIL` or `FREE` node.

The next optimization we can make is for the primitive types *Int*, *Char*, and *Float*. Since these constructors have an argument, namely the primitive value, we cannot make a single static node for them. We might try to create a single static node for every literal value used in the program. Unfortunately this does not tend to help us that much. Consider the standard factorial program:

```
fac n = case n of
            0 → 1
            n → n ∗ fac (n − 1)
```

Now if we evaluate *fac* 42, we will allocate memory up front for 0, 1, and 42. This will certainly save some memory, but not as much as we would hope. We will still construct every number between 2 and 41.

A better solution is to employ the flyweight pattern similar to the JVM. The idea is that small integers are likely to come up often. So, we statically allocate all of the integers between -128 and 128. We do a similar allocation for characters. Unfortunately, this patterns did not show improved performance for floating point numbers.

Now that we have seen how to represent boxes, we can work on removing them, and see what we actually gain from it.

7.1.2 Unboxed Values

In order to get an idea of the effectiveness of unboxing, let us look at an example. Consider the function to compute Fibonacci numbers if figure **??**. We will work with this example extensively in the next couple of optimizations, in an attempt to see how much we can optimize it.

Unfortunately, using the optimizations we have already discussed, this function can not be optimized any further. The *fib* function is recursive, so we can not reduce it, and $n - 1$ contains a primitive operation. However, we allocate a lot of memory while evaluating this function. We create 5 nodes for each recursive call, $cont, n_1, f_1, n_2, f_2$. We do not create a node for $f_1 + f_2$ since that will replace the root node during evaluation. We can statically allocate a node for each integer, because the integers are constant. However, there is still no need for this much allocation. The problem is that each of our primitive operations and recursive calls must be represented as a node to fit in with our definition of an expression graph.

However, after explicitly representing the boxes, and using our new definitions for $+$ and $\leqslant$, we can optimize *fib* to the program given in figure **??**.

As we can see, the code is significantly longer, but now we have included the primitive operations in our code. The variables v_2, n_1, n_2, p_1, p_2 are all primitive values, so we do not need to allocate any memory for them, so they will not be represented as nodes in our expression graph. This seems like a big win, but it is a little deceptive. We are still allocating 1 node for $cond, f_1, f_2$ and 2 nodes for the *Int* constructors. So, we are still allocating 5 nodes, which is just as much memory as before. This is an improvement in efficiency, but we can certainly do better.

7.1.3 Primitive Conditions

The first optimization is that we really do not need to allocate memory for *cond*. $x \leqslant_{\text{prim}} y$ should compile down to an expression involving the primitive `<=` operation in C, and return a Boolean value. However, right now there is no way to signal that to the code generator, so we introduce the **pcase** construct.

The *primCond* must be a primitive condition expression, which is either $==_{\text{prim}}$ or $\leqslant_{\text{prim}}$, and the arguments must be primitive values. The semantics of **pcase** are exactly what be expected, but now we can translate it into a simple `if` statement in C, as shown in figure **??**.

This has several advantages. First we do not construct a node for the boolean value. Even if we are statically allocating a single node value for *True* and *False*, we avoid the cost of switch case loop, and the cost of checking if *primcond* is non-deterministic. It must be deterministic, because both of its operands are primitive values. After implementing this construct, the new version is in figure **??**. Now we are down to 4 nodes, but we can still do better. The next challenge is unboxing the arguments in the call to *fib*.

7.1.4 Strictness Analysis

The problems with eliminating boxes from arguments of function calls is strongly related to the run-time system and how we represent nodes in our expression graph. Recall that our expression graph is made up of node structs that point to other node structs. If we have a *fib* node, then the argument to this node is expected to be another node. In C we can get around this by using a union. We created a union `field`, defined in figure **??** that can either represent an *Int*, *Char*, *Float*, `Node`, or an array of `Node*` in case a node has more than 3 children.

The problem with storing a primitive value in a node, instead moves to identifying when a value is primitive. There is no way to distinguish between `Node*` and `unsigned long`. Instead of trying to figure out when a child of a node is supposed to represent a primitive value at run-time. We need to keep track of this information at compile time. Fortunately, this is a well studied problem [64, 75, 92].

Lazy functional languages often try to remove laziness for efficiency reasons. We do not want to create an expression for a primitive value if we are only going to deconstruct it, so it becomes useful to know what parameters in a function must be evaluated, A parameter that must be evaluated by a function is called *strict*. Formally, a function f is strict in its parameter if $f\ \bot = \bot$.

We use $\bot$ here to mean that the value of it is parameter does not evaluate to a value, this can come from a call to the *error* function, or an infinite computation. It does not mean that f failed to return a value. We explicitly exclude that case, because that can change the results of some Curry programs. For example consider the function:

$f\ x = head\ [\,]$

If we were to mark x as strict, then we may try to evaluate x before computing f. This could cause an infinite loop in the following program:

$loop = loop$

$main = (f\ loop)\ ?\ 1$

This should return a single result, and never try to evaluate *loop*. For this reason we consider failing computations to be similar to expressions rooted by constructors for the purposes of strictness analysis.

We implemented an earlier form of strictness analysis described by Peyton Jones et al. [84]. The idea is that we start by assuming every function is strict in all its parameters. Then as we analyze a function we determine which parameters can be relaxed. For example consider the following function:

$$
\begin{aligned}
&f\ x\ y\ z = \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad\quad \mathit{True}\ \rightarrow y\\
&\qquad\qquad\quad \mathit{False} \rightarrow y + z
\end{aligned}
$$

It is clear that x must be strict, but we do not know about y or z. After analyzing the case branches, we see that since y appears in both branches, and $+$ is strict in both of it is arguments, y must be strict as well. Finally, since there is a branch that z does not appear in, z may not be evaluated, so it is not strict.

This syntactic traversal of an expression is useful, but it fails when working with a recursively defined function. Consider the factorial function with an accumulator:

$$
\begin{aligned}
&\mathit{faca}\ n\ \mathit{acc} = \textbf{case}\ n{=}{=}0\ \textbf{of}\\
&\qquad\qquad\qquad \mathit{True}\ \rightarrow \mathit{acc}\\
&\qquad\qquad\qquad \mathit{False} \rightarrow n * \mathit{faca}\ (n-1)\ (n * \mathit{acc})
\end{aligned}
$$

We can see with a syntactic check that n is strict, but what about *acc*. *acc* does appear in both branches, but it is the argument to the recursive call of *faca*. Therefore we have *acc*, the second parameter of *faca*, is strict if, and only if, the second parameter of *faca* is strict. We can solve this problem by iteratively running the strictness analyzer on *faca* until it converges to a single set of strict parameters. Formally, since a variable can be either strict or not strict, we can represent it with a 2 element set $\{0, 1\}$, and our strictness analyzer is a monotonic function, so we are computing a least fixed point in the strictness analyzer over the set 2^n where n is the arity of the function.

There are much more sophisticated implementations of strictness analysis. We do not analyze deeper than a single pattern, and we are very conservative in regard to recursive functions. Mycroft's original work was to interpret functions as Boolean formulas [75]. This can find several cases of strict parameters that our implementation does not. There has also been a lot of work on projection based strictness analysis [64]. The current state of the art for Haskell is backwards projection analysis [92]. Studying the validity and implementation of these strictness analyzers in regard to Curry would all be great candidates for future work.

Once we know which arguments are strict we can split the function into a wrapper function and a worker function [92]. We can see this with *fib*. We take the current optimized version in figure **??**, and apply the worker/wrapper split in figure **??**. This creates two functions, *fib*, which simply evaluates and unboxes the parameter, ,and *fib#worker*, which does the rest of the computation. Then we optimize the function again resulting in figure **??**. Notice that since *fib* is no longer recursive we can inline it. So we can inline the call to *fib* in the following code:

$\mathbf{let}\ f_1 = \mathit{fib}\ (\mathit{Int}\ n_1)$

This results in:

$$\begin{aligned}&\mathbf{let}\ f_1 = \mathbf{case}\ (\mathit{Int}\ n_1)\ \mathbf{of}\\&\qquad \mathit{Int}\ v_2 \rightarrow \mathit{fib\#worker}\ v_2\end{aligned}$$

Which can be optimized to:

$\mathbf{let}\ f_1 = \mathit{fib\#worker}\ n_1$

We are down to allocating 2 nodes. We only need to allocate nodes for the calls to *fib#worker*. This means that we have reduced our memory consumption by 60%. That is a huge improvement, but we can still do better. With the next optimization we look at how to remove the remaining allocations.

