Making Curry with Rice
An Optimizing Curry Compiler

Steven Libby

October 15, 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. [**?**, **?**, **?**] While there are other semantics [**?**, **?**, **?**], 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 $\to$. We write $a \to b$ instead of $(a, b) \in \to$, and we have several modifiers on our relation.

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

	$(x \cdot x + 1)(2 + x)$	
$\rightarrow$	$(x \cdot x + 1)(x + 2)$	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 1.1.

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.

	$3 \cdot (4 + 5)$
$\rightarrow$	$3 \cdot 4 + 3 \cdot 5$
$\rightarrow$	$12 + 15$
$\rightarrow$	27

(a) distributing first

	$3 \cdot (4 + 5)$
$\rightarrow$	$3 \cdot 9$
$\rightarrow$	27

(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.

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.

$$\begin{array}{rl} & a \cdot (b + c) \\ \rightarrow & a \cdot b + a \cdot c \end{array}$$

$$\begin{array}{rl} & x + y \\ \rightarrow & y + x \\ \rightarrow & x + y \\ \ldots & \end{array}$$

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.

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.

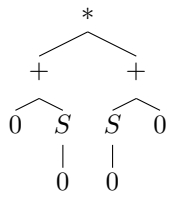


Figure 1.4: Tree representation of the term $*(+(0, S(0)), +(S(0), 0))$.

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 **??**

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

Figure 1.5: algorithm for finding a subterm of t.

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.

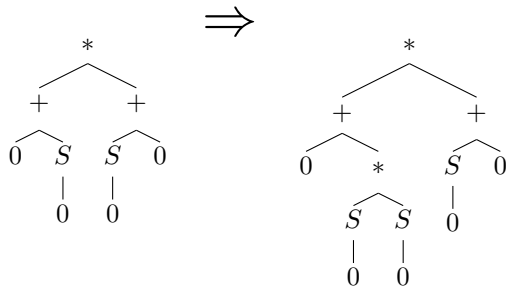


Figure 1.6: The result of the computation $t[[0, 1] \leftarrow S(0)]$

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 $s(v_i) = t_i$ 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

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.

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

Figure 1.7: Algorithm for applying a substitution.

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.

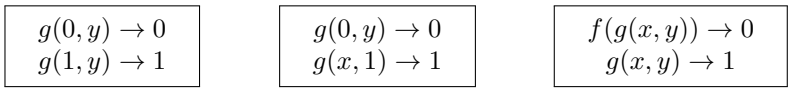
There are a few important properties of rewrite rules $l \to r$. A rule is left (right) linear if $l(r)$ is linear. A rule is collapsing if $r \in V$. A rule is duplicating if there is an $x \in V$ that occurs more often in r than in l.

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. [**?**]

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.

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

The two sets are $C = \{0, S, \top, \bot\}$ and $F = \{+, *, -, \leq\}$, and the root of the left hand side of each rule is a function symbol. In contrast, the SKI system is



(a) A non-overlapping system (b) A system that overlaps at the root (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.

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

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

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.

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

Figure 1.10: The SKI system from combinatorial logic.

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 most

$$\begin{array}{lllll} gcd(x,x) & & \rightarrow & x \\ gcd(x,y) & \mid \; y \leq x & \rightarrow & gcd(x-y,y) \\ gcd(x,y) & \mid \; x \leq y & \rightarrow & gcd(x,y-x) \end{array}$$

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

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 [**?**, **?**] 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.

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. [**?**] 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 $l \rightarrow r$, 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

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

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

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* $\mathcal{S}$:$T(\Sigma,V) \to Pos$ 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.

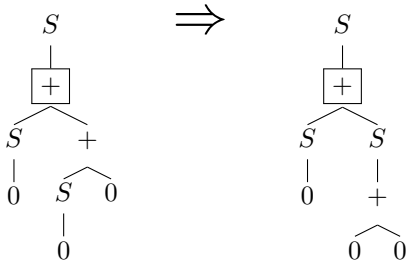
While both of these strategies are well understood, we can actually make a stronger guarantee. We want to reduce only the redexes that are necessary to find a normal form. To formalize this we need to understand what can happen as a term is rewritten. Specifically for a redex s that is a subterm of t, 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 $Des(s, A)$

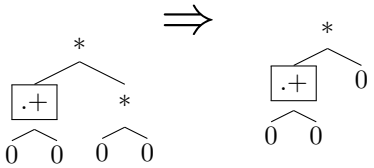
$$Des(s,A) = \begin{cases} \emptyset & \text{if } v = u \\ \{s\} & \text{if } p \not\leqslant v \\ \{t|_{u \cdot w \cdot q} : r|_w = x\} & \text{if } p = u \cdot v \cdot q \text{ and } t|_v = x \text{ and } x \in V \end{cases}$$

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

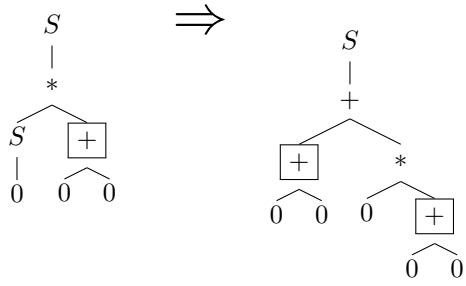
The first part of the definition is formalizing the notion of descendant. The second part is extending it to a rewrite derivation. The extension is straightforward. Calculate the descendants for the first rewrite, then for each descendant, calculate the descendants for the rest of the rewrites. With 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.



