Making Curry with Rice

An Optimizing Curry Compiler

Steven Libby

December 28, 2021

Contents

Chapter 1

MATHEMATICAL BACKGROUND

When cooking, it is very important to follow the rules. You don't 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 isn't a single way to compile Curry programs, however we do need to know the rules of the game. Throughout this compiler, we'll be transforming Curry programs in many different ways, and it's important to make sure that all of these transformations respect the rules of Curry. As we'll see, if we break these rules, then we may get some unexpected results.

1.1 REWRITING

In programing language terms, the rules of Curry are its semantics. The semantics of Curry are generally given in terms of rewriting. [30, 7, 12] While there are other semantics [3, 29, 56], rewriting is a good fit for Curry. We'll give a definition of rewrite systems, then we'll 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 Rewrite System (ARS) is a set A along with a relation $\rightarrow$. We write $a \rightarrow b$ instead of $(a, b) \in \rightarrow$, and we have several modifiers on our relation.

- $a \rightarrow^n b$ iff $a = x_0 \rightarrow x_1 \rightarrow \ldots x_n = b$.
- $a \rightarrow^{\leq n} b$ iff $a \rightarrow^i b$ and $i \leqslant n$.
- reflexive closure: $a \rightarrow^= b$ iff $a = b$ or $a \rightarrow b$.
- symmetric closure: $a \leftrightarrow b$ iff $a \rightarrow b$ or $b \rightarrow a$.

	$(x \cdot x + 1)(2 + x)$	
$\rightarrow$	$(x \cdot x + 1)(x + 2)$	by commutativity of addition
$\rightarrow$	$(x^2 + 1)(x + 2)$	by definition of x^2
$\rightarrow$	$x^2 \cdot x + 2 \cdot x^2 + 1 \cdot x + 1 \cdot 2$	by FOIL
$\rightarrow$	$x^2 \cdot x + 2x^2 + x + 2$	by identity of multiplication
$\rightarrow$	$x^3 + 2x^2 + x + 2$	by definition of x^3

Figure 1.1: reducing $(x \cdot x + 1)(2 + x)$ using the standard rules of algebra

- transitive closure: $a \rightarrow^+ b$ iff $\exists n \in \mathbb{N}. a \rightarrow^n b$.
- reflexive transitive closure: $a \rightarrow^* b$ iff $a \rightarrow^= b$ or $a \rightarrow^+ b$.
- rewrite derivation: a sequence of rewrite steps $a_0 \rightarrow a_1 \rightarrow \ldots a_n$.
- a is in *normal form* if no rewrite rules can apply.

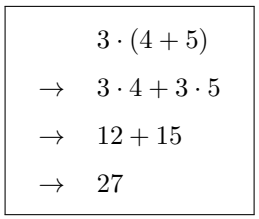
A rewrite system is meant to invoke the feeling of algebra. In fact, rewrite system are much more general, but they can still retain the feeling. If we have an expression $(x \cdot x + 1)(2 + x)$, we might reduce this with the reduction in figure 4.7.

We can conclude that $(x \cdot x + 1)(x + 2) \rightarrow^+ x^3 + 2x^2 + x + 2$. This idea of rewriting invokes the feel of algebraic rules. The mechanical process of rewriting allows for a straightforward implementation on a computer. Therefore, it shouldn't be surprising that most systems have a straightforward translation to rewrite systems.

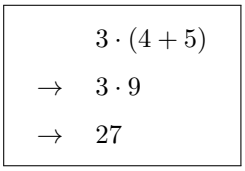
It's worth understanding the properties and limitations of these rewrite systems. Traditionally there are two important questions to answer about any rewrite system. Is it *confluent*? Is it *terminating*?

A confluent system is a system where the order of the rewrites doesn't 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 1.2.

A terminating system will always halt. That means that eventually there are no rules that can be applied. Distributivity is terminating, whereas commutativity is not terminating. See figure 1.3.

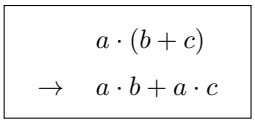


(a) distributing first



(b) reducing $4 + 5$ first

Figure 1.2: Two possible reductions of $3 \cdot (4 + 5)$. Because they both can rewrite to 27, this is a confluent system.



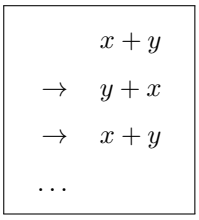


Figure 1.3: A system with a single rule for distribution is terminating, but any system with a commutative rule is not. Note that $x + y \rightarrow^2 x + y$

Confluence and termination are important topics in rewriting, but we will largely ignore them. After all, Curry is neither confluent nor terminating. However, there will be a few cases where these concepts will be important. For example, if our optimizer isn't terminating, then we'll 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, the purpose of term rewriting is to transform 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. 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's a signature $f^{(n)}$ with children $t_1, t_2, \ldots t_n$, where $t_1, t_2, \ldots t_n$ 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.

This inductive definition gives us a tree structure for terms. As an example consider Peano arithmetic $\Sigma = \{+^2, *^2, -^2, <^2, 0^0, S^1, \top^0, \bot^0\}$. 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.

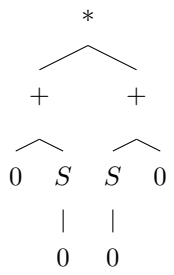


Figure 1.4: Tree representation of the term $*(+(0, S(0)), +(S(0), 0))$.

A *child* c of term $t = f(t_1, t_2, \ldots t_n)$ is one of $t_1, t_2, \ldots t_n$. A *subterm* s of t is either t itself, or it is a subterm of a child of t. We write $s = t|_{[i_1, i_2, \ldots i_n]}$ 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 $t|_{[i_1, i_2, \ldots i_n]} = t_{i_1}|_{[i_2, \ldots i_n]}$, which matches our definition for subterm. We call $[i_1, i_2, \ldots i_n]$ the *path* from t to s. 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 $t[p \leftarrow r]$ to denote replacing subterm $t|_p$ with r. Algorithmically we can define this as in figure **??**

In our above example $t = *(+(0, S(0), +(S(0), 0)))$, We can compute the rewrite $t[[0, 1] \leftarrow *(S(0), S(0))]$, and we get the term $*(+(0, *(S(0), S(0))), +(S(0), 0))$, with the tree in figure 1.6.

A substitution replaces variables with terms. Formally, a *substitution* is a mapping from $\sigma{:}V \rightarrow T(\Sigma, V)$. We write $\sigma = \{v_1 \mapsto t_1, \ldots v_n \mapsto t_n\}$ to denote the substitution where $s(v_i) = t_i$

$t[\epsilon \leftarrow r] = r$

$f(t_1, \ldots t_i, \ldots t_n)[i{:}p \leftarrow r] = f(t_1, \ldots t_i[p \leftarrow r], \ldots t_n)$

Figure 1.5: algorithm for finding a subterm of t.

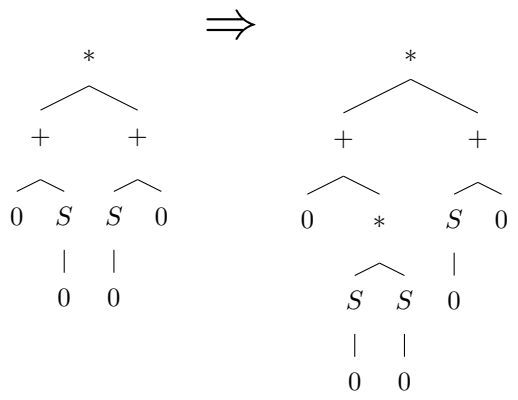


Figure 1.6: The result of the computation $t[[0,1] \leftarrow S(0)]$

for $i \in \{1 \ldots n\}$, and $s(v) = v$ otherwise. We can uniquely extend σ to a function on terms by figure 1.7

$\sigma'(v) = \sigma(v)$

$\sigma'(f(t_1, \ldots t_n) = f(\sigma'(t_1) \ldots \sigma'(t_n))$

Figure 1.7: Algorithm for applying a substitution.

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 $t_1 = \sigma(t_2)$, Two terms t_1 and t_2 *unify* if there exists some substitution σ such that $\sigma(t_1) = \sigma(t_2)$. 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, $v \leq u$. Mgu's are unique up to renaming of variables. That is, if u_1 and u_2 are mgu's 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 $l \to r$ is a pair of terms. However this time we require that $Var(r) \subseteq Var(l)$, and that $l \notin V$. A Term Rewrite 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 $l \to r$, we say that t rewrites to s if, l matches $t|_p$ with matcher σ, and $t[p \leftarrow \sigma(r)] = s$. 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 $l \to r$. A rule is left (right) linear if $l(r)$ is linear. 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.

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. Two rules $l_1 \to r_1$ and $l_2 \to r_2$ if l_1 and l_2 overlap. A rewrite system is overlapping if any two rules overlap. Otherwise it's non-overlapping. Any non-overlapping left linear system is *orthogonal*. It is well known that all orthogonal TRS's are confluent. [47]

As an example, in figure 1.8 examples (b) and (c) both overlap. It's clear that these systems aren't confluent, but non-confluence can arise in more subtle ways. The converse to theorem **??** isn't true. There can be overlapping systems which are confluent.

$$\boxed{\begin{array}{l} g(0, y) \to 0 \\ g(1, y) \to 1 \end{array}}$$

(a) A non-overlapping system

$$\boxed{\begin{array}{l} g(0, y) \to 0 \\ g(x, 1) \to 1 \end{array}}$$

(b) A system that overlaps at the root

$$\boxed{\begin{array}{l} f(g(x, y)) \to 0 \\ g(x, y) \to 1 \end{array}}$$

(c) A system that overlaps at a subterm

Figure 1.8: Three TRSs demonstrating how rules can overlap. In (a) they don't overlap at all, In (b) both rules overlap at the root, and in (c) rule 2 overlaps with a subterm of rule 1.

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.

$$
\begin{array}{lcll}
R_1 & : & 0 + y & \to & y \\
R_2 & : & S(x) + y & \to & S(x + y) \\
R_3 & : & 0 * y & \to & 0 \\
R_4 & : & S(x) * y & \to & y + (x * y) \\
R_5 & : & 0 - y & \to & 0 \\
R_6 & : & S(x) - 0 & \to & S(x) \\
R_7 & : & S(x) - S(y) & \to & x - y \\
R_8 & : & 0 \leq y & \to & \top \\
R_9 & : & S(x) \leq 0 & \to & \bot \\
R_{10} & : & S(x) \leq S(y) & \to & x < y \\
R_{11} & : & 0 = 0 & \to & \top \\
R_{12} & : & S(x) = 0 & \to & \bot \\
R_{13} & : & 0 = S(y) & \to & \bot \\
R_{14} & : & S(x) = S(y) & \to & x = y
\end{array}
$$

Figure 1.9: The rewrite rules for Peano Arithmetic with addition, multiplicaton, subtraction, and comparison. All operators use infix notation.

The two sets are $C = \{0, S, \top, \bot\}$ 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. W e will do something similar to implement higher order functions. This means that Curry programs won't 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 don't unify, then they cannot overlap. We don't need to check if subterms overlap. Furthermore, any term that consists entirely of constructors and variables is in normal form. For this reason, it's not surprising that

$$
\begin{array}{lcl}
Ap(I, x) & \rightarrow & x \\
Ap(Ap(K, x), y) & \rightarrow & x \\
Ap(Ap(Ap(S, x), y), z) & \rightarrow & Ap(Ap(x, z), Ap(y, z))x
\end{array}
$$

Figure 1.10: The SKI system from combinatorial logic.

$$
\begin{array}{lcccl}
gcd(x, x) & & & \rightarrow & x \\
gcd(x, y) & | & y \leq x & \rightarrow & gcd(x - y, y) \\
gcd(x, y) & | & x \leq y & \rightarrow & gcd(x, y - x)
\end{array}
$$

Figure 1.11: Conditional rewrite system for computing greatest common divisor.

most functional languages are based on constructor systems.

Finally, we can introduce conditions to rewriting systems. We introduce a new symbol $\top$ to the rewrite system's alphabet Σ. A conditional rewrite rule is a rule $l|c \rightarrow r$ where l, c, and r are terms. A term t conditionally rewrites to s with rule $l|c \rightarrow r$ if there exists a path p and substitution σ such that $t_p = \sigma(l)$, $\sigma(c) \rightarrow^* \top$, and $s = \sigma(r)$.

The idea is actually a pretty simple extension. In order to rewrite a term, we must satisfy a condition. If the condition is true, then the rule is applied. In order to simplify the semantics of this system, we determine if a condition is true by rewriting it to the value $\top$. Figure 1.11 gives an example of a conditional rewrite system for computing greatest common divisor. It uses the rule defined in 1.9.

While most treatments of conditional rewriting [30, 37] define a condition as a pair $s = t$ where s and t mutually rewrite to each other, We chose this definition because it's closer to the definition of Curry, where the condition must reduce to the boolean value `True` for the rule to apply.

Curry uses conditional rewriting extensively, and efficiently evaluating conditional rewrite systems is the core problem in most functional logic languages. The solution to this problem comes from the theory of narrowing.

1.3 NARROWING

Narrowing was originally developed to solve the problem of semantic unification. The goal was, given a set of equations $E = \{a_1 = b_2, a_2 = b_2, \ldots a_n = b_n\}$ how do you solve the $t_1 = t_2$ 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 1.12.

$$
\begin{aligned}
&\sigma(*(1, +(x, 3))) && = \\
&*(1, +(+(4, 5), 3)) && = \\
&+(*(1, +(4, 5)), *(1, 3)) && = \\
&+(+(*(1, 4), *(1, 5)), *(1, 3)) && = \\
&\sigma(+(+(*(1, 4), *(y, 5)), *(z, 3)))
\end{aligned}
$$

Figure 1.12: Derivation of $*(1, +(x, 3)) = +(+(*(1, 4), *(y, 5)), *(z, 3))$ with $\sigma = \{x \mapsto +(4, 5), y \mapsto 1, z \mapsto 1\}$.

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. [39] 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. 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 $t[p \leftarrow \sigma(r)] = s$. We write $t \rightsquigarrow_{p,l\to r,\sigma} s$. We may write $t \rightsquigarrow_\sigma s$ 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 gives us a way to solve equations with a rewrite system, but for our purposes it's more important that narrowing allows us to rewrite terms with free variables.

At this point, rewrite systems are a nice curiosity, but they are completely impractical. This is because we don't have a plan for solving 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 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'd be sure to finish. This is the best possible case for rewrite systems, and even this is still terribly inefficient. 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 find a normal form for a 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 1.10 $Ap(Ap(K,I),Ap(Ap(S,Ap(I,I)),Ap(S,Ap(I,I))))$. 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.

A *Rewriting Strategy* $\mathcal{S}$:$T(\Sigma,V) \rightarrow Pos$ is a function from terms to positions, such that for any term t, $\mathcal{S}(t)$ is a redex. The idea is that $S(t)$ should give us a position to rewrite, and we find the rule that matches $S(t)$.

For orthogonal rewriting systems, there are two common rewriting strategies, innermost and outermost rewriting. 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 strategies correspond roughly to lazy evaluation. We rewrite the highest possible term that matches a rewrite rule.

A strategy is normalizing if a term t has a normal form, then the strategy will eventually find it. While outermost rewriting isn't normalizing in general, it is for a large subclass of orthogonal rewrite systems. This matches the intuition from programming languages. Lazy languages can perform computations that would loop 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 as a term is rewritten. Specifically for a redex s that

is a subterm of t, how can s change as t is being rewritten. If we're 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.
Case 2: s is either above $t|_p$, or they are completely disjoint. In this case s doesn't 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 1.13 We can formalize this with the notion of descendants with the following definition.

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

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

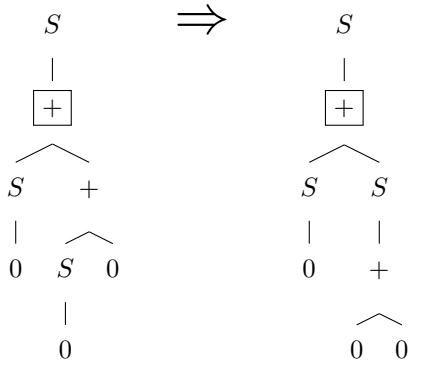
This definition extends to derivation $t \to_{A_1} t_1 \to_{A_2} t_2 \to_{A_2} \ldots \to_{A_n}, t_{n+1}$. $Des(s, A_1, A_2 \ldots A_n) = \bigcup_{s' \in Des(s,A_1)} Des(s', A_2, \ldots A_n)$.

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.

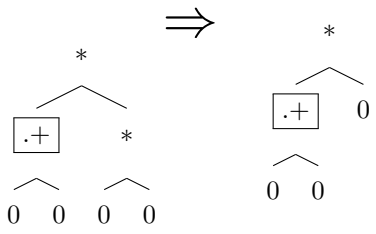
Definition 1.4.2. Needed: A redex $s \leq t$ is *needed* if, for every derivation of t to a normal form, a descendant of s is the root of a rewrite.

This definition is good because it's immediately clear that, if we're 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.

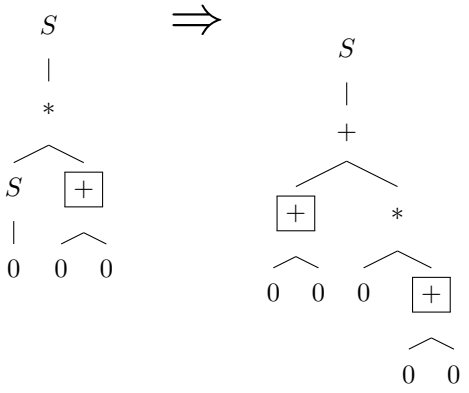
Theorem 1. *For an orthogonal TRS, any term that is not in normal form contains a needed redex. Furthermore, a rewrite strategy that rewrites only needed redexes is normalizing.*



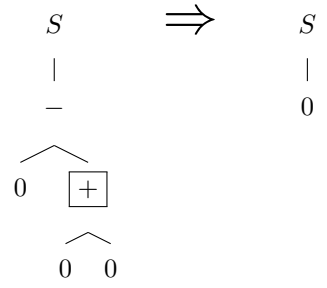
(a) rewrite R_1 at position $[0, 1]$ doesn't affect $t|_{[0]}$.



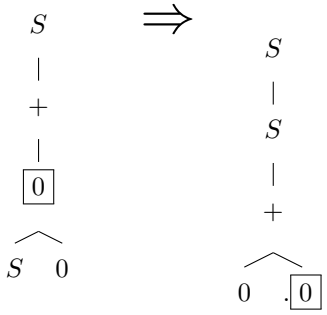
(b) rewrite R_4 at position [1] doesn't affect $t|_{[0]}$.



(c) rewrite R_3 at position [0] duplicates $t|_{[}0, 1]$.



(d) rewrite R_5 at position [0] erases term at $t|_{[}0, 1]$.



(e) rewrite R_2 at position [0] moves $t|_{[}0, 1]$ to position $[0, 0, 1]$.

Figure 1.13: four cases for the descendants for a term after a single rewrite. The boxed term is either left alone, duplicated, or erased, or moved.

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's 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.

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, $\sigma(v_i)$ 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'll 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 subterms.

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 [34], innermost [27], outermost [57], standard [25], and lazy [43]. Unfortunately, each of these strategies are too restrictive on the rewrite systems they allow.

$(x + x) + x = 0$

(a) This fails for eager narrowing, because evaluating $x + x$ can produce infinitely many answers. However This is fine for lazy narrowing. We will get $(0 + 0) + 0 = 0, \{x = 0\}$ or $S(S(y + S(y)) + S(y)) = 0\{x = S(y)\}$ and the second one will fail.

$x \leq y + y$

(b) With a lazy narrowing strategy we may end up computing more than is necessary. If x is instantiated to 0, then we don't need to evaluate $y + y$ at all.

Fortunately there exists a narrowing strategy that's 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 straightforward. Since we are working with constructor rewrite systems, we can group all of the rules defined for the same function symbol together. We'll 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.

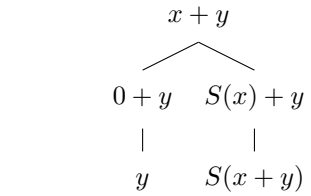
Definition 1.5.1. T is a *partial definitional tree* if T is one of the following.
$T = exempt(\pi)$ where π is a pattern.
$T = leaf(\pi \to r)$ where π is a pattern, and $\pi \to r$ is a rewrite rule.
$T = branch(\pi, o, T_1, \ldots T_k)$, where π is a pattern, o is a path, $\pi|_o$ is a variable, $c_1,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

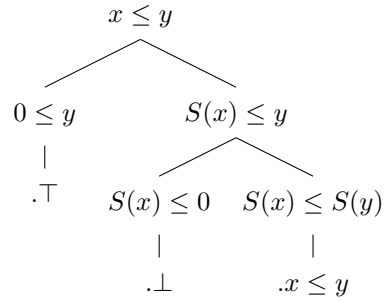
The name "inductively sequential" is justified because there is a narrowing strategy that only reduces needed redexes for any of these systems. This definition can be difficult to follow mathematically, but it is usually much easier to understand with a few examples. In figure 1.15 we show the definitional 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.

The extension from rewriting to narrowing has two complications. The first is that we need to compute a substitution. This is pretty straightforward, but it leads to the second complication. What does it mean 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 mgu's, which are unique up to renaming of variables. However, this turned out to be the wrong approach. Computing mgu's 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 $x \leq y + z \rightsquigarrow_{\epsilon, R_8, \{x \mapsto 0\}} \top$. This step never needs to compute a substitution for y. Therefore we need a definition that isn't dependent on substitutions that might be computed

$$
\begin{array}{lcl}
0 + y & \to & y \\
S(x) + y & \to & S(x + y)
\end{array}
$$



$$
\begin{array}{lcl}
0 \leq y & \to & \top \\
S(x) \leq 0 & \to & \bot \\
S(x) \leq S(y) & \to & x < y
\end{array}
$$



$$
\begin{array}{lcl}
0 = 0 & \to & \top \\
S(x) = 0 & \to & \bot \\
0 = S(y) & \to & \bot \\
S(x) = S(y) & \to & x = y
\end{array}
$$

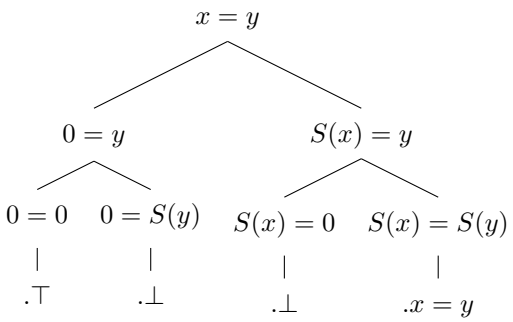


Figure 1.15: Definitional trees for $x + y$, $x \leq y$, and $x = y$.

later.

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

Here we don't require that σ be an mgu, but, for any less general substitution, it must be the case that we're rewriting a needed redex. So our example, $x \leq y + z \leadsto_{2\dot{\epsilon},R_1,\{y\mapsto 0\}} x \leq z$, isn't a needed narrowing step because $x \leq y + z \leadsto_{2\dot{\epsilon},R_1,\{x\mapsto 0,y\mapsto 0\}} 0 \leq z$, Isn't a needed rewriting step.

Unfortunately, this definition raises a new problem. Since we are no longer using mgu's for our unifiers, we may not have a unique step for an expression. For example, $x < y \leadsto_{\epsilon,R_8,\{x\mapsto 0\}} \top$, and $x < y \leadsto_{\epsilon,R_9,\{x\to S(u),t\mapsto S(v)\}} u \leq v$ 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 e be an expression rooted by function symbol f. Let T be the definitional tree for f.

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

The function λ simply traverses the definitional tree for a function symbol. 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're looking at isn't a constructor, then we need to narrow that subterm first.

Theorem 2. *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, this says nothing about the underlying completeness of the rewrite system we're narrowing. We may still have non-terminating derivations.