7.2 SHORTCUTTING

In the last section were able to optimize the *fib* function from allocating 5 nodes per recursive call to only allocating 2 nodes per recursive call. However, we were left with a problem that we can not solve by a code transformation.

$$\begin{aligned}&\mathbf{let}\ f_1 = \mathit{fib\#worker}\ n_1\\&\mathbf{in\ case}\ f_1\ \mathbf{of}\\&\qquad\quad \mathit{Int}\ p_1 \rightarrow\ \ldots\end{aligned}$$

In this section we aim to eliminate these final two nodes. However, in order to do this, we will have to step outside of compile-time optimizations, and look into previous work on run-time optimizations for Curry. Specifically we are going to use an idea inspired by the shortcutting optimizations [19]. We first look into the shortcutting optimization itself, then we show how a couple of ideas from it can be applied to our compiler. However, the analogy will be a bit imprecise, since shortcutting was developed for Graph Rewrite Systems.

7.2.1 The Original Shortcutting Optimizations

Initially shortcutting was developed for Inductively Sequential Graph Rewrite Systems [19]. However, the application to functional logic languages was clearly in mind, and the ideas were even implemented, and shown to be effective in the Packs compiler.

We begin with a set of rewrite rules R, for an Inductively Sequential system, and define a `compile` function to translate the rules into a rewrite system that simulates an innermost rewriting system by adding two new rules **H** and **N**. These rules are roughly analogous to our `hnf` and `nf` functions from chapter 3. The **H** function computes an expression to head constructor form, and the **N** function computes an expression to constructor normal form by repeatedly calling the **H** function. We will focus on the **H** function itself, since that is where the optimization occurs.

To translate R to a new rewritings system that simulates an innermost strategy, we traverse the definitional tree in post-order, and emit rules as we reach the leaves of the tree. We gloss over the details here, but an example of the **H** function for the curry function $+\!\!+$ and *length* are shown in figure **??**. Our transformed code is actually very similar to what the compiler already does. We generate a new **H** function for every function symbol. For example, $\mathbf{H}_{length}$ computes the length of a list to head constructor form. This corresponds to the O_R code described in the original paper. When looking at the $\mathbf{H}_{length}$ function, we notice something interesting.

$$\mathbf{H}_{length}(x : xs) = \mathbf{H}_{+}(1, \mathbf{H}_{length}(xs))$$

We do not actually construct a node for + or *length xs*. Even with our current implementation, and unboxing strategy, we would still have to construct a node for *length xs* in order to evaluate it. It is recursive, so we could not inline the call to length, and it is possible that it could be non-deterministic. The shortcutting compiler has managed to avoid constructing a node that our compiler has to construct. This is worth looking into.

In keeping with our example, let us see what the shortcutting compiler produces with our *fib* example. We have restructured the example as rewrite rules, but it is still performing the same computation. The results are shown in figure **??**. As we can see, we can avoid constructing nodes for $\leqslant$, $+$ and $-$, but we already managed to avoid these node constructions with unboxing and inlining. The more interesting point here is that we can avoid the construction of the recursive call to *fib*. So, how do we avoid constructing these nodes? There are two key pieces of information that will help us here. First, the node labeled by *fib* is not the root of the right hand side. Second, the node is needed in that expression. If both of these conditions are met, then we do not need to construct the node. We can *shortcut* the construction, and just compute the value.

Now that we have a theoretical idea of how to eliminate nodes with shortcutting, we need to apply it to our actual generated code. By our two conditions, we are looking for a node that is generated by the right hand side of a rewrite rule, and we are looking for a node that is needed in that expression. The equivalent in our compiled code is when a let bound variable is rooted by a function, and is used exactly once as the scrutinee of a case expression. In the Fibonacci example in figure **??** both f_1 and f_2 meet these conditions. Typically a compiler for a lazy language would recognize this situation, and instead of generating a node only to evaluate it, the compiler would produce a function call that would return the value of that node.

It is worth looking at an attempt to try to replace the node with a function call. One possibility would be to try to statically analyze a function f and determine if it is deterministic. This is a reasonable idea, but it has two major drawbacks. First, determining if a function is non-deterministic is undecidable, so the best we could do is an approximation. Second, even if f is deterministic, the expression $f\ x$ could still be non-deterministic if x is. This is going to be very restrictive for any possible optimization.

In the example in figure **??** we need a node to hold the value for $\mathit{fib\#worker}\ n_1$, but this value will only be used in the case expression. In fact, it is not possible for this node to be shared with any part of the expression graph. If this expression is only ever scrutinized by the case expression, then we only need to keep the value around temporarily. The idea here is simple, but the implementation becomes tricky. We want to use a single, statically allocated, node for every variable that is only used as the scrutinee of a case.

There are two steps to the optimization. The first step is marking every node that is only used as the scrutinee. The second step happens during code generation. Instead of dynamically allocating memory for a marked node, we store all of the information in a single, statically allocated, node. We call this node RET for return.

This effectively removes the rest of the dynamically allocated nodes from our *fib* function, but before we celebrate, we need to make sure that code generated using this transformation actually produces the same results. There are a few things the can potentially go wrong.

First, let us look at the case where the scrutinee is deterministic. In that case, there is only one thing that could go wrong. It is possible that in order to reduce the scrutinee we need to reduce another expression that could be stored in RET. For example, consider the following program:

$$
\begin{aligned}
&f\ x = \mathbf{case}\ g\ x\ \mathbf{of}\\
&\qquad\qquad True\ \rightarrow False\\
&\qquad\qquad False \rightarrow True\\
&main = \mathbf{case}\ f\ 3\ \mathbf{of}\\
&\qquad\qquad True\ \rightarrow 0\\
&\qquad\qquad False \rightarrow 1
\end{aligned}
$$

In the evaluation of *main*, $f\ 3$ can be stored in the RET node, then we can evaluate $f\ 3$ to head constructor form, but while we are evaluating $f\ 3$, we store $g\ 3$ in the same RET node. While this is concerning, it is not actually a problem. As shown in chapter 3, at the beginning of `f_hnf`, we store all of the children of `root` as local variables, and then when we have computed the value, we overwrite the `root` node. In our case the `root` node in our function is RET. However, aside from the very start and end of the function, we never interact with the `root` node, so even if we reuse RET in the middle of evaluating f, it does not actually affect the results.

A portion of the generated code for f can be seen in figure **??**. As we can see, the only time that we use the `root` node is at the very start of the function to store the variables, and right before we return. Even if `root` happens to be RET, this does not actually affect the evaluation. The RET node is overwritten with the contents of $g\ x$, then it is evaluated, and finally it is overwritten with the result of f right before returning.

It seems like we should be able to store these marked variables in the RET node, and then just call the appropriate `_hnf` function. In fact this was the first idea we tried. The generated code for main is given in figure **??**.

This initial version actually works very well. In fact, for *fib#worker* we are able to remove the remaining 2 allocations. This is fantastic, and we will come back to this point later, but we need to deal with a looming problem.

7.2.2 Non-deterministic RET Nodes

The problem with the scheme we have developed so far is that if `RET` is non-deterministic, then the rewrite rule we push on the backtracking stack may contain a pointer to `RET`. This is a major problem with this optimization, because `RET` will almost certainly have been reused by the time backtracking occurs.

This optimization was built on the idea that `RET` is only ever used in a single case expression. Therefore, it is important that we never put `RET` on the backtracking stack. We need rethink on our idea. Initially, we wanted to avoid allocating a node if a variable is used in a single case. Instead, we will only allocate a node if `RET` is non-deterministic. This means that for deterministic expression, we do not allocate any memory, but for non-deterministic expression, we still have a persistent variable on the stack. This lead to the second implementation in figure **??**.

7.2.3 RET hnf Functions

This solution is better, because any rewrites we push onto the stack contain a copy of `RET`, but it is still not correct. Three things can still go wrong here. These are very subtle errors that are very easy to overlook, and even harder to track down the real cause of the errors.

The first problem is that `RET` might have been reduced to a forwarding node, so it might be deterministic, but forward to a non-deterministic node. For example, in $\mathbf{case}\ id\ (0\ ?\ 1)\ \mathbf{of} \ldots$ there is clearly non-determinism, but the *id* node is not the cause of it, so that rewrite should not be pushed on the backtracking stack.

Another problem is that, if `RET` is a forwarding node, when evaluating the node it forwards to, we might have reused `RET`. Consider the following program.

$$
\begin{array}{l}
h\ x = \mathbf{case}\ x > 3\ \mathbf{of} \\
\qquad\qquad\quad \mathit{False} \rightarrow 3 \\
\qquad\qquad\quad \mathit{True}\ \ \rightarrow 4 \\
\mathit{main} = \mathbf{case}\ (h\ 4\ ?\ h\ 2)\ \mathbf{of} \\
\qquad\qquad\qquad 4 \rightarrow \mathit{True}
\end{array}
$$

Here *main* evaluates $h\ 4\,?\,h\ 2$. Since ? is non-deterministic, and reduces to a forwarding node, we need to make a copy of RET as part of the rewrite we push on the backtracking stack. However, before we can even do that, we need to evaluate h 4, and the expression $x > 3$ will be stored in RET. Now we have lost the information in RET before we can copy it.

