Practical Programming with Total Functions

Karl Voelker

July 27, 2010

Submitted in fulfillment of the requirements for the degree of
Master of Science.

Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

Chair, Professor Matthew Fluet

Reader, Professor James Heliotis

Observer, Professor Stanisław P. Radziszowski

Graduate Advisor, Professor Hans-Peter Bischof

Abstract

Functional programming offers an advantage over imperative programming: functional programs are easier to reason about and understand, which makes certain classes of errors less common. Yet, the two disciplines have some pitfalls in common: any computation, functional or not, may be non-terminating, or may terminate in a run-time error. Turner describes a discipline called "Total Functional Programming" (TFP) in which these pitfalls are impossible, due to some easily-checked rules which require all recursion to be done structurally. [30]

In this report, I detail my findings about the practical benefits and limitations of total functional programming, as well as the interactions which arise between TFP and the design and implementation of a rich functional programming language. These findings are the result of implementing a TFP compiler as a modification of the Glasgow Haskell Compiler (GHC), as well as a total standard library and a variety of total example programs.

Contents

List of Figures

Chapter 1

Background

Functional programming (FP) languages are known for their great expressive power. Even better, by allowing—or, in the case of *pure* FP, mandating—a referentially-transparent programming style, reasoning about the behavior of such programs is made easier. This is advantageous to students who are learning to write programs, because they can easily work out what their programs are doing, and they can think in the mathematical terms they've already learned.

For example, consider two functions which sum the numbers in a list:

```
sum :: [Int] -> Int
sum []       = 0
sum (x : xs) = x + sum xs
```

```
int sum(int n, int *arr) {
  int acc = 0;
  for (int i = 0; i < n; ++i) {
    acc += arr[i];
  }
  return acc;
}
```

The first function, written in Haskell, can be understood via equational reasoning, a process fundamentally comprehensible to anyone who has studied basic algebra, which looks like this:

$$sum([2,3]) = 2 + sum([3]) = 2 + 3 + sum([]) = 2 + 3 + 0 = 5$$

Equational reasoning does not suffice for the second function, written in C. Understanding how the C function computes the sum requires understanding the imperative model of a program as a sequence of steps *and* the concept of rewritable computer memory.

Unfortunately, even a pure functional program can encounter a run-time error, such as dividing by zero or taking the head of an empty list, and even a pure functional program may fail to terminate. It can thus be said that many of the functions in such a program are *partial* functions. Furthermore, if all the functions in a program were *total* rather than partial, run-time errors and non-termination would be impossible. Of course, deciding the totality of a function is, in general, an undecidable problem.

Despite the impossibility of the problem when applied to the language of all functions, it is entirely feasible to define a more restrictive language in which all functions happen to be total. Programs written in such a language would have many desirable properties: they could not encounter run-time errors or infinite loops, theorems about their meaning could be more easily proven, including by automated provers, and as such they could be more aggressively optimized and analyzed by compilers.

1.1 Implementing a Total Language

Turner describes a programming discipline in which all functions are total: Total Functional Programming (TFP) [30]. This is ensured by a set of simple rules which all functions must obey. Importantly, these rules do not add significant complication beyond that of a typical functional programming language. The

rules are:

- All data must be immutable.
- All pattern-matching must be exhaustive, since clearly any function which employs non-exhaustive pattern-matching must not be total.
- Datatype definitions must not recurse on the left-hand side of an $\rightarrow$ operator. Types violating this rule can allow arbitrary recursion to sneak back into the language via the creation of a fixpoint operator, as in this example [30]:

  ```
  data F a b = F (F a b -> a -> b)

  fix :: (F a b -> a -> b) -> (a -> b)
  fix f = f (F f)

  up :: Int -> Int
  up = fix (\(F f) a -> f (F f) (a + 1))
  ```

- Recursion must be structural. Thus, all recursive calls must be made by "syntactic descent on data constructors." This is also known as "primitive recursion." An example is this definition of `map`:

  ```
  map :: (a -> b) -> [a] -> [b]
  map f []       = []
  map f (x : xs) = f x : map f xs
  ```

  In that example, when `map` is called on a non-empty list, `(x : xs)`, the recursive call is made on `xs`, which is known by the use of deconstructor syntax to be a subcomponent of `(x : xs)`. Conceptually, requiring structural recursion allows the compiler to infer a proof by induction of the recursive function's termination. As such, the second parameter of `map` may be described as an "induction variable."

  An example of invalid recursion is this function:

```
log2_approx 1 = 0
log2_approx n = 1 + log2_approx (n `div` 2)
```

That example is invalid because, when `log2_approx` is called on n, the recursive call is made on $\lfloor \frac{n}{2} \rfloor$, which is not the direct result of any use of deconstructor syntax on n.

1.2 Codata and Corecursion

If a language were limited entirely to structurally recursive functions or Walther recursive functions [20] (described further in Section 6.3), it would be able to express many useful algorithms, but would not be well-suited to the implementation of operating systems, which may be intended to run forever. As such, Turner suggests that in addition to data and functions, which are necessarily finite, a language should also provide codata and cofunctions, which are potentially infinite. [30]

Codata constructors are defined in the same manner as data constructors. A cofunction is one which produces a result of a codata type. If a cofunction recurses, that corecursive call must be directly wrapped by a coconstructor call. This ensures that although a codata structure may be infinite, the computation needed to reach down one layer into the structure is always finite. Cofunctions meeting this criterion are described as "productive."

Although codata appear syntactically similar to data, one important difference is that recursing on the substructure of a codata value does not make recursion valid, since in an infinite codata structure, a substructure is no closer to the "bottom" than the structure which contained it.

There is a "principle of coinduction" described by Turner which enables proofs of the equivalence of two different corecursive functions to be written in an inductive style. [30] This principle enhances the usefulness of TFP to programmers, but the paper does not suggest any mechanism by which such proofs might be created and used automatically by a compiler.

Turner suggests two possible uses of codata with little elaboration [30]:

- A language may provide input and output facilities via special codata structures. A simple approach similar to that used by Miranda [19] is to have each input stream represented by a `Colist Char`. If an earlier part of the colist is read a second time, the data will be provided from a cache so that it is guaranteed to match what was read the first time.
- A program may represent infinite concepts, such as the Fibonnaci sequence, directly.

Consider this simple example borrowed from Turner [30]:

```
codata Colist a = Nil | Cons a (Colist a)


comap :: (a -> b) -> Colist a -> Colist b
comap _ Nil         = Nil
comap f (Cons h t) = Cons (f h) (comap f t)
```

In the example above, the corecursive call to *comap* is an argument to the coconstructor *Cons*. In contrast, consider this unfortunately invalid example, also borrowed from Turner [30]:

```
evens = Cons 2 (comap (+ 2) evens)
```

This example is invalid because the corecursive call to *evens* is not directly wrapped in a coconstructor due to the intervening call to *comap*. Despite not being regarded as valid, *evens* is nonetheless productive. Telford and Turner describe an abstract interpretation which would allow this definition of *evens*. [27] Implementing that abstract interpretation would be a significant effort on its own, and as such is beyond the scope of this work.

1.3 Numbers in TFP

Numeric data types and operations are an important consideration in any programming language. The structurally recursive nature of Total Functional Pro-

gramming suggests that any such language should include a type for natural numbers. Runciman suggests that natural numbers should be given greater weight in any programming language, as well as giving rationale for certain definitions of arithmetic operations on the natural numbers. [24] Some features of the given definitions are:

- $0 - n = 0$

  The rationale for this definition is an analogy to the concrete situations these numbers frequently represent. One concrete example is a list. Consider this typical definition of `drop`:

  ```
  drop _        []       = []
  drop 0        xs       = xs
  drop (n + 1) (x : xs) = drop n xs
  ```

  Combining `drop` with the given definition of subtraction, we have the identity `length (drop n xs) == (length xs) - n`.

- $\frac{n}{0} = n$

  The rationale for this definition is based on another function defined by Runciman, $slice(n, d) = \frac{n}{d+1}$. [24] This function can be understood as finding the size of one slice of n when d cuts are made. Division is then defined as $div(n, d) = slice(n, d - 1)$. So, when $d > 0$, $div(n, d) = \frac{n}{d}$, but when $d = 0$, $div(n, 0) = slice(n, 0 - 1) = slice(n, 0) = \frac{n}{1} = n$, using the aforementioned definition of subtraction. It is noted that this definition of division is monotonic, but no more practical suggestions are made regarding its value.

  An alternative to this kind of division, also mentioned by Runciman, is to use $\frac{n}{0} = \infty$. Although infinity is often represented with a special data constructor, Runciman suggests instead that infinity could be defined as the codata value `infinity = Succ infinity`. [24] Such a representation is appealing for its simplicity: basic arithmetic operations like `+` and `*` can

be easily written as cofunctions without treating infinity as a special case. But, the representation has drawbacks: such a number cannot be easily formatted for printing, and many other operations will not work as desired. Consider, for example, `infinity - infinity`, which would presumably diverge. In a total system, using this representation would require the use of a codata type. This would prevent divergence but would not make difficult operations any easier.

Chapter 2

Summary of Work

2.1 The Compiler

I adapted Turner's rules for sound recursion and productive corecursion [30] to the complex Haskell 98 language. Then, inside the Glasgow Haskell Compiler (GHC), version 6.12, a popular Haskell compiler, I implemented the adapted rules. The rules are enabled by a GHC "language option" so that the compiler can be used normally or in "total mode," in which case a total module is produced. In addition to implementing the rules, I also added codata declarations to the input syntax accepted by GHC in total mode.