This needed narrowing strategy is the underlying mechanism for evaluating 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 [1] 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 DAG's. 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

[1] Some authors will use walk and tour and reserve path and cycle for the cases where there are no repeated vertices. This distinction isn't relevant for our work.

a rooted graph G with nodes in N where each node n has a label in $\Sigma \cup V$. We'll 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 doesn't 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 will be implemented using pointers. 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. The second is that this representation corresponds closely to the representation in a computer. The notation these graphs is given by the grammar

Graph $\rightarrow$ Node

Node $\rightarrow$ n:L(Node, ... Node) | n

We start with the root node, and for each node in the graph, If we haven't 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 doesn't have any successors, then we'll omit the parentheses entirely, and just write down the label.

A few examples are shown in figure 1.16. 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 don't 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 $g[p \leftarrow u]$) is given by the

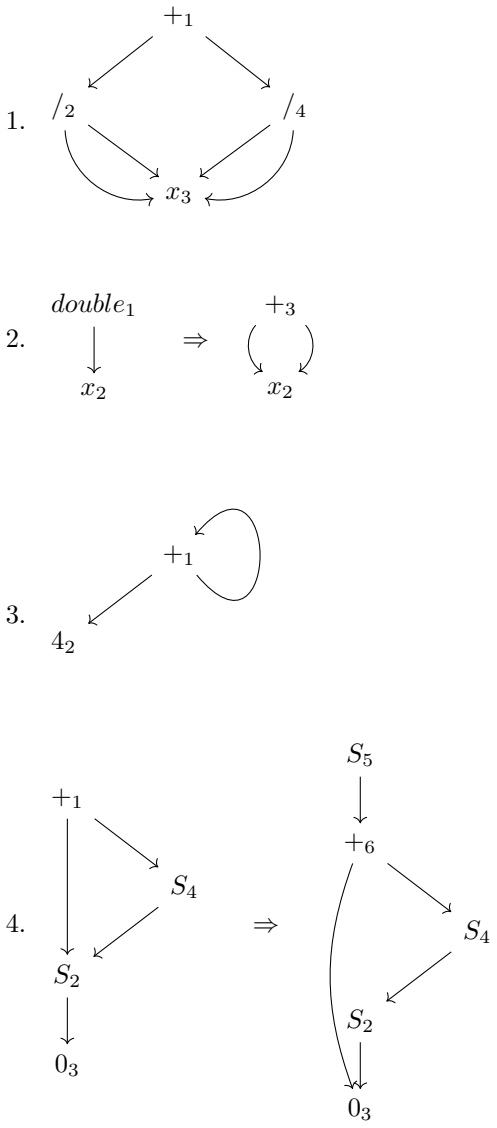


Figure 1.16: 1. $1{:}+(2{:}/(3{:}x,3),4{:}/(3,3))$,

2. $1{:}double(2{:}x) \Rightarrow 3{:}+(2{:}x,2)$

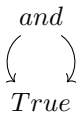
3. $1{:}+(2{:}4,1)$

4. $1{:}+(2{:}S(3{:}0),4{:}S(2)) \Rightarrow 5{:}S(6{:}+(3{:}0),4{:}S(2{:}S(3)))$

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 don't 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 needed to be the same, 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 doesn't 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, but they can be found in Echaned and Janodet [24]. 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 [38], and Baader and Nipkow [17]. The notation and ideas in this section largely come from Ohlebusch [47], although they are very similar to the previous two summaries. The foundations of term rewriting were laid by Church, Rosser, Curry, Feys, Newman. [22, 23, 46] Most of the work on rewriting has centered on confluence and termination. [38] Narrowing has been developed by Slagle [55]. Sequential strategies were developed by Huet and Levy [33], who gave a decidable criteria for a subset of sequential systems. This led to the work of Antoy on inductively sequential systems [11]. The needed narrowing strategy came from Hanus, Antoy, and Echahed [12]. Graph rewriting is a bit more disconnected. Currently there isn't a consensus on how to represent graphs mathematically. We went with the presentation in [24], but there are also alternatives in [38, 17, 47]

Here we saw how we can rewrite terms and graphs. We'll 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's time to find out how to make Curry!

Chapter 2

THE CURRY LANGUAGE

2.1 THE CURRY LANGUAGE

In order to write a compiler for Curry, we need to understand how Curry works. We'll start by looking at some examples of Curry programs. We'll see how Curry programs differ from Haskell and Prolog programs. Then we'll move on to defining a small interpreter for Curry. Finally we'll 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 [8]. For an exhaustive explanation of the syntax and semantics of Curry see [31].

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's 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's not the most elegant way, but I chose it for two reasons. There is no syntactic sugar and no libraries hiding any of the computations, and the algorithm for permuting a list is similar to the algorithm we will use in Curry.

$$
\begin{aligned}
&\mathit{perms} && :: [a] \to [[a]] \\
&\mathit{perms}\ [\,] && = [[\,]] \\
&\mathit{perms}\ (x : xs) && = \mathit{concat}\ (\mathit{map}\ (\mathit{insert}\ x)\ (\mathit{perms}\ xs)) \\
&\quad \textbf{where}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert}\ x\ [\,] &&= [[x]]\\
&\mathit{insert}\ x\ (y : ys) &&= (x : y : ys) : \mathit{map}\ (y:)\ (\mathit{insert}\ x\ ys)
\end{aligned}
$$

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: $\mathit{insert}\ 1\ [2,3]$ 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's 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 doesn't 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:

$$
\begin{aligned}
&x\ ?\ y = x\\
&x\ ?\ y = y
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{perm} &&:: [a] \to [a]\\
&\mathit{perm}\ [\,] &&= [\,]\\
&\mathit{perm}\ (x : xs) &&= \mathit{insert}\ x\ (\mathit{perm}\ xs)\\
&\quad \textbf{where}\\
&\quad\ \mathit{insert}\ x\ [\,] &&= [x]\\
&\quad\ \mathit{insert}\ x\ (y : ys) &&= x : y : ys\ ?\ y : \mathit{insert}\ x\ ys
\end{aligned}
$$

In many cases functions that return multiple results can lead to much simpler code. Curry has another feature that's just as useful. We can declare a *free variable* in Curry. This is a variable that hasn't 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're 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.

$$
\begin{aligned}
&perm \quad :: [a] \to [a] \\
&perm\ [\,] = [\,] \\
&perm\ xs \\
&\quad \mid xs \equiv (begin +\!\!+ [x] +\!\!+ end) = x : perm\ (begin +\!\!+ end) \\
&\quad \textbf{where}\ begin, x, end\ \textbf{free}
\end{aligned}
$$

Look at that. We've 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.

$$
\begin{aligned}
&perm && :: [a] \to [a] \\
&perm\ [\,] && = [\,] \\
&perm\ (begin +\!\!+ [x] +\!\!+ end) && = x : perm\ (begin +\!\!+ end)
\end{aligned}
$$

Now the real work of our algorithm is a single line. Even better, it's 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've already seen it, but a more explicit example would be helpful. We've 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 don't need, or even want, every permutation. So, how do we filter out the permutations we don't 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's sorted.

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

$$
\begin{aligned}
&sort :: (Ord\ a) \Rightarrow [a] \to [a] \\
&sort\ xs \mid sorted\ ys = ys \\
&\quad \textbf{where} \\
&\qquad ys = perm\ xs \\
&\qquad sorted\ [\,] \qquad\quad = True \\
&\qquad sorted\ [x] \qquad\quad = True \\
&\qquad sorted\ (x : y : ys) = x \leqslant y \wedge sorted\ (y : ys)
\end{aligned}
$$

In this example every permutation of xs that isn't sorted will fail in the guard. Once an expression has failed, computation on it stops, and other alternatives are tried. As we'll 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 for programming, we need to understand how they work if we are going to implement them in a compiler.

2.2 SEMANTICS

As we've 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 is a Limited Overlapping Inductively Sequential (LOIS) Rewrite system. Haskell, on the other hand, requires all rules to be non-overlapping.

To see the difference consider the usual definition of factorial.

$$
\begin{aligned}
&fac :: Int \to Int \\
&fac\ 0 = 1 \\
&fac\ n = n * fac\ (n - 1)
\end{aligned}
$$

This seems like an innocuous Haskell program, however It's 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.

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

Here $\neg\ p_i$ means that we don't 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 $e \to^* \{ v_1, v_2 \}$.

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

$$coin = 0 \,?\, 1$$
$$double\ x = x + x$$

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.

$$double\ coin \Rightarrow coin + coin \Rightarrow (0 \,?\, 1) + (0 \,?\, 1) \Rightarrow 0 + (0 \,?\, 1) \Rightarrow 0 + 1 \Rightarrow 1$$

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. [35].

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 use graph rewriting instead of term rewriting. 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 Flat Curry

One of the earliest steps in the compilation process is to form definitional trees out of Curry functions. These trees are then turned into an intermediate representation where each branch of the Tree is replaced by a case expression. This IR is called FlatCurry [3], and we will be working exclusively with FlatCurry Programs. FlatCurry is a simple language to represent Curry programs. All syntactic sugar has been removed, and we are left with a language similar to Haskell's Core. The syntax is given in figure 2.2. It has been modified from the original in two ways. First, the original relied on transforming free variables into non-determinism. Since I do not use that transformation in my compiler, I represent free variables explicitly with a **let** ... **free** expression. The second change is a little more substantial. I've added an expression $\bot$ to represent failure. This comes with the assumption no **case** expressions have missing branches. While this is not common in Curry compilers, It's easy enough to enforce, and leads to an easier implementation. It also allows for more optimizations. An example of the *fac* function in both Curry and FlatCurry is given in figure 2.1

Curry

$$fac\ 0 = 1$$
$$fac\ n = n * fac\ (n - 1)$$

FlatCurry

$$fac\ v_1 = (\textbf{case}\ v_1\ \textbf{of}\ 0 \to 1)\ ?\ (v_1 * fac\ (v_1 - 1))$$

Figure 2.1: the factorial function in Curry and FlatCurry

2.2.2 Evaluation

We'll start off with a small interpreter for a first order functional language. Then we'll make incremental improvements until we have all of the features of Curry. It's important to start here, because each time we add a feature, it may interact with features that came before.

Let's look at the first interpreter. The goal is to rewrite a term in our language to normal form. In this language, a normal form is simple. It can consist of Constructors, Literals, and nothing else.

$$\begin{array}{lll} n \Rightarrow l & & \textit{literal} \\ \quad \mid\ C_k\ n_1 \ldots n_k & & \textit{constructor} \end{array}$$

In a lazy language, to compute an expression to normal form, we first compute head normal form. Head normal form is just the restriction that the root of the expression must be a constructor or literal. That is, the *head* is in normal form.

So the algorithm for computing an expression to normal form is

$$\begin{array}{l} nf :: Expr \to Expr \\ nf\ e = \textbf{case}\ hnf\ e\ \textbf{of} \\ \qquad\qquad C\ e_1\ e_2 \ldots e_n \to C\ (nf\ e_1)\ (nf\ e_2) \ldots (nf\ e_n) \\ \qquad\qquad l \qquad\qquad\quad \to l \end{array}$$

We can build a simple interpreter for a first order language by giving the *hnf* function.

$$\begin{array}{ll} hnf :: Expr \to Expr & \\ hnf\ [\![l]\!] & = l \end{array}$$

$$
\begin{array}{lll}
f \Rightarrow f\ v_1\ v_2 \ldots v_n = e & & \\
e \Rightarrow v & & \textit{Variable} \\
\quad \mid\ l & & \textit{Literal} \\
\quad \mid\ e :: t & & \textit{Typed} \\
\quad \mid\ e_1\ ?\ e_2 & & \textit{Choice} \\
\quad \mid\ \bot & & \textit{Failed} \\
\quad \mid\ f_k\ e_1\ e_2 \ldots e_n & & \textit{Function Application} \\
\quad \mid\ C_k\ e_1\ e_2 \ldots e_n & & \textit{Constructor Application} \\
\quad \mid\ \textbf{let}\ v_1 = e_2 \ldots v_n = e_n\ \textbf{in}\ e & & \textit{Variable Declaration} \\
\quad \mid\ \textbf{let}\ v_1, v_2, \ldots v_n\ \textbf{free in}\ e & & \textit{Free Variable Declaration} \\
\quad \mid\ \textbf{case}\ e\ \textbf{of}\ \{ p_1 \rightarrow e_1; \ldots p_n \rightarrow e_n \} & & \textit{Case Expression} \\
p \Rightarrow C\ v_1\ v_2 \ldots v_n & & \textit{Constructor Pattern} \\
\quad \mid\ l & & \textit{Literal Pattern}
\end{array}
$$

Figure 2.2: FlatCurry This is largely the same as other presentations [3, 9] but we have elected to add more information that will become relevant for optimizations later.

$$
\begin{aligned}
&\mathit{hnf}\ [\![e :: t]\!] && = \mathit{hnf}\ e \\
&\mathit{hnf}\ [\![C\ e_1 \ldots e_n]\!] && = C\ e_1 \ldots e_n \\
&\mathit{hnf}\ [\![f\ e_1 \ldots e_n]\!] && = \mathbf{let}\ v_1 \ldots v_n = \mathit{vars}\ f \\
&&&\qquad e_f = B_{\mathcal{F}}\ f \\
&&&\qquad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
&&&\ \mathbf{in}\ \ \mathit{hnf}\ (\sigma\ e_f) \\
&\mathit{hnf}\ [\![\ \mathbf{let}\ v_1 = e_1\ \mathbf{in}\ e]\!] && = \mathbf{let}\ \sigma = \{\, v_1 \to e_1 \,\} \\
&&&\ \mathbf{in}\ \ \mathit{hnf}\ (\sigma\ e) \\
&\mathit{hnf}\ [\![\ \mathbf{case}\ e\ \mathbf{of}\ \mathit{bs}]\!] \\
&\quad \mid \mathit{hnf}\ [\![e]\!] = l && = \mathbf{let}\ (l \to e) \in \mathit{bs} \\
&&&\ \mathbf{in}\ \ \mathit{hnf}\ e \\
&\quad \mid \mathit{hnf}\ [\![e]\!] = (C\ e_1 \ldots e_n) && = \mathbf{let}\ (C\ v_1 \ldots v_n \to e) \in \mathit{bs} \\
&&&\qquad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
&&&\ \mathbf{in}\ \ \mathit{hnf}\ (\sigma\ e)
\end{aligned}
$$

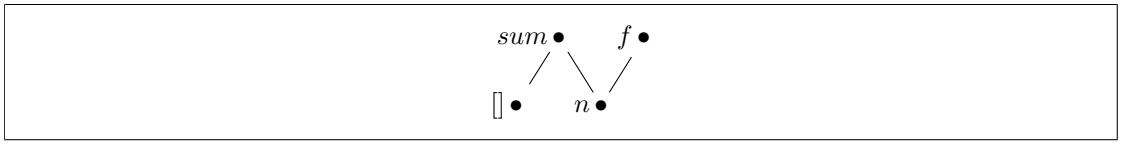
Notice that we don't need to interpret a variable, since they will all be replaced by substitutions. While this interpreter is compact, it suffers from a number of problems. One is that we can't handle recursive definitions in let expressions. Recursive functions are allowed, but a recursive let expression will crash the interpreter. The second problem is that we aren't actually using lazy evaluation. Since we may copy terms by performing a substitution, we may reevaluate the same expression multiple times.

We can get around this by moving to graph rewriting. Surprisingly, not much changes for the interpreter. The only real change is that all expressions are graphs, and make substitution work by pointer redirection. Now our interpreter is lazy and will work correctly with recursive let bindings. Even mutually recursive let bindings are still fine. This also solves the problem of enforcing call time choice semantics.

There is one issue with graph rewriting that requires a little more care. A *collapsing rule* is a function that returns a single variable. A simple example is the *id* function, but collapsing rules can be much more complicated as shown by the following *sum* function.

$$
\begin{aligned}
&\mathit{sum}\ \mathit{xs}\ \mathit{acc} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad\qquad [\,] \qquad\ \to \mathit{acc} \\
&\qquad\qquad (y : \mathit{ys}) \to \mathit{sum}\ \mathit{ys}\ (y + \mathit{acc})
\end{aligned}
$$

The first branch of the case statement here is collapsing. Collapsing rules will cause several problems throughout this compiler, but the first one we need to deal with is sharing. Suppose we have the following expression graph:



Now, if we reduce *sum* [ ] n to n, then we have a problem. Do we overwrite the *sum* node with the value of n? This seems like it would be a problem. After all we'd need to copy the expression, which was the very thing that graph rewriting was supposed to help us avoid. However, there's another possibility. According to our evaluation strategy we must evaluate n to head normal form. So, we can evaluate n, and then copy the value of the constructor over to the *sum* node. This is the strategy use by GHC [1].

Unfortunately, this strategy of copying the constructor is also going to fail. The problem here is non-determinism. The ? operator is a non-deterministic collapsing functions that is used heavily throughout Curry. We will justify why copying can't work in the next section, but for now we can find a solution using forwarding nodes. A forwarding node is very simple. It's just a node with a single child that we represent as $(FORWARD\ e)$. We can think of forwarding nodes like references in other languages. Now, we expand the interpreter by replacing every collapsing rule with a forwarding node. For example, the *sum* function now becomes:

$$
\begin{aligned}
&sum\ xs\ acc = \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad [\,] \qquad\ \rightarrow FORWARD\ acc \\
&\qquad\qquad (y : ys) \rightarrow sum\ ys\ (y + acc)
\end{aligned}
$$

With this we can extend our interpreter to graph rewriting.

$$
\begin{aligned}
&hnf :: Expr \rightarrow Expr \\
&hnf\ [\![l]\!] &&= l \\
&hnf\ [\![e :: t]\!] &&= FORWARD\ (hnf\ [\![e]\!]) \\
&hnf\ [\![C\ e_1 \ldots e_n]\!] &&= C\ e_1 \ldots e_n \\
&hnf\ [\![f\ e_1 \ldots e_n]\!] &&= \textbf{let}\ v_1 \ldots v_n = vars\ f \\
&&&\qquad e_f = B_{\mathcal{F}}\ f \\
&&&\qquad \sigma = \{\, v_1 \rightarrow e_1, \ldots v_n \rightarrow e_n \,\}
\end{aligned}
$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{hnf}\ (\sigma\ e_f)\\
&\mathit{hnf}\ [\![\mathit{FORWARD}\ e]\!] \qquad = \mathit{FORWARD}\ (\mathit{hnf}\ [\![e]\!])\\
&\mathit{hnf}\ [\![\ \textbf{let}\ v_1 = e_1\ \textbf{in}\ e]\!] \quad = \textbf{let}\ \sigma = \{\, v_1 \to e_1 \,\}\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{hnf}\ (\sigma\ e)\\
&\mathit{hnf}\ (\textbf{case}\ e\ \textbf{of}\ bs)\\
&\quad \mid e \equiv [\![\mathit{FORWARD}\ e']\!] = \mathit{FORWARD}\ (\mathit{hnf}\ [\![\ \textbf{case}\ e'\ \textbf{of}\ bs]\!])\\
&\quad \mid \mathit{hnf}\ e \equiv [\![l]\!] \qquad\quad = \textbf{let}\ (l \to e) \in bs\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \mathit{hnf}\ e\\
&\quad \mid \mathit{hnf}\ e \equiv [\![C\ e_1 \ldots e_n]\!] = \textbf{let}\ (C\ v_1 \ldots v_n \to e) \in bs\\
&\qquad\qquad\qquad\qquad\qquad\quad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\}\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{hnf}\ (\sigma\ e)
\end{aligned}
$$

2.2.3 Non-determinism

The next problem is to add non-determinism. The change to the interpreter is small. We only need to add a two types of expression, $e_1\ ?\ e_2$ and $\bot$. However, we now need to find a strategy for evaluating non-deterministic expressions.

This has recently been the subject of a lot of research. Currently there are four options for representing non-determinism. Backtracking, Copying, Pull-tabbing, and Bubbling. All of these options are incomplete in their naive implementations. However, all of them can be made complete. [15]

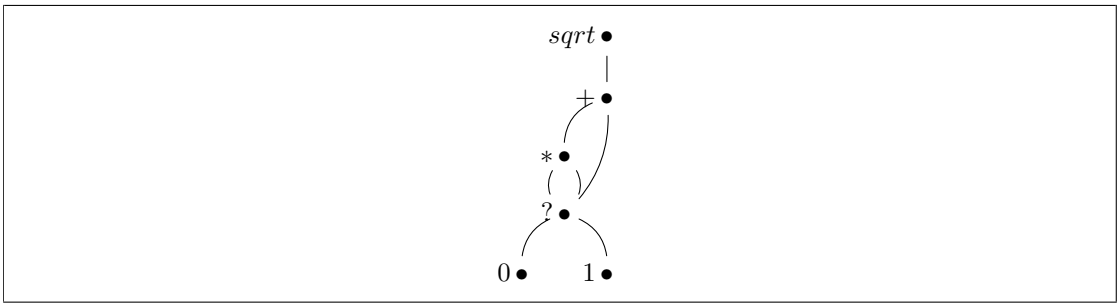
Backtracking is conceptually the simplest mechanism for non-determinism. We evaluate an expression normally, and every time we hit a choice operator, we pick one option. If we finish the computation, either by producing an answer or failing, then we undo each of the computations until the last choice expression. We continue until we've tried every possible choice.

There are a few issues with backtracking. Aside from being incomplete, a naive backtracking implementation relies on copying each node as we evaluate it, so we can undo the computation. Solving incompleteness is a simple matter of using iterative deepening instead of backtracking. This poses its own set of issues, such as how to avoid producing the same answer multiple times, however these are not difficult problems to solve. The issue of copying every node we evaluate is a bigger issue, as it directly competes with any attempt to build an optimizing compiler. However, we'll show how we can avoid creating many of these backtracking nodes.

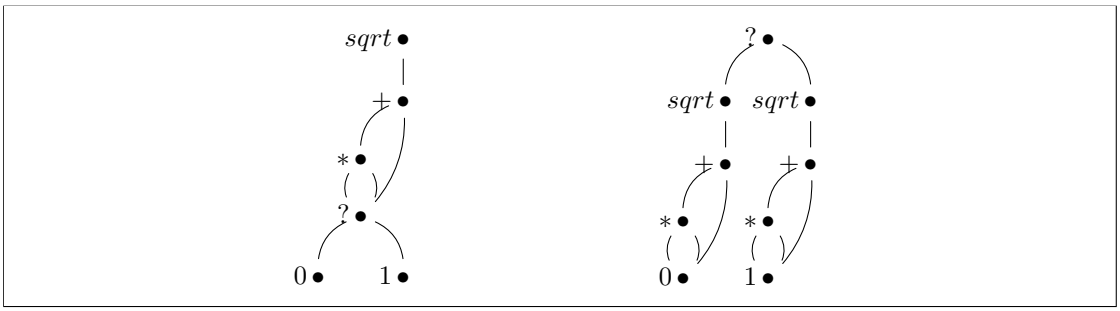
The following three mechanisms are all based on the idea of copying part of the expression graph. All of them are incomplete with a naive implementation, however they can all be made complete using the fair scheme [15]. I'll demonstrate each of these mechanisms with the following expression.

let $x = 0 \,?\, 1$
in $sqrt\ ((x * x) + x)$

This expression has the following graph:



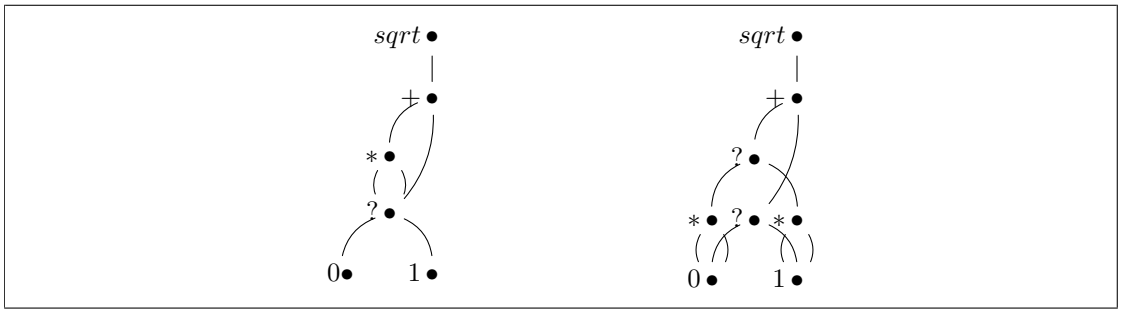
Copying is a different take on non-determinism. The idea is straightforward. Any time we encounter a choice node in an expression, move the choice node up to the root of the graph, and copy every node that was on the path to that choice. We can see the results of copying on our expression below.