Finally, we still have not avoided putting RET on the backtracking stack. Recall our program from before.

$$\begin{aligned}
&f\ x = \textbf{case}\ g\ x\ \textbf{of} \\
&\qquad\qquad\quad \mathit{True}\ \rightarrow \mathit{False} \\
&\qquad\qquad\quad \mathit{False} \rightarrow \mathit{True} \\
&\mathit{main} = \textbf{case}\ f\ 3\ \textbf{of} \\
&\qquad\qquad\quad\ \mathit{True}\ \rightarrow 0 \\
&\qquad\qquad\quad\ \mathit{False} \rightarrow 1
\end{aligned}$$

If the expression $g\ x$ is non-deterministic, then the node containing f will be marked as non-deterministic. However, f 3 was stored in the RET node, so the root parameter will be RET. Now RET is still pushed on the backtracking stack, but this time it is on the left hand side of the rewrite.

This is starting to seem hopeless, when we fix one problem, 3 much more subtle problems pop up. How can we avoid creating nodes for deterministic expressions, but still only create a single node that the caller and callee agree on if the expression is non-deterministic?

The answer is that we need to change how RET nodes are reduced. Specifically, we create a new reduction function that only handles nodes stored in RET. In the case of f, we would create a f_hnf, a f_1_hnf and a f_RET_hnf. The third function only reduces f that has been stored in a RET node.

The difference between f_hnf and f_RET_hnf is that instead of passing the root node, we pass Node* backup. The backup node is where we will store the contents of RET if we discover evaluating the expression rooted by f is non-deterministic. Finally we return backup. Now both the caller and callee agree on backup. Furthermore, since backup is a local variable, it is not affected if f reuses RET over the course of its evaluation. We can see the final implementation of shortcutting for *main* in figure ??. We also give the definition for f_RET_hnf in figure ??.

Now, we finally have a working function. We only allocate memory if the scrutinee of the case is non-deterministic. If the expression is non-deterministic in multiple places, then the same backup node is pushed on the stack, so our expression graphs stay consistent.

This also works well if we have multiple reductions in a row. Suppose we have the following Curry code:

$$
\begin{array}{l}
\mathit{main} = \textbf{case}\ f\ 4\ \textbf{of} \\
\qquad\qquad \mathit{True} \rightarrow \mathit{False} \\
\qquad\qquad \mathit{False} \rightarrow \mathit{False} \\
f\ n = \textbf{case}\ n\ \textbf{of} \\
\qquad\qquad 0 \rightarrow \mathit{True} \\
\qquad\qquad \_ \rightarrow f\ (n-1)
\end{array}
$$

In this case f is a recursive function, so when we reduce f 4, we need to reduce f 3. This is no problem at all, because we are reducing f 4 with `f_RET_hnf`. Ignoring the complications of Unboxing for the moment, we can generate the following code for the return of f.

```
field v2 = make_int(n-1)
set_f(RET, v2);
return f_RET_hnf(backup);
```

7.2.4 Shortcutting Results

Before we move onto our next optimization, we should look back at what we have done so far. Initially, we had a *fib* function that allocated 5 nodes for every recursive call. Then, through Unboxing, we were able to cut that down to only 2 allocations per call. Finally, using Shortcutting, we were able to eliminate those two allocations. We would expect a substantial speedup by reducing memory consumption by 60%, but removing those last two allocations is a difference in kind. The *fib* function runs in exponential time, and since each step allocates some memory, the original *fib* function allocated an exponential amount of memory on the heap. However, our fully optimized *fib* function only allocates a static node at startup. We have moved from exponential memory allocated on the heap to constant space. While *fib* still runs in exponential time, it runs much faster, since it does not need to allocate memory. Surprisingly, *fib* is still just as efficient with non-deterministic arguments. If the argument is non-deterministic, the wrapper function will evaluate it before calling the worker.

Now that we have removed most of the implicitly allocated memory with Unboxing and Shortcutting, we can work on removing explicitly allocated memory with a technique from functional languages.

7.3 DEFORESTATION

We now turn to our final optimization, Deforestation. The goal of this optimization is to remove intermediate data structures. Programmers often write in a pipeline style when writing functional programs. For example, consider the program:

$$\mathit{sumPrimes} = \mathit{sum} \circ \mathit{filter}\ \mathit{isPrime} \circ \mathit{enumFromTo}\ 2$$

While this style is concise and readable, it is not efficient. First, we create a list of integers, then we create a new list of all of the integers in our list that are prime, and finally we sum the values in that list. It would be much more efficient to compute this sum directly.

```
$\mathit{sumPrimes}\ n = \mathit{go}\ 2\ n$
  where $\mathit{go}\ k\ n$
          $\mid k \geqslant n \qquad = 0$
          $\mid \mathit{isPrime}\ k = k + \mathit{go}\ (k+1)\ n$
          $\mid \mathit{otherwise}\ = \mathit{go}\ (k+1)\ n$
```

This pipeline pattern is pervasive in functional programming, so it is worth understanding and optimizing it. In particular, we want to eliminate the two intermediate lists created here. This is the goal of Deforestation.

7.3.1 The Original Scheme

Deforestation has actually gone through several forms throughout it is history. The original optimization proposed by Wadler [99] was very general, but it required a complicated algorithm, and it could fail to terminate. There have been various attempts to improve this algorithm [97] and [37] that have focused on restricting the form of programs.

An alternative was proposed by Gill in his dissertation [40,41] called foldr-build Deforestation or short-cut Deforestation. This approach is much simpler, always terminates, and has a nice correctness proof, but it comes at the cost of generality. Foldr-build Deforestation only works with functions that produce and consume lists. Still, lists are common enough in functional languages that this optimization has proven to be effective.

Since then foldr-build Deforestation has been extended to Stream Fusion [32]. While this optimization is able to cover more cases than foldr-build Deforestation, it relies on more advanced compiler technology.

The foldr-build optimization itself is actually very simple. It relies on an observation about the structure of a list. All lists in Curry are built up from cons and nil cells. The list $[1,2,3,4]$ is really $1:2:3:4:[\,]$. One very common list processing technique is a fold, which takes a binary operation and a starting element, and reduces a list to a single value. In Curry, the *foldr* function is defined as:

$$\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [\,a\,] \to b\\
&\mathit{foldr} \oplus z\ [\,] \qquad\quad = z\\
&\mathit{foldr} \oplus z\ (x : \mathit{xs}) = x \oplus \mathit{foldr}\ f\ \ z\ \mathit{xs}
\end{aligned}$$

As an example, we can define the *sum* function as $\mathit{sum}\ \mathit{xs} = \mathit{foldr}\ (+)\ 0$. To see what this is really doing we can unroll the recursion. Suppose we evaluate $\mathit{foldr}\ (+)\ 0\ [1,2,3,4,5]$, then we have:

$$\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ [1,2,3,4,5]\\
&\Rightarrow 1 + \mathit{foldr}\ (+)\ 0\ [2,3,4,5]\\
&\Rightarrow 1 + (2 + \mathit{foldr}\ (+)\ 0\ [3,4,5]))\\
&\Rightarrow 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [4,5]))\\
&\Rightarrow 1 + (2 + (3 + (4 + \mathit{foldr}\ (+)\ 0\ [5])))\\
&\Rightarrow 1 + (2 + (3 + (4 + (5 + \mathit{foldr}\ (+)\ 0\ [\,])))))\\
&\Rightarrow 1 + (2 + (3 + (4 + (5 + 0))))
\end{aligned}$$

But wait, this looks very similar to our construction of a list.

$$\begin{aligned}
&1\ :\ (2\ :\ (3\ :\ (4\ :\ (5\ :\ ([\,])))))\\
&1 + (2 + (3 + (4 + (5 + (0)))))
\end{aligned}$$

We have just replaced the : with + and the [ ] with 0. If the compiler can find where we will do this replacement, then we do not need to construct the list. On its own, this is a very hard problem, but we can help the compiler along. We just need a standard way to construct a list. This can be done with the *build* function [40].

$$\begin{aligned}
&\mathit{build} :: (\forall\ b\ (a \to b \to b) \to b \to b) \to [\,a\,]\\
&\mathit{build}\ g = g\ (:)\ [\,]
\end{aligned}$$

The *build* function takes a function that constructs a list. However, instead of construction the list with : and [ ], we abstract this by passing the constructors in as arguments, which we call c

and n respectively. Now, with *build*, we can define what we mean by deforestation with a simple theorem from [40].