2.2 The Libraries

I adapted the Haskell standard libraries, and some important libraries included with GHC, to only expose total functions. This was accomplished partly by exporting existing safe functions, partly by implementing wrappers to make existing functions safe, and partly by writing completely new functions.

The more novel components of the total standard libraries include a natural number library, a colist library, and a filesystem navigation library.

2.3 The Examples

To determine the practicality of TFP, I planned and implemented a wide variety of complex example programs which compile entirely in total mode. I observed patterns in these programs which are included in this report.

Chapter 3

The Compiler

3.1 The Sound Recursion Rule

The rule described by Turner requiring structural recursion [30] is simple, so simple that no formalized description was necessary to make it clear. My work in the context of Haskell [11], a much more complicated language, has produced a rule which demands a formal description.

I will begin by formalizing Turner's rule, Figure 3.1, so that I may formalize mine in the same terms. All of the formalizations in this chapter operate on syntactic entities, primarily named function definitions. Also, at the stage in the compiler where the rule is implemented, all variables have been given unique identifiers, so two mentions of a variable are equal iff they mention the same variable.

Note that in Figure 3.1, the function u, which identifies the bindings that result from pattern-matching deconstruction of a parameter, returns bindings regardless of their depth in the structure of the pattern. For example, consider this function:

```
f (x1 : (x2 : xs)) = xs
```

Note that $u(\texttt{f}, 0) = \{\texttt{x1}, \texttt{x2}, \texttt{xs}\}$.

The predicate p is the critical component of Figure 3.1. A rough explanation of the meaning of p is that, among the arguments to a recursive call:

- one argument must be a substructure of the corresponding input parameter, and
- all preceding arguments must either be substructures of or equivalent to their corresponding input parameters.

Let $n(f)$ be the arity of f.

Let $a(f, i)$ be the ith parameter binding of f for $i \in [0, n(f))$.

Let $u(f, i)$ be the set of all bindings made by case analysis on $a(f, i)$ in patterns of the form $D\ X^*$, where D is a data constructor and X is a pattern.

Let $r(f)$ be the number of recursive applications of f.

Let $g(f, i, j)$ be the ith argument of the jth recursive application of f for $j \in [0, r(f))$ and $i \in [0, n(f))$.

Let $p(f)$ be true iff $\forall j \in [0, r(f))\ \ \exists i \in [0, n(f))$ such that:

- $g(f, i, j) \in u(f, i)$ and
- $\forall h \in [0, i)\ \ [g(f, h, j) = a(f, i)]$.

A function f follows Turner's rule iff $p(f)$.

Figure 3.1: Turner's Rule

The meaning of Turner's rule can be demonstrated with an example. We will consider two functions, one valid and one invalid. To imitate the abstract syntax upon which the rule is actually executed, each binding has a unique name and pattern-matching is only done in `case` expressions.

```
f :: [Int] -> [Int]
f xs = case xs of
  []         -> 0
  (x : xs') -> x + f xs'
```

```
g :: [Int] -> [Int]
g ys = 1 : g ys
```

Now, let us trace the evaluation of the rule on these examples step by step:

$n(\mathtt{f}) = 1$

$a(\mathtt{f}, 0) = \mathtt{xs}$

$u(\mathtt{f}, 0) = \{\mathtt{x}, \mathtt{xs}'\}$

$r(\mathtt{f}) = 1$

$g(\mathtt{f}, 0, 0) = \mathtt{xs}'$

$g(\mathtt{f}, 0, 0) \in u(\mathtt{f}, 0)$

$p(\mathtt{f})$

$n(\mathtt{g}) = 1$

$a(\mathtt{g}, 0) = \mathtt{ys}$

$u(\mathtt{g}, 0) = \emptyset$

$r(\mathtt{g}) = 1$

$g(\mathtt{g}, 0, 0) = \mathtt{ys}$

$g(\mathtt{g}, 0, 0) \notin u(\mathtt{g}, 0)$

$\neg p(\mathtt{g})$

3.1.1 My Rule

I will first present a basic form of my rule for recursion, Figure 3.3, followed by three important enhancements. Informally, my rule is different in that it accounts for mutual recursion, a feature of Haskell 98.

Mutual recursion is handled by the rule that in any mutually-recursive cycle of functions, there must be one function in which all calls to any functions in the cycle follow Turner's rule as if those were directly recursive calls. This is the first part of $p(c)$, which begins with $\exists f \in c$.

Furthermore, all other functions in the cycle must follow a slightly looser rule: rather than having to descend on the structure of their arguments, they must merely not expand the structure of their arguments. This is the second part of $p(c)$, which begins with $\forall f \in c$.

Mutual recursion between functions of differing arity must also be considered. In the rule, we find the minimum arity of all functions in a cycle ($n'(c)$) and effectively truncate all parameter and argument lists to that minimum.

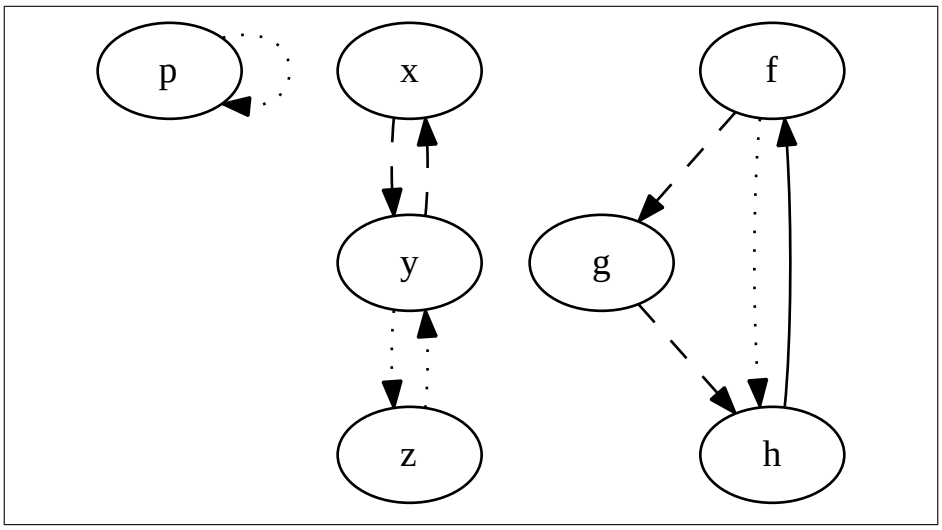

Figure 3.2: Mutually-recursive cycles: each cycle in a graph is drawn in a distinct style, with a solid edge being in all the graph's cycles.

The rule does not formally define what a mutually-recursive cycle is, but I will give a definition here: a mutually-recursive cycle is a cycle in the static call graph which does not visit any function more than once. A mutually recursive cycle may consist of a single function which calls itself. This isn't really "mutual recursion," but that doesn't matter for the purposes of these rules. Figure 3.2 contains some example cycles.

Partial Application

Extending Turner's rule to support partial applications, a feature of Haskell 98, is trivial. The only real problem with the formal rule as written is that the definition of p may go out-of-bounds in its use of g.

Extending the previous rule, which encompasses mutual recursion, is even more trivial, but for the sake of completeness I will formalize it here as Figure 3.4.

Let n, a, and u be as in Figure 3.1.

Let $C(f)$ be the set of all mutually-recursive cycles which include f.

Let $n'(c) = \min_{f \in c} n(f)$.

Let $r(f, c)$ be the number of applications in the body of f of any function in c.

Let $g(f, c, i, j)$ be the ith argument of the jth application in the body of f of any function in c for $j \in [0, r(f, c))$ and $i \in [0, n(f))$.

Let $p(c)$ be true iff:

- $\exists f \in c \;\; \forall j \in [0, r(f, c)) \;\; \exists i \in [0, n'(c))$ such that:
  - $g(f, c, i, j) \in u(f, i)$ and
  - $\forall h \in [0, i) \;\; [g(f, c, h, j) = a(f, i)]$.

  and
- $\forall f \in c \;\; \forall j \in [0, r(f, c)) \;\; \forall i \in [0, n'(c))$:
  - $g(f, c, i, j) \in u(f, i)$ or
  - $g(f, c, i, j) = a(f, i)$.

A function f follows this rule iff $\forall c \in C(f) \;\; [p(c)]$.

Figure 3.3: My initial rule

Let n, a, and u be as in Figure 3.1.

Let C, r, g, and p be as in Figure 3.3.

Let $s(f, c, j)$ be the number of arguments to the jth application in f of any function in c for $j \in [0, r(f, c))$.

Let $n'(c) = \min_{f \in c}\{n(f), \min_{j=0}^{r(f,c)-1} s(f, c, j)\}$.

A function f follows this rule iff $\forall c \in C(f) \;\; [p(c)]$.

Figure 3.4: My initial rule, with partial application

Nested Functions

Haskell 98 supports the nested definition of functions. Nested functions have all the power of functions bound at the top level, including recursion and mutual recursion. Therefore, nested functions cannot simply be ignored and treated as part of their enclosing functions, as this example demonstrates:

```
f xs = g xs
  where
    g xs = g xs
```

Treating nested functions the same as top-level functions gives a sound rule, but one which is excessively restrictive, as demonstrated by this example:

```
f []                = []
f xss@(xs : xss') = g xs
  where
    g []        = f xss'
    g (x : xs') = x + g xs'
```

The problem with this example is that `xss'` is not related to any parameter of `g`, so the call `f xss'` in `g` is invalid, according to the rules developed so far.