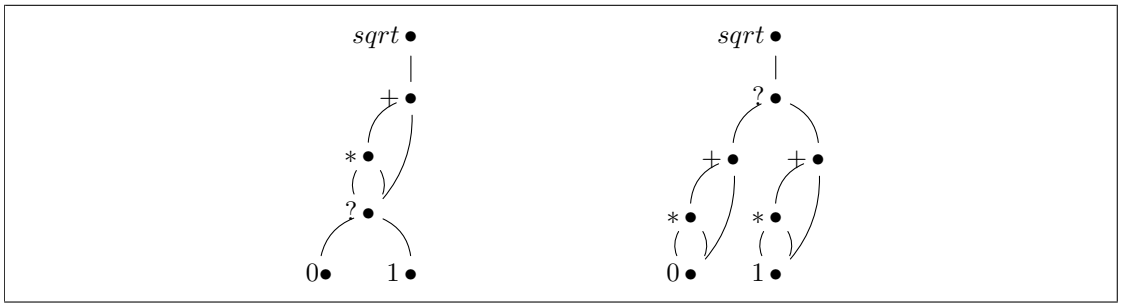
The advantage to copying is its simplicity. We just move the non-determinism to the root, and copy everything on the way up. This is simple to do, and we end up with an expression of the form $answer \,?\, answer \,?\, answer \ldots$. The down side is that we must copy the entire expression. This usually leads to a lot of wasted copying. Especially if one of the branches of the choice will fail.

Pull-tabbing is the other extreme for moving non-determinism. Instead of moving the choice node to the root of the graph, we move the choice node up one level. [4] A naive implementation

of pull-tabbing isn't even valid, so an identifier must be included for each variable to represent which branch it is on. There is a significant cost to keeping track of these identifiers.



Bubbling is a more sophisticated approach to moving non-determinism. Instead of moving the choice node to the root, we move it to its dominator. [5] Bubbling is always valid, and we aren't copying the entire graph. Unfortunately, computing dominators at runtime is expensive. There are strategies of keeping track of the current dominator, [10] but as of this time, there are no known bubbling implementations.



We've elected to implement non-determinism using backtracking for a few reasons. It is the simplest one to implement, and it is known to be efficient. In order for backtracking to work, we need to augment the interpreter with a stack. We'll keep things simple. A stack will be a list of `frames`. Each frame will represent a single rewrite, and a bit to mark if this rewrite was the result of a choice.

type $Frame = (Expr, Expr, Bool)$
type $Stack = [Frame]$

We define an auxiliary function *push* to handle the stack. The idea is that if we rewrite an expression, then we push a frame with the rewrite and the original expression. This avoids cluttering the code with $hnf\ bt\ e@[\![\ldots]\!]$.

$$
\begin{aligned}
&\mathit{hnf}\ \mathit{bt}\ e = \mathbf{let}\ e' = \ldots \\
&\quad \mathbf{in}\ \mathit{push}\ e'\ \mathit{bt} \\
&\quad \mathbf{where}\ \mathit{push}\ \mathit{exp}\ \mathit{stack} = (\mathit{exp}, (\mathit{exp}, e, \mathit{False}) : \mathit{stack})
\end{aligned}
$$

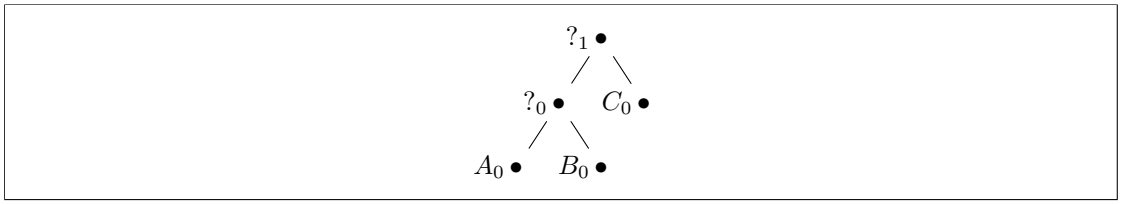
The *pushChoice* is defined similarly, except the frame is $(\mathit{exp}, e, \mathit{True})$ since it's a choice frame.

$$
\begin{aligned}
&\mathit{hnf} :: \mathit{Stack} \to \mathit{Expr} \to (\mathit{Expr}, \mathit{Stack}) \\
&\mathit{hnf}\ \mathit{bt}\ [\![\bot]\!] && = (\bot, \mathit{bt}) \\
&\mathit{hnf}\ \mathit{bt}\ [\![l]\!] && = (l, \mathit{bt}) \\
&\mathit{hnf}\ \mathit{bt}\ [\![e :: t]\!] && = \mathbf{let}\ (e', \mathit{bt}') = \mathit{FORWARD}\ (\mathit{hnf}\ \mathit{bt}\ [\![e]\!]) \\
& && \quad \mathbf{in}\ (\mathit{FORWARD}\ e', \mathit{bt}') \\
&\mathit{hnf}\ \mathit{bt}\ [\![C\ e_1 \ldots e_n]\!] && = (C\ e_1 \ldots e_n, \mathit{bt}) \\
&\mathit{hnf}\ \mathit{bt}\ [\![f\ e_1 \ldots e_n]\!] && = \mathbf{let}\ v_1 \ldots v_n = \mathit{vars}\ f \\
& && \qquad e_f = B_{\mathcal{F}}\ f \\
& && \qquad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
& && \qquad (e'_f, \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ (\sigma\ e_f) \\
& && \quad \mathbf{in}\ \ \mathit{push}\ e'_f\ \mathit{bt}' \\
&\mathit{hnf}\ \mathit{bt}\ [\![\mathit{FORWARD}\ e]\!] && = \mathbf{let}\ (e', \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ e \\
& && \quad \mathbf{in}\ \ \mathit{push}\ (\mathit{FORWARD}\ e')\ \mathit{bt}' \\
&\mathit{hnf}\ \mathit{bt}\ [\![e_1\ ?\ e_2]\!] && = \mathbf{let}\ (e'_{-1}, \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ e_1 \\
& && \quad \mathbf{in}\ \ \mathit{pushChoice}\ (\mathit{FORWARD}\ e'_{-1})\ \mathit{bt}') \\
&\mathit{hnf}\ \mathit{bt}\ [\![\ \mathbf{let}\ v_1 = e_1\ \mathbf{in}\ e]\!] && = \mathbf{let}\ \sigma = \{\, v_1 \to e_1 \,\} \\
& && \quad \mathbf{in}\ \ \mathit{hnf}\ \mathit{bt}\ (\sigma\ e) \\
&\mathit{hnf}\ \mathit{bt}\ [\![\ \mathbf{case}\ e\ \mathbf{of}\ \mathit{bs}]\!] \\
&\quad \mid e \equiv [\![\mathit{FORWARD}\ e']\!] && \quad = \mathbf{let}\ (e', \mathit{bt}') = (\mathit{hnf}\ \mathit{bt}\ [\![\ \mathbf{case}\ e'\ \mathbf{of}\ \mathit{bs}]\!]) \\
& && \qquad \mathbf{in}\ \ \mathit{push}\ (\mathit{FORWARD}\ e')\ \mathit{bt}' \\
&\quad \mid \mathit{hnf}\ e \equiv ([\![l]\!], \mathit{bt}') && \quad = \mathbf{let}\ (l \to \mathit{be}) \in \mathit{bs} \\
& && \qquad\quad (e', \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ \mathit{be} \\
& && \qquad \mathbf{in}\ \ \mathit{push}\ e'\ \mathit{bt}' \\
&\quad \mid \mathit{hnf}\ e \equiv (([\![C\ e_1 \ldots e_n]\!], \mathit{bt}') && = \mathbf{let}\ (C\ v_1 \ldots v_n \to \mathit{be}) \in \mathit{bs} \\
& && \qquad\quad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
& && \qquad\quad (e', \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ (\sigma\ \mathit{be}) \\
& && \qquad \mathbf{in}\ \ \mathit{push}\ e'\ \mathit{bt}'
\end{aligned}
$$

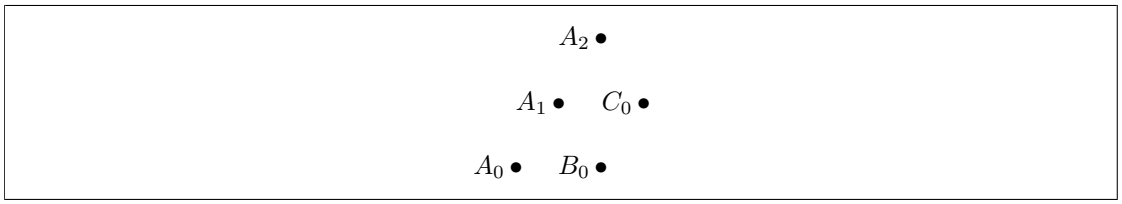
Notice that we don't need to push values in head normal from onto the stack, since there is no evaluation. We also don't push let expressions onto the stack, since they already represent an expression graph.

Due to our recursive evaluation of a **case** expression with a forwarding node, we may add several *phantom rewrites* to the backtracking stack. For example if we have **case** $(FORWARD\ (FORWARD \ldots e))$, we'll add one frame for every forwarding node. They do not affect the semantics because e' will have another frame higher in the stack that will undo that rewrite. This can be proved by induction on the derivation of e'. In practice we evaluate the case expression with forwarding nodes iteratively, so these phantom rewrites are never added to the stack. This will be discussed further in the next section.

To justify the use of forwarding nodes from the last section consider the expression $(A\,?\,B)\,?\,C$ for some constructors A, B, C, If we are backtracking, and attempting to copy values onto the stack, then there is no way produce only the three required answers with copying. The problem is that, in the first evaluation we will replace both ? nodes with a copy of A. So, we start with the expression graph:



After evaluating the expression we'll end up with the following graph and stack.



$$[(A\_2, (?_0\ ?_1\ C\_0), True), (A\_1, (A\_0\ ?_0\ B\_0), True)]$$

This looks fine, but remember that any node pushed on the backtracking stack is a copy of the original node. So the $?_0$ in the first frame does not refer to the $?_0$ in the second frame. Ultimately copying will lead to either terminating programs failing to produce valid answers, or producing duplicate answers. Neither one of these options are acceptable, so we are forced to use forwarding nodes.

2.2.4 Free Variables

Now that we've developed a semantics for non-determinism, free variables and narrowing are pretty easy to implement. We add a new type of node. *FREE* represents a free variable. We use $:=$ as a destructive update operation, so that we can replace a free variable with a different expression.

$$
\begin{aligned}
&hnf :: Stack \to Expr \to (Expr, Stack)\\
&hnf\ bt\ [\![\bot]\!] &&= (\bot, bt)\\
&hnf\ bt\ [\![l]\!] &&= (l, bt)\\
&hnf\ bt\ [\![e :: t]\!] &&= \textbf{let}\ (e', bt') = FORWARD\ (hnf\ bt\ [\![e]\!])\\
&&&\quad \textbf{in}\ (FORWARD\ e', bt')\\
&hnf\ bt\ [\![C\ e_1 \ldots e_n]\!] &&= (C\ e_1 \ldots e_n, bt)\\
&hnf\ bt\ [\![FREE]\!] &&= (FREE, bt)\\
&hnf\ bt\ [\![f\ e_1 \ldots e_n]\!] &&= \textbf{let}\ v_1 \ldots v_n = vars\ f\\
&&&\qquad e_f = B_{\mathcal{F}}\ f\\
&&&\qquad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\}\\
&&&\qquad (e'_f, bt') = hnf\ bt\ (\sigma\ e_f)\\
&&&\quad \textbf{in}\ \ push\ e'_f\ bt'\\
&hnf\ bt\ [\![FORWARD\ e]\!] &&= \textbf{let}\ (e', bt') = hnf\ bt\ e\\
&&&\quad \textbf{in}\ \ push\ (FORWARD\ e')\ bt'\\
&hnf\ bt\ [\![e_1\ ?\ e_2]\!] &&= \textbf{let}\ (e'_{-1}, bt') = hnf\ bt\ e_1\\
&&&\quad \textbf{in}\ \ pushChoice\ (FORWARD\ e'_{-1})\ bt')\\
&hnf\ bt\ [\![\ \textbf{let}\ v_1 = e_1\ \textbf{in}\ e]\!] &&= \textbf{let}\ \sigma = \{\, v_1 \to e_1 \,\}\\
&&&\quad \textbf{in}\ \ hnf\ bt\ (\sigma\ e)\\
&hnf\ bt\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!]\\
&\quad \mid e \equiv [\![FORWARD\ e']\!] &&= \textbf{let}\ (e', bt') = (hnf\ bt\ [\![\ \textbf{case}\ e'\ \textbf{of}\ bs]\!])\\
&&&\qquad fwd = FORWARD\ e'\\
&&&\quad \textbf{in}\ \ push\ (FORWARD\ e')\ bt'\\
&\quad \mid e \equiv [\![FREE]\!] &&= \textbf{let}\ \{\, C_1 \ldots \to \_, \ldots, C_n \ldots \to \_ \,\} = bs\\
&&&\qquad e_1 = C_1\ FREE \ldots FREE\\
&&&\qquad \ldots\\
&&&\qquad e_n = C_n\ FREE \ldots FREE
\end{aligned}
$$

$$
\begin{aligned}
& & & e := e_1 \,?\, \ldots \,?\, e_n \\
& & \textbf{in} \;\; & \mathit{hnf}\ \mathit{bt}\ [\![\, \textbf{case}\ e\ \textbf{of}\ \mathit{bs}]\!] \\
& \mid \mathit{hnf}\ e \equiv ([\![l]\!], \mathit{bt}') & = \textbf{let}\ & (l \to \mathit{be}) \in \mathit{bs} \\
& & & (e', \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ \mathit{be} \\
& & \textbf{in} \;\; & \mathit{push}\ e'\ \mathit{bt}' \\
& \mid \mathit{hnf}\ e \equiv (([\![C\ e_1 \ldots e_n]\!], \mathit{bt}') & = \textbf{let}\ & (C\ v_1 \ldots v_n \to \mathit{be}) \in \mathit{bs} \\
& & & \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
& & & (e', \mathit{bt}') = \mathit{hnf}\ \mathit{bt}\ (\sigma\ \mathit{be}) \\
& & \textbf{in} \;\; & \mathit{push}\ e'\ \mathit{bt}'
\end{aligned}
$$

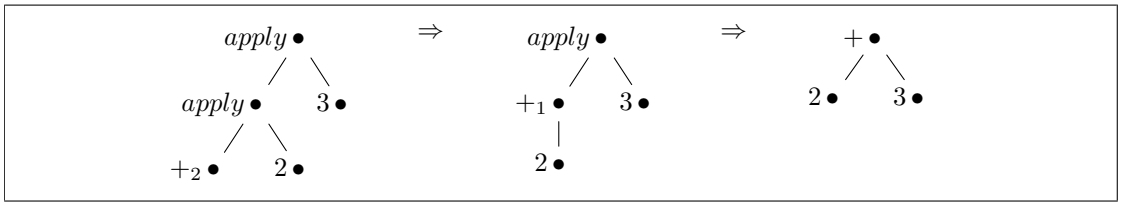
2.2.5 Higher Order Functions

The last feature we need to add is higher order functions. This is typically done with defunctionalization. [52]. The idea is simple. We add a new function called *apply* with the definition.

$$\mathit{apply}\ f\ x = f\ x$$

This means that our system is no longer strictly a rewriting system. But this also introduces a new problem. What is f? If we look closely we see that f actually has two different meanings. On the left hand side f is a symbol that is passed to apply, however on the right hand side f is a function that needs to be reduced.

We make this definition precise by introducing a *partial application*. If f is a function symbol with arity n, then f_k where $k \leqslant n$ is the partial application that is missing k arguments. For example, $+_2$ represents the usual + function, but it is missing two arguments. We can then apply it to arguments with $\mathit{apply}\ (\mathit{apply}\ +_2\ 2)\ 3$. The evaluation is shown below graphically.



Applying functions one argument at a time will always work, but in practice this is very slow. We can improve performance drastically by making *apply* into a variadic function.

$$\mathit{apply}\ f_n\ [x_1, \ldots x_n] = f\ x_1 \ldots x_n$$

Unfortunately, this definition only works if the length of the argument list is exactly the same as the number of missing arguments in f. This is rarely the case. So, we need to change the definition of *apply* to handle the three different possibilities.

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

This is the only change we need to support higher order functions, but is this change valid? How does it interact with non-determinism and free variables? The answer is that there aren't any complicated interactions to worry about. If f_k is non-deterministic, then we push the apply node on the backtracking stack. This is no different than any variable evaluated by a case statement. If f_k is a free variable, then we return $\perp$, since we cannot narrow functions.

2.3 THE GENERATED CODE

This gives us a working semantics for the FlatCurry language. However this is not the semantics we used. Unfortunately, while this semantics isn't too complicated, its simplicity comes at the cost of speed. There are two major problems. The first is that we explicitly represent **case** and **let** expressions as nodes in our graph. These should be translated down to flow of control and assignment statements. The second problem is that every time we rewrite a node, we push a frame on the backtracking stack.

In order to fix these problems, we need to move from the world of abstract interpreters into compiled code. We compile to C code, since C is low level enough to apply all of our optimizations, but modern C compilers are able to take care of optimizations not related to functional logic programs.

Since we are constructing a graph rewriting system, we need to decide on the representation of the graph. I've started with a simplified version of a `Node` of the graph. We will expand it as we add features.

```
typedef struct Node
{
    unsigned int missing;
```

```
    unsigned int tag;
    const void (*hnf)(struct Node*);
    Node* children[4];
} Node;
```

As we can see, a node doesn't contain a lot of information. It only contains the number of arguments it's missing, a `tag`, a function pointer to some `hnf` function, and an array of 4 children. The `missing` variable will only be relevant if this node represents a partially applied function or constructor. Most of the time it will be set to 0. While the node can have four children, we can extend this by having the final child point to an array of more children.

The `tag` field tells us what kind of node this represents. There are five global tags, `FAIL, FUNCTION, CHOICE, FORWARD,` and `FREE`. These tags are given the values 0,1,2,3, and 4 respectively. Then, for every data type, each constructor is given a unique tag for that type. For example the type *Bool* has two constructors *True* and *False*. The tag for *False* is 5, and the tag for *True* is 6. Curry's type system guarantees that expressions of one type will remain in that type, so we only need tags to be unique for each type. It's not an issue that both *False* and *Nothing* from the *Maybe* type share the tag 5, because no boolean expression could become a value of type *Maybe*.

Finally the `hnf` field is a function pointer to the code that can reduce this node. For every function f in Curry, we will generate a `f_hnf` C function. An example of the *id* hnf function is given below.

```
void Prelude_id_hnf(field root)
{
  Node* x = root->children[0]
  x->hnf(x);
  root->hnf = &forward_hnf;
  root->tag = FORWARD_TAG;
  push(bt_stack, root, make_id(x));
  return;
}
```

Here each `hnf` function takes the root of the expression as a parameter. So if we're evaluating the expression *id* 5, then `root` is *id* and `x` is 5. We get this first child of `root`, since *id* only

takes one argument. Then we evaluate the child `x` to head normal form, using `x`'s hnf function. Finally we set `root` to be a forwarding node, and push `root` and a copy of the id node onto the backtracking stack.

This matches what our semantics would do exactly, but *id* is a simple function. What happens when we have a function with a case statement. We'll use the Curry function *not* as an example.

$$
\begin{aligned}
&\mathit{not} :: \mathit{Bool} \to \mathit{Bool} \\
&\mathit{not}\ x = \textbf{case}\ x\ \textbf{of} \\
&\quad \mathit{True} \to \mathit{False} \\
&\quad \mathit{False} \to \mathit{True}
\end{aligned}
$$

The generated code for these functions becomes complex quickly, so we'll start with a simplified version. Initially we might generate the following.

```
void not_hnf(Node* root)
{
  Node* x = root->children[0];
  switch(x->tag)
  {
      case False_TAG:
          root->tag = True_TAG;
          root->hnf = CTR_hnf;
          push(bt_stack, root, make_not(x));
          break;

      case True_TAG:
          root->tag = False_TAG;
          root->hnf = CTR_hnf;
          push(bt_stack, root, make_not(x));
          break;
  }
}
```

This looks great. The only surprising part is why we are assigning a `hnf` function to a node

in head normal form. The `CTR_hnf` function doesn't actually do anything. It's just there because every node needs an hnf function.

Right now this code only works if `x` is *True* or *False*. But `x` could be any other expression. It could be a `FAIL` node, a `FUNCTION` call, a `CHOICE` expression, a `FORWARD` node, or a `FREE` variable. We'll tackle these one at a time. Fortunately `FAIL` nodes are easy. If the scrutiny of a case is a failure, then the whole expression should fail. We just need to add the case:

```
case FAIL_TAG:
    root->tag = FAIL_TAG;
    root->hnf = CTR_hnf;
    push(bt_stack, root, make_not(x));
    break;
```

We can reuse `CTR_hnf` because `FAIL` is a head normal form. This is simple enough, but now we need to add `FUNCTION` nodes to our case. The problem is, if our case expression is a function node, then we need to evaluate that to head normal form, and then we need to re-examine the tag. The solution here is simple. We just put the whole case in a loop. Surprisingly, this code is about as efficient as using a more complicated scheme like a jump table [53]. So our function node becomes

```
case FUNCTION_TAG:
    root->hnf(root);
    break;
```

We can actually do the same for choice and free nodes. A choice node is reduced to one of its two values, and a free node is replaces with one of the two constructors. After this is done, we reevaluate the expression.

```
case CHOICE_TAG:
    x->choice_hnf(x);
    break;


case FREE_TAG:
    x->TAG = CHOICE_TAG;
    x->hnf = &choice_hnf;
```

```
    x->children[0] = make_True();
    x->children[1] = make_False();
    x->choice_hnf(x);
    break;
```

The `choice_hnf` function chooses between the two options, and will be described later. Any `make_*` function will construct new a new node.

Finally we have the `FORWARD` nodes. Unfortunately, these nodes are more complicated. A naive implementation could set the value of the forward node to the node that it points to, such as the following code.

```
case FORWARD_TAG:
    x = x->children[0]
    break;
```

Unfortunately, this solution fails if we need the original node. Suppose we have the Curry program

$$
\begin{aligned}
&\mathit{makeJustBool}\ x = \textbf{case}\ x\ \textbf{of}\\
&\quad \mathit{True} \to \mathit{Just}\ x\\
&\quad \mathit{False} \to \mathit{Just}\ x\\
&\mathit{main} = \mathit{makeJustBool}\ (\mathit{False}\ ?\ \mathit{True})
\end{aligned}
$$

If we were to use the naive forwarding method then we would evaluate *main* to the expression *Just True*, when it should really be *Just* (*FORWARD True*). Therefore, we need to keep the original variable around. This leads to an unfortunate problem of keeping two values of each variable. The variable itself, and a forwarding position. This makes the generated code harder to read, but it doesn't effect performance much. The C optimizer can easily remove unused duplicates. This finally leads to the full code for the *not* function given below. There are a few more technical issues to resolve, but this is the core idea behind how we generate code.

```
void not_hnf(Node* root)
{
  Node* x = root->children[0];
  Node* x_forward = x;
```

```
while(true)
{
  switch(x_forward->tag)
  {
    case FAIL_TAG:
      root->tag = FAIL_TAG;
      root->hnf = CTR_hnf;
      push(bt_stack, root, make_not(x));
      return;

    case FORWARD_TAG:
      x_forward = x_forward->children[0]
      break;

    case FUNCTION_TAG:
      root->hnf(root);
      break;

    case CHOICE_TAG:
      x_forward->choice_hnf(x_forward);
      break;

    case FREE_TAG:
      x_forward->TAG = CHOICE_TAG;
      x_forward->hnf = &choice_hnf;
      x_forward->children[0] = make_True();
      x_forward->children[1] = make_False();
      x_forward->choice_hnf(x_forward);
      break;

    case False_TAG:
      root->tag = True_TAG;
```

```
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_not(x));
        return;

      case True_TAG:
        root->tag = False_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_not(x));
        return;
    }
}
```

2.3.1 Let Expression

The semantics here seem fine, but we actually encounter a surprising problem when we add let expressions. Consider the following function:

$$
\begin{aligned}
\mathit{weird} = \ & \textbf{let } x = \mathit{True} \ ? \ \mathit{False} \\
& \quad \textbf{in case } x \textbf{ of} \\
& \qquad \mathit{False} \rightarrow \mathit{False} \\
& \qquad \mathit{True} \ \rightarrow \mathit{True}
\end{aligned}
$$

This would be a silly function to write, but its meaning should be clear. It will produce both *True* and *False*.