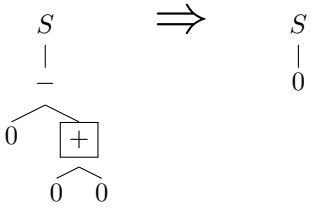
(a) rewrite R_1 at position $[0,1]$ doesn't affect $t|_{[0]}$.



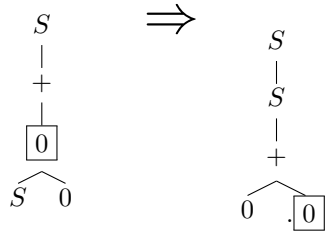
(b) rewrite R_4 at position $[1]$ doesn't affect $t|_{[0]}$.



(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.

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.*

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 $v_1, v_2 \ldots v_n$, 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 [**?**], innermost [**?**], outermost [**?**], standard [**?**], and lazy [**?**]. 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 $x \leq y + z \leadsto_{2 \cdot \epsilon, R_1, \{y \mapsto 0\}} x \leq z$. 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 $x \leq y + z \leadsto_{\epsilon, R_8, \{x \mapsto 0\}} \top$. 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 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,

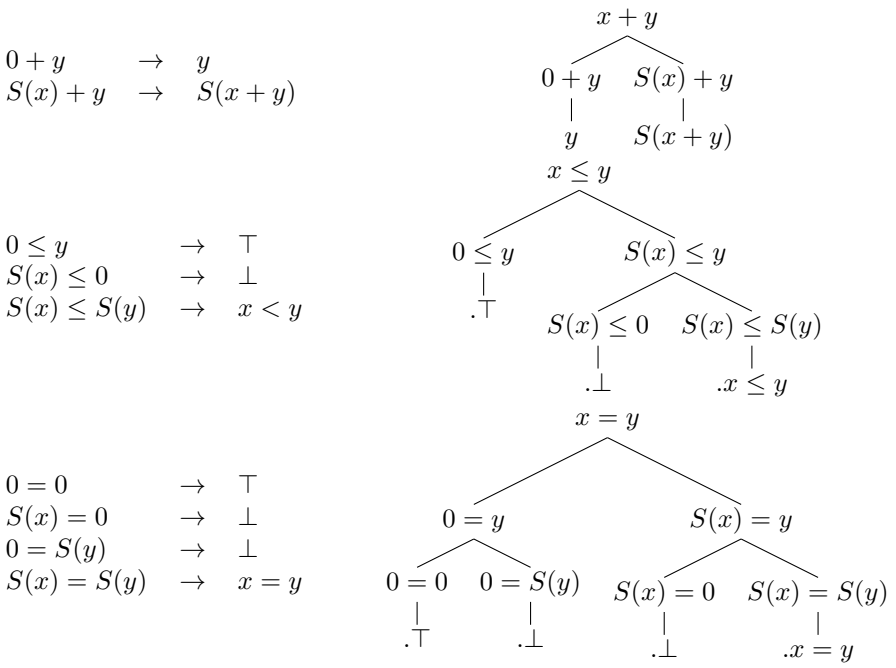


Figure 1.15: Definitional trees for $x + y$, $x \leq y$, and $x = y$.

$x \leq y + z \rightsquigarrow_{2\dot{\epsilon},R_1,\{y\mapsto 0\}} x \leq z$, isn't a needed narrowing step because $x \leq y + z \rightsquigarrow_{2\dot{\epsilon},R_1,\{x\mapsto 0,y\mapsto 0\}} 0 \leq z$, 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, $x < y \rightsquigarrow_{\epsilon,R_8,\{x\mapsto 0\}} \top$, and $x < y \rightsquigarrow_{\epsilon,R_9,\{x\to S(u),t\mapsto S(v)\}} u \leq v$ 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 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. 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) $\mid$ 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

[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.

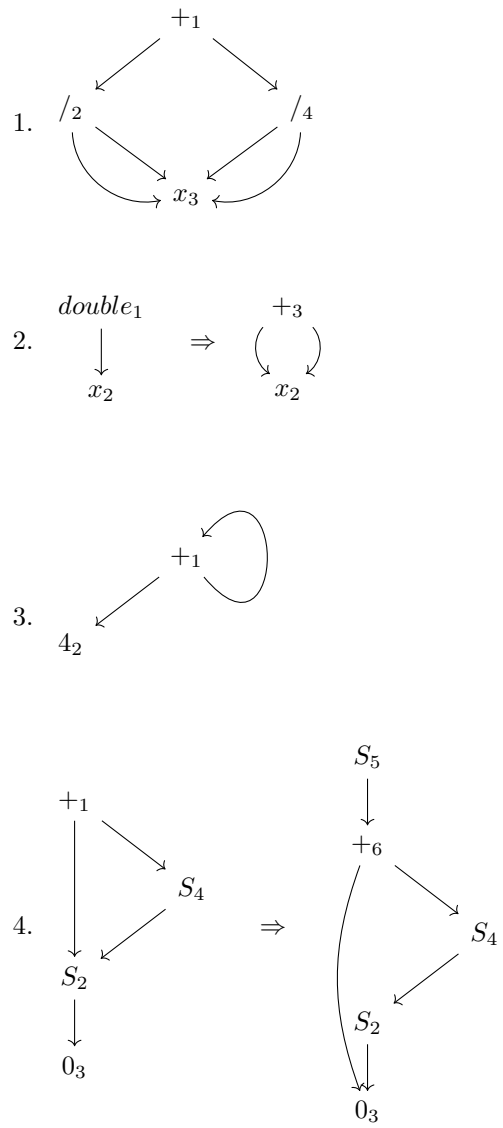


Figure 1.16: 1. $1{:}+(2{:}/(3{:}x,3),4{:}/(3,3))$,
2. $1{:}double(2{:}x) \Rightarrow 3{:}+(2{:}x,2)$
3. $1{:}+(2{:}4,1)$
4. $1{:}+(2{:}S(3{:}0),4{:}S(2)) \Rightarrow 5{:}S(6{:}+(3{:}0),4{:}S(2{:}S(3)))$

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 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.