This example could easily be rewritten to follow the rule. This example may even seem pointless. However, GHC routinely generates code of roughly this form in its desugaring phase, and my implementation of this rule comes after that phase, so accepting functions of this form is critical for my implementation. The rule that supports nested functions is Figure 3.5.

The solution to the problem of nested functions is simply to observe that a nested function can utilize all the parameters of its enclosing functions as if they were its own parameters. As such, the nested function rule uses a function $w(f)$, the function in which f is nested, in numerous recursive definitions. For example, $a(f, i)$, the ith parameter of f, now recurses on $w(f)$ in order to treat the parameters of $w(f)$ as parameters of f. Other parts of the rule have undergone a similar transformation.

Unfortunately, there is one remaining problem with this rule which prevents the function `f` in the example above from being accepted. The problem is that $n'(\{\texttt{f}, \texttt{g}\}) = 1$ because $n(\texttt{f}) = 1$. The rule effectively rewrites the application `g xs'` to be `g xss xs'`, and since $n'(\{\texttt{f}, \texttt{g}\}) = 1$, the argument `xs'`, which correctly functions as the induction variable in this instance, is never considered.

Let u be as in Figure 3.1.

Let C, r, and p be as in Figure 3.3.

Let n' be as in Figure 3.4.

Let n_{prev} and a_{prev} be n and a from Figure 3.1.

Let g_{prev} be g from Figure 3.3.

Let $w(f)$ be the function in which f is immediately nested, if one exists, or 0 otherwise.

Let $t(f, c, j)$ be the function applied by the jth application in the body of f of a function in c.

Let $n(f)$ be 0 if $f = 0$, or otherwise $n(w(f)) + n_{prev}(f)$.

Let $s(f, c, j) = y + n(w(t(f, c, j)))$ for $j \in [0, r(f, c))$ where y is the number of arguments to the jth application in the body of f of any function in c.

Let $a(f, i)$ be $a(x, i)$ if $i < n(x)$, or otherwise $a_{prev}(f, i - n(x))$ where $x = w(f)$.

Let $g(f, c, i, j)$ be:

- $a(x, i)$ if $i < n(x)$
- $g_{prev}(f, c, i - n(x))$ otherwise

where $x = w(t(f, c, j))$.

A function f follows the rule iff $\forall c \in C(f)\ \ [p(c)]$.

Figure 3.5: My initial rule, with partial application and nested functions

Parameter Permutations

The problem previously described is only one example of a broader problem with the rule: even when it is possible, it is not always convenient to put a function's parameters in the correct order necessary for the function to be valid.

I have implemented a solution to this problem in which the compiler puts the function's parameters in the correct order. It discovers the correct order by checking different permutations of the function's parameters against the rule until one succeeds or all fail. The number of permutations grows as $n!$ with the number of parameters, but in practice, nearly all functions have a small enough

Let $w(f)$ be the function in which f is immediately nested, if one exists, or 0 otherwise.

Let $n(f)$ be 0 if $f = 0$ or $n(w(f)) + x$ otherwise, where x is the arity of f.

Let $a(f, i)$ for $i \in [0, n(f))$ be:

- $a(x, i)$ if $i < n(x)$
- the yth parameter binding of f otherwise, where $y = i - n(x)$.

where $x = w(f)$.

Let $u(f, i)$ be the set of all bindings made by case analysis on $a(f, i)$ in patterns of the form $D\ X^*$, where D is a data constructor and X is a pattern.

Let $r(f, c)$ be the number of applications in the body of f of any function in c.

Let $C(f)$ be the set of all mutually-recursive cycles which include f.

Let $s(f, c, j) = y + n(w(t(f, c, j)))$ for $j \in [0, r(f, c))$ where y is the number of arguments to the jth application in the body of f of any function in c.

Let $n'(c) = \min_{f \in c}\{n(f), \min_{j=0}^{r(f,c)-1} s(f, c, j)\}$.
Note that $n'(c) \geq \min_{f \in c}\{n(w(f))\}$.

Let $t(f, c, j)$ be the function applied by the jth application in the body of f of a function in c.

Figure 3.6: My final rule, continued in Figure 3.7

number of parameters so that checking all permutations can be done in only a few seconds.

In the case of mutually recursive cycles, it is necessary to check all ways in which one permutation can be taken for each of the functions in the cycle.

My implementation first calculates the number of permutations which will have to be considered to do an exhaustive search. If that number is beyond a limit, no permutations are considered other than the one which appears in the program as written. After some experimentation, the limit was set at $7!^2 = 25,401,600$, enough for a mutually recursive cycle of two functions, each of which has seven arguments.

This rule, Figures 3.6 and 3.7, is the complete rule that my implementation uses (except when there are too many permutations, in which case it uses Figure

Let $g(f, c, i, j)$ for $i \in [0, n'(f))$ and $j \in [0, r(f, c))$ be:

- $a(x, i)$ if $i < n(x)$
- the yth argument of the jth application in f of any function in c otherwise, where $y = i - n(x)$.

where $x = w(t(f, c, j))$.

Let $g'(\mu, f, c, i, j) = g(f, c, \mu(t(f, c, j))(i), j)$.

Let $p(c, \mu)$ be true iff:

- $\exists f \in c \;\; \forall j \in [0, r(f, c)) \;\; \exists i \in [0, n'(c))$:
  - $g'(\mu, f, c, i, j) \in u(f, \mu(f)(i))$ and
  - $\forall h \in [0, i) \;\; [g'(\mu, f, c, h, j) = a(f, \mu(f)(i))]$,

  and
- $\forall f \in c \;\; \forall j \in [0, r(f, c)) \;\; \forall i \in [0, n'(c))$:
  - $g'(\mu, f, c, i, j) \in u(f, \mu(f)(i))$ or
  - $g'(\mu, f, c, i, j) = a(f, \mu(f)(i))$.

Let $\delta(f)$ be the set of all bijections from $\mathbb{Z}_{n(f)}$ to $\mathbb{Z}_{n(f)}$.

Let $\beta(c)$ be the set of all functions from c to $\bigcup_{f \in c} \delta(f)$ such that $\forall f \in c \;\; \forall \gamma \in \beta(c) \;\; [\gamma(f) \in \delta(f)]$.

A function f follows the rule iff $\forall c \in C(f) \;\; \exists \mu \in \beta(c) \;\; [p(c, \mu)]$.

Figure 3.7: My final rule, continued from Figure 3.6

3.5). I will later discuss some possible rules which I have not implemented.

It seems likely that there is a more efficient algorithm for solving this problem than a brute-force search of all permutations.

Now, consider these example functions:

```
-- correct
f :: [Int] -> [Int] -> Int
f xs ys = g xs ys where
  g ms ns = case ms of
    [] -> case ns of
      [] -> 0
      (n : ns') -> n * g xs ns'
```

```
    (m : ms') -> m + g ms' ns

-- incorrect
f :: [Int] -> [Int] -> Int
f xs ys = g xs ys where
  g ms ns = case ms of
    [] -> case ns of
      [] -> 0
      (n : ns') -> n * g xs ns -- typo
    (m : ms') -> m + g ms' ns
```

The first definition of `f` is correct. The second definition is almost the same, but contains a typo which invalidates it. Now, we will see, step by step, how my complete rule, as implemented, is applied to these functions.

Facts common to both definitions

$w(\mathtt{f}) = 0 \qquad w(\mathtt{g}) = \mathtt{f} \qquad n(\mathtt{f}) = 2 \qquad n(\mathtt{g}) = 4$

$a(\mathtt{f}, 0) = a(\mathtt{g}, 0) = \mathtt{xs} \qquad a(\mathtt{f}, 1) = a(\mathtt{g}, 1) = \mathtt{ys}$

$a(\mathtt{g}, 2) = \mathtt{ms} \qquad a(\mathtt{g}, 3) = \mathtt{ns}$

$u(\mathtt{f}, 0) = u(\mathtt{f}, 1) = u(\mathtt{g}, 0) = u(\mathtt{g}, 1) = \emptyset$

$u(\mathtt{g}, 2) = \{\mathtt{m}, \mathtt{ms}'\} \qquad u(\mathtt{g}, 3) = \{\mathtt{n}, \mathtt{ns}'\}$

$C(\mathtt{f}) = \emptyset \qquad C(\mathtt{g}) = \{\{\mathtt{g}\}\}$

Let $c = \{\mathtt{g}\}$, noting that $C(\mathtt{g}) = \{c\}$.

$s(\mathtt{g}, c, 0) = s(\mathtt{g}, c, 1) = 4 \qquad t(\mathtt{g}, c, 0) = t(\mathtt{g}, c, 1) = \mathtt{g}$

$r(\mathtt{g}, c) = 2 \qquad n'(\mathtt{g}) = 4$

$g(\mathtt{g}, c, 0, 0) = g(\mathtt{g}, c, 0, 1) = a(\mathtt{f}, 0) = \mathtt{xs}$

$g(\mathtt{g}, c, 1, 0) = g(\mathtt{g}, c, 1, 1) = a(\mathtt{f}, 1) = \mathtt{ys}$

$g(\mathtt{g}, c, 2, 0) = \mathtt{xs}$

$g(\mathtt{g}, c, 2, 1) = \mathtt{ms}'$ $\quad$ $g(\mathtt{g}, c, 3, 1) = \mathtt{ns}$

Facts about correct definition $\quad$ **Facts about incorrect definition**

$g(\mathtt{g}, c, 3, 0) = \mathtt{ns}'$ $\quad$ $g(\mathtt{g}, c, 3, 0) = \mathtt{ns}$ (the typo)

Now, note that $|\delta(\mathtt{g})| = 4! = 24$ and $c = \{\mathtt{g}\}$, so $|\beta(c)| = 24$. To show that the correct definition of `f` is valid, it must be shown that $\exists \mu \in \beta(c) \;\; [p(c, \mu)]$. However, to show that the incorrect definition of `f` is invalid, it must be shown that $\forall \mu \in \beta(c) \;\; [\neg p(c, \mu)]$. This would be more tedious than enlightening, so instead I will only show that the one particular value of μ with which I validate the correct definition of `f` does not validate the incorrect definition of `f`.