However, if we were to run this code with our current implementation, we'd get surprising behavior.

```
: eval weird
False
False
False
False
...
```

What went wrong here? Well, we can look at the generated code for *weird* to find a clue.

```
void weird_hnf(Node* root)
{
  Node* x = make_choice(make_True(), make_False());
  Node* x_forward = x;
  while(true)
  {
    switch(x_forward->tag)
    {
      ...

      case False_TAG:
        root->tag = False_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird);
        return;

      case True_TAG:
        root->tag = True_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird);
        return;
    }
}
```

When we push a rewrite onto the backtracking stack, we can only backtrack to the calling function. In this case that means that when we backtrack, `root` is replaced by `weird`. But weird actually has some important state. It created a local node that was non-deterministic. So, when we backtrack, we need to keep the state around. This turns out to be a hard problem to solve. Both Pakcs and Kics2 sidestep this problem by transforming the program so that there is at most one case in each function. [18, 32] This can solve the problem, but it increases the number of function calls substantially. We propose a novel solution where there are no extra function calls.

The idea is pretty straightforward. Notice that the problem from our *weird* example happened because we reached a case statement with some local state. So, when we backtrack, we would

want to backtrack to that specific point in the function. This leads to a new definition. Let e be an expression, then the *case path* $e|_p$ of an expression is a path through the branches of case statements. This is analogous to the path through a definitional tree. Now for each function, we can define a *path function* as $f|_p\ x_1 \ldots x_1 = e|_p$ where $x_1 \ldots x_n$ are undefined variables in $e|_p$. The full definition for $e|_p$ is given with the following non-deterministic function.

$$
\begin{aligned}
&\mathit{casePath}\ (\mathbf{let} \ldots \mathbf{in}\ e) = \mathit{casePath}\ e \\
&\mathit{casePath}\ (e_1\ ?\ e_2) = \mathit{casePath}\ e_1\ ?\ \mathit{casePath}\ e_2 \\
&\mathit{casePath}\ (\mathbf{case} \ldots \mathbf{in} \ldots) = [\,] \\
&\mathit{casePath}\ (\mathbf{case} \ldots \mathbf{in}\ \{\ldots p_i \to e_i \ldots\}) = i : \mathit{casePath}\ e_i
\end{aligned}
$$

The idea here is that we make a new function starting at each case statement. Then when we're at the case at position p, we push $f|_p$ onto the backtracking stack instead of f. In C we represent $f|_p$ as `f_p`, and `f_` is the function at the empty path, which is just before the first case statement. We can use this to solve our *wierd* problem. We generate two function.

```
void weird_hnf(Node* root)
{
  Node* x = make_choice(make_True(), make_False());
  Node* x_forward = x;
  while(true)
  {
    switch(x_forward->tag)
    {
      ...

      case False_TAG:
        root->tag = False_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird_());
        return;

      case True_TAG:
        root->tag = True_TAG;
```

```
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird_());
        return;
    }
}

void weird__hnf(Node* root)
{
  Node* x = root->childrent[0];
  Node* x_forward = x;
  while(true)
  {
    switch(x_forward->tag)
    {
      ...

      case False_TAG:
        root->tag = False_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird_());
        return;

      case True_TAG:
        root->tag = True_TAG;
        root->hnf = CTR_hnf;
        push(bt_stack, root, make_weird_());
        return;
    }
}
```

As we can see this will duplicate a lot of code, and the problem gets worse when we have more nested case statements. However, the problem is not as bad as it might seem at first. While we will have more duplication with nested case statements, we're duplicating smaller functions

each time. Also, while we're duplicating a lot of code, the duplicate code is part of a separate function, so having more code won't evict the running code from the cache. It may be possible to eliminate the duplicate code with a clever use of gotos, but it's not clear that it would be more efficient, and is outside the scope of this research.

2.3.2 Choice Nodes

At this point our compiler is correct, but there are still some details to work out. How do we actually implement non-determinism? So far we've swept it under the rug with the `choice_hnf` function. This function isn't terribly complicated conceptually, but it hides a lot of details.

```
typedef struct
{
    bool choice;
    field lhs;
    field rhs;
} Frame;

typedef struct
{
    Frame* array;
    size_t size;
    size_t capacity;
} Stack;

Stack bt_stack;
```

In our C implementation, choice frames and the backtracking stack are both straightforward. A choice frame has a left hand side, right hand side, and a marker denoting if the frame came from a choice node. Our backtracking function is almost as simple. We just copy the `rhs` over `lhs`

```
bool undo()
{
    if(empty(bt_stack))
```

```
        return false;

    Frame* frame;
    do
    {
        frame = pop(bt_stack);
        memcpy(frame->lhs.n, frame->rhs.n, sizeof(Node));
    } while(!(frame->choice || empty(bt_stack)));

    return frame->choice;
}
```

This is all easy, but what about the choice node itself? Well, that's not much more complicated. A choice node is just a node, but we give a specific meaning to each child. A choice node has a left child `children[0]`, a right child `children[1]`, and a marker for which side to reduce `children[2]`. When we first encounter a choice node, we reduce it to the left hand side, then after we've backtracked, we reduce it to the right hand side. Notice that if `children[2]` is 0 then reduce the left child and mark this node in the backtracking stack, otherwise reduce the right child. This leads to the following algorithm for evaluating a choice node.

```
void choice_hnf(field root)
{
    Node* choices[2] = {root->children[0], root->children[1]};
    int side = root->children[2];

    Node* saved = (Node*)malloc(sizeof(Node));
    memcpy(saved.n, root.n, sizeof(Node));
    saved->children[2] = !side;

    choices[side]->hnf(choices[side]);
    set_forward(root,choices[side]);

    push(bt_stack, root, saved, side == 0);
```

```
}
```

2.3.3 Optimization: Removing Backtracking Frames

Surprisingly, the code in the previous section is the only piece of the runtime system that is needed for non-determinism. However, while this works, there's a major efficiency problem here. We're pushing nodes on the backtracking stack for every rewrite. There are a lot of cases where we don't need to push most of these nodes, such as the following code

$$
\begin{array}{l}
\mathit{fib}\ n = \textbf{if}\ n < 2\ \textbf{then}\ n\ \textbf{else}\ \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \\
\mathit{main} \\
\quad \mid \mathit{fib}\ 20 \equiv (1\ ?\ 6765) = \mathit{putStrLn}\ \texttt{"found answer"}
\end{array}
$$

This program will compute *fib* 20, then it will push all of those nodes onto the stack, Then, when it discoverers that $\mathit{fib}\ 20 \not\equiv 1$, 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, can't we just avoid pushing those values on the stack? The short answer is no. There are two reasons. First, determining if a function is non-deterministic is undecidable. Second, a function may have a non-deterministic argument. For example, we could easily change the above program to:

$$
\begin{array}{l}
\mathit{fib}\ n = \textbf{if}\ n < 2\ \textbf{then}\ n\ \textbf{else}\ \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \\
\mathit{main} \\
\quad \mid \mathit{fib}\ (1\ ?\ 20) \equiv 6765 = \mathit{putStrLn}\ \texttt{"found answer"}
\end{array}
$$

Now the expression with *fib* is no longer deterministic. So, do we need to just give up and accept this loss of efficiency? Surprisingly, we don't. While it's impossible to tell statically if an expression is non-deterministic, its very easy to tell dynamically if it is.

As far as we're aware, this is another novel solution. The idea is simple. Each expression contains a boolean flag that marks if it is non-deterministic. We've called these `nondet` flags.

The rules for determining if an expression node e is `nondet` are simple. If e is a choice, then e is `nondet`. If e has a case who's scrutenee is `nondet`, or is a forward to a `nondet`, then e is `nondet`. All other nodes are deterministic.

It's easy to see that any node not marked as `nondet` doesn't need to be pushed on the stack. It's not part of a choice, all of its case statements used deterministic nodes, and it's not forwarding

to a non-deterministic node. However proving this is a more substantial problem.

In order to prove the correctness of this modification we need to take a step back into graph rewriting. First we need a couple of definitions.

Definition 2.3.1. Given a rewrite system $\mathbf{R}$ with strategy S, a computation space of expression e, $C\_S\ (e)$ is a directed graph where nodes are expressions, there is an edge (e_1, e_2) iff $(\mathbf{R}, p) \in S\ (e_1)$ and $e_1 - > \_\ (\mathbf{R}, p)\ e_2$, and $C\_S\ (e)$ is rooted by e.

Definition 2.3.2. given an expression e and a rewrite rule $l \to r$, the redex pattern of e is the set of non-variable nodes in e that match l. [?, Def. 2.7.3] that the **redex pattern**

Now we can start to reason about computation spaces. The first thing to point out is pretty straightforward. A path with no branches corresponds to a deterministic expression.

Theorem 3. *If $C\_S\ (e)$ is a path, then e is deterministic.*

Proof. if e is non-deterministic, then $e \to^* f$ where f has two rules that can apply. Which means that there is a branch in $C\_S\ (e)$, and therefore $C\_S\ (e)$ isn't a path. □

Next we can start contracting deterministic paths. The idea is straightforward. If a reduction is deterministic, then it doesn't matter when the reduction happens. This would be directly implied by confluence in an orthogonal system, but Curry isn't confluent, so we need to prove it.

Lemma 4 (redex compression)**.** *if $a \to \{\, b \,\}$, and $e \to^* n$ then $e[a \mapsto b] \to^* n$*

Proof. If a is not needed in the derivation of e, the $e[a \mapsto b]$ will not change the computed values. If a is needed, then we can break the computation into 3 parts $e \to^* e_a \to_a e_b \to^* n$. Since a is a redex, and $\mathbf{R}$ is a constructor rooted system, everything in the redex pattern for a must be a constructor. Since a can't be duplicated, we don't need to worry about parallel rewrites. Therefore by our definition of rewriting none of the nodes in the redex pattern of a will be changed, since constructors have no rewrite rules. Therefore if we replace a with b at the beginning, our derivation $e[a \mapsto b] \to e_b \to^* n$ proceeds as before, except we've remove the step $e_a \to_a e_b$. □

Finally we can prove our main theorem. If an expression a deterministically reduces to b, then we don't really need to do that reduction. There's only one possible choice, so if we just replace a with b at the start, then we get the same answers.

Theorem 5 (path compressions)**.** *if* $a \rightarrow^* \{\, b \,\}$*, and* $e \rightarrow^* n$ *then* $e[a \mapsto b] \rightarrow^* n$

Proof. we proceed by induction. Base case: $a \rightarrow \{\, b \,\}$. This is the redex compression theorem. Inductive case: $a \rightarrow \{\, a_1 \,\} \rightarrow^* \{\, b \,\}$. It must be the case that a_1 is a single reduct of a, because otherwise $a \rightarrow^* b$ would not be deterministic. Assume that $e \rightarrow^* n$ where a is a redex of e. By the redex compression theorem $e[a \mapsto a_1] \rightarrow^* n$, and by the inductive hypothesis $e[a \mapsto a_1][a_1 \mapsto b] \rightarrow^* n$ □

This is really all we need to prove that our backtracking strategy is valid. The effect of this theorem is that if I know $a \rightarrow^* b$ in the computation space, and $a \rightarrow^* b$ must be deterministic, then I can just replace a with b, and skip the entire rewriting process. So, In the computation space for $C\_S\ (e)$, I can remove the path $a \rightarrow^* b$, and it won't have any effect on the final answers.

This means that if we have an expression that we know is deterministic, then we can replace that expression with any of it's reducts. This directly implies that if we don't backtrack a deterministic expression, the we still produce the same set of solutions.

Now we need to prove that our backtracking scheme only ever omits deterministic redexes. Recall that e is only marked nondet is it's a choice node, or if it's a case that depends on a nondet node. Taking the contrapositive of this definition we find that the following nodes are not nondet, any literal, forwarding node, fail node, any constructor application, or any function application that does not contain a nondet node in it's redex pattern, and does not reduce to a nondet node.

Now we can justify our using the term nondet for these nodes.

Lemma 6. *Any Curry expression that is not marked as nondet are deterministic.*

Proof. Literals, forwarding nodes, fail nodes, and constructors are already in head normal form. If we have a function node, then by assumption we know that all of the nodes that it evaluates are not marked as nondet, and therefore produce only a single value. Furthermore, since the function does not evaluate to a nondet node, it must also produce a single answer. □

Finally we can establish correctness.

Theorem 6.1 (Correctness of fast backtracking)**.** *Any Curry expression will produce the same answers if only nodes marked as nondet are pushed on the stack.*

Proof. By the previous lemma we know that all nodes that aren't marked nondet are deterministic. Suppose we have an expression e, and $e \to^* e'$, and $e_1 \,?\, e_2$ is needed in e'. Now if $a \to^* b$ is deterministic and a is needed by e' then e' will evaluate normally taking choice e_1, however when we backtrack to e' and take choice e_2, we are left with rewriting $e'[a \mapsto b]$. By the path compression theorem this is the same as if we hadn't reduced a. □

Now that we've established correctness, we can look at how we need to change the generated code. Fortunately the only change to the code is that instead of just pushing rewrites on the stack, we check if the variable is `nondet`. For example, the `not_hnf` example is changed to:

```
void not_hnf(Node* root)
{
  Node* x = root->childrent[0];
  Node* x_forward = x;
  bool nondet = false;
  while(true)
  {
    nondet = x_forward->nondet;
    switch(x_forward->tag)
    {
      ...

      case False_TAG:
        root->tag = False_TAG;
        root->hnf = CTR_hnf;
        if(nondet)
        {
          root->nondet = true;
          push(bt_stack, root, make_not(x), false);
        }
        return;

    ...
```

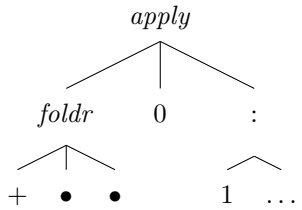
```
    }
}
```

2.3.4 Apply Nodes

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 to apply nodes:

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

Here if f is missing any arguments, then we call f a partial application. Let's look at a concrete examples. In the expression $foldr_2\ (+_2)$, *foldr* is a partial application that is missing 2 arguments. We will write this as $\mathit{foldr}\ (+_2)\ \bullet\ \bullet$ where $\bullet$ denotes a missing argument. Now suppose that we want to apply



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's not true for *apply* nodes. In fact, and *apply* node may have any positive arity. Furthermore, by definition, an *apply* node can't be missing any arguments. For this reason, we use the `missing` field to hold how many arguments the node is applied to. In reality we set missing to the negative value of the arity to distinguish an apply node from a partial application.

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's

consider the *foldr* example above. But this time let's decompose it into 3 apply nodes, so we have $apply\ (apply\ (apply\ foldr_3\ (+_2))\ 0)\ [1,2,3]$. 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're storing the arguments in reverse order, then we get a really handy result. Given function f_k that is missing k arguments, then $apply\ f_k\ x$ 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.

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

The algorithm is given below. 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 $apply\ f\bullet\bullet arg\_n\ldots arg\_1$ This isn't done in the runtime system because it would be inefficient, but it simplifies the code for presentation. We also make use of the `set_child_at` macro, which simplifies setting child nodes, since the first three children are part of the node, but any more are part of an external array. Finally, the loop to put the partial function in head normal 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.

The algorithm, shown below, is pretty simple. 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. Now we split into two cases. If we're under applied, or have exactly the right amount of arguments, then copy the contents of `f` into the root, and move the arguments over and reduce. If we're over applied, then make a new copy of `f`, and copy arguments into it until it's fully applied. reduce the fully applied copy of `f`, and finally apply the rest of the arguments.

```
void apply_hnf(field root)
{
```

```
field f = root.n->children[0];
field* children = root.n->children[3].a;

while(f.n->missing <= 0)
{
    // Normal HNF loop
}

if(f.n->nondet)
{
    save_copy(root);
}

int nargs = -root.n->missing;
int missing = f.n->missing;

if(missing <= nargs)
{
    set_copy(root, f);
    for(int i = nargs; i > 0; i--, missing--)
    {
        set_child_at(root, missing-1, children[i-1]);
    }

    root.n->missing = missing;

    if(missing == 0)
    {
        root->symbol->hnf(root);
    }
}
else
```

```
    {
        field newf = copy(f);

        while(missing > 0)
        {
            set_child_at(newf, missing-1, children[nargs-1]);
            nargs--;
            missing--;
        }

        newf.n->missing = 0;
        newf->symbol->hnf(newf);

        set_child_at(root,0,newf);
        root.n->missing = -nargs;
        apply_hnf(root);
    }
}
```

And with that, we've arrived at our complete semantics for our compiler. In the next section, we describe how compiler transformations are implemented, and we give a short description of the compiler using these transformations. Now that we have our recipe, it's time to make some Curry!

Chapter 3

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. We then show an extended example of using the GAS system to transform FlatCurry programs into a canonical form so that we can compile them to C code, as discussed in the last chapter.

3.0.1 Building Optimizations

Optimization is usually considered the most difficult aspect of writing a modern compiler. It's 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's 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.

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

This will be translated into FlatCurry as

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

While this transformation is obviously inefficient, it is general and easy to implement. A good optimizer should be able to recognize that *otherwise* is really a synonym for *True*, and reduce the case-expression. So for this one example, we have two different optimizations we need to implement.

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's 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. [41, 48] 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. [50]

The aim of our solution is to try to get the best of all three worlds. We've 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 didn't work, we could look at it and test it in isolation. Fourth, this code is much smaller than a traditional optimizer. This isn't 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're 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 didn't 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 significant disadvantages to the GAS system as well. This biggest disadvantage is that there are some optimizations and transformations that are not easily described by rewriting. Another disadvantage is that, while we've improved the efficiency of the algorithm considerably, it still takes longer to optimize programs then we'd like.

The first problem isn't really a problem at all. If there is an optimization that doesn't lend itself well to rewriting, we can always write it as a traditional optimization. Furthermore, as we'll see later, we don't 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. [2] We also believe that an optimizing compiler would help with this problem [42].

3.0.2 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.

type *Opt* = *Expr* → *Expr*

We can describe an optimization by simply describing what it does to each expression. As an

example consider the definition for floating let-expressions:

$$\mathit{float}\ [\![f\ (as +\!\!+ [\textbf{let}\ vs\ \textbf{in}\ e] +\!\!+ bs)]\!] = [\![\ \textbf{let}\ vs\ \textbf{in}\ (f\ (as +\!\!+ [e] +\!\!+ bs))]\!]$$

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's a free variable declaration inside of a function? Well, we can define that case with another rule.

$$\begin{aligned}\mathit{float}\ [\![f\ (as +\!\!+ [\textbf{let}\ vs\ \textbf{in}\ e] +\!\!+ bs)]\!] &= [\![\ \textbf{let}\ vs\ \textbf{in}\ (f\ (as +\!\!+ [e] +\!\!+ bs))]\!] \\ \mathit{float}\ [\![f\ (as +\!\!+ [\textbf{let}\ vs\ \textbf{free in}\ e] +\!\!+ bs)]\!] &= [\![\ \textbf{let}\ vs\ \textbf{free in}\ (f\ (as +\!\!+ [e] +\!\!+ bs))]\!]\end{aligned}$$

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 don't care which rule matches, as long as they both do eventually. This will be transformed into one of the following:

$$[\![\ \textbf{let}\ r\ \textbf{free in let}\ [x = 1]\ \textbf{in}\ f\ x\ 2]\!]$$
$$[\![\ \textbf{let}\ [x = 1]\ \textbf{in let}\ r\ \textbf{free in}\ f\ x\ 2]\!]$$

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.

$$\begin{aligned}\mathit{float}\ [\![f\ (as +\!\!+ [\textbf{let}\ vs\ \textbf{in}\ e] +\!\!+ bs)]\!] &= [\![\ \textbf{let}\ vs\ \textbf{in}\ (f\ (as +\!\!+ [e] +\!\!+ bs))]\!] \\ \mathit{float}\ [\![f\ (as +\!\!+ [\textbf{let}\ vs\ \textbf{free in}\ e] +\!\!+ bs)]\!] &= [\![\ \textbf{let}\ vs\ \textbf{free in}\ (f\ (as +\!\!+ [e] +\!\!+ bs))]\!] \\ \mathit{float}\ [\![[\textbf{let}\ vs\ \textbf{in let}\ ws\ \textbf{free in}\ e]]\!] &= [\![\ \textbf{let}\ ws\ \textbf{free in let}\ vs\ \textbf{in}\ e]\!]\end{aligned}$$

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

Now that we can do simple examples, let's 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:

$$\begin{aligned}&\textbf{let}\ a = b \\ &\quad\ \ b = c\end{aligned}$$

```
        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's 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 isn't 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.