Theorem 9. *For all* $f : a \to b \to b$*,* $z : b$*, and* $g : (\forall\ b\ (a \to b \to b) \to b \to b) \to [a]$*,*

$$foldr\ f\ z\ (build\ g) = g\ f\ z$$

So, if we can construct standard list functions using *build* and *foldr*, then we can remove these function using the above theorem. As an example, let us look at the function *enumFromTo* a b that constructs a list of integers from a to b.

$$\begin{aligned}
&enumFromTo\ a\ b\\
&\quad \mid a > b \qquad\quad = [\,]\\
&\quad \mid otherwise = a : enumFromTo\ (a+1)\ b
\end{aligned}$$

We can turn this into a build function.

$$\begin{aligned}
&enumFromTo\ a\ b = build\ (enumFromTo\_build\ a\ b)\\
&enumFromTo\_build\ a\ b\ c\ n\\
&\quad \mid a > b \qquad\quad = n\\
&\quad \mid otherwise = a\ `c`\ enumFromTo\_build\ (a+1)\ b\ c\ n
\end{aligned}$$

We can create build functions for several list creation functions found in the standard library. In fact, for this optimization we replace several functions in both the Prelude and List library with equivalent functions constructed with *foldr* and *build*. Now we are ready to apply Deforestation to Curry. Unfortunately there are two problems we need to solve. The first is an implementation problem, and the second is a theoretical problem. First, while we can apply foldr/build Deforestation, we can not actually optimize the results. Second, we still need to show it is valid for curry.

7.3.2 The Combinator Problem

Let us look back at the motivating example, and see how it could be optimized in Haskell, or any language that can inline lambda expressions. The derivation in figure **??** comes from the original paper [40].

This looks good. In fact, we obtained the original expression we were trying for. Unfortunately we do not get the same optimization in Rice. The problem is actually the definition of *filter*.

$$\mathit{filter}\ f = \mathit{build}\ (\lambda c\ n \rightarrow \mathit{foldr}\ (\lambda x\ y \rightarrow \mathbf{if}\ f\ x\ \mathbf{then}\ x\ \text{‘}c\text{‘}\ y\ \mathbf{else}\ y)\ n)$$

Functions that transform lists, such as *filter*, *map*, and *concat*, are rewritten in the standard library as a build applied to a fold. Unfortunately our inliner can not produce this derivation. We do not inline lambda expressions, and reductions can only be applied to let bound variables, so we simply can not do this reduction. Instead we need a new solution.

7.3.3 Solution build_fold

Our solution to this problem is to introduce a new combinator for transforming lists. We call this *build_fold* since it is a build applied to a fold.

$$\mathit{build\_fold} :: ((c \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b \rightarrow b)) \rightarrow (b \rightarrow b) \rightarrow [\,a\,] \rightarrow b$$
$$\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs} = \mathit{foldr}\ (\mathit{mkf}\ (:))\ (\mathit{mkz}\ [\,])\ \mathit{xs}$$

The idea behind this combinator is a combination of a build and a fold. This function was designed to be easily composable with both build and fold. Ideally, it could fit in the middle of build and fold and still reduce. As and example:

$$\mathit{foldr}\ (+)\ 0\ (\mathit{build\_fold}\ \mathit{filter\_mkf}\ \mathit{filter\_mkz}\ (\mathit{build}\ \mathit{enumFromTo\_build}))$$

Ideally, this function should reduce into something relatively efficient, Furthermore we wanted *build_fold* to compose nicely with itself. For example, $\mathit{map}\ f \circ \mathit{map}\ g$ should compose to something like $\mathit{map}\ (f \circ g)$.

We achieve this by combining pieces of both *build* and *foldr*. The two functions *mkf* and *mkz* make the f and z functions from fold, however they take c and n as arguments similar to *build*. The idea is that *mkf* takes an f from *foldr* as a parameter, and returns a new f. The *map* and *filter* implementations are given below.

$$\mathit{map}\ f = \mathit{build\_fold}\ (\mathit{map\_mkf}\ f)\ \mathit{map\_mkz}$$
$$\mathit{map\_mkf}\ f\ c\ x\ y = f\ x\ \text{‘}c\text{‘}\ y$$
$$\mathit{map\_mkz}\ n = n$$

$$\mathit{filter}\ p = \mathit{build\_fold}\ (\mathit{filter\_mkc}\ p)\ \mathit{filter\_mkz}$$
$$\mathit{filter\_mkf}\ p\ c\ x\ y = \mathbf{if}\ p\ x\ \mathbf{then}\ x\ \text{‘}c\text{‘}\ y\ \mathbf{else}\ y$$
$$\mathit{filter\_mkz}\ n = n$$

The purpose of the convoluted definition of *build_fold* is that it plays nicely with *build* and *foldr*. We have the following three theorems about *build_fold*, which we will prove later. These are analogous to the *foldr* / *build* theorem.

Theorem 10. *For all functions of the appropriate type that evaluate no ? expressions, the following qualities hold.*

$$\begin{aligned}
&\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = \mathit{build}\ (\lambda c\ n \to g\ (\mathit{mkf}\ c)\ (\mathit{mkz}\ n))\\
&\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) = \mathit{foldr}\ (\mathit{mkf}\ f)\ (\mathit{mkz}\ z)\ \mathit{xs}\\
&\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}) = \mathit{build\_fold}\ (\mathit{mkf}_2 \circ \mathit{mkf}_1)\ (\mathit{mkz}_2 \circ \mathit{mkz}_1)\ \mathit{xs}
\end{aligned}$$

Now that we have removed all of the lambdas from our definitions, we can look at the implementation.

7.3.4 Implementation

Deforestation turned out to be one of the easiest optimizations to implement. The implementation is entirely in GAS, and it proceeds in two steps. First we find any case where a *build* or *build_fold* occurs exactly once in either a *build_fold* or *fold*. If this is the case, we inline the variable that *build* is bound to into it is single use. This temporarily takes our expression out of A-Normal Form, but we will restore that with the second step, which is the actual Deforestation transformation, which applies either the *foldr* / *build* theorem, or one of the three *build_fold* theorems from above. The definitions for the deforest transformation are given in figure **??** The optimization derivation for *sumPrimes* is in figure **??**.

So far we have done a decent job. It is not as efficient as the Haskell version, but that is not surprising. However, we can still improve this. The main problem here is that we can not optimize a partial application. This is unfortunate, because the *build_fold* function tends to create large expressions of partially applied functions. Fortunately we have already solved this problem earlier in our compiler. We already have a way to detect if an expression is partially applied, so, in the post processing phase, we do a scan for any partially applied functions. If we find one, then we move the code into a newly created function, and attempt to optimize it. We call this function outlining, since it is the opposite of inlining. If we can not optimize the outlined function, then we do nothing. Otherwise, we make a new function, and replace the call to the partially applied function with a call to the outlined function. This would actually be worth doing even if we did not implement Deforestation. With function outlining our final optimized code is given below.

$$
\begin{array}{l}
\mathit{sumPrimes}\ n = \mathit{enumFromTo\_build}\ 2\ n\ f'\ 0 \\
f'\ x\ y = \mathbf{if}\ \mathit{isPrime}\ x\ \mathbf{then}\ x + y\ \mathbf{else}\ y \\
\mathit{enumFromTo\_build}\ a\ b\ c\ n \\
\quad \mid a > b \qquad\quad = n \\
\quad \mid \mathit{otherwise} = a\ `c`\ \mathit{enumFromTo\_build}\ (a+1)\ b\ c\ n
\end{array}
$$

This certainly is not perfect, but it is much closer to what we were hoping for. Combining this with Unboxing and Shortcutting gives us some very efficient code. While these results are very promising, we still need to know if Deforestation is even valid for Curry.

7.3.5 Correctness

First we show that the *build_fold* theorems are valid for a deterministic subset of Curry using the same reasoning as the original foldr-build rule. Without non-determinism and free variables, we can apply the same arguments as the original paper on shortcut deforestation [40].