Facts common to both definitions

Let $\mu(\mathtt{g}) = \lambda n.3 - n$.

$g'(\mu, \mathtt{g}, c, 3, 0) = g'(\mu, \mathtt{g}, c, 3, 1) = a(f, 0) = \mathtt{xs}$

$g'(\mu, \mathtt{g}, c, 2, 0) = g'(\mu, \mathtt{g}, c, 2, 1) = a(f, 1) = \mathtt{ys}$

$g'(\mu, \mathtt{g}, c, 1, 0) = \mathtt{xs}$

$g'(\mu, \mathtt{g}, c, 1, 1) = \mathtt{ms}' \in u(\mathtt{g}, 2)$ $\quad$ $g'(\mu, \mathtt{g}, c, 0, 1) = \mathtt{ns} = a(\mathtt{g}, 3)$

Facts about correct definition $\quad$ **Facts about incorrect definition**

$g'(\mu, \mathtt{g}, c, 0, 0) = \mathtt{ns}'$ $\quad$ $g'(\mu, \mathtt{g}, c, 0, 0) = \mathtt{ns}$ (the typo)

$g'(\mu, \mathtt{g}, c, 0, 0) \in [u(\mathtt{g}, 3) = \{\mathtt{n}, \mathtt{ns}'\}]$ $\quad$ $\forall i \in [0, 3] \;\; [g'(\mu, \mathtt{g}, c, i, 0) \notin u(\mathtt{g}, 3-i)]$

$p(c, \mu)$ $\quad$ $\neg p(c, \mu)$

3.1.2 Other Considered Rules

Some analysis of the sound recursion rule which I implemented has led to some examples showing that the rule could be tweaked to gain additional generality.

However, since these cases have not arisen in practice, and accounting for them may increase the cost of checking adherence to the rule, it's possible that they are not worth implementing.

First, consider this example:

```
foo 0      x = x
foo (n+1)  x = bar n (x+1)


bar n      x = foo n (x+1)
```

Let $f =$`bar`. Thus, $C(f) = \{\{$`foo`,`bar`$\}\}$. Let $c = \{$`foo`,`bar`$\}$. Thus we have:

- $n'(c) = 2$
- $g(f, c, 0, 0) =$`n`
- $g(f, c, 1, 0) =$`(x+1)`
- $\forall \mu \in \beta(c), \exists i \in \{0, 1\}$ such that:
  - $g'(\mu, f, c, i, 0) \notin u(f, \mu(f)(i))$ and
  - $g'(\mu, f, c, i, 0) \neq a(f, \mu(f)(i))$

Thus, $\forall \mu \in \beta(c), \neg p(c)$.

Informally, the problem is that my rule blindly looks at both arguments in both applications. In the application of `foo`, that's a problem: if only the first argument had been considered, the rule would have recognized that it was the same as the input, but when considering both arguments, the arguments on the whole are bigger than the input.

The reason why it would be safe to only look at the first argument to the application of `foo` is that we only need to look at the first argument to the application of `bar` to discern that in that application, the arguments on the whole are smaller than the input.

The solution to this problem is to determine how many parameters need to be considered. This could mean, instead of using the fixed value of $n'(c)$, trying

all values in $\{1..n'(c)\}$. Alternately, the rule could first find all functions which recurse on values smaller than their inputs, and then find the one which reliably does so while considering the least number of the parameters.

Note that simply picking an arbitrary function which recurses on values smaller than its input is not enough. There could be a cycle with two candidate functions, where one candidate is better than the other, such as in this example:

```
foo 0     x     = x
foo (n+1) x     = bar n (x+1)

bar 0     0     = 0
bar (n+1) 0     = baz n 1
bar n     (x+1) = baz n x

baz n     x     = foo n (x+2)
```

From looking at `foo`, we would decide to only consider one parameter, whereas from looking at `bar`, we would decide to consider two parameters, which would make the cycle be considered invalid because of the recursion in `baz`.

3.2 The Productive Corecursion Rule

The goal of a rule for productive corecursion is to ensure that descending a finite depth into a recursive codata structure will not require infinite computation.

3.2.1 Turner's Rule

Turner's rule is that a corecursive function is accepted iff each of its recursive applications is an argument to a codata constructor.

3.2.2 My Rule

My rule is that a corecursive function f is accepted iff in each cycle c of mutually corecursive functions containing f, there exists a function g in c such that each application in g of any function in c is an argument to a codata constructor.

3.3 Mixed Recursive-Corecursive Cycles

Having considered only briefly the possibility of mutual recursion between functions and cofunctions, I simply banned them in my implementation. However, upon further reflection, it does seem that such a stringent rule is not necessary.

If all the functions in a mixed cycle are judged by the standard for total recursion, the result types don't matter: the functions terminate.

If all the functions in a mixed cycle are judged by the standard for productive corecursion, then if there are infinite recursive applications, each one is wrapped directly in a coconstructor, which is exactly where infinite data are supposed to be. No finite function can accidentally begin some nonterminating recursion by descending down this data, since the inductive inference cannot be made by pattern-matching on a coconstructor.

So, it seems that either standard can be applied to a cycle, so long as it is applied to all the functions in that cycle. Thus, the ideal check would be to check each cycle against both standards, succeeding if either subcheck succeeds. However, the need for this check never arose in my examples, which are described in Chapter 5.

3.4 Other Interactions with Haskell and GHC

Haskell and GHC are both complex and ornate. As such, I expected many issues to arise while implementing TFP atop GHC, and indeed many issues did. My discussion of these issues is broken into two categories: those which arise as a result of the Haskell language, and those which arise from GHC specifically.

3.4.1 With Haskell 98

Record Accessors

Haskell specifies a "record syntax" which can be used to make data constructors of high arity more convenient [9], but a record accessor produces a run-time error if used on a value made with the wrong data constructor [10]. Consider this example:

```
data T a = Foo { x :: a } | Bar { y :: a }
x :: T a -> a  -- implicit
n :: Int
n = x (Bar { y = 3 })  -- error
```

The solution to this problem which I implemented is a simple check: if a data constructor uses record syntax, it must be the only constructor of its type. That way, the convenience of the record syntax is not lost in cases where it is safe.

Another solution would have been to make record accessor functions return a `Maybe`. It might be ideal to change record accessors in this way only when there are multiple data constructors for a type, to avoid unnecessary `Maybe`s. Then again, this would mean that adding a data constructor to a type could break a lot of existing code.

Record Construction

Haskell's record syntax causes further trouble in that an application of a record constructor is not required to provide values for all the fields [10]. Unmentioned fields are undefined:

```
data T a = Foo { x :: a }
n :: Int
n = x (Foo { })  -- error
```

The implemented solution to this problem is to catch these errors at compile-time.

Type Classes

There are four ways in which Haskell's type classes [14] interact poorly with my implementation of TFP.

First, an instance declaration is not required to implement all the methods in the class. Any methods without an implementation produce a run-time error. This behavior was simple to change in the compiler, since it already issues a warning about undefined methods.

Second, corecursive functions cannot be class methods. This is due to the prohibition of mutually-recursive cycles which mix data and codata. Such cycles arise between a corecursive method and the corresponding instance dictionary. This limitation has been problematic in only one case so far in my experiments, that being the `enumFrom` and `enumFromThen` methods of the `Enum` class, which produce potentially-infinite lists of values from an enumeration.

Third, methods cannot be implemented in terms of each other. This is because of recursion which arises between the instance method and the instance dictionary. Consider this example:

```
class Foo a where
  bar :: a -> a
  baz :: a -> a

instance Foo Int where
  bar = id
  baz = bar
```

After desugaring, the code looks roughly like this:

```
-- class Foo a where ...
Foo     bar       baz = (bar, baz)
Foo_bar (bar, _)      = bar
Foo_baz (_, baz)      = baz
```

```
-- instance Foo Int where ...
Foo_Int     = Foo Foo_Int_bar Foo_Int_baz
Foo_Int_bar = id
Foo_Int_baz = Foo_bar Foo_Int
```

The invalid mutual recursion is between `Foo_Int` and `Foo_Int_baz`.

Fourth, an instance must define all the methods of the class, even if the class provides default implementations of some methods. This problem has a cause similar to the previous one.

Modules

A Haskell program is formed from a set of modules [13]. Clearly, a total module must not import a non-total module, and this restriction is implemented. Total modules may, however, be imported by any other module. Although a non-total module may "poison" its use of a total module by, for example, passing an infinite list into a total function, it cannot poison the total module in any global sense because of the pure functional nature of Haskell.

Incomplete Patterns

A case analysis in Haskell is not required to be complete [10], whereas this completeness is one of Turner's requirements [30]. I have implemented this requirement, although doing so was quite trivial, because GHC has the ability to warn about incomplete case analysis.