From here we can define rewriting and narrowing similarly to how we did for terms. We do not give the definitions here, but they can be found in Echaned and Janodet [**?**]. 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 [**?**], and Baader and Nipkow [**?**]. The notation and ideas in this section largely come from Ohlebusch [**?**], although they are very similar to the previous two summaries. The foundations of term rewriting were laid by Church, Rosser, Curry, Feys, Newman. [**?**, **?**, **?**] Most of the work on rewriting has centered on confluence and termination. [**?**] Narrowing has been developed by Slagle [**?**]. Sequential strategies were developed by Huet and Levy [**?**], who gave a decidable criteria for a subset of sequential systems. This led to the work of Antoy on inductively sequential systems [**?**]. The needed narrowing strategy came from Hanus, Antoy, and Echahed [**?**]. 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 [**?**], but there are also alternatives in [**?**, **?**, **?**]

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 [**?**]. For an exhaustive explanation of the syntax and semantics of Curry see [**?**].

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.

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

$$insert\ x\ (y : ys) = (x : y : ys) : map\ (y:)\ (insert\ x\ ys)$$

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

While this algorithm is not terribly complex, it'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:

$$x\ ?\ y = x$$
$$x\ ?\ y = y$$

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

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

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.

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

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.

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

Now the real work of our algorithm is a single line. Even better, it'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.

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

In this example every permutation of xs that 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.

[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.

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:

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

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

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

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. [**?**].

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 [**?**], 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 $\perp$ 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.

$$\begin{array}{lll} n \Rightarrow l & & \textit{literal} \\ \quad | \ C_k\ n\_1 \ldots n\_k & & \textit{constructor} \end{array}$$

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.

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

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

$$
\begin{array}{lll}
hnf :: Expr \rightarrow Expr & & \\
hnf\ [\![l]\!] & = l & \\
hnf\ [\![e :: t]\!] & = hnf\ e & \\
hnf\ [\![C\ e_1 \ldots e_n]\!] & = C\ e_1 \ldots e_n & \\
hnf\ [\![f\ e_1 \ldots e_n]\!] & = \textbf{let}\ v_1 \ldots v_n = vars\ f & \\
 & \qquad e_f = body\ f & \\
 & \qquad \sigma = \{ v_1 \rightarrow e_1, \ldots v_n \rightarrow e_n \} & \\
 & \quad \textbf{in}\ \ hnf\ (\sigma\ e_f) & \\
hnf\ [\![\ \textbf{let}\ v_1 = e_1\ \textbf{in}\ e]\!] & = \textbf{let}\ \sigma = \{ v_1 \rightarrow e_1 \} & \\
 & \quad \textbf{in}\ \ hnf\ (\sigma\ e) & \\
hnf\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!] & & \\
\quad \mid hnf\ [\![e]\!] = l & = \textbf{let}\ (l \rightarrow e) \in bs & \\
 & \quad \textbf{in}\ \ hnf\ e & \\
\quad \mid hnf\ [\![e]\!] = (C\ e_1 \ldots e_n) & = \textbf{let}\ (C\ v_1 \ldots v_n \rightarrow e) \in bs & \\
 & \qquad \sigma = \{ v_1 \rightarrow e_1, \ldots v_n \rightarrow e_n \} & \\
 & \quad \textbf{in}\ \ hnf\ (\sigma\ e) &
\end{array}
$$

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.

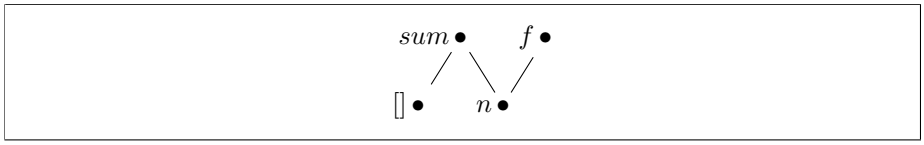
We can get around this by moving to graph rewriting. Surprisingly, not much changes for the interpreter. The only real change is that all expressions

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.

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

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 [**?**].

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:

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

With this we can extend our interpreter to graph rewriting.

$$\begin{aligned} &hnf :: Expr \to Expr \\ &hnf\ [\![l]\!] &&= l \\ &hnf\ [\![e :: t]\!] &&= FORWARD\ (hnf\ [\![e]\!]) \end{aligned}$$

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

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. [**?**]

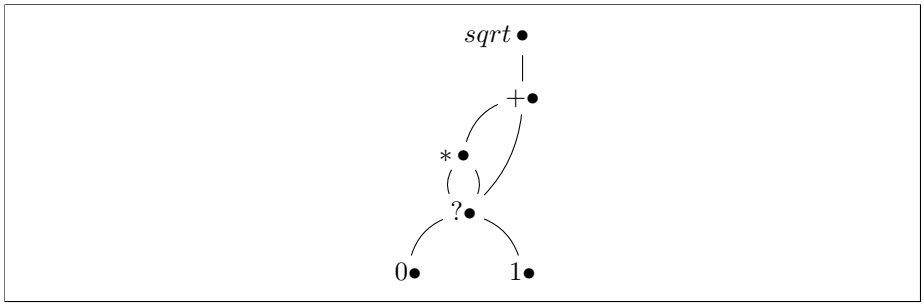
Backtracking is conceptually the simplest mechanism for non-determinism. We evaluate an expression normally, and every time we hit a choice operator, we pick one option. If we finish the computation, either by producing an answer or failing, then we undo each of the computations until the last choice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