```
blocks _ [[Let vs e]] | numBlocks > 1 = e'
  where (e', numBlocks) = makeBlocks es e


makeBlocks vs e = (letExp, length comps)
  where letExp = foldr makeBlock e comps
        makeBlock comp = λexp → [[ let (map getExp comp) in exp]]
        getExp n | [[n = exp]] ∈ vs = [[n = exp]]
        comps = scc (vs >>= makeEdges)
        makeEdges [[v = exp]] = [(v, f) | f ← freeVars exp ∩ map fst vs]
```

While this optimization is significantly more complicated then the $float$ example, We can still implement it in our system. Furthermore, we're 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 our 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 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.

$$floatBlocks = float \; ? \; blocks$$

This is a new optimization that will apply both $float$ and *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.

3.0.3 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 [19, 6] The full code is given in figure 3.1

While this attempt is really simple, there's a problem with it. It is unusably slow. While looking at the code, it's 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.

3.0.4 A Second Attempt: Multiple Transformations Per Pass

Our second attempt is much more successful. 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've 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 3.2

```
fix :: (a → a) → a → a
fix f x
  | f x ≡ ∅    = x
  | otherwise = fix f (f x)

subExpr :: Expr → Expr
subExpr e = e
subExpr ⟦f es⟧                = subExpr (foldr1 (?) es)
subExpr ⟦ let vs in e⟧        = subExpr (foldr1 (?) (map snd es))
subExpr ⟦ let vs free in e⟧   = subExpr e
subExpr ⟦e : t⟧               = subExpr e
subExpr ⟦e1 ? e2⟧             = subExpr e1 ? subExpr e2
subExpr ⟦ case e of bs⟧       = subExpr (e ? map branchExpr bs)
  where branchExpr ⟦pat → e⟧ = e

reduce :: Opt → Expr → Expr
reduce opt e = opt (subExpr e)

simplify :: Opt → Expr → Expr
simplify opt e = fix (reduce opt) e
```

Figure 3.1: A first attempt at an optimization engine. Pick an arbitrary subexpression and try to optimize it.

$$
\begin{aligned}
&\mathit{fix} :: (a \to a) \to a \to a \\
&\mathit{fix}\ f\ x \\
&\quad \mid f\ x \equiv x = x \\
&\quad \mid \mathit{otherwise} = \mathit{fix}\ f\ (f\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{Opt} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{reduce}\ \mathit{opt}\ [\![v]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![v]\!] \\
&\mathit{reduce}\ \mathit{opt}\ [\![l]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![l]\!] \\
&\mathit{reduce}\ \mathit{opt}\ [\![f\ es]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![f\ es']\!] \\
&\quad \textbf{where}\ es' = \mathit{map}\ (\mathit{run}\ \mathit{opt})\ es \\
&\mathit{reduce}\ \mathit{opt}\ [\![\ \textbf{let}\ vs\ \textbf{in}\ e]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![\ \textbf{let}\ vs'\ \textbf{in}\ e']\!] \\
&\quad \textbf{where}\ vs' = \mathit{map}\ (\lambda[\![v = e]\!] \to [\![v = \mathit{run}\ \mathit{opt}\ e]\!])\ vs \\
&\qquad\quad e' = \mathit{run}\ \mathit{opt}\ e \\
&\mathit{reduce}\ \mathit{opt}\ [\![\ \textbf{let}\ vs\ \textbf{free in}\ e]\!] = \mathit{runOpts}\ \mathit{opt}\ [\![\ \textbf{let}\ vs\ \textbf{free in}\ e']\!] \\
&\quad \textbf{where}\ e' = \mathit{run}\ \mathit{opt}\ e \\
&\mathit{reduce}\ \mathit{opt}\ [\![e : t]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![e' : t]\!] \\
&\quad \textbf{where}\ e' = \mathit{run}\ \mathit{opt}\ e \\
&\mathit{reduce}\ \mathit{opt}\ [\![e_1\ ?\ e_2]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![e_{-1}'\ ?\ e_{-2}']\!] \\
&\quad \textbf{where}\ e_{-1}' = \mathit{run}\ \mathit{opt}\ e_1 \\
&\qquad\quad e_{-2}' = \mathit{run}\ \mathit{opt}\ e_2 \\
&\mathit{reduce}\ \mathit{opt}\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!] && = \mathit{runOpts}\ \mathit{opt}\ [\![\ \textbf{case}\ e'\ \textbf{of}\ bs']\!] \\
&\quad \textbf{where}\ e' = \mathit{run}\ \mathit{opt}\ e \\
&\qquad\quad bs' = \mathit{map}\ (\lambda[\![\mathit{pat} \to e]\!] \to [\![\mathit{pat} \to (\mathit{run}\ \mathit{opt}\ e')]\!])\ bs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{runOpts} :: \mathit{Opt} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{runOpts}\ \mathit{opt}\ e = \textbf{if}\ \mathit{opt}\ e \equiv \emptyset \\
&\qquad\qquad\qquad \textbf{then}\ e \\
&\qquad\qquad\qquad \textbf{else let}\ e' \in \mathit{opt}\ e\ \textbf{in}\ e'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{simplify} :: \mathit{Opt} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{simplify}\ \mathit{opt}\ e = \mathit{fix}\ (\mathit{reduce}\ \mathit{opt})\ e
\end{aligned}
$$

Figure 3.2: A second attempt. Traverse the expression and, at each node, check if an optimization applies.

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's still more we can do. The new approach makes the GAS library usable on larger Curry programs, like the standard Prelude.

3.0.5 Adding More Information

Rather surprisingly our current approach is actually sufficient for compiling Curry. However, to optimize Curry we're going to need more information when we apply a transformation. Specifically, we'll be able to create new variables. To simplify optimizations, we'll 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're allowed to use. We may also need to know if we're rewriting the root of an expression. Fortunately, both of these changes are easy to add. We just change the definition of *Opt* to take all the information as an argument. For each optimization, we'll pass in an $n :: Int$ that represents the next variable v_n that is guaranteed to be fresh. We'll also pass in a $top :: Bool$ that tells us if we're at the top of the stack. We also return a pair of $(Expr, Int)$ to denote the optimized expression, and the number of new variables we used.

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

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're 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 3.3.

3.0.6 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'll 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

```
reduce :: Opt → Bool → Expr → State Int Expr
reduce opt top ⟦v⟧                  = runOpts opt top ⟦v⟧
reduce opt top ⟦l⟧                  = runOpts opt top ⟦l⟧
reduce opt top ⟦f es⟧               = do es' ← mapM (run opt False ⟦es⟧)
                                         runOpts opt top ⟦f es⟧
reduce opt top ⟦ let vs in e⟧       = do vs' ← mapM runVar vs
                                         e'  ← mapM run opt False e
                                         runOpts opt top ⟦ let vs' in e'⟧
  where runVar ⟦v = e⟧ = do e' ← run opt False e
                            return ⟦v = e'⟧
reduce opt top ⟦ let vs free in e⟧ = do e' ← run opt False e
                                        runOpts opt top ⟦ let vs free in e'⟧
reduce opt top ⟦e : t⟧              = do e' ← run opt False e
                                         runOpts opt top ⟦e' : t⟧
reduce opt top ⟦e1 ? e2⟧            = do e_1' ← run opt False e1
                                         e_2' ← run opt False e2
                                         runOpts opt ⟦e_1' ? e_2'⟧
reduce opt top ⟦ case e of bs⟧      = do e'  ← run opt False e
                                         bs' ← mapM runBranch bs
                                         runOpts opt ⟦ case e' of bs'⟧
  where runBranch ⟦pat → e⟧ = do e' ← run opt False e
                                 return ⟦pat → e'⟧

runOpts :: Opt → Bool → Expr → State Int Expr
runOpts opt top e = do v ← get
                       if opt (v, top) e ≡ ∅
                       then return e
                       else do let (e', dv) ∈ opt e
                               put (v + dv)
                               return e'

fix :: (a → State b a) → a → b → a
fix f x s = let (x', s') = runState (f x) s
  in        if x ≡ x'
            then x
            else fix f x' s'
```

steps, where each step contains the rule and position of the rewrite. We describe paths in figure 3.4. To make reconstruction easier, we also include the expression that is the result of the rewrite. This gives us the type:

$$\textbf{type } \mathit{Path} = [\mathit{Int}]$$
$$\textbf{type } \mathit{Step} = (\mathit{String}, \mathit{Path}, \mathit{Expr})$$
$$\textbf{type } \mathit{Derivation} = [\mathit{Step}]$$

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.

$$\textbf{type } \mathit{Opt} = (\mathit{Int}, \mathit{Bool}) \to \mathit{Expr} \to (\mathit{Expr}, \mathit{String}, \mathit{Int})$$

So, what changes do we need to make to the algorithm? Again, there aren't many. 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've elected to call this the *ReWriter* monad.[1] We add a function $\mathit{update} :: \mathit{Expr} \to \mathit{Step} \to \mathit{Int} \to \mathit{ReWriter}\ \mathit{Expr}$ 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's null. Finally *fix* and *simplify* are modified to remember the rewrite steps we've already computed. We change the return type of *simplify* so that we have the list of steps. The full implementation is in figure 3.5

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

$$\begin{aligned}
&\mathit{reconstruct} :: \mathit{Expr} \to [\mathit{Step}] \to \mathit{String}\\
&\mathit{reconstruct}\ _\ [\,] = \texttt{""}\\
&\mathit{reconstruct}\ e\ ((\mathit{rule}, p, \mathit{rhs}) : \mathit{steps}) = \textbf{let}\ e' = e[p \mapsto \mathit{rhs}]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \texttt{"=>_"} +\!\!+ \mathit{rule} +\!\!+ \texttt{" "} +\!\!+ (\mathit{show}\ p) +\!\!+ \texttt{"\textbackslash n"} +\!\!+\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{pPrint}\ e' +\!\!+ \texttt{"\textbackslash n"} +\!\!+\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{reconstruct}\ e'\ \mathit{steps}
\end{aligned}$$

[1] We are still disappointed that we were not able to come up with a way to construct this using a combination of the *Reader* and *Writer* monads. In our opinion this is the single great failing of this dissertation.

$f\ E_0\ E_1\ E_2 \ldots E_n$

let $v\_0 = E_0$
  $v_1 = E_1$
  $\ldots$
in E_{-1}
let vs **in** E_0

$E_0 :: t$

$E_0\ ?\ E_1$

case E_{-1} **of**
  $p_0 \rightarrow E_0$
  $p_1 \rightarrow E_1$
  $\ldots$
  $p_n \rightarrow E_n$

Figure 3.4: The definition of a path for Curry expressions.
E_i denotes that expression E is at position i relative to the current expression. Note that variables and literals don't have subexpressions, so they're excluded. E_{-1} is used for expressions that have a variable number of children, such as let expressions.

```
reduce                                  :: Opt → Path → Expr → ReWriter Expr
reduce opt p ⟦v⟧                        = runOpts opt p ⟦v⟧
reduce opt p ⟦l⟧                        = runOpts opt p ⟦l⟧
reduce opt p ⟦f es⟧                     = do es' ← mapM runArg (zip [0..] es)
                                             runOpts opt p ⟦f es⟧
  where runArg (n, ⟦e⟧) = do e' ← run opt (n : p) e
                              return ⟦e'⟧
reduce opt p ⟦ let vs in e⟧             = do vs' ← mapM runVar (zip [0..] vs)
                                             e'  ← mapM run opt (−1 : p) e
                                             runOpts opt p ⟦ let vs' in e'⟧
  where runVar (n, ⟦v = e⟧) = do e' ← run opt (n : p) e
                                 return ⟦v = e'⟧
reduce opt p ⟦ let vs free in e⟧ = do e' ← run opt (0 : p) e
                                     runOpts opt p ⟦ let vs free in e'⟧
reduce opt p ⟦e : t⟧                    = do e' ← run opt (0 : p) e
                                             runOpts opt p ⟦e' : t⟧
reduce opt p ⟦e1 ? e2⟧                  = do e_1' ← run opt (0 : p) e1
                                             e_2' ← run opt (1 : p) e2
                                             runOpts opt ⟦e_1' ? e_2'⟧
reduce opt p ⟦ case e of bs⟧            = do e'  ← run opt (−1 : p) e
                                             bs' ← mapM runBranch (zip [0..] bs)
                                             runOpts opt ⟦ case e' of bs'⟧
  where runBranch (n, ⟦pat → e⟧) = do e' ← run opt (n : p) e
                                      return ⟦pat → e'⟧


runOpts :: Opt → Path → Expr → ReWriter Expr
runOpts opt p e = do v ← get
                     if opt (v, null p) e ≡ ∅
                     then return e
                     else do let (e', rule, dv) ∈ opt e
                                 update (e', rule, p) dv
                                 return e'


fix :: (a → ReWriter a) → a → Int → [Step] → (a, [Step])
fix f x n steps = let (x', n', steps') = runRewriter (f x) n
                  in  if x ≡ x'
                      then x
                      else fix f x' n' (steps ++ steps')
```

3.0.7 Optimizing the Optimizer

Remember that our optimizing engine is going to run for every optimization, so it's worth taking the time to tune it to be as efficient as possible. There are a few tricks we can use to make the optimization process faster. 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. In fact we can optimize type expressions by just removing the type when we encounter it. We won't ever use the type in the rest of the compiler. 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 the next section we discus the compiler pipeline, and how to transform the FlatCurry into C.

3.1 THE COMPILER PIPELINE

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 Code to ICurry, an intermediate representation that is closer to C. Finally, we compile the code to C. These steps are referred to as preprocess, optimize, postprocess, toICurry, and toC within the compiler. While there are several small details that are important to constructing a working Curry compiler, we'll concern ourselves with the big picture here.

3.1.1 Canonical FlatCurry

The preproces and postprocess 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. We will discuss the

optimization phase in the next section, but for now we can see how transformations work.

Let's 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 can translate the new expression to C in a direct manner. This is the purpose of the preprocess and postprocess steps. Rewrite Curry a expression that doesn't make sense in C to an equivalent Curry expression that we can translate directly to C. Most of the transformations aren't complicated, and consist of disallowing certain syntactic constructs.

The rules are listed in table **??** For clarity, the rules are presented as rewrite rules. However, the implementation is not much more difficult, just more cluttered. As an example, the first let floating rule could be implemented as:

$$\begin{aligned} &\mathit{float}\ \_\ [\![\ \textbf{let } x = (\textbf{let } y = e_1 \textbf{ in } e_2) \textbf{ in } e_3]\!] \\ &\quad = ([\![\ \textbf{let } y = e_1 \textbf{ in let } x = e_2 \textbf{ in } e_3]\!], \texttt{"float 1"}, 0) \end{aligned}$$

We did not need any fresh variable names, so the first parameter is ignored, and we return 0 new variables. The name `"float 1"` is returned for reconstructing the rewrite derivation.

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 straightforward.

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 isn't 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 is beneficial 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 is an infinite loop.

Let Floating

$$\begin{aligned}
&\textbf{let } x = (\textbf{let } y = e_1 \textbf{ in } e_2) \textbf{ in } e_3 \Rightarrow \textbf{let } y = e1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in let } x = e_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in } e_3 \\
&\textbf{let } x = (\textbf{let } y \textbf{ free in } e_1) \textbf{ in } e_2 \Rightarrow \textbf{let } y \textbf{ free} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in let } x = e_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in } e_3 \\
&(\textbf{let } x = e_1 \textbf{ in } e_2)\,?\,e_3 \qquad\quad \Rightarrow \textbf{let } x = e_1 \textbf{ in } (e_2\,?\,e_3) \\
&(\textbf{let } x \textbf{ free in } e_1)\,?\,e_2 \qquad\quad \Rightarrow \textbf{let } x \textbf{ free in } (e_1\,?\,e_2) \\
&f\ (\textbf{let } x = e_1 \textbf{ in } e_2) \qquad\quad \Rightarrow \textbf{let } x = e_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } f\ e_2 \\
&f\ (\textbf{let } x \textbf{ free in } e_2) \qquad\quad \Rightarrow \textbf{let } x \textbf{ free} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } f\ e_2 \\
&\textbf{case } (\textbf{let } x = e_1 \textbf{ in } e_2) \textbf{ of } \ldots \quad \Rightarrow \textbf{let } x = e_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in case } e_2 \textbf{ of } \ldots \\
&\textbf{case } (\textbf{let } x \textbf{ free in } e) \textbf{ of } \ldots \quad \Rightarrow \textbf{let } x \textbf{ free} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in case } e \textbf{ of } \ldots
\end{aligned}$$

Case in Case

$$\begin{aligned}
&\textbf{case } (\textbf{case } e_1 \textbf{ of } b_21 \to e_{2,1} \Rightarrow \textbf{case } e_1 \textbf{ of } b_21 \to \textbf{case } e_{2,1} \textbf{ of } b_11 \to e_{1,1} \\
&\qquad\qquad\qquad\quad b_22 \to e_{2,2} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad b_12 \to e_{2,2} \\
&\qquad\qquad\qquad\quad \ldots) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
&\qquad\quad \textbf{of } b_11 \to e_{1,1} \qquad\qquad\qquad\qquad\quad b_22 \to \textbf{case } e_{2,2} \textbf{ of } b_11 \to e_{1,1} \\
&\qquad\qquad b_12 \to b_12 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad b_12 \to e_{2,2} \\
&\qquad\qquad \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots
\end{aligned}$$

Figure 3.6: GAS rules for putting FlatCurry programs into canonical form

Double Apply

$$\mathit{apply}\ (\mathit{apply}\ f\ [x])\ [y] \quad \Rightarrow \mathit{apply}\ f\ [x, y]$$

Case Apply

$$\mathit{apply}\ (\textbf{case}\ e\ \textbf{of}\ (C\ (\ldots) \to f))\ x \Rightarrow \textbf{case}\ e\ \textbf{of}\ C\ (\ldots) \to \mathit{apply}\ f\ x$$

String Constant

$$c_1 : c_2 \quad \ldots c_n : [\,] \quad \Rightarrow \mathit{StrConst}\ \texttt{"c1c2...cn"}$$

where $c_1\ c_2 \ldots c_n$ *are character literals*

Blocks

$$\begin{aligned}
&\textbf{let}\ x_1 = e_2 && \Rightarrow \mathit{blocks}\ [(x_1, e_1), (x_2, e_2) \ldots (x_n, e_n)] \\
&\quad x_2 = e_2 \quad \textbf{where}\ \mathit{blocks} = \mathit{foldr}\ \mathit{Let}\ e \circ \mathit{scc} \\
&\quad \ldots \\
&\quad x_n = e_n \\
&\textbf{in}\ e
\end{aligned}$$

Alias

$$\begin{aligned}
&\textbf{let}\ x = y\ \textbf{in}\ e && \Rightarrow e\ [x \mapsto y] \\
&\textbf{let}\ x = x\ \textbf{in}\ e && \Rightarrow \textbf{let}\ x = \mathit{loop}\ \textbf{in}\ e
\end{aligned}$$

fill cases

$$\begin{aligned}
&\textbf{case}\ e\ \textbf{of}\ C_1 \ldots \to e_1 && \Rightarrow \textbf{case}\ e\ \textbf{of}\ C_1 \ldots \to e_1 \\
&\qquad\qquad C_2 \ldots \to e_2 && \qquad\qquad\quad C_2 \ldots \to e_2 \\
&\qquad\qquad \ldots && \qquad\qquad\quad \ldots \\
&\qquad\qquad C_k \ldots \to e_k && \qquad\qquad\quad C_k \ldots \to e_k \\
&\qquad\qquad\qquad\qquad && \qquad\qquad\quad C_{k+1} \ldots \to \bot \\
&\qquad\qquad\qquad\qquad && \qquad\qquad\quad \ldots \\
&\qquad\qquad\qquad\qquad && \qquad\qquad\quad C_n \ldots \to \bot \\
&\qquad\qquad \textbf{where}\ C_{k+1} \ldots C_n \in \mathit{Ctrs} - \{\, C_1, C_2, \ldots C_k \,\}
\end{aligned}$$

Figure 3.7: GAS rules for putting FlatCurry programs into canonical form (continued)

Let Case

$$
\begin{array}{lll}
f\ v_1 \ldots v_n = \ldots & \Rightarrow f\ v_1 \ldots v_n & = \ldots \\
\quad \textbf{let}\ x = \textbf{case}\ e\ \textbf{of}\ C_1 \rightarrow e_1 & & \quad \textbf{let}\ x = f\#1\ x_1 \ldots x_k \\
\qquad\qquad\qquad\quad C_2 \rightarrow e_2 & & \quad \textbf{in} \ldots \\
\qquad\qquad\qquad\quad \ldots & f\#1\ x_1 \ldots x_k = & \textbf{case}\ e\ \textbf{of}\ C_1 \rightarrow e_1 \\
\qquad\qquad\qquad\quad C_n \rightarrow e_n & & \qquad\qquad\quad C_2 \rightarrow e_2 \\
\quad \textbf{in} \ldots & & \qquad\qquad\quad \ldots \\
 & & \qquad\qquad\quad C_n \rightarrow e_n
\end{array}
$$

$\textbf{where}\ \{x_1 \ldots x_k\} \equiv \mathit{freeVars}\ [e, e_1, e_2 \ldots e_n]$

Var Case

$$\textbf{case}\ e\ \textbf{of} \ldots \Rightarrow \textbf{let}\ x = e\ \textbf{in case}\ x\ \textbf{of} \ldots$$

Figure 3.8: Rule for moving a let bound case out of a function, and eliminating compound expressions in case-expressions.

Finally the **Fill cases** transformation completes the definitional tree. If we have a case with branches for constructors $C_1, C_2 \ldots C_k$, then we look up the type T that all of these constructor belong to. Then 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 $C_i \rightarrow \bot$.

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. Currently, we only need two transformations for post processing. 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 don't do this transformation in pre-processing because we don't want to split functions apart during optimizations. The **Let-Case** transformation has a single rule given in figure 3.8.

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 preprecess derivation is given in 3.9. At this point point we are ready to transform the cononicalized FlatCurry into

ICurry.

3.1.2 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. It should be noted that restricting the scrutinee of the case statement to a single variable will cause efficiency problems, but we'll address this later.

After we've finished transforming the FlatCurry, the transformation to ICurry is almost trivial. The algorithm from [**?**] can be applied directly 3.11 Currently we need to come up with a new variable for the scrutinee of a case-expression. However, This can either be taken care of in the FlatCurry transformations with the following rule, or we can create a new variable during the transformation itself.

$$\textbf{case}\ e\ \textbf{of}\ \ldots \mid not\ (isVar\ e) \Rightarrow \textbf{let}\ x\_e = e\ \textbf{in case}\ x\_e\ \textbf{of}\ \ldots$$

However, as we'll see in the next section, this will become a moot point. All case-expressions will already only contain variables. Finally we can move onto code generation.

3.1.3 C

Generating C code from ICurry isn't as easy as generating the ICurry code, but it's not much more complicated. We already have a good idea of what the C code should look like, and our ICurry structure fits closely with this. This major 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's not surprising that we take the same approach.

For each Data type D and function f, we generate both a *makeD* function and a *setD*. The difference is that *makeD* will allocate memory for a new node, while *setD* takes an existing node