3.4.2 With GHC

Type Synonyms

The Haskell report says that a type synonym is exactly equivalent to its referent and that no type synonym may be used without all of its parameters applied [9]. Given this, one might expect a Haskell compiler's core syntax to make no mention of type synonyms, since they could be fully expanded for simplicity's

sake. However, GHC does not do this. To some extent, this may be explainable by GHC's support for certain extensions to the Haskell language related to the treatment of type synonyms.

Implementing Turner's requirement that datatypes be covariant [30] was made somewhat trickier by the possible presence of type synonyms in types.

Rebindable Syntax

Standard Haskell specifies various forms of "syntactic sugar" which map onto functions from the standard libraries. The scope in which such sugar is used does not matter: the sugar always references the same, standard functions [10].

Two forms of this sugar, `do`-notation and enumeration sequences (`..`-notation), are based on classes for which the total standard libraries have substitutes: `Monad` and `Enum` respectively. As such, I needed some mechanism by which the altered versions of these classes would be used in the desugaring process. Fortunately, GHC includes such a mechanism, called "rebindable syntax." [29] With this option enabled, certain syntactic forms look for the relevant functions in their scope rather than grabbing the standard function out of nowhere.

Incompatible Language Options

GHC supports many extensions to the Haskell language [28]. Many seemed dubious within the context of TFP: either they might cause trouble for my implementation in particular, or they might provide backdoors for breaking totality in general. Proving the safety of all of GHC's extensions is beyond the scope of this work, so those which are not obviously safe have been disabled.

3.5 Important Implementation Details

A full description of the design of my changes to the compiler is not included in this report. However, I have collected here some thoughts about the implementation which would likely be relevant to anyone attempting a similar

implementation.

3.5.1 Location in the Compilation Pipeline

Initially, I began to implement Turner's sound recursion rule in the front-end of GHC, operating on the `HsSyn` syntax, which represents the full syntactic complexity of Haskell. However, I soon realized that adapting Turner's rule to such a wide variety of syntactic forms, many with great semantic depth, would be a time-consuming task.

Since the original goal of this project was to put as much focus on the libraries and example programs as on the compiler, I didn't want to become mired in work at only the first stage of the project. As such, I decided to implement Turner's rule in the middle-end of GHC, operating on the vastly simpler core syntax.

That choice probably made my work easier, but not by as much as I had expected. Furthermore, I would highly recommend that any production-quality implementation of this concept operate at the highest syntactic level possible. The fundamental reason for this is that Turner's check is syntactic in nature, so the more changes the syntax has undergone internally, the greater the chance that a program which ought to be valid will be transformed into one which is invalid.

Now, I continue with details about the specific problems caused by working with the core syntax, as well as addressing issues which would arise when working with the rich syntax.

Poor Error Messages

GHC's core syntax carries no information about the location of definitions in the source code. As such, the error messages produced by my implementation are terse, simply stating the kind of problem and the name of the entities involved.

Furthermore, the desugaring process often introduces extra functions, with names that are meaningless to the programmer, which may end up in error

messages.

`n+k` Patterns

Identifying uses of `n+k` patterns in the core syntax is a challenge, since these patterns have disappeared and been replaced by expressions involving various arithmetic and comparison operators. My implementation looks for expressions which match the pattern generated by the compiler from `n+k` patterns.

It might have been easier to identify `n+k` patterns earlier in the compiler and record the identifiers involved for later use, but at the time it seemed that it would be more expeditious to identify the expression patterns than to acquaint myself with more gigantic portions of the GHC source code which bore little resemblance to the code with which I had already become familiar.

These patterns also cause an issue in identifying complete pattern matching. Since `n+k` patterns were originally intended for use on integral types, the compiler correctly considers {`0`, `n+1`} to be an incomplete set of patterns, since the pattern `n+1` only matches positive values of `n`, as per the specification. Yet, in the case of the natural number types, such a set of patterns is complete. No solution to this problem is implemented, since there is a fairly trivial workaround: put the `0` case last, and replace `0` with `_`.

Handling `do`-Notation

In retrospect, the semantics of `do`-notation may not be as daunting for Turner's rule as I had guessed. The only thing hidden by the notation is the use of the functions defined in the `Monad` class. So, except in the definition of an instance of `Monad`, the applications which are missing from the syntax will not be relevant to the detection of recursion.

3.5.2 The Implementation of Codata

One factor motivating the choice of Haskell as the base language for the implementation was that its lazy evaluation would make the implementation of

codata easier. In my implementation, data and codata are indistinguishable but for a Boolean flag carried by all the types which are used to describe data types in the various syntaxes, including the syntax which gets serialized into interface files.

3.5.3 Use of Language Options

GHC, as a result of its many language extensions [28], already has a system in place for enabling and disabling extensions on a per-module basis. These "language options" can be enabled and disabled on the command line or in a module file with a special pragma. My implementation uses three language options:

- `Codata`, which allows the `codata` syntax.
- `Total`, which enables checking of all the rules (and implies `Codata`).
- `FakeTotal`, which is used by certain system libraries to declare that they are believed to be total, although they should not be checked.

3.5.4 Use of Annotations

GHC has a generic system of annotations which can be added to any module, type, or function [1]. My implementation adds an annotation to a module which was compiled with the `Total` or `FakeTotal` options.

Chapter 4

The Libraries

4.1 Natural Number Types

Natural numbers, as noted by Runciman [24], are to be found everywhere in computer programs, as descriptors of the size of or a position in any finite data structure or computation. Yet, natural numbers have traditionally been ignored in favor of integers, which avoid certain fundamental problems such as how to define subtraction.

It is precisely because of the use of natural numbers to describe finite structures that Runciman's definition of $0 - n = 0$ makes sense. This is the definition used in my implementation.

Another important question is how natural numbers should be implemented. The "textbook" approach is to define them as any other data type:

```
data Natural = Zero | Succ Natural
```

This definition provides simplicity of implementation, and by implementing instances of Haskell's numeric type classes, such a type can also be made simple to the programmer who uses it. Integer literals can be implicitly converted to this type, the standard arithmetic operators can be defined, and pattern-matching on the data constructors can be used to write recursive functions

which rely on a natural number as the induction variable.

However, the run-time efficiency of nearly any operation on this type is dismal when compared to that of a machine integer. This desire for efficiency, which in a practical context may turn out to be quite critical, can be realized with this definition:

```
newtype Natural a = Natural a
```

This type is parameterized by what integer type we wish to use as its backing. In implementing the standard numeric instances for this type, we can indicate the constraint that the backing must indeed be an integer:

```
instance (Integral a) => Integral (Natural a) where
  fromInteger = Natural . fromInteger
  ...
```

The drawback of this implementation is that we can no longer use a `Natural` as an induction variable, which largely eliminates our motivation for having such a type in the first place. The ideal solution would be to use the latter implementation while providing the former means of pattern-matching.

If Haskell had support for views, a way of using distinct types for the internal and external representations of some data, such an ideal solution might be reachable without any change to the compiler, although the question of how to ensure the totality of a system of views is beyond the scope of this work. GHC does have support for "view patterns," but they are not adequate for our purposes, as this example demonstrates:

```
data    Natural' = Zero | Succ Natural'
newtype Natural  = Natural Int

view :: Natural -> Natural'
view = ...

view' :: Natural' -> Natural
```

```
view' = ...

replicate :: Natural -> a -> [a]
replicate (view -> Zero)   _ = []
replicate (view -> Succ n) x =
  x : replicate (view' n) x
```

The problem is that GHC's views do not really provide another way of looking at the same type; they just apply a casting function inside a pattern. So, the recursive call in `replicate` has to convert the view type back into the original type, and in doing so it is no longer structurally recursive.

Haskell does provide a compromise solution to this problem, known as "`n+k` patterns." [9] Such a pattern takes the form `n+k`, where `n` is of an `Integral` type and `k` is a non-negative integer literal. In GHC's desugaring phase, these patterns are converted to code like this:

```
-- foo (n + k) = bar
foo n' =
  if n' >= k then let n = n' - k in bar else NEXT
```

Of course, these patterns do not look or act like data constructor patterns internally, so my structural recursion analysis does have to treat these patterns as a special case. The details of this, including some other problems which arise, can be found in Chapter 3.

4.2 Simple Changes to Haskell 98 Functions

Many Haskell library functions are already total and require no changes at all to be included in the total library. Most of the remaining functions only require simple changes, and these changes generally fall into a few categories.

The first category consists of functions which, although possibly total, are made somewhat useless by the fact that they use integers instead of natural

numbers. These functions include `length`, `replicate`, and the list indexing operator, `(!!)`. [15]

Another category consists of functions which raise run-time errors when given bad input. These functions have been rewritten so that where they once returned a value of type `a`, they now return a value of type `Maybe a`. Functions in this category include `read`, `(!!)`, `toEnum`, `pred`, `succ`, `quotRem`, `divMod`, `digitToInt`, and `intToDigit`. [15] [8]

One simple but exceedingly useful addition I made to the library is the operator `(!!>)`:

```
infixl 9 !!>  -- same fixity as !!
(!!>) :: Maybe [a] -> Nat -> Maybe a
(!!>) xs i = xs >>= (!! i)
```

This operator takes advantage of `Maybe` as a monad to index a `Maybe [a]`, providing a result of type `Maybe a`. This is useful when indexing into a multi-dimensional list:

```
mat = [[1, 2, 3], [4, 5, 6]]
foo = mat !! 1 !!> 2          -- Just 6
bar = mat !! 2 !!> 1          -- Nothing
```

You might have noticed that `head` and `tail` were left out of the list of functions that return a `Maybe`. This is not an accident. Although such functions could be written, `head` would be equivalent to `listToMaybe`, and `tail` would probably not be as useful as `drop 1`. An even more extreme case along these lines is the function `fromJust`, which when transformed to return a `Maybe` becomes the identity function.