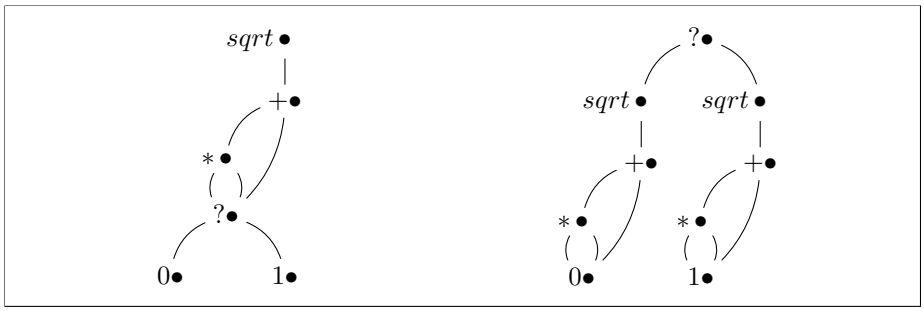
The following three mechanisms are all based on the idea of copying part of the expression graph. All of them are incomplete with a naive implementation, however they can all be made complete using the fair scheme [**?**]. I'll demonstrate each of these mechanisms with the following expression.

$$
\begin{array}{l}
\textbf{let}\ x = 0\ ?\ 1 \\
\textbf{in}\ \mathit{sqrt}\ ((x * x) + x)
\end{array}
$$

This expression has the following graph:

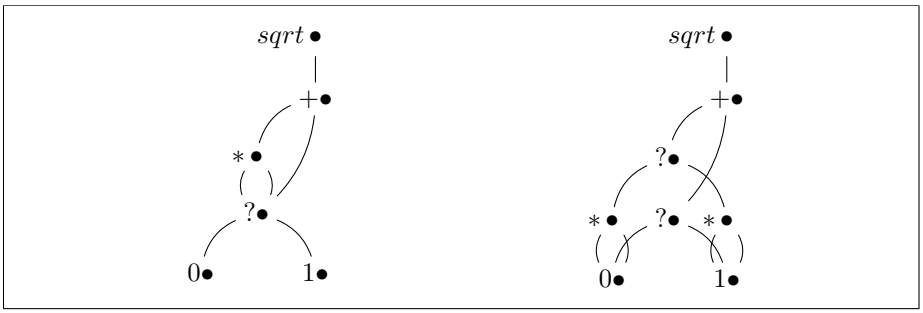


Copying is a different take on non-determinism. The idea is straightforward. Any time we encounter a choice node in an expression, move the choice node up to the root of the graph, and copy every node that was on the path to that choice. We can see the results of copying on our expression below.

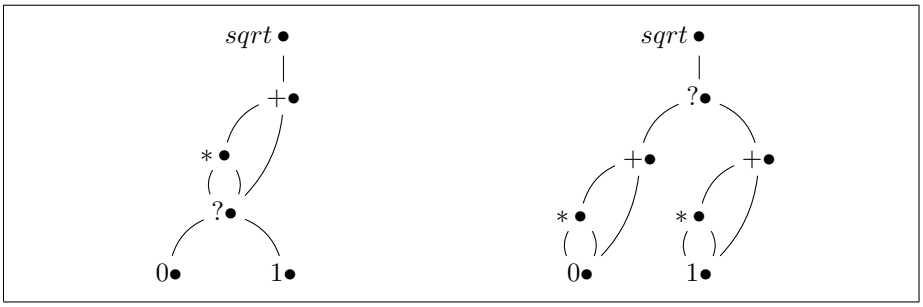


The advantage to copying is its simplicity. We just move the non-determinism to the root, and copy everything on the way up. This is simple to do, and we end up with an expression of the form *answer* ? *answer* ? *answer* .... The down side is that we must copy the entire expression. This usually leads 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. [**?**] 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. [**?**] 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, [**?**] 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.

$$\textbf{type } \mathit{Frame} = (\mathit{Expr}, \mathit{Expr}, \mathit{Bool})$$
$$\textbf{type } \mathit{Stack} = [\mathit{Frame}]$$

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 $\mathit{hnf}\ \mathit{bt}\ e@[\![\ldots]\!]$.

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

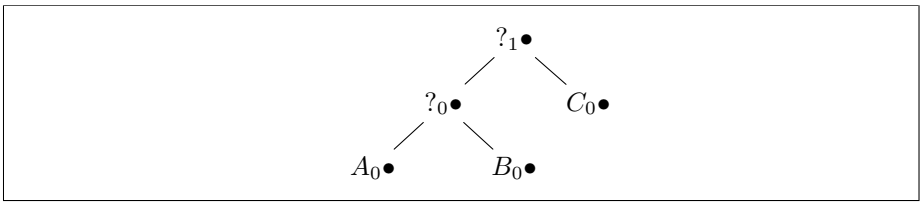
The *pushChoice* is defined similarly, except the frame is $(\mathit{exp}, e, \mathit{True})$ since it's a choice frame.