```
poweraux v1 v2 v3 = case (≡) v3 0 of
                        True → v1
                        False → let v4 = square v2
                                    v5 = halve v3
                                in case (≡) (apply (apply mod v3) 2) 1 of
                                       True → powaux ((∗) v1 v2) v4 v5
                                       False → powaux v1 v4 v5
⇒ Double Apply [1, −1, −1, 0]
poweraux v1 v2 v3 = case (≡) v3 0 of
                        True → v1
                        False → let v4 = square v2
                                    v5 = halve v3
                                in case (≡) (apply mod v3 2) 1 of
                                       True → powaux ((∗) v1 v2) v4 v5
                                       False → powaux v1 v4 v5
⇒ Blocks [1]
poweraux v1 v2 v3 = case ((≡) v3 0) of
                        True → v1
                        False → let v4 = square v2
                                in let v5 = halve v3
                                   in case (≡) (apply mod v3 2) 1 of
                                          True → powaux ((∗) v1 v2) v4 v5
                                          False → powaux v1 v4 v5
```

Figure 3.9: Reducing the *powaux* function defined in the standard Float library. The first reduction occurs in the *False* branch [1] or the in expression [-1] in the scrutinee of the case [-1] in the first argument of the apply node [0], so it has a path of [1,-1,-1,0].

as a parameter, and transforms it to the given type of node. 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`, the make and set functions reset the nondet flag to `false`, and set any children that were passed into the node.

As mentioned in the last chapter, for every function f, we need to generate a function $f|_p$ where p is a path to a case statement. The translation to C code is where we finally generate these functions. Surprisingly, this doesn't have too much of an effect on the code generator. Instead of generating code for a function f, we collect a list of pairs (f, p), where p is a path to a case statement, then we generate code for each (f, p) pair. In practice, it is useful to keep track of the declared variables, so we actually generate code for a triple $(f, vars, p)$. While generating code for (f, p), we track the current position p' of the code we're generating. When we need to generate code to push f onto the stack, such as when the code evaluates a non-deterministic expression, then we push `f_p'` onto the stack.

Aside from that change, generating the C code proceeds exactly as you would expect. Declarations are turned into C declarations; assignments are turned into calls to make functions; return statements are turned into calls to set functions; and case statements are turned into the while/switch construct from the last chapter.

At this point we can now produce a running program. In fact, the code produced by this compiler is already relatively efficient, In many cases it outperforms the current state-of-the-art Curry compilers.

In this chapter, we've built up the GAS tool for performing transformations on FlatCurry code in a simple declarative way. We've already seen how this transformation tool eases the process of compiling programs, but it turns out to be much more powerful. As we'll see in the next chapter, when we apply GAS to optimizations, we can produce some complex optimizations, such as inlining, with rather minimal effort. This allows us to implement new optimizations faster, and test them more thoroughly, than if we had written each optimization by hand. In fact, it's often faster to implement an optimization, and just see what it does to the code, than to try to reason it out ourselves. This lets us iterate quickly on designing new optimizations, and produce more complex and powerful optimizations. Now we're cooking with GAS!

$p \Rightarrow$	$t^* f^*$	*program*
$t \Rightarrow$	$C_1\ C_2 \ldots C_n$	*datatype*
$f \Rightarrow$	$f = b$	*function*
$b \Rightarrow$	d_1	*block*
	$\ldots$	
	d_k	
	a_1	
	$\ldots$	
	a_n	
	s	
$d \Rightarrow$	declare x	*variable declaration*
$\mid$	*declfree* x	**free** *variable declaration*
$a \Rightarrow$	$v = e$	
$s \Rightarrow$	*return* e	*return statement*
$\mid$	$\bot$	*failure*
$\mid$	**case** x **of**	**case** *statement*
	$C_1 \rightarrow b_1$	
	$\ldots$	
	$C_n \rightarrow b_n$	
$e \Rightarrow$	v	*variable expression*
$\mid$	$NODE\ (l, e_1, \ldots, e_n)$	*node creation*
$\mid$	$e_1\ ?\ e_2$	*choice expression*
$v \Rightarrow$	x	*local variable*
$\mid$	$v\ [i]$	*variable access*
$\mid$	$ROOT$	*root variable*
$l \Rightarrow$	C	*constructor label*
$\mid$	f	*function label*
$\mid$	$LABEL\ (v)$	*variable label*

Figure 3.10: Abstract syntax of function definitions in ICurry

$\mathcal{F}\ (f\ (x_1, \ldots, x_n) = e) : = f = \mathcal{B}\ (x_1, \ldots, x_n, e, ROOT)$

$\mathcal{B}\ (x_1, \ldots, x_n, \bot, root) : = \bot$

$\mathcal{B}\ (x_1, \ldots, x_n, e, root)\ : =$

declare x_1

$\ldots$

declare x_n

$\mathcal{D}\ (e)$

$x_1 = root\ [1]$

$\ldots$

$x_n = root\ [n]$

$\mathcal{A}\ (e)$

$\mathcal{R}\ (e)$

$\mathcal{D}\ (\textbf{let}\ x_1, \ldots, x_n\ \textbf{free in}\ e) : =$

free x_1

$\ldots$

free x_n

$\mathcal{D}\ (\textbf{let}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \textbf{in}\ e) : =$

declare x_1

$\ldots$

declare x_n

$\mathcal{D}\ (\textbf{case}\ e\ \textbf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\}) : =$ declare x_e

$\mathcal{A}\ (\textbf{let}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \textbf{in}\ e) : =$

$x_1 = \mathcal{E}\ (e_1)$

$\ldots$

$x_n = \mathcal{E}\ (e_n)$

$[x\_i\ [p] = x\_j \mid x\_j\text{`}\ \textbf{in}\ \text{`}\{x_1, \ldots, x_n\}, e_i \mid\_\ p = x\_j]$

$\mathcal{A}\ (\textbf{case}\ e\ \textbf{of}\ \_) : = x\_e = \mathcal{E}\ (e)$

$\mathcal{R}\ (\textbf{case}\ e\ \textbf{of}\ C_1\ (x_{1,1}, \ldots, x\_1m) \to e_1 : = \textbf{case}\ E\ (e)\ \textbf{of}\ \mathcal{B}\ (x_{1,1}, \ldots, x\_1m, e_1, x\_e)$

$\ldots \qquad\qquad \ldots$

$C_n\ (x_{n,1}, \ldots, x\_nk) \to e_n) \qquad \mathcal{B}\ (x_{n,1}, \ldots, x\_nk, e_n, x\_e)$

$\mathcal{R}\ (e) \qquad\qquad : = return\ E\ (e)$

$\mathcal{E}\ (x) \qquad\qquad : = x$

$\mathcal{E}\ (c\ e_1 \ldots e_n) \qquad : = NODE\ (c, \mathcal{E}\ (e_1), \ldots, \mathcal{E}\ (e_n))$

$\mathcal{E}\ (f\ e_1 \ldots e_n) \qquad : = NODE\ (f, \mathcal{E}\ (e_1), \ldots, \mathcal{E}\ (e_n))$

$\mathcal{E}\ (e_1\ ?\ e_2) \qquad : = \mathcal{E}\ (e_1)\ ?\ \mathcal{E}\ (e_2)$

$\mathcal{E}\ (\textbf{let}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \textbf{in}\ e) : = \mathcal{E}\ (e)$

$\mathcal{E}\ (\textbf{let}\ x_1, \ldots, x_n\ \textbf{free in}\ e) \qquad : = \mathcal{E}\ (e)$

Chapter 4

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's 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'll 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's 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's 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.

4.0.1 A-Normal Form

Before we discuss any substantial optimizations, we need to deal with a significant roadblock to optimizing Curry. Equational reasoning is not valid for Curry programs. Consider the following program.

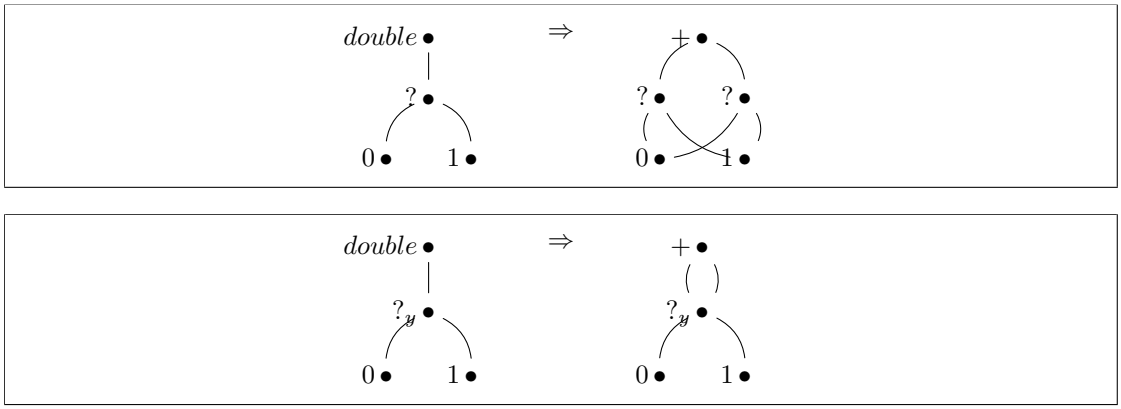
$$\begin{aligned} &\mathit{double}\ x = x + x \\ &\mathit{main} \quad\ = \mathit{double}\ (0\ ?\ 1) \end{aligned}$$

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 isn't the only problem, for a trivial example inlining *id* $(0\,?\,1)$ is fine. We can even duplicate non-deterministic expressions with the following example.

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

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 just fine. So, what is the problem with our first example?

The real problem 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 didn't 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. Every expression that could be copied should be explicitly stored in a 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

$a \Rightarrow v$	*Variable*
$\mid\ l$	*Literal*
$\mid\ \bot$	*Failed*
$e \Rightarrow a_1\ ?\ a\_2$	*Choice*
$\mid\ f_k\ a_1\ a\_2 \ldots a\_n$	*Function Application*
$\mid\ C_k\ a_1\ a\_2 \ldots a\_n$	*Constructor Application*
$\mid$ **let** $v_1 = e_2 \ldots v_n = e_n$ **in** e	*Variable Declaration*
$\mid$ **let** $v_1, v_2, \ldots v_n$ **free in** e	*Free Variable Declaration*
$\mid$ **case** a **of** $\{ p_1 \rightarrow e_1; \ldots p_n \rightarrow e_n \}$	*Case Expression*
$p \Rightarrow C\ v_1\ v_2 \ldots v_n$	*Constructor Pattern*
$\mid\ l$	*Literal Pattern*

Figure 4.1: Restricting Curry expressions to A-Normal Form.
An atom is either a variable, a literal, or a failure. Compound expressions are only allowed to contain atoms.

(ANF) [26]. 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 4.1 and we implement the transformation using GAS in figure 4.2.

As long as we enforce this A-Normal Form structure, we restore equational reasoning for Curry programs. We don't even need to enforce A-Normal Form strictly here. During optimization, it's often useful to be able to inline constructors and partial applications. Since a constructor is computationally inert, we can inline one without fear of problems with non-deterministic expressions. This will be referred to as limited A-Normal Form.

In fact, this idea isn't even new to Curry. In [**?**] FlatCurry programs are translated into a "Normal Form", before evaluation begins. Although, it should be noted that this form is more limited because case statements and choice expressions may contain compound expressions. We choose to flatten these expressions as well for two reasons. It produces more uniform, albeit longer,

$$
\begin{array}{ll}
\textbf{case } e \textbf{ of } \ldots & \Rightarrow \textbf{let } x = e \ \textbf{ in case } x \textbf{ of } \ldots \\
f\ a_1\ a\_2 \ldots e_k \ldots e_n & \Rightarrow \textbf{let } x = e_k \textbf{ in } f\ a_1\ a\_2 \ldots x \ldots e_n \\
e_1\ ?\ e_2 & \Rightarrow \textbf{let } x = e_1 \textbf{ in } x\ ?\ e_2 \\
a_1\ ?\ e_2 & \Rightarrow \textbf{let } x = e_2 \textbf{ in } a_1\ ?\ x
\end{array}
$$

Figure 4.2: Rules for transforming Curry expression to A-Normal Form.
a is used for atoms, e is used for arbitrary expressions, and x is a fresh variable name.

programs, and more optimizing transformations become valid. Some examples of programs in ANF are given in figure 4.3.

4.1 CASE CANCELING

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

$$
\begin{array}{l}
\mathit{notTrue} = \textbf{case } \mathit{True} \textbf{ of} \\
\qquad\qquad\quad \mathit{True} \rightarrow \mathit{False} \\
\qquad\qquad\quad \mathit{False} \rightarrow \mathit{True}
\end{array}
$$

While it is very unlikely that a programmer would actually write this code, but as we'll see, this code comes up frequently when inlining. This is fantastic, because it's clear what we should do here. We know that the *True* branch will be taken, so we might as well discard the case expression altogether.

$$\mathit{notTrue} = \mathit{False}$$

This transformation is called Case Canceling, and it's the workhorse of all of our other optimizations. The transformation, given in 4.4 looks intimidating at first, but it's very straightforward. If the scrutinee of a case is 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

```
fib n = case n < 1                          | fib n = let x = n < 1
            True  -> n                      |         in  case x of
            False -> fib (n - 1) + fib (n - 2)|                 True  -> n
                                            |                 False -> let n1 = n - 1
                                            |                              n2 = n - 2
                                            |                              f1 = fib n1
                                            |                              f2 = fib n2
                                            |                          in  f1 + f2
                                            |
sumPrimes = foldr (+) 0                     |sumPrimes = let v1 = (+)
  . filter isPrime                          |in let v2 = foldr v1 0
  . enumFromTo 1                            |in let v3 = isPrime
                                            |in let v4 = filter v3
                                            |in let v5 = enumFromTo 1
                                            |in let v6 = v4 . v5
                                            |in v2 . v6
```

Figure 4.3: Examples of Curry programs translated to A-Normal Form

after Case Canceling runs. The first is Case Variable elimination. The idea is also simple. Consider the following expression from the optimization of *compare* for *Bool*:

```
...
  in case v2 of
          True  → LT
          False → case v2 of
                      True  → EQ
                      False → case v1 of
                                  True  → GT
                                  False → EQ
...
⇒ Case Var [−1, 0, −1]
...
  in case v2 of
          True  → LT
          False → case False of
                      True  → EQ
                      False → case v1 of
                                  True  → GT
                                  False → EQ
...
⇒ Case Cancel [−1, 0, −1, 1]
...
  in case v2 of
          True  → LT
          False → case v1 of
                      True  → GT
                      False → EQ
...
```

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. If we're

replacing a variable, which represents a node in our expression graph, with a completely distinct literal value, how do we know that this replacement is valid? This isn't clear. In fact, in general it's not valid to replace a variable. That variable could be shared, and it could represent a non-deterministic expression. Case Canceling is fine, because we only change a case statement, which only affects flow of control. It doesn't actually affect any of the data in the expression graph. However, we can justify this by looking at the computation space for our expression again. Suppose that we have the expression **case** x **in** e. If x is deterministic, then there is no problem. If x is non-deterministic, then x has already been reduced to head normal form, and been pushed on the backtracking stack. Furthermore, the root of this function has also been pushed on the backtracking stack, since **case** x **of** e depended on a non-deterministic variable. So, if x has changed while backtracking, the current expression has been undone. Therefore replacing x with a constructor in this expression has no effect.

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. Furthermore if a variable is never used, then it can also be removed. Finally, if we have **let** $x = e$ **in** x, then we don't need to create the variable x. These are all clearly correct, as long as we're careful to make sure that our variable definitions aren't recursive.

Now that we've 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, Inlining isn't obviously correct, and, in fact, we have to do a lot of work to inline functions in Curry.

4.2 INLINING

As mentioned at the start of this chapter, inlining isn't generally valid in Curry. So, we need to establish cases when inlining is valid, determine when it's a good idea to inline, and ensure that our inlining algorithm is correct. This work is largely based on [**?**, 20].

Similarly to [49], 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's definition. For example **let** $x = True$ **in** $not\ x$ could inline to $not\ True$. When we use the term *reduction*, we are referring to replacing a function call with the body of the function. Again, as an example **let** $x = True$ **in** $not\ x$ could reduce to:

Case Cancel

$$\begin{array}{lll}
\textbf{case}\ (C_i\ e_{i,1} \ldots e_{k_i})\ \textbf{of} & \Rightarrow \textbf{let} & x_{i1} = e_1 \\
\quad C_1\ x_{1,1} \ldots x_{1,k_1} \rightarrow e_1 & & \ldots \\
\quad C_2\ x_{2,1} \ldots x_{2,k_2} \rightarrow e_2 & \textbf{in let}\ x_{i,k_i} = e_{k_i} & \\
\quad \ldots & \textbf{in} \quad e_i & \\
\quad C_i\ x_{i1} \ldots x_{i,k_i} \rightarrow e_i & & \\
\quad \ldots & & \\
\quad C_n\ x_{n,1} \ldots x_{n,k_n} \rightarrow e_n & &
\end{array}$$

Case Var

$$\begin{array}{ll}
\textbf{case}\ x\ \textbf{of} & \Rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad (C \ldots) \rightarrow \ldots x \ldots & \qquad (C \ldots) \rightarrow \ldots (C \ldots) \ldots
\end{array}$$

Dead Code

$$\begin{array}{ll}
\textbf{let free in}\ e & \Rightarrow e \\
\textbf{let in}\ e & \Rightarrow e \\
\textbf{let}\ v\ \textbf{free in}\ e & \mid v \notin e \Rightarrow e \\
\textbf{let}\ v = \ldots \textbf{in}\ e & \mid v \notin e \Rightarrow e \\
\textbf{let}\ v = e\ \ \textbf{in}\ v & \mid v \notin e \Rightarrow e
\end{array}$$

Figure 4.4: Case Canceling, Case Variable, and Dead Code Elimination optimizations.

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

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've 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 wouldn't be able to optimize it anyway, because $\circ$ is defined in the standard Prelude. If we can't inline the definition of $\circ$, then we're 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 IR's like Haskell's Core, we could solve this problem by inlining a lambda expression, but it's 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 inline all functions that we've already optimized.

4.2.1 Partial Applications

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

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

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

$$f \circ g = compLambda_1\ f\ g$$
$$compLambda\ f\ g\ x = f\ (g\ x)$$

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

let *v1* $= (+)$
in let *v2* $= foldr\ v1\ 0$
in let *v3* $= isPrime$
in let *v4* $= filter\ v3$
in let *v5* $= enumFromTo\ 1$
in let *v6* $= v4 \circ v5$
in *v2* $\circ$ *v6*
$REDUCE \Rightarrow [-1,-1,-1,-1,-1,-1]$
let *v1* $= (+)$
in let *v2* $= foldr_1\ v1\ 0$
in let *v3* $= isPrime\_1$
in let *v4* $= filter\_1\ v3$
in let *v5* $= enumFromTo\_1\ 1$
in let *v6* $= v4 \circ v5$
in $compLambda_1$
$REDUCE \Rightarrow [-1,-1,-1,-1,-1]$
let *v1* $= p_2$
in let *v2* $= foldr_1\ v1\ 0$

in let *v3* = *isPrime_1*
in let *v4* = *filter_1 v3*
in let *v5* = *enumFromTo_1* 1
in let *v6* = $compLambda_1$ *v4 v5*
in $compLambda_1$ *v2 v6*

At this point there's no more optimization that can be done, because everything is a partial function. But this is clearly ridiculous. We've 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 function is a partial application, then we can rewrite our definition.

$f\ v_1 \ldots v_k = g_n \ldots$
$\Rightarrow$ **Add Missing Variables**
$f\ v_1 \ldots v_k\ x_1 \ldots x_n = apply\ (g_n \ldots)\ x_1 \ldots x_n$
where $x_1 \ldots x_n$ *are fresh variables*

The *sumPrimes* functions is transformed with the derivation in 4.5 and we can continue to optimize the function.

Unfortunately, since this involves the definition of the function itself, and not just its body, we can't use the GAS system here. However, this does solve our problem. It leads to a new problem though. Since we are changing the arity of functions, any function calls may have the wrong arity.

4.2.2 The Function Table

In order to keep track of all of the functions we've optimized we create a function lookup table called $\mathcal{F}$. 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 the heuristic we use is easy to tweak, but at the very least f should not be external, or a loop breaker, and should not be too big. $U_{\mathcal{F}}\ x\ f\ e$ 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's used as the scrutinee of a case expression, or it's used in a function that's likely to be reduced. $S_{\mathcal{F}}\ f$ returns True if

let $v_1 = (+)$
in let $v_2 = \mathit{foldr}\ v_1\ 0$
in let $v_3 = \mathit{isPrime}$
in let $v_4 = \mathit{filter}\ v_3$
in let $v_5 = \mathit{enumFromTo}\ 1$
in let $v_6 = \mathit{compLambda}_1\ v_4\ v_5$
in $\mathit{compLambda}_1\ v_2\ v_6$

$\Rightarrow$ **Add Missing Variables**

apply (**let** $v_1 = (+)$
in let $v_2 = \mathit{foldr}\ v_1\ 0$
in let $v_3 = \mathit{isPrime}$
in let $v_4 = \mathit{filter}\ v_3$
in let $v_5 = \mathit{enumFromTo}\ 1$
in let $v_6 = \mathit{compLambda}_1\ v_4\ v_5$
in $\mathit{compLambda}_1\ v_2\ v_6$) x_1

...

$\Rightarrow$ **Let Floating**

let $v_1 = (+)$
in let $v_2 = \mathit{foldr}\ v_1\ 0$
in let $v_3 = \mathit{isPrime}$
in let $v_4 = \mathit{filter}\ v_3$
in let $v_5 = \mathit{enumFromTo}\ 1$
in let $v_6 = \mathit{compLambda}_1\ v_4\ v_5$
in $\mathit{apply}\ (\mathit{compLambda}_1\ v_2\ v_6)\ x_1$

$\Rightarrow$ **Unapply**

let $v_1 = (+)$
in let $v_2 = \mathit{foldr}\ v_1\ 0$
in let $v_3 = \mathit{isPrime}$
in let $v_4 = \mathit{filter}\ v_3$
in let $v_5 = \mathit{enumFromTo}\ 1$
in let $v_6 = \mathit{compLambda}_1\ v_4\ v_5$
in $\mathit{compLambda}\ v_2\ v_6\ x_1$

Figure 4.5: Adding a missing variable to *sumPrimes*

f is a simple reduction with no case expressions. It's 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.

4.2.3 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 [20, 49], and the solutions for other languages apply equally well to Curry.

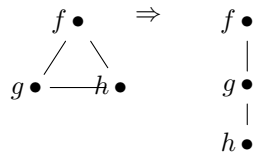
The problem seems very complicated at the start. We want to know what is the best order to optimize functions. Fortunately there's a very natural solution. If possible we should optimize a function before we optimize any function that calls it. This turns out to be an exercise in Graph Theory.

We define the Call Graph of a set of function $\mathbf{F} = \{f_1, f_2, \ldots f_n\}$ to be the graph $G_{\mathbf{F}} = (\mathbf{F}, \{f_i \rightarrow f_j | f_i \ \mathit{calls}\ f_j\})$. This problem reduces to finding the topological ordering of $G_{\mathbf{F}}$. Unfortunately, if $\mathbf{F}$ contains any recursion, then the topological ordering isn't 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 a "loop breaker" which is removed from the graph, and attempt to find the topological order of each component again. This process repeats until our graph is acyclic.

These loop breakers are marked in $\mathcal{F}$, 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:

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



The graph for this function is a triangle, because f calles 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.

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

4.2.4 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 isn't valid in Curry. This is a large change from lazy languages like Haskell, where it's 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 [49] 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't inline a lambda since they don't exist. The second is that we've translated FlatCurry to A-Normal Form. While Haskell programs are put into A-Normal Form when translating to STG code [51], 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, a literal or a variable, then we don't affect the computed results.

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

Proof. The cases for literals and variables are trivial, so we omit them here. Also, note that a partial application functions identically to a constructor, since the only place they can be reduced is the constructor case of `apply_hnf`.

Now let's consider the case of **let** $x = C \ldots$ **in** e. Assume that C is applied to either variables or literal values. It may be the case that $e[x \mapsto C \ldots]$ duplicates C. However, since C is already constructor, it is a deterministic node. By the path compression theorem $e[x \mapsto C \ldots]$ computes the same results.

If C is applied to other constructors or partial applications, then the result follows from structural induction. □

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 **Alias**. We add two new rules. **Let Folding** allows us to move variable definitions closer to where they're 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 4.6. Note that the **Unapply** rule corresponds exactly to the evaluation step for application nodes in our semantics.

4.2.5 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've already done the hard work. At this point, the only place a function is allowed to appear in our expressions is as the root of the expression, 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's now valid to reduce any function we come across.

Theorem 8 (reduction)**.** *let* e *be an expression in limited A-Normal Form, and let* $e|_p = f\ e_1 \ldots e_n$*. then* $e[p \mapsto (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]]$ *computes the same results as* e*.*

Let Folding

$\mathbf{let}\ v = e_1\ \mathbf{in\ case}\ e\ \mathbf{of} \ldots \mid v \notin e \quad \Rightarrow \mathbf{case}\ e\ \mathbf{of}\ C_i \ldots \to \mathbf{let}\ v = e_1\ \mathbf{in} \ldots$

Unapply

$apply\ f_k\ as \mid k < n \quad \Rightarrow apply\ (apply\ f\ as_1)\ as_2$

$\mid k \equiv n \quad \Rightarrow f\ as$

$\mid k > n \quad \Rightarrow f_{k-n}\ as$

Inline Constructor

$\mathbf{let}\ x = C\ v_1 \ldots v_n\ \mathbf{in}\ e \mid x \notin \{v_1, \ldots v_n\} \Rightarrow e[x \mapsto C\ v_1\ v_2 \ldots v_n]$

Inline Partial

$\mathbf{let}\ x = f_k\ v_1 \ldots v_n\ \mathbf{in}\ e \mid x \notin \{v_1, \ldots v_n\} \Rightarrow e[x \mapsto f_k\ v_1\ v_2 \ldots v_n]$

Inline Literal

$\mathbf{let}\ x = l\ \mathbf{in}\ e \quad \Rightarrow e[x \mapsto l]$

Figure 4.6: rules for variable inlining.
We need to ensure that x isn't used recursively before we inline it.

Proof. If f is the root of the expression, or f is the root of a branch or let, then this is equivalent to taking a step in our semantics. If that particular branch of the case is never evaluated at runtime, then reducing f has no effect.

If f is a let bound variable, i.e. $\mathbf{let}\ x = f \ldots \mathbf{in} \ldots$, when we replace f it's still bound to a single node in the graph.

Furthermore, only nodes that can be duplicated while reducing f are the arguments. However, the arguments are constrained to be literals, constructors, variables, or partial applications. Therefore, we are never duplicating any choice nodes.

Finally, replacing f with it's definition is a single deterministic step, so we can apply the path compression theorem again to obtain our result.

□

We give the GAS rules for reduction in figure 4.7. The rules are straightforward, and 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 [49] and it works very well. Although, since FlatCurry uses numbers exclusively to represent variables, we don't get

Reduce Base		
$f\ e_1 \ldots e_n$	$\mid top\ \&\ I_{\mathcal{F}}\ f$	$\Rightarrow (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$
Reduce Branch		
case e **of** $Ctr \rightarrow f\ e_1 \ldots e_n$	$\mid I_{\mathcal{F}}\ f$	$\Rightarrow$ **case** e **of** $Ctr \rightarrow (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$
Reduce Let		
let $\ldots$ **in** $f\ e_1 \ldots e_n$	$\mid I_{\mathcal{F}}\ f$	$\Rightarrow$ **let** $\ldots$ **in** $(B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$
Reduce Useful		
let $x = f\ e_1 \ldots e_n$ **in** e	$\mid U_{\mathcal{F}}\ x\ f\ e$	$\Rightarrow$ **let** $x = (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$ **in** e
Reduce Simple		
let $x = f\ e_1 \ldots e_n$ **in** e	$\mid S_{\mathcal{F}}\ f$	$\Rightarrow$ **let** $x = (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$ **in** e
Reduce Cancels		
let $x = f\ e_1 \ldots e_n$ **in** e	$\mid C_{\mathcal{F}}\ f\ e$	$\Rightarrow$ **let** $x = (B_{\mathcal{F}} f)[v_1 \mapsto e_1, \ldots v_n \mapsto e_n]$ **in** e

Figure 4.7: Rules for reduction.
All expressions are kept in A-Normal Form. **Reduce Base** is only run if we are at the root of the body. While the last three rules are effectively the same, tt's useful to keep them separated for debugging reduction derivations.

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 don't make a needless call to *double*, but we've 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, 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. Shortcutting is a new optimization to Curry.

$double\ x = x + x$

$main = double\ (0\ ?\ 1)$

$double\ (0\ ?\ 1)$

$ANF\ [1]$

let $v_1 = 0\ ?\ 1$

in $double\ v_1$

Reduce Let $[1]$

let $v_1 = 0\ ?\ 1$

in $v_1 + v_1$

Figure 4.8: Derivation of $double\ (0\ ?\ 1)$ showing that we still arrive at an equivalent expression.

$$
\begin{aligned}
&\mathit{not}\ v_1 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{False}\\
&\qquad \mathit{False} \to \mathit{True}\\
&v_1 \wedge v_2 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \quad \to v_2\\
&\qquad \mathit{False} \quad \to \mathit{False}\\
&v_1 \vee v_2 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \quad \to \mathit{True}\\
&\qquad \mathit{False} \quad \to v_2\\
&v_1 \leqslant v_2 = \mathit{not}\ v_1 \vee v_2
\end{aligned}
$$

$(\mathit{not}\ v_1) \vee v_2$

$\Rightarrow$ **ANF App** $[\,]$

$$
\begin{aligned}
&\textbf{let}\ v_3 = \mathit{not}\ v_1\\
&\textbf{in}\ v_3 \vee v_2
\end{aligned}
$$

$\Rightarrow$ **Reduce Useful** $[\,]$

$$
\begin{aligned}
&\textbf{let}\ v_3 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{False}\\
&\qquad \mathit{False} \to \mathit{True}\\
&\textbf{in}\ v_3 \vee v_2
\end{aligned}
$$

$\Rightarrow$ **Reduce Let** $[\,]$

$$
\begin{aligned}
&\textbf{let}\ v_3 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{False}\\
&\qquad \mathit{False} \to \mathit{True}\\
&\textbf{in case}\ v_3\ \textbf{of}\\
&\qquad \mathit{True} \to v_3\\
&\qquad \mathit{False} \to v_2
\end{aligned}
$$

$\Rightarrow$ **Case Var** $[-1]$

$$
\begin{aligned}
&\textbf{let}\ v_3 = \textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{False}\\
&\qquad \mathit{False} \to \mathit{True}\\
&\textbf{in case}\ v_3\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{True}\\
&\qquad \mathit{False} \to v_2
\end{aligned}
$$

$\Rightarrow$ **Inline Case in Case** $[\,]$

$$
\begin{aligned}
&\textbf{case}\ (\textbf{case}\ v_1\ \textbf{of}\\
&\qquad \mathit{True} \to \mathit{False}\\
&\qquad \mathit{False} \to \mathit{True})\ \textbf{of}\\
&\quad \mathit{True} \to \mathit{True}
\end{aligned}
$$

$\Rightarrow$ **Inline Constructor** $[0]$

case v_1 **of**

$\quad$ *True* $\rightarrow$ **case** *False* **of**

$\quad\quad\quad\quad\quad\quad$ *True* $\rightarrow$ *True*

$\quad\quad\quad\quad\quad\quad$ *False* $\rightarrow v_2$

$\quad$ *False* $\rightarrow$ **let** *v8* $=$ *True*

$\quad\quad\quad$ **in case** *v8* **of**

$\quad\quad\quad\quad\quad$ *True* $\rightarrow$ *True*

$\quad\quad\quad\quad\quad$ *False* $\rightarrow v_2$

$\Rightarrow$ **Case Cancel Constructor** $[0]$

case v_1 **of**

$\quad$ *True* $\rightarrow v_2$

$\quad$ *False* $\rightarrow$ **let** *v8* $=$ *True* **in**

$\quad\quad\quad$ **in case** *v8* **of**

$\quad\quad\quad\quad\quad$ *True* $\rightarrow$ *True*

$\quad\quad\quad\quad\quad$ *False* $\rightarrow v_2$

$\Rightarrow$ **Inline Constructor** $[1]$

case v_1 **of**

$\quad$ *True* $\rightarrow v_2$

$\quad$ *False* $\rightarrow$ **case** *True* **of**

$\quad\quad\quad\quad\quad\quad$ *True* $\rightarrow$ *True*

$\quad\quad\quad\quad\quad\quad$ *False* $\rightarrow v_2$

$\Rightarrow$ **Case Cancel Constructor** $[1]$

case v_1 **of**

$\quad$ *True* $\rightarrow v_2$

$\quad$ *False* $\rightarrow$ *True*

Figure 4.10: Derivation of $\leqslant$ for *Bool* condinued

Chapter 5

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, Shortcutting is an attempt to remove the node created as the scrutinee of a case expression. In lazy functional languages like Haskell, this isn't a problem, but in Curry the scrutinee may be non-deterministic, and so we need a plan for dealing with that. Finally, 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.

5.1 UNBOXING

So far we've avoided talking about operations in Curry for primitive data types *Int*, *Char*, and *Float*. This is primarily because all primitive values in Curry are boxed, and 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 that representation from the beginning.

We chose to follow the style of Unboxing from Launchbury et al. [**?**] and represent all boxes explicitly in FlatCurry. This has several advantages, but one of the most important is that we can apply optimizations to the boxes themselves.

Let's look at an example to see how this works. Consider the function to compute Fibonacci numbers. We will work with this example extensively in the next couple of optimizations, in an attempt to see how much we can optimize it.

$$
\begin{aligned}
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n = \textbf{case}\ n \leqslant 1\ \textbf{of} \\
&\qquad\qquad \mathit{True} \to n
\end{aligned}
$$

$$\mathit{False} \rightarrow \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$$

After translating to A-Normal Form we have:

$$\begin{aligned}
&\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{fib}\ n = \textbf{let}\ \mathit{cond} = n \leqslant 1\\
&\qquad \textbf{in case}\ \mathit{cond}\ \textbf{of}\\
&\qquad\qquad \mathit{True}\ \rightarrow n\\
&\qquad\qquad \mathit{False} \rightarrow \textbf{let}\ n_1 = n - 1\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ f_1 = \mathit{fib}\ n_1\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ n_2 = n - 2\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ f_2 = \mathit{fib}\ n_2\\
&\qquad\qquad\qquad\qquad \textbf{in}\ f_1 + f_2
\end{aligned}$$

Unfortunately, this function can't really be optimized. The *fib* function is recursive, so we can't reduce it, and $n-1$ is a primitive operation. However, we use a lot of memory for this function. We create 8 nodes for each recursive call, and there's really no need for this. The problem is that each of our primitive operations must be represented as a node to have a uniform representation in Curry.

5.1.1 The Transformation

The idea behind the Unboxing transformation is that we represent every box around primitive operations explicitly. The expression $1+2$ is transformed into $\mathit{Int}\ 1 + \mathit{Int}\ 2$, and the *fib* function is transformed into:

$$\begin{aligned}
&\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{fib}\ n = \textbf{let}\ \mathit{cond} = n \leqslant \mathit{Int}\ 1\\
&\qquad \textbf{in case}\ \mathit{cond}\ \textbf{of}\\
&\qquad\qquad \mathit{True}\ \rightarrow n\\
&\qquad\qquad \mathit{False} \rightarrow \textbf{let}\ n_1 = n - \mathit{Int}\ 1\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ f_1 = \mathit{fib}\ n_1\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ n_2 = n - \mathit{Int}\ 2\\
&\qquad\qquad\qquad\qquad \textbf{in let}\ f_2 = \mathit{fib}\ n_2\\
&\qquad\qquad\qquad\qquad \textbf{in}\ f_1 + f_2
\end{aligned}$$

This doesn't seem like we've actually helped at all, but the real improvement comes in the implementation of $x + y$, $x - y$ and $x \leqslant y$. Let's look at $x + y$ first. We can implement this operation using a primitive add operation. This will be translated into an add instruction at the C level.

$$
\begin{aligned}
x + y = \ & \textbf{case } x \textbf{ of} \\
& \quad \mathit{Int}\ x_{\text{prim}} \rightarrow \textbf{case } y \textbf{ of} \\
& \qquad \mathit{Int}\ y_{\text{prim}} \rightarrow \textbf{let } v = +_{\text{prim}}\ x_{\text{prim}}\ y_{\text{prim}} \\
& \qquad\qquad \textbf{in } \mathit{Int}\ v
\end{aligned}
$$

We can implement $x \leqslant y$ in a similar fashion:

$$
\begin{aligned}
x \leqslant y = \ & \textbf{case } x \textbf{ of} \\
& \quad \mathit{Int}\ x_{\text{prim}} \rightarrow \textbf{case } y \textbf{ of} \\
& \qquad \mathit{Int}\ y_{\text{prim}} \rightarrow \leqslant_{\text{prim}}\ x_{\text{prim}}\ y_{\text{prim}}
\end{aligned}
$$

The difference is that $\leqslant_{\text{prim}}$ needs to return a *True* or *False* value. We can implement this as `x_prim <= y_prim ? make_Prelude_True() : make_Prelude_False()`. Now we actually have something we can optimize. We still can't do a lot of with the recursive calls to *fib*, but let's see what happens. We end up with the definition for *fib* in figure 5.1. The full derivation can be seen in the appendix.

As we can see, the code is significantly longer, but now we've 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 don't need to allocate any memory for them. Unfortunately, there's still a problem. We're still allocating 1 node for $cond, f_1, f_2$ and 2 nodes for the Int constructors. So, we're still allocating 5 nodes. This is an improvement, but we can certainly do better.

5.1.2 Primitive Conditions

The first optimization is that we really don't need to allocate memory for *cond*. $x \leqslant y$ really should be a primitive operation returning a Boolean value, but this doesn't have an equivalent in FlatCurry, so we introduce the **pcase** construct.

$$
\begin{aligned}
& \textbf{pcase } \mathit{primCond} \textbf{ of} \\
& \quad \mathit{True} \rightarrow e_t \\
& \quad \mathit{False} \rightarrow e_f
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{fib}\ n = \textbf{case}\ n\ \textbf{of} \\
\quad \mathit{Int}\ v_2 \rightarrow \textbf{let}\ \mathit{cond} = \leqslant_{\text{prim}}\ v_2\ 1 \\
\qquad\quad \textbf{in case}\ \mathit{cond} \\
\qquad\qquad\quad \mathit{True} \rightarrow n \\
\qquad\qquad\quad \mathit{False} \rightarrow \textbf{let}\ n_1 = -_{\text{prim}}\ v_2\ 1 \\
\qquad\qquad\qquad\quad \textbf{in let}\ f_1 = \mathit{fib}\ (\mathit{int}\ n_1) \\
\qquad\qquad\qquad\quad \textbf{in case}\ f_1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Int}\ p_1 \rightarrow \textbf{let}\ n_2 = -_{\text{prim}}\ v_2\ 2 \\
\qquad\qquad\qquad\qquad\qquad \textbf{in let}\ f_2 = \mathit{fib}\ (\mathit{int}\ n_2) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in case}\ f_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{Int}\ p_2 \rightarrow \textbf{let}\ r = +_{\text{prim}}\ p_1\ p_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{Int}\ r
\end{array}
$$

Figure 5.1: Optimized *fib* after Unboxing

The *primCond* must be a primative condition expression, which is either $\equiv_{\text{prim}}$ 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.