Theorem 11. *For any deterministic f, z, g, mkf, and mkz, the following equations hold.*

$$
\begin{array}{l}
\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = \mathit{build}\ (\lambda c\ n \rightarrow g\ (\mathit{mkf}\ c)\ (\mathit{mkz}\ n)) \\
\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) = \mathit{foldr}\ (\mathit{mkf}\ f)\ (\mathit{mkz}\ z)\ \mathit{xs} \\
\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}) = \mathit{build\_fold}\ (\mathit{mkf}_2 \circ \mathit{mkf}_1)\ (\mathit{mkz}_2 \circ \mathit{mkz}_1)\ \mathit{xs}
\end{array}
$$

Proof. Recall that the free theorem [98] for *build* is for all h, f, and f' of the appropriate type:

$$
\begin{array}{l}
(\forall\ (a : A)\ (\forall\ (b : B)\ h\ (f\ a\ b) = f'\ a\ (h\ b))) \Rightarrow \\
\forall\ (b : B)\ h\ (g_B\ f\ b) = g'_B\ f'\ (h\ b)
\end{array}
$$

We substitute $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}$ for h, $(:)$ for f and $\mathit{mkf}\ (:)$ for f'. From the definition of *build_fold* we have $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (a : b) = (\mathit{mkf}\ (:))\ a\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ b)$
and $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ [\,] = \mathit{mkz}\ [\,]$. Therefore we have $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (g\ (:)\ b) = g\ (\mathit{mkf}\ (:))\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ b)$
This gives us the following result.

$$
\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = g\ (\mathit{mkf}\ (:))\ (\mathit{mkz}\ [\,])
$$

Finally, working backwards from the definition of *build* we have our theorem.

$$\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = \mathit{build}\ (\lambda c\ n \rightarrow g\ (\mathit{mkf}\ c)\ (\mathit{mkz}\ n))$$

Again with *foldr* we have the free theorem
if $\forall\ (a : A)\ (\forall\ (b : B)\ b\ (x\ \oplus\ y) = (a\ x)\ \otimes\ (b\ y)$ and $b\ u = u'$
then $b \circ \mathit{foldr}\ \oplus\ u = \mathit{foldr}\ \otimes\ u' \circ (\mathit{map}\ a)$
Here we take $b = \mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}$, $\oplus = f$, and $\otimes = \mathit{mkf}\ f\ \ a = \mathit{id}$
then the statement becomes:

if $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (f\ x\ y) = (\mathit{mkf}\ f)\ x\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ y)$
and $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ [\,] = \mathit{mkz}\ [\,]$
then $\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz} \circ \mathit{foldr}\ f\ z = \mathit{foldr}\ (\mathit{mkf}\ f)\ (\mathit{mkz}\ z)$

Since both conditions follow directly from the definition of *build_fold* we are left with

$$\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz} \circ \mathit{foldr}\ f\ z = \mathit{foldr}\ (\mathit{mkf}\ f)\ (\mathit{mkz}\ z)$$

which is exactly what we wanted. Free theorems are fun!

Finally for *build_fold* / *build_fold* rule suppose we have the expression

$$\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}))$$

From the previous result we have:

$$\begin{aligned}
&\mathit{foldr}\ (\mathit{mkf}_1\ f)\ (\mathit{mkz}_1\ z)\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs})\\
&= \mathit{foldr}\ (\mathit{mkf}_2\ (\mathit{mkf}_1\ f))\ (\mathit{mkz}_2\ (\mathit{mkz}_1\ z))\ \mathit{xs}\\
&= \mathit{foldr}\ ((\mathit{mkf}_2 \circ \mathit{mkf}_1)\ f)\ ((\mathit{mkz}_2 \circ \mathit{mkz}_1)\ z)\ \mathit{xs}\\
&= \mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ (\mathit{mkf}_2 \circ \mathit{mkf}_1)\ (\mathit{mkz}_2 \circ \mathit{mkz}_1)\ \mathit{xs})
\end{aligned}$$

which establishes our result:

$$\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2) = \mathit{build\_fold}\ (\mathit{mkf}_2 \circ \mathit{mkf}_1)\ (\mathit{mkz}_2 \circ \mathit{mkz}_1)$$

□

While this gives us confidence that Deforestation is a possible optimization, we have already seen that referential transparency [55], and therefore equational reasoning, does not always apply in Curry. We need to show that both expressions will evaluate to the same set of values in any contest. In fact, as they are currently stated, These theorems do not actually hold for Curry. However, with a few assumptions, we can remedy this problem. First, we need to rewrite our rules so that the reduced expression is in A-Normal form.

$$\begin{aligned}
\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = \mathbf{let}\ g' = (\lambda c\ n \rightarrow\ &\mathbf{let}\ f = \mathit{mkf}\ c\\
&\phantom{\mathbf{let}}\ z = \mathit{mkz}\ n\\
&\mathbf{in}\ \ g\ f\ z)\\
\mathbf{in}\ \mathit{build}\ g'&
\end{aligned}$$

$$\begin{aligned}
\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) = \mathbf{let}\ &f' = \mathit{mkf}\ f\\
&z' = \mathit{mkz}\ z\\
\mathbf{in}\ \ &\mathit{foldr}\ f'\ z'\ \mathit{xs}
\end{aligned}$$

$$\begin{aligned}
\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}) = \mathbf{let}\ &\mathit{mkf} = \mathit{mkf}_2 \circ \mathit{mkf}_1\\
&\mathit{mkz} = \mathit{mkz}_2 \circ \mathit{mkz}_1\\
\mathbf{in}\ \ &\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}
\end{aligned}$$

Now we are ready to state our result.

Theorem 12. *suppose f, z, g, mkf, and mkz are all FlatCurry functions whose right had side is an expression in A-Normal form, then the following equations are valid.*

$$\begin{aligned}
\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) = \mathbf{let}\ g' = (\lambda c\ n \rightarrow\ &\mathbf{let}\ f = \mathit{mkf}\ c\\
&\phantom{\mathbf{let}}\ z = \mathit{mkz}\ n\\
&\mathbf{in}\ \ g\ f\ z)\\
\mathbf{in}\ \mathit{build}\ g'&
\end{aligned}$$

$$\begin{aligned}
\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) = \mathbf{let}\ &f' = \mathit{mkf}\ f\\
&z' = \mathit{mkz}\ z\\
\mathbf{in}\ \ &\mathit{foldr}\ f'\ z'\ \mathit{xs}
\end{aligned}$$

$$\begin{aligned}
\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_2\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}) = \mathbf{let}\ &\mathit{mkf} = \mathit{mkf}_2 \circ \mathit{mkf}_1\\
&\mathit{mkz} = \mathit{mkz12} \circ \mathit{mkz}_1\\
\mathbf{in}\ \ &\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}
\end{aligned}$$

Proof. We show the result for foldr-build, and the rest are similar calculations. We intend to show that for any f, z, and g that the following equation holds.

$$\mathit{foldr}\ f\ z\ (\mathit{build}\ g\ (:)\ [\,]) = g\ f\ z$$

That is, we show that $\mathit{fold}\ f\ z\ (\mathit{build}\ g\ (:)\ [\,])$ reduces to the same values as $g\ f\ z$.

We proceed in a manner similar to [30]. First, notice that $\mathit{build}\ g\ (:)\ [\,]$ is constructing a list. However, since g is potentially non-deterministic, and it might fail, we may have a non-deterministic alternation of lists when evaluating this expression. Let us make this explicit. After evaluating $\mathit{build}\ g\ (:)\ [\,]$ we will produce an alternation of several lists.

$$\begin{aligned}\mathit{build}\ g\ (:)\ [\,] = {} & g_{1,1} : g_{1,2} : g_{1,3} : \ldots \mathit{end}_1\\ & ?\ g_{2,1} : g_{2,2} : g_{2,3} : \ldots \mathit{end}_2\\ & \quad \ldots\\ & ?\ g_{k,1} : g_{k,2} : g_{k,3} : \ldots \mathit{end}_k\end{aligned}$$

Where, for all i, $\mathit{end}_i = [\,]\ ?\ \bot$.

Here we have a alternation of k lists, and each list ends either with the empty list, or the computation may have failed along the way. Therefore, end_i may be either $[\,]$ or $\bot$. In fact, it might be the case that an entire list is $\bot$, but this is fine, because that would still fit this form defined above.

We can generalize this by passing arbitrary arguments to build. The expression $\mathit{build}\ g \oplus z$ evaluates to the following alternation of values.

$$\begin{aligned}&\mathit{build}\ g \oplus z\\ &\quad = (g_{1,1} \oplus g_{1,2} \oplus g_{1,3} \oplus \ldots z_{\mathit{end}_1})\ ?\\ &\qquad (g_{2,1} \oplus g_{2,2} \oplus g_{2,3} \oplus \ldots z_{\mathit{end}_2})\ ?\\ &\qquad\quad \ldots\\ &\qquad (g_{k,1} \oplus g_{k,2} \oplus g_{k,3} \oplus \ldots z_{\mathit{end}_k})\end{aligned}$$

Where, for all i, z_{end_i} is $\bot$ if end_i is $\bot$ and z otherwise.