4.3 Changes to Haskell 98 Classes

Due to limitations which are explained in Chapter 3, my implementation will not consider valid any typeclass instance which relies on default method implementations or which has method definitions that rely on each other. Therefore,

it is necessary to subtly change many of the standard typeclasses. Here, for example, is my reimplementation of the `Eq` class:

```
class Eq a where
  (==) :: a -> a -> Bool


(/=) :: (Eq a) => a -> a -> Bool
(/=) a b = not (a == b)
```

Unlike the standard implementation [15], the "not-equal-to" function is no longer a method, but rather a normal function with a class constraint in its type. This allows that function to have the same sane definition automatically which, under the standard definition, it has by default.

This change seems reasonable anyway, given that this entire exercise in total programming has the goal of increasing the strength of the rules enforced by the compiler. Where once it was simply recommended by the documentation that any instance of `Eq` should obey the identity `(x == y) == not (x /= y)`, that identity is now rigidly enforced.

Other classes which were changed in this manner include `Ord`, `Enum`, `Show`, `Read`, and `Monad`. [15]

4.4 The `Total.Data.CoList` Library

One of the fundamental codata types is the colist. It is defined as follows:

```
codata CoList a = Nil | a :* CoList a
```

You may have noticed that the colist type has a `Nil` constructor, which means that a colist can be finite. Since colists are used heavily by the total IO library, and any input stream could eventually come to an end, this makes sense. There are situations where infinite-only colists are more useful, such as when modeling numerical sequences. Hinze has demonstrated this use of infinite streams. [17]

It would be ideal to have both infinite and potentially-finite colist types. Unfortunately, with the rule for productive corecursion which I have implemented, it is impossible to generalize the colist functions to polymorphically operate on either type. This is because a polymorphic cofunction cannot possibly refer to a coconstructor by name, since the particular coconstructor is not known, yet a direct reference to the coconstructor is required by the rule. As such, implementing a complete library for both types of colists would be extremely tedious.

The colist library includes many functions, which can be divided into three categories: functions which operate on both lists and colists, functions which are identical to ones from `Data.List` but for their type, and entirely new functions.

4.4.1 Generalizing Lists and Colists

Some list functions produce a finite result even when their input is infinite. Such functions can be generalized over lists and colists. I've created a typeclass for this purpose, `Cons`:

```
class Cons c where
  nil    :: c a
  cons   :: a -> c a -> c a
  uncons :: c a -> Maybe (a, c a)
```

Here is one interesting example of a generalized list function, `splitAt`, which combines the functionality of `take` and `drop` [15]. Notice that the second part of the list is returned as the same type as the input, `c a`, which could be `CoList a`. This polymorphic function can return a colist because that colist is not built up recursively: it already exists as a substructure of the input.

```
splitAt :: (Cons c) => Nat -> c a -> ([a], c a)
splitAt n xs = f n xs []
  where
    f (n + 1) xs acc = case uncons xs of
```

```
      Nothing -> (acc, nil)
      Just (x, xs') -> f (n :: Nat) xs' (x : acc)
  f _ xs acc = (reverse acc, xs)
```

The `take` and `drop` functions are both implemented in terms of `splitAt`. Curiously, this breaks the old symmetry between them, in that they no longer have the same type:

```
take :: (Cons c) => Nat -> c a -> [a]
take n = fst . splitAt n

drop :: (Cons c) => Nat -> c a -> c a
drop n = snd . splitAt n
```

Also generalized across both list types are the indexing operators mentioned earlier, `(!!)` and `(!!>)`.

4.4.2 Functions Borrowed from `Data.List`

Many functions in the standard `Data.List` [12] are applicable to colists as-is, except for their constructors and destructors they use. Some of these functions are not exposed for lists by `Total.Data.List` because of their potential to produce infinite output from finite input. Others are available in both list and colist forms.

I followed the naming convention that functions which are available separately for lists and colists should have `co` prefixed to their colist forms, while functions that are only available for colists should simply have their original names.

The colist functions in this category include `coMap`, `coMapAccum`, `coZip`, `iterate`, `repeat`, `cycle`, and `coTakeWhile`.

Another function in this category is `coScan`, which is analogous to `scanl`. It is not named `coScanl` because there is no matching `coScanr`, for reasons I will discuss a little later.

4.4.3 New Functions

Some of the functions in the colist library are more novel: either they are adaptations of standard Haskell functions which had to be changed in some significant way, or they simply have no standard counterpart.

To understand why many of the functions in this category differ from their standard counterparts, consider this direct translation of `filter` to colists:

```
coFilter :: (a -> Bool) -> CoList a -> CoList a
coFilter _ Nil       = Nil
coFilter p (x :* xs) =
  if p x then x :* coFilter p xs else coFilter p xs
```

This function is invalid because the second recursive reference to `coFilter` is not wrapped in a coconstructor. Generally speaking, any cofunction which may skip an infinitely large number of inputs is invalid. However, the concept of filtering remains useful. One alternative is to leave a placeholder whenever an element of the input is dropped. Thus, we end up with a `coFilter` with this type:

```
coFilter :: (a -> Bool) -> CoList a -> CoList (Maybe a)
```

Other functions which have been transformed in this way are `coPartition`, `coNub`, and `coDelete`. This common idiom also suggests another useful library function, `coTakeWhileJust`:

```
coTakeWhileJust :: CoList (Maybe a) -> CoList a
coTakeWhileJust (Just x :* xs) = x :* coTakeWhileJust xs
coTakeWhileJust _              = Nil
```

One final function in this category, suggested by Hinze, is `interleave`. [17] This function alternates between two colists, taking an element from one and then from the other. Hinze gives many clever definitions of numeric sequences using `interleave`, although it is not clear that these definitions have any advantage over another style which relies more on `coZipWith`.

4.4.4 Missing Functions

Finally, it is worth mentioning some functions which are not in the colist library because they are impossible. Two obvious ones are `foldl` and `foldr` [15]. Indeed, any fold over a colist is impossible because a fold can produce a single, finite value, which would not be known until the entire colist has been traversed.

Related to the folds are the scans, `scanl` and `scanr` [15], which are like the folds but produce a list of every intermediate result. As mentioned earlier, it is possible to implement `scanl` but not `scanr`. To understand why, observe the functions in this example:

```
foo  = [1, 2, 3]
sum  = foldr (+) 0 foo  -- 1+(2+(3+(0)))
sums = scanr (+) 0 foo  -- [0, 3+0, 2+3+0, 1+2+3+0]
prd  = foldl (*) 1 foo  -- (((1)*1)*2)*3
prds = scanl (*) 1 foo  -- [1, 1*1, 1*1*2, 1*1*2*3]
```

Notice that both `sums` and `prds` begin with the innermost computation, but in the case of `sums`, which used `scanr`, the innermost computation involves the last element of the list. This is why `scanr` cannot be made to operate on colists: it would have to reach the end of the colist before producing a single result value.

Another notable omission is `coConcat`. Actually, there is a function by that name, but it doesn't have the type we really wish it did:

```
-- impossible
coConcat :: CoList (CoList a) -> CoList a


-- possible
coConcat :: (Cons c) => [c a] -> CoList a
```

The first type for `coConcat` is impossible because the input could be an infinite colist of `Nil` colists. Of course, it's not the presence of the `Nil` constructor that is the problem: although getting rid of `Nil` would allow us to write `coConcat`, it would be equivalent to `take 1`!

Interestingly, `coSplitEvery`, which feels like an opposite to `coConcat`, is not only possible, it has proven useful.

4.5 The `Total.Data.Map` Library

The `Total.Data.Map` library is a thin wrapper around GHC's `Data.Map` [3]. One change made to the interface is the type of `(!)`, the indexing operator, which returns `Maybe a` instead of `a`. Along with this operator is `(!>)`, which allows easy indexing into nested maps in the same way that `(!!>)` allows easy indexing into nested lists.

The `Map` type implemented by GHC is abstract, so there are no data constructors available with which to use a `Map` as the induction variable of a recursive function.

Finally, it should be noted that `Data.Map` exposes a number of functions which have unchecked preconditions. If used improperly, these functions can produce an invalid `Map`. Such a `Map` will not produce run-time errors, but most of the library's functions are unlikely to work correctly on an invalid map. These dangerous functions are provided for performance reasons.

Consider this example:

```
import Data.Map

s, s' :: Map String Bool
s  = fromList [("03", True), ("01", False), ("14", True)]
s' = fromList [("3", True),  ("1", False),  ("14", True)]

n, n' :: Map Int Bool
n  = mapKeysMonotonic read s
n' = mapKeysMonotonic read s'
```

The function `mapKeysMonotonic` allows the keys of a map to be transformed but requires that the transformation not change the ordering of the keys. In the

example, `n` is valid and `n'` is invalid.

Whether or not the dangerous functions should be included in the interface is an interesting question. They don't break totality, but do they violate the spirit and goals of the system? One possibility is to check the validity of the map after each dangerous operation, but this may thwart the performance benefits of some of the dangerous operations.