```
if([compile primCond])
{
    [compile e_t]
}
else
{
    [compile e_f]
}
```

We don't need to worry about the backtracking stack, because a primitive condition can't be non-deterministic. Now, we can eliminate the *cond* node. After implementing this construct, the new version is in figure 5.2. Now we're down to 4 nodes, but we can still do better. The next challenge is Unboxing parameters.

$$
\begin{array}{l}
\mathit{fib}\ n = \textbf{case}\ n\ \textbf{of} \\
\quad \mathit{Int}\ v_2 \rightarrow \textbf{pcase}\ \leqslant_{\text{prim}}\ v_2\ 1 \\
\quad\quad \mathit{True} \rightarrow n \\
\quad\quad \mathit{False} \rightarrow \textbf{let}\ n_1 = -_{\text{prim}}\ v_2\ 1 \\
\quad\quad\quad\quad \textbf{in let}\ f_1 = \mathit{fib}\ (\mathit{int}\ n_1) \\
\quad\quad\quad\quad \textbf{in case}\ f_1\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad \mathit{Int}\ p_1 \rightarrow \textbf{let}\ n_2 = -_{\text{prim}}\ v_2\ 2 \\
\quad\quad\quad\quad\quad\quad\quad \textbf{in let}\ f_2 = \mathit{fib}\ (\mathit{int}\ n_2) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{in case}\ f_2\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Int}\ p_2 \rightarrow \textbf{let}\ r = +_{\text{prim}}\ p_1\ p_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \mathit{Int}\ r
\end{array}
$$

Figure 5.2: The *fib* function with primitive cases

5.1.3 Strictness Analysis

Unfortunately, Unboxing parameters is slightly more complicated. Earlier we eliminated boxes from local variables in the function. However, if a parameter is unboxed, then we may need to store a primitive value as a child of a node. This requires a fundamental change to our node datatype. Fortunately, C already has a mechanism for this. We use a `union` to tie integers, characters, floating point numbers, and nodes together. The full definition for a node in our expression graph is given below.

```
typedef union field
{
    struct Node*  n; //normal node child
    union field*  a; //array child (for children[3])
    unsigned long c; //primitive character
    long          i; //primitive int
    double        f; //primitive float
} field ;
```

```
typedef struct Node
{
    Int missing;
    bool nondet;
    Symbol* symbol;
    field children[4];
} Node;
```

Now that we have the ability to store primitive values in a node, we need to figure out when storing a primitive value is actually valid. Fortunately, this is a well studied problem [44, 40, 54].

Lazy functional languages often try to remove laziness for efficiency reasons. We don't want to create an expression for a primitive value if we're only going to deconstruct it, so it becomes useful to know what parameters in a function must be evaluated, or what parameters in a function are strict. This strictness analysis has been a major focus of research in the lazy functional community.

We implemented an earlier form of strictness analysis discovered by Mycroft [44]. This works by determining if a parameter is strict by abstract interpretation, which sounds complicated, but in reality it's actually a easy idea. Each parameter is represented as a variable, and the body of the function is converted into a Boolean expression. The translation is similar to [44], so we don't go through it here. There are newer ideas for strictness analysis [40, 54], but Mycroft's solution is sufficient for our purposes, so better implementations are outside the scope of this research.

Once we know which arguments are strict we, can split the function into a wrapper function and a worker function [54]. We can see this with *fib* in figure 5.3, and the optimized version in figure 5.4. Notice that *fib* is no longer recursive, so we can inline it. After optimization we have the following definition of *fib#worker*.

We're down to allocating 2 nodes. We only need to allocate nodes for the calls to *fib#worker*. This means that we've reduced our memory consumption by 75%. That's a huge improvement, but we can still do better. With the next optimization we look at how to remove the remaining allocations.

```
fib n = case n of
  Int v2 → fib#worker v2

fib#worker v1 = let n = Int v1
  in case n of
    Int v2 →
      in pcase ⩽prim v2 1
        True → n
        False → let n1 = −prim v2 1
          in let f1 = fib (int n1)
          in case f1 of
            Int p1 → let n2 = −prim v2 2
              in let f2 = fib (int n2)
              in case f2 of
                Int p2 → let r = +prim p1 p2
                  in Int r
```

Figure 5.3: The *fib* function after strictness analysis.

```
fib#worker v2 =
  pcase ⩽prim v2 1 of
        True → Int v1
        False → let n1 = −prim v2 1
                  in let f1 = fib#worker n1
                  in case f1 of
                        Int p1 → let n2 = −prim v2 2
                                   in let f2 = fib#worker n2
                                   in case f2 of
                                         Int p2 → let r = +prim p1 p2
                                                    in Int r
```

Figure 5.4: The *fib* function after strictness analysis and optimization.

5.2 SHORTCUTTING

In the last section were able to optimize the *fib* function from allocating 8 nodes per recursive call to only allocating 2 nodes per recursive call. However, we were left with a problem that we can't solve by a code transformation.

let $f_1 = \mathit{fib\#worker}\ n_1$
in case f_1 **of**
$\mathit{Int}\ p_1 \rightarrow \ldots$

This is unfortunate, because we never use f_1 after the **case** expression. It seems like we should be able to avoid constructing the node, and in fact, implementations of functional languages do avoid this. Unfortunately, we really do need a physical node. The reason is because $\mathit{fib\#worker}\ n_1$ could be non-deterministic.

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 *fib#worker* and determine if it's 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 *fib#worker* is deterministic, the expression $\mathit{fib\#worker}\ n_1$ could still be non-deterministic, so any sort of determinism analysis is going to fail for any expression that contains a parameter to the function. This is going to be very restrictive for any possible optimization.

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's not possible for this node to be shared with any part of the expression graph. This leads to a new idea. If we need a node for f_1 could we avoid allocating memory for that node? Well, sometimes we can.

The idea here is simple, but the implementation becomes tricky. We want to use a single, statically allocated, node for every variable that's only used as the scrutinee of a case.

There are two steps to the optimization. The first step is marking every node that's only used as the scrutinee, and the second is swaping that variable for the statically allocated node during code generation. We call this node `RET` for return.

Marking the node can be done in FlatCurry using GAS. The only effect of this rule is to mark x' as a variable that can be stored in the `RET` node.

Case Call
let $x = e_1$ **in case** x **of** $e_2 \mid x \notin e_1, e_2 \Rightarrow$ **let** $x' = e_1$ **in case** x' **of** e_2

While this transformation is simple enough, we need to determine if it's valid.

First we assume that e_1 is a deterministic expression. In that case, there is only one thing that could go wrong. It's possible that e_1 also reduces an expression that could be stored in `RET`. For example, consider the program:

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

In the evaluation of *main*, we can store $f\ 3$ in the `RET` node, but while we are evaluating $f\ 3$, we store $g\ 3$ in the same `RET` node. While this is concerning, it's not actually a problem. At the beginning of `f_hnf`, we store all of the children of `root` as local variables, and then when we've computed the value, we overwrite the `root` node. 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 doesn't actually affect the results.

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 5.5.

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

5.2.1 Non-deterministic RET Nodes

The problem with the scheme we've developed so far is that if `RET` is non-deterministic, then we may push it, or an expression containing it, onto the backtracking stack. 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's 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

```
void main_hnf(Node* root)
{
    set_f(RET, make_int(3));
    Node* RET_forward = RET;
    nondet = false;
    while(true)
    {
        switch(RET_forward->tag)
        {
            ...
            case True:
                if(nondet)
                    ...
                set_int(root, 0);
                return;
                ...
        }
    }
```

Figure 5.5: First attempt at compiling *main* with Shortcutting.

case. Instead, we will only allocate a variable if `RET` is non-deterministic. This means that for deterministic expression, we don't 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 5.6.

5.2.2 RET hnf Functions

While this solution is better, it's still not correct. Three things can still go wrong here. The first problem is that `RET` might have been reduced to a forwarding node, so it might not be responsible for the non-determinism, such as **case** *id* (0 ? 1) **of** .... There's clearly non-determinism here, but the *id* node isn't the cause of it, so it shouldn't be pushed on the backtracking stack.

Another problem is that, if `RET` is a forwarding node, then the node it forwards to might have reused `RET`. This is a much more serious problem, because we would push the wrong value on the backtracking stack.

Finally, we still haven't avoided putting `RET` on the backtracking stack, because if `RET` is non-deterministic, it will be pushed on the stack as the left hand side of a stack frame. While we're reducing `RET`, we need to know what node to push on the stack. This means that both the caller and the callee need to know what node we created.

This is starting to seem hopeless. 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__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'll store the contents of `RET` if we discover evaluating f is non-deterministic. At the end of the function, we return `backup`. Now both the caller and callee agree on `backup`. Furthermore, since `backup` is a local variable, it's not affected if f reuses `RET` over the course of its evaluation. This leads to the definition for `f_RET_hnf` in figure 5.7.

Now, we finally have a working function. We only allocate memory if the expression is non-deterministic. If the expression is non-deterministic in multiple places, we reuse that same `backup` node.

```
void main_hnf(Node* root)
{
    set_f(RET, make_int(3));
    Node* RET_forward = RET;
    nondet = false;
    while(true)
    {
        switch(RET_forward->tag)
        {
            ...
            case True:
                if(nondet)
                {
                    Node* backup = copy(RET);
                    stack_push(bt_stack, root, main_(backup));
                }
                set_int(root, 0);
                return;
                ...
        }
    }
```

Figure 5.6: Second attempt at compiling *main* with Shortcutting.