$$
\begin{array}{lll}
hnf :: Stack \to Expr \to (Expr, Stack) & & \\
hnf\ bt\ [\![\bot]\!] & = (\bot, bt) & \\
hnf\ bt\ [\![l]\!] & = (l, bt) & \\
hnf\ bt\ [\![e :: t]\!] & = \textbf{let}\ (e', bt') = FORWARD\ (hnf\ bt\ [\![e]\!]) & \\
 & \quad \textbf{in}\ (FORWARD\ e', bt') & \\
hnf\ bt\ [\![C\ e_1 \ldots e_n]\!] & = (C\ e_1 \ldots e_n, bt) & \\
hnf\ bt\ [\![f\ e_1 \ldots e_n]\!] & = \textbf{let}\ v_1 \ldots v_n = vars\ f & \\
 & \quad\ e_f = body\ f & \\
 & \quad\ \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} & \\
 & \quad\ (e'_f, bt') = hnf\ bt\ (\sigma\ e_f) & \\
 & \quad \textbf{in}\ \ push\ e'_f\ bt' & \\
hnf\ bt\ [\![FORWARD\ e]\!] & = \textbf{let}\ (e', bt') = hnf\ bt\ e & \\
 & \quad \textbf{in}\ \ push\ (FORWARD\ e')\ bt' & \\
hnf\ bt\ [\![e_1\ ?\ e_2]\!] & = \textbf{let}\ (e_{-1}', bt') = hnf\ bt\ e_1 & \\
 & \quad \textbf{in}\ \ pushChoice\ (FORWARD\ e_{-1}')\ bt') & \\
hnf\ bt\ [\![\ \textbf{let}\ v_1 = e_1\ \textbf{in}\ e]\!] & = \textbf{let}\ \sigma = \{\, v_1 \to e_1 \,\} & \\
 & \quad \textbf{in}\ \ hnf\ bt\ (\sigma\ e) & \\
hnf\ bt\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!] & & \\
\quad \mid e \equiv [\![FORWARD\ e']\!] & = \textbf{let}\ (e', bt') = (hnf\ bt\ [\![\ \textbf{case}\ e'\ \textbf{of}\ bs]\!]) & \\
 & \quad\ fwd = FORWARD\ e' & \\
 & \quad \textbf{in}\ \ push\ (FORWARD\ e')\ bt' & \\
\quad \mid hnf\ e \equiv ([\![l]\!], bt') & = \textbf{let}\ (l \to be) \in bs & \\
 & \quad\ (e', bt') = hnf\ bt\ be & \\
 & \quad \textbf{in}\ \ push\ e'\ bt' & \\
\quad \mid hnf\ e \equiv ([\![C\ e_1 \ldots e_n]\!], bt') & = \textbf{let}\ (C\ v_1 \ldots v_n \to be) \in bs & \\
 & \quad\ \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} & \\
 & \quad\ (e', bt') = hnf\ bt\ (\sigma\ be) & \\
 & \quad \textbf{in}\ \ push\ e'\ bt' &
\end{array}
$$

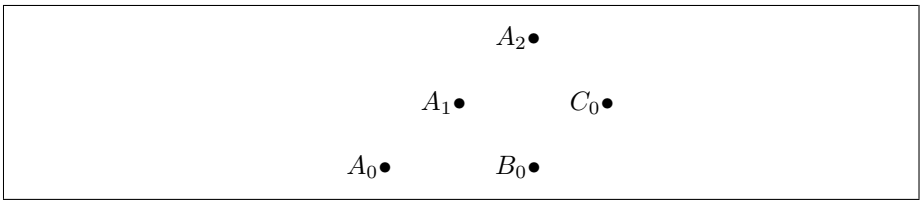
Notice that we don't need to push values in head normal from onto the stack, since there is no evaluation. We also don't push let expressions onto the stack, since they already represent an expression graph.

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.

$$
\begin{array}{ll}
hnf :: Stack \to Expr \to (Expr, Stack) & \\
hnf\ bt\ [\![\bot]\!] & = (\bot, bt) \\
hnf\ bt\ [\![l]\!] & = (l, bt) \\
hnf\ bt\ [\![e :: t]\!] & = \textbf{let}\ (e', bt') = FORWARD\ (hnf\ bt\ [\![e]\!]) \\
 & \quad \textbf{in}\ (FORWARD\ e', bt') \\
hnf\ bt\ [\![C\ e_1 \ldots e_n]\!] & = (C\ e_1 \ldots e_n, bt) \\
hnf\ bt\ [\![FREE]\!] & = (FREE, bt) \\
hnf\ bt\ [\![f\ e_1 \ldots e_n]\!] & = \textbf{let}\ v_1 \ldots v_n = vars\ f \\
 & \qquad e_f = body\ f \\
 & \qquad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
 & \qquad (e'_f, bt') = hnf\ bt\ (\sigma\ e_f) \\
 & \quad \textbf{in}\ \ push\ e'_f\ bt' \\
hnf\ bt\ [\![FORWARD\ e]\!] & = \textbf{let}\ (e', bt') = hnf\ bt\ e \\
 & \quad \textbf{in}\ \ push\ (FORWARD\ e')\ bt' \\
hnf\ bt\ [\![e_1\ ?\ e_2]\!] & = \textbf{let}\ (e'_{\_1}, bt') = hnf\ bt\ e_1
\end{array}
$$