Now, let us see what happens when we normalize the entire expression. Recall that if f is a dominator of $a\ ?\ b$, then $f\ (a\ ?\ b) = f\ a\ ?\ f\ b$ [9]. Therefore if all arguments are in A-Normal form, then function application distributes over choice. Since *foldr* is a dominator of everything in $\mathit{foldr} \oplus z\ (\mathit{build}\ g\ (:)\ [\,]$ we have the following derivation.

$$
\begin{array}{l}
\mathit{foldr} \oplus z\ (\mathit{build}\ g\ (:)\ [\,])\\
\quad = \textbf{let}\ \mathit{fold} = \mathit{foldr} \oplus z\\
\qquad \textbf{in}\ \mathit{fold}\ (\mathit{build}\ g\ (:)\ [\,])\\
\quad = \textbf{let}\ \mathit{fold} = \mathit{foldr} \oplus z\\
\qquad \textbf{in}\ \mathit{fold}\ (g_{1,1} : g_{1,2} : g_{1,3} : \ldots \mathit{end}_1\ ?\\
\qquad\qquad\qquad g_{2,1} : g_{2,2} : g_{2,3} : \ldots \mathit{end}_2\ ?\\
\qquad\qquad\qquad \ldots\\
\qquad\qquad\qquad g_{k,1} : g_{k,2} : g_{k,3} : \ldots \mathit{end}_k)\\
\quad = \textbf{let}\ \mathit{fold} = \mathit{foldr} \oplus z\\
\qquad \textbf{in}\ \mathit{fold}\ (g_{1,1} : g_{1,2} : g_{1,3} : \ldots \mathit{end}_1)\ ?\\
\qquad\quad \mathit{fold}\ (g_{2,1} : g_{2,2} : g_{2,3} : \ldots \mathit{end}_2)\ ?\\
\qquad\qquad\qquad \ldots\\
\qquad\quad \mathit{fold}\ (g_{k,1} : g_{k,2} : g_{k,3} : \ldots \mathit{end}_k)\\
\quad = (g_{1,1} \oplus g_{1,2} \oplus g_{1,3} \oplus \ldots z_{\mathit{end}_1})\ ?\\
\qquad (g_{2,1} \oplus g_{2,2} \oplus g_{2,3} \oplus \ldots z_{\mathit{end}_2})\ ?\\
\qquad\quad \ldots\\
\qquad (g_{k,1} \oplus g_{k,2} \oplus g_{k,3} \oplus \ldots z_{\mathit{end}_k})\\
\quad = g \oplus z
\end{array}
$$

Where, for all i, z_{end_i} is $\bot$ if end_i is $\bot$ and z otherwise.

This proves the result.

□

Note that while this does prove the result, there are still some interesting points here. First, we never made any assumptions about f or z. In fact, we did not really make any assumptions about g, but we did at least give an explicit form for its values. This form is guaranteed by the type. This line of reasoning looks like a promising direction for future explorations into parametricity for functional-logic programming.

Second, it should be noted that branches in g that produce $\bot$ do not necessarily fail when evaluated. If f is strict, then any failure in the list will cause the entire branch to fail. Consider the following expression:

$$\mathit{foldr}\ (\lambda x\ y \to 1)\ 0\ (\mathit{build}\ (\lambda c\ n \to 0\ \text{‘}c\text{‘}\ 1\ \text{‘}c\text{‘}\ \bot))$$

Evaluating the expression rooted by *build* to constructor normal form would produce a failure, since the tail of the list is $\bot$. However, since the first parameter in the expression rooted by *foldr* never looks at either of it is arguments, this branch of the computation can still return a result.

In this chapter we have developed three optimizations to help reduce the memory allocated by Curry programs. These optimizations seem effective, and we have shown why they are correct, but we still need to find out how effective they are. In the next chapter we show how well our compiler compares to Pakcs, Kics2, and MCC on the benchmarking suite provided by Kics2. We also show the results for each optimization individually, and then combined.

Chapter 8

RESULTS

Now that we have finally implemented all of the optimizations, we need to see if they were actually effective. Before we can look at the results, we need to discuss methodology. The test suite is based on the test suite from the Kics2 compiler [26]. We have removed some tests, and added others in order to test specific properties of our compiler.

Specifically, we removed all of the tests that evaluated the unification operator =:= or the functional pattern operator =:≪=. While the `RICE` compiler does support these operations, they are primitive operations with respect to Curry that can potentially do a substantial amount of work. This means that the operators are typically implemented in the target language of the compiler. While `RICE` does perform well with code containing these operators, we felt that it was an unfair comparison. It measured the implementation of the operators, instead of the quality of the generated code.

Furthermore, we added a few tests to demonstrate the effectiveness of deforestation. The benchmark suite for Kics2 contained very few examples of code with multiple list operations.

In order to characterize the effectiveness of our optimizations, we are interested in two measurements. First, we want to show that the execution time of the programs is improved. Second, we want to show that optimized programs consume less memory. The second goal is very easy to achieve. We simply augment the runtime system with a counter that we increment every time we allocate memory. When the program is finished running, we simply print out the number of memory allocations.

Execution time turns out to be much more difficult to measure. There are many factors which can affect the execution time of a program. To help aliviate these problems, we took the approach outlined by Mytkowicz et al. [76]. All programs were run multiple times, and compiled in multiple environments for each compiler. We took the lowest execution time. We believe these results are as unbiased as we can hope for; however, it is important to remember that our results may vary across machines and environments. For most of our results the `RICE` compiler is a clear winner.

8.1 TESTS

Our test suite is based on the Kics2 test suite [26]. We split the functions into three groups: Numeric computations meant to test Unboxing; non-deterministic computations; and list computations meant to test Deforestation. We do not have any specific tests for shortcutting, because it applies in almost every program.

- **Numeric computations:**
  - **fib** is the Fibonacci program from chapter 5.
  - **fibNondet** This is the same program, but we call it with a non-deterministic argument.
  - **tak** computes a long, mutually recursive, function with many numeric calculations.
- **Non-deterministic computations:**
  - **cent** attempts to find all expressions containing the numbers 1 to 5 that evaluate to 100.
  - **half** computes half of a number defined using piano arithmetic by trial and error starting from 0.

    ```
    half n | x + x==n = x
      where x free
    ```

  - **ndTest** computes a variant of *fib* that non-deterministically returns many results.

    ```
    fib n
      | n < 2 = 0 ? 1
      | otherwise = fib (n − 1) + fib (n − 2)
    ```

  - **perm** computes all of the permutations of a list by computing a single permutation non-deterministically.
  - **queensPerm** is the program from the introduction. It computes solutions to the n-queens problem by permuting a list, and checking if it is a valid solution.

  - **primesort** non-deterministically sorts a list of very large prime numbers.
  - **sort** sorts a list by finding a sorted permutation.

- **Deforestation:**
  - **queensDet** computes solutions to the n-queens problem using a backtracking solution and list comprehension.
  - **reverseBuiltin** reverses a list without using functions or data types defined in the standard Prelude.
  - **reverseFoldr** reverses a list using a reverse function written as a fold.
  - **reversePrim** reverses a list using the built-in reverse function and primitive numbers.
  - **sumSquares** computes $sum \circ map\ sqaure \circ filter\ odd \circ enumFromTo\ 1$.
  - **buildFold** computes a long chain of list processing functions.
  - **primes** computes a list of primes.
  - **sieve** computes *sumPrimes* from Chapter 5.

The results of running the tests are given in figure **??** for timing, and **??** for memory. All times are normalized. In figure **??** the times are normalized to `RICE`, and in figure **??** all results normalized to the unoptimized version in order to see the improvement of optimizations. Memory results are measured in the number of allocations of nodes. We also include a comparison all of 3 prominent Curry compilers, Pakcs, Kics2, and Mcc, against `RICE` in figure **??**. We optimized these compilers as much as possible to get the best results. For example Kics2 executed much quicker when run in the primitive depth first search mode. We increased the input size for tak, buildFold, and sieve in order to get a better comparison with these compilers. However, we were not able to run the buildFold test, or the reverseBuiltin test, for the Pakcs compiler. They were both killed by the Operating System before they could complete. We timed every program with Kics2 [26], Pakcs [52], and the Mcc [71] compiler. Unfortunately we were not able to get an accurate result on how much memory any of these compilers allocated, so we were unable to compare our memory results.

There are a lot of interesting results in tables **??**, **??**, and **??** that we feel are worth pointing out. First, it should be noted that the Mcc compiler performed very well, not only against both Kics2 and Pakcs, but it also performed well against `RICE`. In most examples it was competitive with the unoptimized code, and ahead of it in several tests. It even outperformed the optimized version in the cent example. We are currently unsure of why this happened, but we have two theories. First, the code generation and run-time system of Mcc may just be more efficient than `RICE`. While we worked to make the run-time system as efficient as possible, it was not the focus of this compiler. Mcc also translated the code to Continuation Passing Style [22] before generating target code. This may be responsible for the faster execution times. Our other theory is that Mcc supports an older version of Curry that does not include type classes. Mcc may have performed better simply by not having to deal with that overhead.