```
Node* f_RET_hnf(Node* backup)
{
    Node* v1 = RET->children[0];
    set_g(RET, v1);
    Node* RET_forward = RET;
    Node* g_backup = g_RET_hnf(NULL);
    bool nondet = g_backup == NULL;
    while(true)
    {
        nondet |= RET_forward->nondet;
        switch(RET_forward->tag)
        {
            ...
            case True:
                if(nondet)
                {
                    if(!backup)
                    {
                        backup = (Node*)calloc(1,sizeof(Node));
                    }
                    set_False(backup);
                    stack_push(bt_stack, backup, g_backup);
                }
                set_False(RET);
                return backup;
            ...
        }
    }
```

Figure 5.7: Compiling f with Shortcutting.

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

$$
\begin{aligned}
main = \textbf{case } & f\ 4 \textbf{ of} \\
& \quad True \rightarrow False \\
& \quad False \rightarrow False \\
f\ n = \textbf{case } & n \textbf{ of} \\
& \quad 0 \rightarrow True \\
& \quad \_ \rightarrow f\ (n-1)
\end{aligned}
$$

In this case f is a recursive function, so when we reduce $f\ 4$, we need to immediately reduce $f\ 3$. This is no problem at all, because we're 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.

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

5.2.3 Shortcutting Results

Shortcutting was originally formulated in the context of the Pakcs compiler [14, **?**], which handled reduction a little differently then we do here. Instead of calling an hnf function directly, it had a giant lookup table that would dispatch the node to be reduced to the correct reduction predicate. The compiler would then take a single step. Shortcutting was an attempt to circumvent, or shortcut, this lookup table, and it produced code that's actually similar to what we have in Rice. Instead of having a monolithic **H** predicate that normalized any expression, the authors split it into several $\mathbf{H}_f$ predicates for each function f. This also had the effect that knowing the argument to be normalized allowed that function to be called directly without ever having to construct the node. While we went about it a different way, we've achieved the same goal as the Shortcutting paper.

Before we move onto our next optimization, we should look back at what we've done so far. Initially, we had a *fib* function that allocated 8 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 75%, 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've moved from exponential memory allocated on the heap to constant space. While *fib* still runs in exponential time, it runs much faster, since it doesn't need to allocate memory. Surprisingly, *fib* is still just as efficient with non-deterministic arguments. If n is non-deterministic, the wrapper function will evaluate n before calling the worker.

Now that we've 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.

5.3 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:

$$sumPrimes = sum \circ filter\ isPrime \circ enumFromTo\ 2$$

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

$$\begin{aligned}
&sumPrimes\ n = go\ 2\ n \\
&\quad \textbf{where}\ go\ k\ n \\
&\qquad\qquad \mid k \geqslant n \qquad = 0 \\
&\qquad\qquad \mid isPrime\ k = k + go\ (k+1)\ n \\
&\qquad\qquad \mid otherwise\ = go\ (k+1)\ n
\end{aligned}$$

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

5.3.1 The Original Scheme

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

An alternative was proposed by Gill in his dissertation [**?**, **?**] 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 [**?**]. 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:

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

As an example, we can define the *sum* function as $\mathit{sum}\ xs = \mathit{foldr}\ (+)\ 0$. 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.

$$1 : 2 : 3 : 4 : 5 : [\,]$$
$$1 + 2 + 3 + 4 + 5 + 0$$

We've just replaced the : with + and the [] with 0. If the compiler can find where we will do this replacement, then we don't 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.

$$build :: (\forall\ b\ (a \to b \to b) \to b \to b) \to [a]$$
$$build\ g = g\ (:)\ [\,]$$

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 c and n respectively. At that point, we can construct the list by calling *build* on our builder function.

As an example, let's look at the function *enumFromTo a b* that constructs a list of integers from a to b.

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

We can turn this into a build function.

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

We can create build functions for several list creation functions found in the standard library, so it looks like we're 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't actually optimize the results. Second, we still need to show it's valid for curry.

5.3.2 The Combinator Problem

Let's 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 5.8 comes from the original paper [28].

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

$$filter\ f = build\ (\lambda c\ n \rightarrow foldr\ (\lambda x\ y \rightarrow \textbf{if}\ f\ x\ \textbf{then}\ x\ `c`\ y\ \textbf{else}\ y)\ n)$$

Functions that transform lists, such as *filter*, *map*, and *concat*, are written as a build applied to a fold. Unfortunately this doesn't work well with our inliner. Since we don't inline lambda expressions, and since reductions can only be applied to let bound variables, we simply can't do this reduction. Instead we need a new solution.

5.3.3 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.

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

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:

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

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

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.

$$
\begin{aligned}
&\mathit{sumPrimes}\ m = \mathit{sum}\ (\mathit{filter}\ \mathit{isPrime}\ (\mathit{enumFromTo}\ 2\ m))\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0\\
&\quad (\mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x\ `c`\ y\ \textbf{else}\ y)\ n)\\
&\qquad (\mathit{build}\ \mathit{enumFromTo\_build}\ 2\ m))\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0\\
&\quad (\mathit{build}\ (\lambda c\ n \to (\mathit{enumFromTo\_build}\ 2\ x)\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x\ `c`\ y\ \textbf{else}\ y)\ n))\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{enumFromTo\_build}\ 2\ m\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ (\lambda x\ y \to x + y)\ x\ y\ \textbf{else}\ y)\ 0\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{enumToFrom\_build}\ 2\ m\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x + y\ \textbf{else}\ y)\ 0\\
&\quad \textbf{where}\ \mathit{enumToFrom\_build}\ k\ m\ c\ z = \textbf{if}\ k > m\\
&\qquad \textbf{then}\ z\\
&\qquad \textbf{else}\ c\ k\ (\mathit{enumToFrom\_build}\ (k+1)\ m\ c\ z)\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{enumToFrom\_build}\ 2\ m\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x + y\ \textbf{else}\ y)\ 0\\
&\quad \textbf{where}\ \mathit{enumToFrom\_build}\ k\ m\ c\ z = \textbf{if}\ k > m\\
&\qquad\qquad\qquad\qquad \textbf{then}\ z\\
&\qquad\qquad\qquad\qquad \textbf{else}\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x + y\ \textbf{else}\ y)\\
&\qquad\qquad\qquad\qquad\quad k\ (\mathit{enumToFrom\_build}\ (k+1)\ m\ c\ z)\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{enumToFrom\_build}\ 2\ m\\
&\quad \textbf{where}\ \mathit{enumToFrom\_build}\ k\ m = \textbf{if}\ k > m\\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ z\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\lambda x\ y \to \textbf{if}\ \mathit{isPrime}\ x\ \textbf{then}\ x + y\ \textbf{else}\ y)\\
&\qquad\qquad\qquad\qquad\qquad k\ (\mathit{enumToFrom\_build}\ (k+1)\ m\ c\ z)\\
&\Rightarrow\\
&\mathit{sumPrimes}\ m = \mathit{enumToFrom\_build}\ 2\ m\\
&\quad \textbf{where}\ \mathit{enumToFrom\_build}\ k\ m = \textbf{if}\ k > m\\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ z\\
&\qquad\qquad\qquad\qquad\quad \textbf{else if}\ \mathit{isPrime}\ k\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ x + (\mathit{enumToFrom\_build}\ (k+1)\ m\ c\ z)\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\mathit{enumToFrom\_build}\ (k+1)\ m\ c\ z)
\end{aligned}
$$

Figure 5.8: Optimization derivation for for short-cut Deforestation

$$map\ f = build\_fold\ (map\_mkf\ f)\ map\_mkz$$
$$map\_mkf\ f\ c\ x\ y = f\ x\ `c`\ y$$
$$map\_mkz\ n = n$$

$$filter\ p = build\_fold\ (filter\_mkc\ p)\ filter\_mkz$$
$$filter\_mkf\ p\ c\ x\ y = \textbf{if}\ p\ x\ \textbf{then}\ x\ `c`\ y\ \textbf{else}\ y$$
$$filter\_mkz\ n = n$$

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.

$$build\_fold\ mkf\ mkz\ (build\ g) = build\ (\lambda c\ n \rightarrow g\ (mkf\ c)\ (mkz\ n))$$
$$foldr\ f\ z\ (build\_fold\ mkf\ mkz\ xs) = foldr\ (mkf\ f)\ (mkz\ z)\ xs$$
$$build\_fold\ mkf_1\ mkz_1\ (build\_fold\ mkf_2\ mkz_2\ xs) = build\_fold\ (mkf_2 \circ mkf_1)\ (mkz_2 \circ mkz_1)\ xs$$

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

5.3.4 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* is used exclusively in either a *build_fold* or *fold*. If this is the case, we inline the variable into it's 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. It simply applies either the *foldr / build* theorem, or one of the three *build_fold* theorems from above. The definitions are given in figure 5.9 The optimization derivation for *sumPrimes* is in figure 5.10.

So far we've done a decent job. It's not as efficient as the Haskell version, but that's not surprising. However, we can still improve this. The main problem here is that we can't optimize a partial application. This is unfortunate, because the *build_fold* function tends to create large expressions of partially applied functions. Fortunately we've 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 outline

Inline foldr/build

$$\textbf{let } x = \mathit{build}\ g \textbf{ in } e \quad | \ e|_p = \mathit{foldr}\ \_\ \_\ x \quad \Rightarrow e[[p,2] \mapsto \mathit{build}\ g]$$

$$\textbf{let } x = \mathit{build}\ g \textbf{ in } e \quad | \ e|_p = \mathit{build\_fold}\ \_\ \_\ x \Rightarrow e[[p,2] \mapsto \mathit{build}\ g]$$

$$\textbf{let } x = \mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz} \textbf{ in } e \mid e|_p = \mathit{foldr}\ f\ z\ x \quad \Rightarrow e[[p,2] \mapsto \mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}]$$

$$\textbf{let } x = \mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz} \textbf{ in } e \mid e|_p = \mathit{build\_fold}\ \_\ \_\ x \Rightarrow e[[p,2] \mapsto \mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}]$$

Deforest foldr/build

$$\mathit{foldr}\ f\ z\ (\mathit{build}\ g) \quad \Rightarrow g\ f\ z$$

Deforest build_fold/build

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

Deforest foldr/build_fold

$$\begin{aligned}\mathit{foldr}\ f\ z\ (\mathit{build\_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) \quad \Rightarrow\ & \textbf{let } f_1 = \mathit{mkf}\ f \\ & \textbf{in let } \mathit{z\_1} = \mathit{mkz}\ z \\ & \textbf{in } \mathit{foldr}\ f_1\ \mathit{z\_1}\ \mathit{xs}\end{aligned}$$

Deforest build_fold/build_fold

$$\begin{aligned}\mathit{build\_fold}\ \mathit{mkf}_1\ \mathit{mkz}_1\ (\mathit{build\_fold}\ \mathit{mkf}_2\ \mathit{mkz}_2\ \mathit{xs}) \Rightarrow\ & \textbf{let } f_1 = \mathit{mkf}_2 \circ \mathit{mkf}_1 \\ & \textbf{in let } \mathit{z\_1} = \mathit{mkz}_2 \circ \mathit{mkz}_1 \\ & \textbf{in } \mathit{build\_fold}\ f_1\ \mathit{z\_1}\ \mathit{xs}\end{aligned}$$

Figure 5.9: The Deforestation optimization.

The lambda in the build rule is a call to a known function.

The lets are added to keep the expression in A-Normal Form.

let $v_1 = enumFromTo\ 2\ n$

in let $v_2 = filter\ isPrime\ v_1$

in $sum\ v_2$

$\Rightarrow$ **Reduce Useful**

let $v_1 = build\ enumFromTo\_build\ 2\ n$

in let $v_2 = build\_fold\ (filter\_mkf\ isPrime)\ id\ v_1$

in $foldr\ (+)\ 0\ v_2$

$\Rightarrow$ **Inline foldr/build**

let $v_1 = build\ enumFromTo\_build\ 2\ n$

in let $v_2 =$

in $foldr\ (+)\ 0\ (build\_fold\ (filter\_mkf\ isPrime)\ id\ v_1)$

$\Rightarrow$ **Deforest build_fold/build**

let $v_1 = build\ enumFromTo\_build\ 2\ n$

in let $v_2 = build\_fold\ (filter\_mkf\ isPrime)\ id\ v_1$

in $foldr\ (+)\ 0$

$(build\ (mk\_build\ (enumFromTo\_build\ 2\ n)\ (filter\_mkf\ isPrime)\ id)$

$\Rightarrow$ **Deforest build_fold/build**

$mk\_build\ (enumFromTo\_build\ 2\ n)\ (filter\_mkf\ isPrime)\ id\ (+)\ 0$

$\Rightarrow ANF$

let $f = filter\_mkf\ isPrime\ (+)$

$z = id\ 0$

in $enumFromTo\_build\ 2\ n\ f\ z$

$\Rightarrow$ **Inline Literal**

let $f = filter\_mkf\ isPrime\ (+)$

in $enumFromTo\_build\ 2\ n\ f\ 0$

Figure 5.10: Derivation for *sumPrimes*

it and attempt to optimize it. If we can't optimize it at all, then we do nothing. Otherwise, we make a new outlined 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 didn't implement Deforestation. With function outlining our final optimized code is given below.

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

This certainly isn't perfect, but it's 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.

5.3.5 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.

Theorem 9. *For any deterministic f, z, g, mkf, and mkz, the following equations hold.*

$$
\begin{aligned}
&build\_fold\ mkf\ mkz\ (build\ g) = build\ (\lambda c\ n \to g\ (mkf\ c)\ (mkz\ n))\\
&foldr\ f\ z\ (build\_fold\ mkf\ mkz\ xs) = foldr\ (mkf\ f)\ (mkz\ z)\ xs\\
&build\_fold\ mkf_1\ mkz_1\ (build\_fold\ mkf_2\ mkz_2\ xs) = build\_fold\ (mkf_2 \circ mkf_1)\ (mkz_2 \circ mkz_1)\ xs
\end{aligned}
$$

Proof. Recall that the free theorem for *build* is

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

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

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

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

$$build\_fold\ mkf\ mkz\ (build\ g) = build\ (\lambda c\ n \to g\ (mkf\ c)\ (mkz\ n))$$

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

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

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

$$build\_fold\ mkf\ mkz \circ fold\ f\ z = fold\ (mkf\ f)\ (mkz\ z)$$

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

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

$$foldr\ f\ z\ (build\_fold\ mkf_1\ mkz_1\ (build\_fold\ mkf_2\ mkz_2\ xs))$$

From the previous result we have:

$$\begin{aligned}
&foldr\ (mkf_1\ f)\ (mkz_1\ z)\ (build_fold\ mkf_2\ mkz_2\ xs)\\
&\Rightarrow foldr\ (mkf_2\ (mkf_1\ f))\ (mkz_2\ (mkz_1\ z))\ xs\\
&\Rightarrow foldr\ ((mkf_2 \circ mkf_1)\ f)\ ((mkz_2 \circ mkz_1)\ z)\ xs\\
&\Rightarrow foldr\ f\ z\ (build_fold\ (mkf_2 \circ mkf_1)\ (mkz_2 \circ mkz_1)\ xs)
\end{aligned}$$

which establishes our result:

$$build\_fold\ mkf_1\ mkz_1\ (build\_fold\ mkf_2\ mkz_2) = build\_fold\ (mkf_2 \circ mkf_1)\ (mkz_2 \circ mkz_1)$$

□

While this gives us confidence that Deforestation is a useful optimization, we've already seen that equational reasoning doesn't always apply in Curry. In fact, as they are currently stated, it's not surprising that these rules don't hold in 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.

$$\begin{aligned}
\mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) &= \textbf{let}\ g' = (\lambda c\ n \rightarrow \textbf{let}\ f = \mathit{mkf}\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad z = \mathit{mkz}\ n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ g\ f\ z) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{build}\ g' \\
\mathit{foldr}\ f\ z\ (\mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) &= \textbf{let}\ f' = \mathit{mkf}\ f \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad z' = \mathit{mkz}\ z \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{foldr}\ f'\ z'\ \mathit{xs} \\
\mathit{build_fold}\ \mathit{mkf1}\ \mathit{mkz2}\ (\mathit{build_fold}\ \mathit{mkf2}\ \mathit{mkz2}\ \mathit{xs}) &= \textbf{let}\ \mathit{mkf} = \mathit{mkf1} \circ \mathit{mkf2} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{mkz} = \mathit{mkz1} \circ \mathit{mkz2} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}
\end{aligned}$$

Now we are ready to state our result.

Theorem 10. *suppose f, z, g, mkf, and mkz are all functions who's right had side is an expression in A-Normal form, then the following reductions are valid.*

$$\begin{aligned}
\mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ (\mathit{build}\ g) &= \textbf{let}\ g' = (\lambda c\ n \rightarrow \textbf{let}\ f = \mathit{mkf}\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad z = \mathit{mkz}\ n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ g\ f\ z) \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{build}\ g' \\
\mathit{foldr}\ f\ z\ (\mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}) &= \textbf{let}\ f' = \mathit{mkf}\ f \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad z' = \mathit{mkz}\ z \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{foldr}\ f'\ z'\ \mathit{xs} \\
\mathit{build_fold}\ \mathit{mkf1}\ \mathit{mkz2}\ (\mathit{build_fold}\ \mathit{mkf2}\ \mathit{mkz2}\ \mathit{xs}) &= \textbf{let}\ \mathit{mkf} = \mathit{mkf1} \circ \mathit{mkf2} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{mkz} = \mathit{mkz1} \circ \mathit{mkz2} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \mathit{build_fold}\ \mathit{mkf}\ \mathit{mkz}\ \mathit{xs}
\end{aligned}$$

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

$$\mathit{fold}\ f\ z\ (\mathit{build}\ g\ (:)\ [\,]) \equiv g\ f\ z$$

We proceed in a manner similar to [21]. 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 collection of lists when normalizing this expression. Let's make this explicit.

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

Here we have a collection 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.

Now, let's see what happens when we normalize the entire expression. Recall that function application distributes over choice. That is, $f\ (a\ ?\ b) = f\ a\ ?\ f\ b$.

$$\begin{aligned}
&\mathit{foldr} \oplus z\ (\mathit{build}\ g\ (:)\ [\,]) \\
&\quad = \mathit{foldr} \oplus z\ (g_{1,1} : g_{1,2} : g_{1,3} : \ldots \mathit{end}_1\ ? \\
&\qquad\qquad\qquad g_{2,1} : g_{2,2} : g_{2,3} : \ldots \mathit{end}_2\ ? \\
&\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad g_{k,1} : g_{k,2} : g_{k,3} : \ldots \mathit{end}_k) \\
&\quad = \mathit{foldr} \oplus z\ (g_{1,1} : g_{1,2} : g_{1,3} : \ldots \mathit{end}_1)\ ? \\
&\qquad \mathit{foldr} \oplus z\ (g_{2,1} : g_{2,2} : g_{2,3} : \ldots \mathit{end}_2)\ ? \\
&\qquad\qquad\qquad \ldots \\
&\qquad \mathit{foldr} \oplus z\ (g_{k,1} : g_{k,2} : g_{k,3} : \ldots \mathit{end}_k) \\
&\quad = (g_{1,1} \oplus g_{1,2} \oplus g_{1,3} \oplus \ldots z_{\mathit{end}_1})\ ? \\
&\qquad (g_{2,1} \oplus g_{2,2} \oplus g_{2,3} \oplus \ldots z_{\mathit{end}_2})\ ? \\
&\qquad\qquad \ldots \\
&\qquad (g_{k,1} \oplus g_{k,2} \oplus g_{k,3} \oplus \ldots z_{\mathit{end}_k}) \\
&\qquad\quad \textbf{where}\ z_{\mathit{end}_i} = \textbf{if}\ \mathit{end}_i \equiv \mathit{failed}\ \textbf{then}\ \mathit{failed}\ \textbf{else}\ z \\
&\quad = g \oplus z
\end{aligned}$$

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 didn't 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 $\perp$ don't 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:

$$\mathit{foldr}\ (\lambda x\ y \to 1)\ 0\ (\mathit{build}\ (\lambda c\ n \to 0\ `c`\ 1\ `c`\ \perp))$$

Evaluating *build* to normal form would produce a failure, since the tail of the list is $\perp$. However, since the f in *foldr* never looks at either of it's arguments, this branch of the computation can still return a result.

In this chapter we've developed three optimizations to help reduce the memory allocated by Curry programs. These optimizations seem effective, and we've shows why they're correct, but we still need to find out how effective they are. In the next chapter we show a small bechmarking suite to test the efficacy of these optimizations, and to show the results of each. We then discuss possible future directions for this research.

Chapter 6

CONCLUSION

6.1 RESULTS

Now that we've 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 tests we ran aren't extensive. We're not trying to show characterize every area of this compiler's performance, we just want a general idea of the time and memory consumption. While a more extensive test that looks at finer details like pipeline stalls and cache misses would be interesting, and no doubt, informative, we are really only concerned with two characteristics. How long did the program take to run? How many allocations were made by the program?

There are many ways to measure memory allocation and execution time. For memory allocation, we could estimate the memory using the operating system, or we could use a tool like valgrind to find the number of allocations, However, and even simpler solution is just to keep a count of the number of times we create a node. This is simple to implement, and doesn't effect the run-time performance noticeably, so for measuring memory allocation, we instrumented the run time system to report how many nodes are created. For measuring time we took the approach from [45], and ran the programs multiple times, while taking the fastest result. We also ran the program in multiple different environments on multiple machines.

6.1.1 Tests

In order to determine how effective our optimizations are, we've developed a small suite of test programs that are meant to test both deterministic and non-deterministic programs. This suit is loosely based on the suite used to test the Kics2 compiler [18]. However, we've made a few alterations. We've added several programs to test non-determinism, and we've removed or modified programs that used functional patterns as that was not part of the Rice compiler.

We split the functions into tree groups: Numeric computations meant to test unboxing; list

computations meant to test deforestation; and non-deterministic computations.

- **numeric computations:**
  - **fib** is the Fibonacci program from chapter 5. We test it with both deterministic and non-deterministic input.
  - **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.

    $$half\ n \mid x + x \equiv n = x$$
    where x **free**

  - **perm** computes all of the permutations of a list.
  - **queens_perm** computes solutions to the n-queens problem by permuting a list, and checking if it's a valid solution.
  - **sort** sorts a list using by finding a sorted permutation.
- **deforestation:**
  - **queens_det** computes solutions to the n-queens problem using a backtracking solution and list comprehension.
  - **reverse_builtin** reverses a list using the built in reverse function.
  - **reverse_foldr** reverses a list using a reverse function written as a fold.
  - **reverse_prim** reverses a list using the built in reverse function and primitive numbers.
  - **sum_squares** computes $sum \circ map\ squre \circ filter\ odd \circ enumFromTo\ 1$.
  - **buildFold** computes a long chain of list processing functions.
  - **primes** computes a list of primes.
  - **sieve** computes *sumPrimes* from Chapter 5.

6.1.2 Results

The results for running the tests are given if figure **??** for timing, and 6.2 for memory. While we attempted to be as fair in our assessments as possible, there are some cases where we couldn't include a compiler. Either because it was taking too long, or it used too much memory, and was killed by the operation system. For the timing results we include Kics2 and Pakcs when we can, however neither compiler offers a way to see how much memory was used during computation. So, they are not included in the memory results. It is worth noting that several programs running in Kics2 were killed by the operation system for using too much memory.

There is also a much more interesting problem. In some cases we simply can't compare the optimized code. The optimized code runs so fast that the we can't accurately time them.

We'll look at *fib* in particular, because it illustrates this problem particularly well. The algorithm for *fib* is an exponential time algorithm. Specifically it runs in $O(1.61^n)$. This means that *fib* n runs rough 1.61 time as fast as *fib* $(n + 1)$. In running our example with a non-deterministic input, both Kics2 and Pakcs were only able to run up to *fib* 27. In contrast fully optimized Rice was able to run *fib* 42 in about the same amount of time. So using this as a very rough approximation we have that our optimized code is running $1.61^{15} \approx 1000$ times as fast.

This is a very impressive speedup, but we've 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 it's 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's not surprising to us that our code ran slower than GHC. 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 isn't inherently slower and Haskell. We believe that a more mature Curry compiler could run as fast as GHC on most, if not all, deterministic functions.

6.2 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 not only could be beat Kics2 in all cases. In many cases the

	fib	fib_nondet	tak
pakcs	25.900	30.244	6867.545
kics2			
unopt			
basic			
unbox			
shortcut			
deforest			
all			

	cent	half	perm	queens_perm	sort
pakcs	65.630	983.572	45.102	568.812	212.331
kics2					
unopt					
basic					
unbox					
shortcut					
deforest					
all					

	queens_det	rev	rev_fold	rev_prim	sum_squares	buildFold	primes	sieve
pakcs	1051.782		262.144	27.963	33.486		7097.473	2845.308
kics2								
unopt								
basic								
unbox								
shortcut								
deforest								
all								

Figure 6.1: Results for running time

	unopt	basic	unbox	shortcut	deforest	all
fib						
tak						
cent						
half						
perm						
queens_perm						
sort						
queens_det						
reverse_builtin						
reverse_foldr						
reverse_prim						
sum_squares						
buildFold						
primes						
sieve						

Figure 6.2: Results for memory allocations.

results were simply incomparable, and in some cases we were even able to compete with GHC itself. Furthermore, we've shown that the memory optimizations really were effective for Curry programs. This isn't much of a surprise. Allocating less memory is a good strategy for making programs run faster. It is good to know that the presence of non-determinism doesn't affect this commonly head belief.

It's 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 that significant, until we look at what optimizations aren't valid. For example, common sub-expression elimination was not included in this compiler. The reason is that it simply isn't a valid Curry transformation. It introduces sharing where none existed. If the common sub-expression is non-deterministic, but we will change the set of results. On the other hand, common sub-expression elimination is fairly innocuous in most other languages.

6.2.1 Future Work

Most currys are made form 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 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't represent lambda expressions in FlatCurry, and inline them. However before we could even attempt this, we would need to know when it's 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 don't 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.

Finally, developing a better run-time system 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 [16] compiler might solve this issue.

6.2.2 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, but in testing these optimizations, we've 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.

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, 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's 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 to the current state-of-the-art for backtracking Curry programs.

When starting this project, shortcutting was already known to be valid for functional logic programs. It was developed for them specifically. 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've made a tasty Curry with Rice.

References

[1] The Glasgow Haskell Compiler. https://www.haskell.org/ghc/.

[2] Synthesizing set functions. 2018.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

[13] 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.

[14] Sergio Antoy and Andy Jost. Are needed redexes really needed? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 61–71, New York, NY, USA, 2013. ACM.

[15] Sergio Antoy and Andy Jost. Compiling a Functional Logic Language: The Fair Scheme. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation*, pages 202–219, Cham, 2014. Springer International Publishing.

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

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

[18] 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.

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

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

[21] 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.

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

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

[24] 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.

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] Steven Libby. *Making Curry with Rice: An Optimizing Compiler for Curry.* PhD thesis, Portland State University, 2012.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

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

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

[51] 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.

[52] 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.

[53] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114, 2015.

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

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

[56] 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.

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