4.6 The `Total.Data.Array` Library

The `Total.Data.Array` library is a wrapper around GHC's `Data.Array` [2]. Substantial modifications to the API were necessary for the library to be total. The `Array` type defined in `Data.Array` is extremely flexible. It has two type parameters: the index type and the element type. The index type must be an instance of class `Ix`, which describes how to map those indices onto integers. An array is created with `listArray`, which takes two parameters: the bounds of the array and the initial elements. If the bounds are greater than the number of elements given, some parts of the array will be undefined.

Having undefined array elements is not an acceptable situation in the total world, so one change made to the API is that the `listArray` function does not allow bounds to be specified: they are calculated from the given list. Another change is that the indexing operator, `(!)`, returns `Maybe a`, and there is a corresponding `(!>)` operator for indexing nested arrays.

4.7 The I/O Libraries

The design of a total I/O library is a complicated and important task. As such, I will discuss the design process in stages, leading up to the actual implementation, with alternatives and notes along the way.

4.7.1 Simple I/O: Standard Input and Output

The problem of input and output in a total functional system is that either one might reasonably be unbounded. We do, however, have a construct for such situations: codata. In particular, a colist seems to be good choice for the description of a potentially-infinite input or output stream. A simple system might then say that the type of main should be:

```
main :: CoList Char -> CoList Char
```

However, a result type of `CoList Char` is not ideal in all situations, as such a program may be non-terminating, which weakens the guarantee of TFP. One appealing alternative is to mandate that the size of the output be $O(x)$ where x is the size of the input. This could be accomplished by having the programmer specify the main function in two parts:

```
init :: a
loop :: a -> Char -> (a, [Char])
```

These two parts would be used by a built-in main function:

```
main :: CoList Char -> CoList [Char]
main = snd . coMapAccum loop init
```

That is, the programmer's loop function would produce one finite string of characters for each input character. An accumulator value is threaded through applications of the loop function. Such a main function would allow the total system to make the fairly simple guarantee that if the program's input is finite, so will its output be.

Another alternative is to simply require that all programs be finite by typing main as:

```
main :: CoList Char -> [Char]
```

But, for the purposes of experimentation, it is not actually necessary to choose any of these alternatives. Rather, we can enable all of them by leaving the type of main unchanged from standard Haskell (`IO ()`) [7], but restricting

the available functions which use the IO monad to these three, by which the programmer may select any of the paradigms described above:

```
finiteMain    ::
  (CoList Char -> [Char]) -> IO ()

boundedMain   ::
  (a -> Char -> (a, [Char])) -> a -> IO ()

unboundedMain ::
  (CoList Char -> CoList Char) -> IO ()
```

4.7.2 I/O with Files: Two Approaches

Our actual implementation is further complicated, however, by our desire to read from and write to the filesystem. How do we maintain referential transparency while allowing the same file to be read more than once? One simple answer is that every file that is read should be cached in memory by the I/O library, so that if a file is read a second time, it can be returned from the cache, and will thus be the same value that was returned the first time. But one goal of this work is practicality, and such caching is not generally practical.

An alternative is to ensure that the function which reads a file can never be applied with exactly the same arguments twice. Languages including Clean [23] and Mercury [4] implement this in a user-visible way with uniqueness types, which means that the program threads a value representing "the outside world" through all I/O function calls.

Standard Haskell also does this, in theory, but the world value is made inaccessible to the programmer. Instead, the threading of the world value is done implicitly when two `IO` computations are sequenced with the monadic bind operator [7]. Since a Haskell program must ultimately produce a single `IO` computation, and since multiple `IO` computations can only become one via the monadic bind operator, the world value is necessarily threaded through all

`IO` computations. Haskell's approach seems to be simpler and more convenient for the programmer. For this reason, and because our TFP system is based on Haskell, we have taken this approach.

I/O Error Handling

Another question regards the handling of errors. An obvious choice is to say that all I/O operations which can fail return some value which indicates that possibility, like a `Maybe a` or `Either Error a`. Since nearly every I/O operation can fail, this means that a sequence of I/O operations requires a lot of error-checking code, which may be highly repetitive. In the common case, in which each I/O operation in a sequence depends on the success of the previous operations, it is useful to have an abstraction which simply aborts the entire sequence of I/O operations at the first sign of failure.

One simple way we can imagine accomplishing this in Haskell is through the `Maybe` monad. A more advanced approach which allows information about an error to be seen by the program, is to create an instance of monad for `Either IOError`. In our case, since we are already using the `IO` monad, we could combine the two monads by using monad transformer techniques.

Standard Haskell takes a similar approach but avoids the complexity of monad transformers by encapsulating the alternative possibility of an error inside the `IO` monad. So, the `IO` monad can be understood as something like a type `(IOWorld, Either IOError a)`. When two `IO` computations are sequenced, if the first one produced an error, the second one is skipped, just as when a `Nothing` is sequenced with a `Just` in the `Maybe` monad. In order to allow the programmer to handle errors which would otherwise be trapped inside the necessarily opaque `IO` monad, functions such as `try` are supplied, which we will define in terms of the previously-given definition of the `IO` type:

```
type IO a = (IOWorld, Either IOError a)

try :: IO a -> IO (Either IOError a)
```

```
try (w, Left err) = (w, Right (Left err))
try (w, Right a)  = (w, Right (Right a))
```

In either case, `try` returns an `IO` value that is not in an erroneous state. However, in the case that an error had occurred, the result value is information about the error, rather than a value of the normal result type of the computation.

Given this understanding of error handling in I/O, it is clear that `try` is total and thus may be exposed by an I/O library.

4.7.3 Three Practical Paradigms

How does this I/O abstraction relate to the three simple I/O paradigms discussed earlier? Consider first the simplest paradigm, unbounded I/O. Recall that the type of the main function would have been:

```
main :: CoList Char -> CoList Char
```

Now, using sequenced `IO` values, we would instead have input and output functions like:

```
readFile  :: Handle -> IO (CoList Char)
writeFile :: Handle -> CoList Char -> IO ()
```

So, that paradigm survives undisturbed. What about the bounded I/O paradigm, where the output size is limited by the input size? Recall that the type of the main function would have been:

```
main :: CoList Char -> CoList [Char]
```

We could instead have functions like:

```
mapAccumFile ::
  Handle -> (a -> Char -> (a, IO ())) -> a -> IO ()


writeFile    ::
  Handle -> [Char] -> IO ()
```

Notice that the function which produces output, `writeFile`, only produces finite output. However, the `mapAccumFile` function would apply the given function once per character in the file, a potentially unbounded quantity, enabling the program to produce output proportional to the input. A wrapper around `withFile` which allows the callback to be invoked once per line instead of once per character would be quite useful in this paradigm.

Finally, in the simplest paradigm, finite I/O, we would have had a main function like:

```
main :: CoList Char -> [Char]
```

In the IO monad, we would instead have:

```
readFile  :: Handle -> IO (CoList Char)
writeFile :: Handle -> [Char] -> IO ()
```

In all three cases, note that the `IO` type itself is not codata.

4.7.4 Implementing `Total.IO`

Since we have taken the standard Haskell approach to I/O, it makes sense to use the standard Haskell I/O library. Of course, some parts of the library would allow the programmer to circumvent the rules of TFP, and these must not be exposed. Some wrapping must also be put around the library to ensure that colists are returned where appropriate.

Separating the Paradigms

To implement all three I/O paradigms requires a new `IO` type:

```
data IO p a
```

The parameter `p` is the paradigm; the parameter `a` is the result type. A monad has only one type parameter; thus, the I/O monad instance looks like:

```
instance Monad (IO p) where ...
```

This means that the type of the monadic bind for `IO` is:

```
IO p a -> (a -> IO p b) -> IO p b
```

Notice that the paradigm type cannot change as a result of sequencing I/O computations. This is the fundamental mechanism enforcing the separation of the paradigms. Each I/O function can include in its type a constraint on the paradigm. For most I/O functions, this is not as simple as specifying the exact value of `p`, because the paradigms can be ordered by their restrictiveness, and a function which is safe in one paradigm is also safe in all less-restrictive paradigms.

The codification of this ordering consists of three types, representing the paradigms, and three classes, representing a set of paradigms and named after the most restrictive member of the set:

Class	`Finite`	`Bounded`	`Unbounded`
`IOFinite`	x	x	x
`IOBounded`		x	x
`IOUnbounded`			x

GHC's run-time system expects the user's `main` function to have the type `System.IO.IO a`. The goal of the three-paradigm design is to allow flexibility in writing the program, while still making the termination properties of the program as clear as possible. To this end, the implementation forces the programmer to visibly declare which paradigm is being used. Thus, rather than providing a single function of type `IO p a -> System.IO.IO a`, three functions are provided:

```
finite    :: IO Finite    a -> System.IO.IO a
bounded   :: IO Bounded   a -> System.IO.IO a
unbounded :: IO Unbounded a -> System.IO.IO a
```

There is just one problem with these types: `System.IO.IO` is a monad, so the programmer could, in that monad, sequence together I/O computations which had been brought out of differing paradigms. This runs counter to the previously-stated design goal. As such, those three functions actually return values of type `AnyIO`, a new type which is not a monad. The compiler then

provides an implicit wrapper around the user's main function of type `AnyIO a -> System.IO.IO a`.

The Finite Paradigm

In the finite paradigm, the contents of any handle can be obtained as a costring, but any output is always a string. It doesn't matter that the costring input might be infinite, because the process of producing a finite output can only possibly examine a finite prefix of an infinite input.