Aside from the surprising performance of Mcc, we found a couple of results in our optimizations that surprised us as well. First, the *half* program used more memory with basic optimizations turned on then it did with no optimizations. This is because strictness analysis created a worker function, but it was not able to cancel out any of the new *Int* constructors. While this did cause memory usage to go up a little, it did not effect the execution time. However, we could disable strictness analysis unless unboxing is turned on. Second, the *ndtest* used a bit (about 0.05%) more memory with the unboxing optimization. This is because of a confluence of two side effects of the optimization. Without unboxing we can not determine that the parameters to primitive operations are needed, so we can not force evaluation. This means that instead of evaluating each piece of the Fibonacci function separately, we need to construct the entire contractum $\mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$ and evaluate it. Because of this, the optimized code only contains a single case expression. The other factor is our solution to the non-determinism problem from section 2.2.3. Since we are returning several results, and the unboxed *fib* function contains several cases, we have to push more case functions onto the backtracking stack. While this does allocate a little more memory, we believe that the 2x speed-up in execution time is worth the sacrifice.

In terms of effectiveness, unboxing seemed to be the clear winner. Deforestation did not seem to be nearly as effective, but we believe this is more related to the test suite than anything else. These are all small programs that do not include many list processing operations. We believe that, on larger programs, deforestation would have more opportunities to fire. Shortcutting typically performed well, and compensated for the lack of unboxing in several situations. We think the most interesting part of these results is the effect of combining these optimizations. In particular, unboxing and shortcutting work very well together, often reducing the amount of memory consumed more than either optimization alone.

Generally `RICE` compares very favorably with all of the current compilers, only losing out to Mcc on the cent example. We focus on the Kics2 compiler, because that was the best performing compiler that is still in active development. With this comparison `RICE` performs very well, showing anywhere form a 2x to 50x execution speed-up on all of the non-deterministic programs, and a 3x to 50x improvement on the deterministic programs. Even comparing against Mcc, we typically see a 2x speed-up. The only excepts are cent, and programs that cannot be optimized, such as perm. We also see a very impressive speedup on *fibNondet* compared to Kics2. However, this is a known issue with the evaluation of non-deterministic expressions with functions with non-linear rules. We do believe that this is important to note, because these programs are common in Curry, and is the reason that we could not use Kics2 to develop `RICE`.

This is a very impressive speed-up, but we have already discussed the reason for it. After we applied Unboxing and Shortcutting, we were able eliminate all but a constant number of heap allocations from the program. This would be a great result on its own, but it gets even better when we compare it to GHC. Compiling the same *fib* algorithm on GHC produced code that ran about twice as fast as our optimized `RICE` code, and when we turned off Optimizations for GHC we ran faster by a factor of 8. It is not surprising to us that our code ran slower than GHC. The run time system is likely much faster than ours, and there are several optimization in GHC that we have not implemented. In fact, we would be shocked if it managed to keep up. What is surprising, and encouraging, is that we were competitive at all. It suggests that Curry is not inherently slower than Haskell. We believe that a more mature Curry compiler could run as fast as GHC for deterministic functions. This would give us the benefits of Curry, such as non-determinism and free variables, without sacrificing the speed of modern functional languages.

In this chapter we have justified the benefit of these optimizations to Curry. In the next chapter we look at possible future directions to take this work, and we conclude.

Chapter 9

CONCLUSION

These results were honestly significantly better than we ever expected with this project. Initially, we hoped to compete with Kics2, since it was leveraging GHC's optimizer to produce efficient code. However, we found that could we beat Kics2 in all cases, and in many cases the results were simply incomparable. In some cases we were even able to compete with GHC itself. Furthermore, we have shown that the memory optimizations really were effective for Curry programs. This is not much of a surprise. Allocating less memory is a good strategy for improving run-time performance. It is good to know that the presence of non-determinism does not affect this commonly held belief.

It is a little more surprising that these optimizations all turned out to be valid in Curry. In fact, a surprising number of optimizations are valid in Curry under suitable conditions. This might not seem very significant until we look at what optimizations are not valid. For example, common sub-expression elimination was not included in this compiler, because it simply is not a valid Curry transformation. It introduces sharing where none existed. If the common sub-expression is non-deterministic, then we will change the set of results. On the other hand, common sub-expression elimination is fairly innocuous in most other languages.

9.0.1 Future Work

Most curries are made from curry powder and coconut milk, however our Curry was mostly made from low hanging fruit. As nice as our results are, we would like to see this work extended in the future. We believe that a better inliner and strictness analyzer would go a long way to producing even more efficient code.

In fact, a general theory of inlining in Curry would be hugely beneficial. One of the biggest drawbacks to this compiler is that we can not represent lambda expressions in FlatCurry, and inline them. Before we could even attempt this, we would need to know when it is safe to inline a lambda in Curry.

We would also like to move from short-cut Deforestation to Stream Fusion. This should be possible, but it would require a more sophisticated strictness analyzer, and we may not be able to get away with our combinator approach.

We would also like to see the development of new, Curry specific, optimizations. Right now the ? operator acts as a hard barrier. We can move let-bound variables outside of it, but we can not move the choice itself. However, there may be an option for using pull-tabbing or bubbling to move the choice to make room for more optimizations.

For personal reasons we would also like to bootstrap `RICE` with itself. This would significantly decrease the time it takes to compile large Curry programs. Right now, `RICE` is compiled using Pakcs. Currently Kics2 is not a feasible option for compiling `RICE`, because of performance issues with non-deterministic function. So, compiling `RICE` in itself would significantly improve the performance of the compiler.

We would also like to move from C to LLVM. This would allow for more optimizations including Tail Call Optimization. We currently are limited by the recursion depth of the machine, and TCO could allow us to compile more programs. Moving to LLVM would also greatly help in the development of a garbage collector.

Finally, developing a better run-time system would also be an important improvement. While we did work to make sure our run time system was efficient, it could certainly be better. Integrating this work with the Sprite [20] compiler might solve this issue.

9.0.2 Conclusion and Related Work

We have presented the `RICE` Optimizing Curry compiler. The compiler was primarily built to test the effectiveness of various optimizations on Curry programs. While testing these optimizations, we have also built an efficient evaluation method for backtracking Curry programs, as well as a general system for describing and implementing optimizations. The compiler itself is written in Curry.

This system incorporated a lot of work from the functional language community, and the Haskell community in particular. The work on general optimizations [84], Inlining [83], Unboxing [58], Deforestation [40], and the STG-machine [82, 87] were all instrumental in the creation of this compiler, as well as the work by Appel and Peyton-Jones about functional compiler construction [21, 22, 86].

While there has been some work on optimizations for functional-logic programs, there does not seem to be a general theory of optimization. Peemöller and Ramos et al. [80, 89] have developed a theory of partial evaluation for Curry programs, and Moreno [73] has worked on the Fold/Unfold transformation from Logic programming. We hope that our work can help bridge the gap to traditional compiler optimizations.

The implementation of the GAS system was instrumental in developing optimizations for this compiler. It not only allowed us to implement optimizations more efficiently, but also to test new optimizations, and through optimization derivations, discover which optimizations were effective, which were never used, and which were wrong. This greatly simplified debugging optimizations, but it also allowed us to test more complicated optimizations. Often we would just try an idea to see what code it produced, and if it fired in unintended places. It is difficult to overstate just how useful this system was in the compiler.

While the run-time system was not the primary focus of this dissertation, we were able to produce some useful results. The path compression theorem, and the resulting improvement to backtracking, is a significant improvement to the current state-of-the-art for backtracking Curry programs.

When starting this project, Shortcutting was already known to be valid for Inductively Sequential Rewrite Systems. It was developed for them specifically, so it is not too surprising that the idea can be translated to Curry programs. However, it was a nice surprise to find that Unboxing and Deforestation were both valid in Curry. It was even more remarkable that, with some simple restrictions, we could make inlining and reduction valid in Curry as well.

We believe that this work is a good start for optimizing Curry compilers, and we would like to see it continue. After having a taste of optimized Curry, we want to turn up the heat, and deliver an even hotter dish. But for now, we have made a tasty Curry with `RICE`.

Index

References

[1] Synthesizing set functions. 2018.

[2] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 219–233, New York, NY, USA, 1986. ACM.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[4] 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.

[5] F.E. Allen. Program optimization. *Annual Review in Automatic Programming*, 5:239–307, 1969.

[6] F.E. Allen and J. Cocke. *A catalogue of optimizing transformations.* Prentice-Hall, 1972.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[8] A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. The pull-tab transformation. In *Third International Workshop on Graph Computation Models*, Enschede, The Netherlands, October 2010.