$$
\begin{array}{lll}
 & & \textbf{in}\ \ \mathit{pushChoice}\ (\mathit{FORWARD}\ e'_{-1})\ bt' \\
\mathit{hnf}\ bt\ [\![\ \textbf{let}\ v_1 = e_1\ \textbf{in}\ e]\!] & = & \textbf{let}\ \sigma = \{\, v_1 \to e_1 \,\} \\
 & & \textbf{in}\ \ \mathit{hnf}\ bt\ (\sigma\ e) \\
\mathit{hnf}\ bt\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!] & & \\
\quad \mid e \equiv [\![\mathit{FORWARD}\ e']\!] & = & \textbf{let}\ (e', bt') = (\mathit{hnf}\ bt\ [\![\ \textbf{case}\ e'\ \textbf{of}\ bs]\!]) \\
 & & \quad \mathit{fwd} = \mathit{FORWARD}\ e' \\
 & & \textbf{in}\ \ \mathit{push}\ (\mathit{FORWARD}\ e')\ bt' \\
\quad \mid e \equiv [\![\mathit{FREE}]\!] & = & \textbf{let}\ \{\, C_1 \ldots \to \_, \ldots, C_n \ldots \to \_ \,\} = bs \\
 & & \quad e_1 = C_1\ \mathit{FREE} \ldots \mathit{FREE} \\
 & & \quad \ldots \\
 & & \quad e_n = C_n\ \mathit{FREE} \ldots \mathit{FREE} \\
 & & \quad e := e_1\ ?\ \ldots\ ?\ e_n \\
 & & \textbf{in}\ \ \mathit{hnf}\ bt\ [\![\ \textbf{case}\ e\ \textbf{of}\ bs]\!] \\
\quad \mid \mathit{hnf}\ e \equiv ([\![l]\!], bt') & = & \textbf{let}\ (l \to be) \in bs \\
 & & \quad (e', bt') = \mathit{hnf}\ bt\ be \\
 & & \textbf{in}\ \ \mathit{push}\ e'\ bt' \\
\quad \mid \mathit{hnf}\ e \equiv (([\![C\ e_1 \ldots e_n]\!], bt') & = & \textbf{let}\ (C\ v_1 \ldots v_n \to be) \in bs \\
 & & \quad \sigma = \{\, v_1 \to e_1, \ldots v_n \to e_n \,\} \\
 & & \quad (e', bt') = \mathit{hnf}\ bt\ (\sigma\ be) \\
 & & \textbf{in}\ \ \mathit{push}\ e'\ bt'
\end{array}
$$

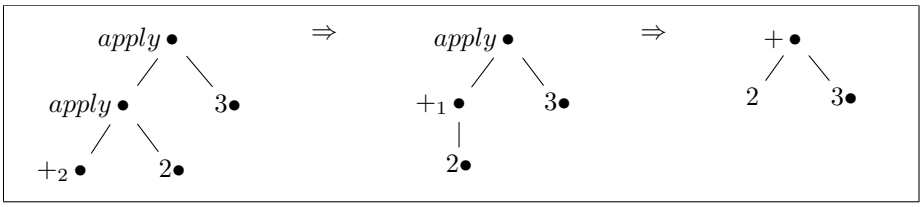
2.2.5 Higher Order Functions

The last feature we need to add is higher order functions. This is typically done with defunctionalization. [**?**]. 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.

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

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.

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

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.

$not :: Bool \to Bool$
$not\ x = \textbf{case}\ x\ \textbf{of}$
$\quad True \to False$
$\quad False \to True$

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 [**?**]. 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:

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

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. [**?**, **?**] 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`, then e is `nondet`. If e is a forward node to a `nondet` node, 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. This doesn't prove the correctness by any means, but it's pretty compelling. Even better, 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:

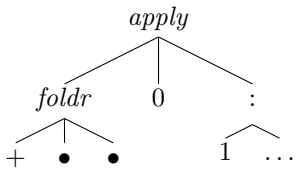
```
case False_TAG:
  root->tag = True_TAG;
  root->hnf = CTR_hnf;
  if(x->nondet)
  {
    push(bt_stack, root, make_not_(x), false);
  }
  return;
```

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 *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.

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

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 versital and is the main workhorse of the compiler and optimizer. We show how to construct, combine, and improve the efficiency of transformation, 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 too. 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, I will be describing a system to apply optimizations to FlatCurry programs. While this is not the only area of the compiler that I've applied optimizations, it is by far the most extensive, so it's worth understanding how my 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 an absolute value function defined below.

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

This will be translated into FlatCurry as

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

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 statement. 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 it's 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 the fact that there is 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. [**?**, **?**] 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 functional 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 the optimizations apply. However, these are typically limited in the optimizations they can apply. [**?**]

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). My approach was to do optimization entirely by rewriting. This has several advantages, and may end up being the most useful result of this work. First, Developing new optimizations is simple. For every new optimization, we can write it down in this system within minutes. It was often easier to write down the optimization and test it, than it was to try and describe the optimization. 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 my compiler, but I was 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 my system. Fifth, Since I'm optimizing by rewrite rules, the

compiler can easily output what rule was used, and the position it was used in 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 my optimizations, and there is 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 the 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. My implementation relies on finding a single value form a set generated by a non-deterministic function. Current implementations are inefficient, but there are new implementations being developed. [**?**] I also believe that an optimizing compiler would help with this problem [**?**].

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 nondeterministic rewriting.