[9] S. Antoy, D. Brown, and S. Chiang. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*, pages 61–70, Vienna, Austria, April 2006.

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

[11] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, August 2006. Springer LNCS 4079.

[12] S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proc. of the 11th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

[13] S. Antoy and M. Hanus. Functional Logic Programming. *Comm. of the ACM*, 53(4):74–85, April 2010.

[14] S. Antoy and M. Hanus. Curry: A Tutorial Introduction, January 13, 2017. Available at http://www-ps.informatik.uni-kiel.de/currywiki/documentation/tutorial.

[15] S. Antoy, M. Hanus, A. Jost, and S. Libby. Icurry. *CoRR*, abs/1908.11101, 2019.

[16] S. Antoy and S. Libby. Making Bubbling Practical. In *Proc. of the 27th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2018)*. Springer, 2018.

[17] Sergio Antoy. Definitional trees. In Hélène Kirchner and Giorgio Levi, editors, *Algebraic and Logic Programming*, pages 143–157, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[18] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, July 2000.

[19] Sergio Antoy, Jacob Johannsen, and Steven Libby. Needed Computations Shortcutting Needed Steps. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 183, pages 18–32. Open Publishing Association, 2015.

[20] Sergio Antoy and Andy Jost. A New Functional-Logic Compiler for Curry: Sprite. *CoRR*, abs/1608.04016, 2016.

[21] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, New York, NY, USA, 1997.

[22] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA, 2007.

[23] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[24] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Towards an intermediate language based on Graph Rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, pages 159–175, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

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

[26] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[27] Bernd Brassel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *J. Funct. Log. Program.*, 2004, 2004.

[28] Chandler Carruth. Understanding compiler optimization - chandler carruth - opening keynote meeting c++ 2015. December 2015.

[29] I. Castiñeiras, J. Correas, S. Estévez-Martín, and F. Sáenz-Pérez. TOY: A CFLP Language and System. *The Association for Logic Programming*, 2012.

[30] Jan Christiansen, Daniel Seidel, and Janis Voigtländer. Free theorems for functional logic programs. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*, PLPV '10, page 39–48, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[32] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion. From Lists to Streams to Nothing at All. In *ICFP'07*, 2007.

[33] Haskell B. Curry and Robert Feys. Combinatory logic. volume i. *Journal of Symbolic Logic*, 32(2):267–268, 1967.

[34] Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages 174–188, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

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

[36] Michael Fay. First-order unification in equational theories. pages 161,167, 1979.

[37] A. B. Ferguson and Philip Wadler. When will deforestation stop? In *In 1988 Glasgow Workshop on Functional Programming*, pages 39–56, 1988.

[38] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, 1993.

[39] L. Fribourg. Slog: A logic programming language interpreter based on clausal superposition and rewriting. In *SLP*, 1985.

[40] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[41] Andrew John Gill and Andrew John Gill. Cheap deforestation for non-strict functional languages, 1996.

[42] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep. Dactl: An experimental graph rewriting language. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, pages 378–395, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[43] J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In Hanne Riis Nielson, editor,

Programming Languages and Systems — ESOP '96, pages 156–172, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[44] Jr. Guy Lewis Steele. Debunking the "Expensive Procedure Call" Myth, or Procedure Call Implementations Considered Harmful, or LAMBDA, the Ultimate GOTO". *ACM Conference Proceedings. 1977*, 1977.

[45] Jr Guy Lewis Steele. *RABBIT: A Compiler for SCHEME*. PhD thesis, May 1978.

[46] Jr. Guy Lewis Steele. Compiler Optimization Based on Viewing LAMBDA as RENAME + GOTO. 1980.

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

[48] Michael Hanus. The integration of functions into logic programming: From theory to practice. *The Journal of Logic Programming*, 19-20:583 – 628, 1994.

[49] Michael Hanus and Ramin Sadre. A concurrent implementation of curry in java. In *In Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[50] Michael Hanus and Ramin Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999, 01 1999.

[51] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0), 2016.

[52] M. Hanus (ed.). PAKCS 1.14.3: The Portland Aachen Kiel Curry System, March 04, 2017. Available at http://www.informatik.uni-kiel.de/ pakcs.

[53] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López, José F. Morales, and German Puebla. *An Overview of the Ciao Multiparadigm Language and Program Development Environment and Its Design Philosophy*, pages 209–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[54] G. Huet and J. Lévy. Computations in orthogonal rewriting systems, i. In *Computational Logic - Essays in Honor of Alan Robinson*, 1991.

[55] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[56] Jean-Marie Hullot. Canonical forms and unification. In Wolfgang Bibel and Robert Kowalski, editors, *5th Conference on Automated Deduction Les Arcs, France, July 8–11, 1980*, pages 318–334, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[57] Heinrich Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *The Journal of Logic Programming*, 12(3):237–255, 1992.

[58] SL Peyton Jones, J Launchbury, and Simon Peyton Jones. Unboxed values as first class citizens. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523, pages 636–666. Springer, January 1991.

[59] Stephane Kaplan. Simplifying conditional term rewriting systems : Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295 – 334, 1987.

[60] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[61] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, page 192–203, New York, NY, USA, 2005. Association for Computing Machinery.

[62] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[63] DONALD E. KNUTH and PETER B. BENDIX. Simple word problems in universal algebras††the work reported in this paper was supported in part by the u.s. office of naval research. In JOHN LEECH, editor, *Computational Problems in Abstract Algebra*, pages 263 – 297. Pergamon, 1970.

[64] Ryszard Kubiak, John Hughes, and John Launchbury. Implementing projection-based strictness analysis. In *Functional Programming*, 1991.

[65] Augustsson L. *Compiling Lazy Functional Languages.* PhD thesis, 1978.

[66] 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.

[67] Steven Libby. Rice. `https://github.com/slibby05/rice`.

[68] Steven Libby. Cooking with gas:
making an optimizing curry compiler. 2019.

[69] Steven Libby. *Making Curry with Rice: An Optimizing Compiler for Curry.* PhD thesis, Portland State University, 2022.

[70] Edward S. Lowry and C. W. Medlock. Object Code Optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[71] Wolfgang Lux and Herbert Kuchen. An efficient abstract machine for curry. In Kurt Beiersdörfer, Gregor Engels, and Wilhelm Schäfer, editors, *Informatik'99*, pages 390–399, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[72] Aart Middeldorp. Strategies for rewrite systems: Normalization and optimality. 2018.

[73] Ginés Moreno. Transformation rules and strategies for functional-logic programs. 2002.

[74] 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.

[75] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming*, pages 269–281, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[76] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.

[77] M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43:223, 1942.

[78] Enno Ohlebusch. *Advanced Topics in Term Rewriting.* Springer-Verlag New York, 2002.

[79] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[80] Björn Peemöller. *Normalization and Partial Evaluation of Functional Logic Programs*. PhD thesis, 2016.

[81] Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming*, pages 4–15, September 2004.

[82] Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming*, pages 4–15, September 2004.

[83] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[84] Simon Peyton Jones and Andre Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1), October 1997.

[85] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. *Haskell 2001*, 04 2001.

[86] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[87] Simon L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201. ACM, 1989.

[88] J. Guadalupe Ramos, Josep Silva, and Germán Vidal. An Offline Partial Evaluator for Curry Programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, WCFLP '05, pages 49–53, New York, NY, USA, 2005. ACM.

[89] J. Guadalupe Ramos, Josep Silva, and Germán Vidal. An Offline Partial Evaluator for Curry Programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and*

Functional Logic Programming, WCFLP '05, pages 49–53, New York, NY, USA, 2005. ACM.

[90] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery.

[91] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz.* PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, December 1998.

[92] Ilya Sergey, Simon Peyton Jones, and Dimitrios Vytiniotis. Theory and practice of demand analysis in haskell. Unpublished draft, June 2014.

[93] James R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *J. ACM*, 21(4):622–642, October 1974.

[94] Zoltan Somogyi and Fergus Henderson. The design and implementation of Mercury. *Joint International Conference and Symposium on Logic Programming*, Sep 1996.

[95] E Stoltz, M.P. Gerlek, and M Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. pages 43 – 52, 02 1994.

[96] Andrew Tolmach and Sergio Antoy. A monadic semantics for core curry. *Electronic Notes in Theoretical Computer Science*, 86(3):16–34, 2003. WFLP 2003, 12th International Workshop on Functional and Constraint Logic Programming.

[97] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, page 292–325, jun 1986.

[98] Philip Wadler. Theorems for free! In *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*, pages 347–359. ACM Press, 1989.

[99] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990.

[100] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, apr 1991.

[101] Jia-Huai You. Enumerating outer narrowing derivationsfor constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7(3):319–341, 1989. Unification: Part 1.