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

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

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. 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 either be transformed into either 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:

$$\begin{aligned}
&\textbf{let } a = b \\
&\quad\ \ b = c \\
&\quad\ \ c = d + e \\
&\quad\ \ d = b \\
&\quad\ \ e = 1 \\
&\textbf{in } \ a
\end{aligned}$$

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

$$\begin{aligned}
&\textbf{let } e = 1 \\
&\textbf{in } \ \textbf{let } b = c \\
&\qquad\qquad\ \ c = d + e \\
&\qquad\qquad\ \ d = b \\
&\qquad\ \ \textbf{in } \ \textbf{let } a = b \\
&\qquad\qquad\ \ \textbf{in } \ a
\end{aligned}$$

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.

$$blocks\ \_\ [\![Let\ vs\ e]\!] \mid numBlocks > 1 = e'$$
$$\textbf{where}\ (e', numBlocks) = makeBlocks\ es\ e$$

$$makeBlocks\ vs\ e = (letExp, length\ comps)$$
$$\textbf{where}\ letExp = foldr\ makeBlock\ e\ comps$$
$$makeBlock\ comp = \lambda exp \rightarrow [\![\ \textbf{let}\ (map\ getExp\ comp)\ \textbf{in}\ exp]\!]$$
$$getExp\ n \mid [\![n = exp]\!] \in vs = [\![n = exp]\!]$$
$$comps = scc\ (vs \ggg\!= makeEdges)$$
$$makeEdges\ [\![v = exp]\!] = [(v, f) \mid f \leftarrow freeVars\ exp \cap map\ fst\ vs]$$

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 I 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 combining two optimizations is simple. I 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 was quite simple really. We simply pick a arbitrary subexpression with *subExpr* and apply an optimization to it. We can then use a non-deterministic fix point operator to find all transformations that can applied to the current expression. We can define the non-deterministic fix point operator using either the Findall library, or Set Function [**?**, **?**] 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.

```
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.

3.0.4 A Second Attempt: Multiple Transformations Per Pass

Our second attempt was much more successful. Instead of picking an arbitrary subexpression, we chose 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

This approach is significantly better. Aside from being able to apply 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 to 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 will also have one occasion where it's important 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 free. 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*.

$$\begin{aligned}
&reduce :: Opt \to Bool \to Expr \to State\ Int\ Expr \\
&reduce\ opt\ top\ [\![v]\!] &&= runOpts\ opt\ top\ [\![v]\!] \\
&reduce\ opt\ top\ [\![l]\!] &&= runOpts\ opt\ top\ [\![l]\!] \\
&reduce\ opt\ top\ [\![f\ es]\!] &&= \textbf{do}\ es' \leftarrow mapM\ (run\ opt\ False\ [\![es]\!]) \\
&&&\quad\ runOpts\ opt\ top\ [\![f\ es]\!] \\
&reduce\ opt\ top\ [\![\ \textbf{let}\ vs\ \textbf{in}\ e]\!] &&= \textbf{do}\ vs' \leftarrow mapM\ runVar\ vs \\
&&&\quad\ e' \leftarrow mapM\ run\ opt\ False\ e
\end{aligned}$$

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


reduce                              :: Opt → Expr → Expr
reduce opt ⟦v⟧                      = runOpts opt ⟦v⟧
reduce opt ⟦l⟧                      = runOpts opt ⟦l⟧
reduce opt ⟦f es⟧                   = runOpts opt ⟦f es′⟧
  where es′ = map (run opt) es
reduce opt ⟦ let vs in e⟧           = runOpts opt ⟦ let vs′ in e′⟧
  where vs′ = map (λ⟦v = e⟧ → ⟦v = run opt e⟧) vs
        e′  = run opt e
reduce opt ⟦ let vs free in e⟧ = runOpts opt ⟦ let vs free in e′⟧
  where e′  = run opt e
reduce opt ⟦e : t⟧                  = runOpts opt ⟦e′ : t⟧
  where e′  = run opt e
reduce opt ⟦e1 ? e2⟧                = runOpts opt ⟦e′1 ? e′2⟧
  where e′1 = run opt e1
        e′2 = run opt e2
reduce opt ⟦ case e of bs⟧          = runOpts opt ⟦ case e′ of bs′⟧
  where e′  = run opt e
        bs′ = map (λ⟦pat → e⟧ → ⟦pat → (run opt e′)⟧) bs


runOpts :: Opt → Expr → Expr
runOpts opt e = if opt e ≡ ∅
                then e
                else let e′ ∈ opt e in e′


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

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

```
                                        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'
```

```
simplify :: Opt → Expr → Expr
simplify opt e = fix (reduce opt True) e (maximum (vars e) + 1)
```

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. 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 addition, then we need to make a few changes. First, we need to decide on a representation for a rewrite derivation. Traditionally

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

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**
  $p_0 \to E_0$
  $p_1 \to E_1$
  $\ldots$
  $p_n \to E_n$

Figure 3.3: 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.

a rewrite derivation is a sequence of rewrite steps, where each step contains the rule and position of the rewrite. We describe paths in figure 3.3 To make reconstruction easier, I'll also include the expression that is the result of the rewrite. This gives us the type:

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

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

type $Opt = (Int, Bool) \to (Expr, String, Int)$

So, what changes do we need to make to the algorithm? Again there isn't a lot. 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 $update :: Expr \to Step \to Int \to ReWriter\ Expr$ that is similar to *put* from *State*. This updates the state variable, and creates a single step. The *reduce* function requires few changes. We change the boolean variable *top* to a more general *Path*. Because of this change, we need to add the correct subexpression position, instead of just changing *top* to *False*. The *RunOpts* function is similar. We just change *top* to a *Path*, and check if it'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.

$$
\begin{array}{lll}
reduce & :: Opt \to Path \to Expr \to ReWriter\ Expr \\
reduce\ opt\ p\ [\![v]\!] & = runOpts\ opt\ p\ [\![v]\!] \\
reduce\ opt\ p\ [\![l]\!] & = runOpts\ opt\ p\ [\![l]\!] \\
reduce\ opt\ p\ [\![f\ es]\!] & = \mathbf{do}\ es' \leftarrow mapM\ runArg\ (zip\ [0\,..]\ es) \\
 & \qquad runOpts\ opt\ p\ [\![f\ es]\!] \\
\quad \mathbf{where}\ runArg\ (n, [\![e]\!]) = \mathbf{do}\ e' \leftarrow run\ opt\ (n:p)\ e & \\
 & \qquad return\ [\![e']\!] \\
reduce\ opt\ p\ [\![\ \mathbf{let}\ vs\ \mathbf{in}\ e]\!] & = \mathbf{do}\ vs' \leftarrow mapM\ runVar\ (zip\ [0\,..]\ vs) \\
 & \qquad e' \leftarrow mapM\ run\ opt\ (-1:p)\ e \\
 & \qquad runOpts\ opt\ p\ [\![\ \mathbf{let}\ vs'\ \mathbf{in}\ e']\!] \\
\quad \mathbf{where}\ runVar\ (n, [\![v = e]\!]) = \mathbf{do}\ e' \leftarrow run\ opt\ (n:p)\ e & \\
 & \qquad return\ [\![v = e']\!] \\
reduce\ opt\ p\ [\![\ \mathbf{let}\ vs\ \mathbf{free\ in}\ e]\!] = \mathbf{do}\ e' \leftarrow run\ opt\ (0:p)\ e & \\
 & \qquad runOpts\ opt\ p\ [\![\ \mathbf{let}\ vs\ \mathbf{free\ in}\ e']\!] \\
reduce\ opt\ p\ [\![e:t]\!] & = \mathbf{do}\ e' \leftarrow run\ opt\ (0:p)\ e \\
 & \qquad runOpts\ opt\ p\ [\![e':t]\!] \\
reduce\ opt\ p\ [\![e_1\ ?\ e_2]\!] & = \mathbf{do}\ e_{-1}' \leftarrow run\ opt\ (0:p)\ e_1 \\
 & \qquad e_{-2}' \leftarrow run\ opt\ (1:p)\ e_2 \\
 & \qquad runOpts\ opt\ [\![e_{-1}'\ ?\ e_{-2}']\!] \\
reduce\ opt\ p\ [\![\ \mathbf{case}\ e\ \mathbf{of}\ bs]\!] & = \mathbf{do}\ e' \leftarrow run\ opt\ (-1:p)\ e \\
 & \qquad bs' \leftarrow mapM\ runBranch\ (zip\ [0\,..]\ bs) \\
 & \qquad runOpts\ opt\ [\![\ \mathbf{case}\ e'\ \mathbf{of}\ bs']\!] \\
\quad \mathbf{where}\ runBranch\ (n, [\![pat \to e]\!]) & = \mathbf{do}\ e' \leftarrow run\ opt\ (n:p)\ e \\
 & \qquad return\ [\![pat \to e']\!]
\end{array}
$$

$$
\begin{array}{l}
runOpts :: Opt \to Path \to Expr \to ReWriter\ Expr \\
runOpts\ opt\ p\ e = \mathbf{do}\ v \leftarrow get \\
\qquad \mathbf{if}\ opt\ (v, null\ p)\ e \equiv \emptyset \\
\qquad \mathbf{then}\ return\ e \\
\qquad \mathbf{else\ do\ let}\ (e', rule, dv) \in opt\ e \\
\qquad\qquad update\ (e', rule, p)\ dv \\
\qquad\qquad return\ e'
\end{array}
$$

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

```
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')


simplify :: Opt → Expr → (Expr, [Step])
simplify opt e = fix (reduce opt []) e (maximum (vars e) + 1) []
```

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

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

3.1 The Compiler Pipeline

3.1.1 Canonical FlatCurry

Let's look at a transformation for *apply* nodes. Recall that *apply f as* applies a function f to the arguments in the list *as*. Now, if f is a known function,

then we can statically determine how many arguments it's missing. We use f_k to denote f missing k arguments. While, in theory, we would never create an apply node of a known function, in practice this comes up often, either because it was easier to compile this way, or this application became exposed over the course of optimization.

In any case, we should eliminate

```
unapply :: Opt
unapply [[apply f_k as]]
  = case compare k n of
      LT -> [[apply (f as_1) as_2]]
      EQ -> [[f as]]
      GT -> [[f_{k-n} as]]
  where n          = length as
        (as_1, as_2) = split k as
```

```
unapply :: Opt
unapply v [[apply f_k as]]
  = case compare k n of
      LT -> [[ let v = f as1 in apply v as2]]
      EQ -> [[f as]]
      GT -> [[f_{k-n} as]]
  where n          = length as
        (as1, as2) = split k as
```

```
case (case e of p_{2,1} -> e_{2,1}
                ...
                p_{2,n} -> e_{2,n}) of
     p_{1,1} -> e_{1,1}
     ...
     p_{1,n} -> e_{1,n})
=>
case e of
     p_{2,1} -> case e_{2,1} of
                  p_{1,1} -> e_{1,1}
                  ...
                  p_{1,n} -> e_{1,n}
     ...
     p_{2,n} -> case e_{2,n} of
                  p_{1,1} -> e_{1,1}
                  ...
                  p_{1,n} -> e_{1,n}
```

3.1.2 ICurry

3.1.3 C