There are many traditional I/O schemes which suffer under this paradigm. A common scheme is to read one line of input at a time, because this is convenient for a human operator interacting with the program through a terminal. Without setting a maximum on the length of an input line, there is no way to simply process an entire line, since the line might be infinite.

There are two important ways of circumventing this problem provided by my finite I/O library. The first takes the suggestion mentioned in the previous paragraph:

```
hGetLineWithMax :: (IOFinite p)
                => Handle -> Nat -> IO p String
```

The second approach is more elaborate. It is based on this premise: if a file has an end, it is finite. This premise is not exactly true: one can imagine ways in which it might be circumvented. But, in practice, it is almost certainly true. Thus, the I/O library includes a family of functions which first check that a file has an end. They do this by seeking to the end of the file and then going back. If the seek fails, no attempt is made to read the contents of the file and an error is raised. This check is encapsulated by a single function:

```
finitely :: (IOFinite p)
         => (Handle -> IO p a) -> Handle -> IO p a
```

Seeking back and forth in terminal input is not possible. So, when standard input is connected to a terminal, as is usually the case, any attempt to use a

finite I/O function on standard input will produce an I/O error. This means that it is much easier to write a program in the finite paradigm if it reads from a file on disk. This can be seen in Chapter 5.

The Bounded Paradigm

The idea of the bounded paradigm is that the program's output may be infinite only if the input is infinite. This is ensured by processing the input in chunks: output is the result of processing a finite chunk of input, of a length greater than zero. To get infinite output, there must be an infinite number of chunks, and therefore, since each input chunk is positive in size, there must be infinite input.

The fundamental function in the bounded I/O library uses a chunk size of one character. The key aspect of this function, other than the properties already described, is that it threads an accumulator value through all applications of the chunk processing function:

```
hMapAccum
  :: (IOBounded p)
  => Handle -> (a -> Char -> (a, IO p ())) -> a
  -> IO p ()
```

It would be possible to provide a slightly more complicated variant of this function which allows the chunk processor to return a result value which is collected into a colist:

```
hMapAccum'
  :: (IOBounded p)
  => Handle -> (a -> Char -> (a, IO p b)) -> a
  -> IO p (CoList b)
```

Since there might be an infinite number of chunks, the results must be collected into a colist, not a list. Furthermore, under no circumstances can the final value of the accumulator be returned, since that single value could require the processing of an infinite number of chunks to compute.

Other bounded I/O functions, much more useful in practice, are derived from `hMapAccum`, such as `hMapAccumLines`, which uses the accumulator to buffer characters until a newline is found.

Many common categories of program should fit within the bounded paradigm. Any event-driven program, such as a typical operating system or graphical word-processor, should respond to each event in a finite amount of time. Otherwise, the system is hung. Such a system may run indefinitely because it continues to receive new events.

The Unbounded Paradigm

In the unbounded paradigm, the program may produce infinite output under any circumstances. The fundamental function of this paradigm is:

```
coWriteFile :: (IOUnbounded p)
            => FilePath -> CoString -> IO p ()
```

It is questionable whether this paradigm is necessary or within the spirit of total programming. I found no need for it in writing the example programs which are described in Chapter 5.

4.7.5 Filesystem Navigation

Navigating the filesystem is important for both systems programming and ad-hoc scripting. My implementation includes a filesystem navigation library which provides a significant abstraction over the underlying syscalls. It provides a view of the entire file heirarchy, from a given root directory, as a tree structure:

```
data File p =
    RegularFile  (IO p Handle)
  | Directory    (DirTree p)
  | SymbolicLink (IO p (File p))
  | SpecialFile
```

```
type DirTree p = [(FilePath, File p)]
```

This structure has many interesting features. First, note that for the structure to be finite, symbolic links cannot be expanded directly into subtrees. However, for convenience, a symbolic link is presented as the I/O action which follows that link. Similarly, a regular file is presented as the I/O action which opens that file.

For my purposes, the finite nature of this structure is critical. But it must also be noted that without lazy evaluation, a structure like this would be tremendously inefficient in practice when used at a high point in the filesystem heirarchy.

This library does make the assumption that filesystems are not malformed or malicious, in that they do not contain any loops of hard-linked directories.

Chapter 5

The Examples

In deciding what examples to implement, I tried to cover a wide variety of algorithms. Yet, in the end, none of the examples demonstrate an interesting application of codata. This could be a flaw in the set of examples, or it could indicate that codata, as implemented, are not frequently useful.

5.1 Calculator

The calculator program demonstrates the use of monadic parser combinators [18] [25] and a recursive interpreter. It accepts a simple language for performing integer arithmetic. The program runs in the bounded I/O paradigm, parsing and interpreting one line at a time with `hMapAccumLines`. The accumulator contains the variable store that the input program may update with assignment statements.

To demonstrate this use of bounded I/O, here is the `main` function of this example, which reads input lines, passes them along to be parsed and executed, and prints the output, while threading the variable store along from one line to the next.

```
main = bounded $ hMapAccumLines stdin f []
  where
```

```
    f acc xs = (fst r, putStrLn $ out $ snd r)
      where
        r = exec (pread readsTop xs) acc
```

The language of one line of the calculator's input is:

```
stmt   ::= assign | expr
assign ::= VAR '=' expr
expr   ::= factor (sum_op factor)*
sum_op ::= '+' | '-'
factor ::= negate (fac_op negate)*
fac_op ::= '*' | '/'
negate ::= '-'* power
power  ::= item ('^' item)*
item   ::= INT | VAR | '(' expr ')'
```

The implementation of this grammar uses a parser type which is both a `Monad`, giving a sequencing operator, and a `MonadPlus`, giving an alternation operator. This monad relies internally on the `Either String` monad to record errors:

```
newtype Parser a = P (String -> Either String (a, String))

instance Monad Parser where
  (P r) >>= f = P (\xs -> do
    (a, xs') <- r xs
    parse (f a) xs')
  return a    = P (\xs -> return (a, xs))
  fail err    = P (\_  -> fail err)

instance MonadPlus Parser where
  mzero             = P (\_  -> Left "mzero")
  mplus (P f) (P g) = P (\xs -> case f xs of
```

```
    Left err    -> g xs
    a@(Right _) -> a)
```

One additional combinator, other than `>>=` and `mplus`, is `many`, which parses zero or more repetitions of a production. The recursive implementation of `many` requires an induction variable. The opaque parser being repeated is of no use in this regard, so the induction must be done on the length of the input. This is not an especially tight bound, unless the parser being repeated consumes exactly one token, but it works.

The weakness of this approach is that, although total programming purports to catch non-terminating functions at compile-time, a non-terminating grammar is only caught at run-time.

```
inputSize :: Parser Nat
inputSize = P (\xs -> return (length xs, xs))

many :: Parser a -> Parser [a]
many p = inputSize >>= fm p

fm :: Parser a -> Nat -> Parser [a]
fm p (n + 1) = (do
  a  <- p
  as <- fm p n
  return (a : as)) `mplus` return []
fm _ _ = return []
```

The `many` combinator is at least convenient in that the programmer does not need to reimplement the induction process when parsing a repetition.

When implementing the expression grammar, however, we find another, more complicated instance of recursion: a large mutually-recursive loop which reaches from `expr` through `factor`, `negate`, `power`, and `item` before cycling.

The induction is based on the length of the input. This means threading an induction variable through all of these parsers, but the mutual recursion

rule which I have implemented makes this less painful than one might expect. The rule only requires one function in the loop to make its recursive calls on a "smaller" value than its input. In this example, the production `item ::= '(' expr ')'` is the obvious choice: it is simple because all recursive loops ultimately pass through this one point, and it is obviously correct, in that this production always consumes at least two characters, the two parentheses.

In fact, the mutual recursion rule is not only helpful, but necessary: there are productions in the loop, such as `expr ::= factor (sum_op factor)*`, which do not necessarily consume any input before recursing.

5.1.1 Making the Induction More Exact

The best induction variable which is obviously feasible to implement is one which, although it is usually an overestimate, is exact for some inputs. In this example, such induction could be accomplished by subtracting from the induction variable in each production the minimum amount of input that production can consume. However, there are problems with this approach. Recall the production for expressions, as well as the same production expanded to avoid the use of `*`:

```
expr  ::= factor (sum_op factor)*


expr' ::= factor rest
rest  ::= sum_op expr' | <empty>
```

To add an exact induction variable to the latter set of productions is fairly straightforward, albeit invasive:

```
expr'(n) ::= factor rest(n-1)
rest(n)  ::= sum_op expr'(n-1) | <empty>
```

In recursing from `expr'` to `factor`, no input has been consumed, so n is passed unchanged. In recursing to `rest`, it is known that parsing `factor` has consumed at least one token, so $n-1$ is passed to `rest`. In its first alternative,

`rest` recurses on `expr'` after parsing `sum_op`, which consumes at least one token, so $n-1$ is passed to `expr'`, relative to the input to `rest`, which is $n-2$ relative to the input to the outer application of `expr'`.

Suppose an induction variable were added to `expr` in the same manner. Then the production might look like this:

```
expr(n) ::= factor(n) (sum_op factor(?))*
```

Notice the question mark. The exact value of the induction variable which should be passed to `factor` varies depending on how many `sum_op factor` sequences have already been parsed. But the value which is, in fact, passed to `factor` must be obtained by pattern-matching deconstruction on `n`, and thus the value must be `n-k` for some `k` which is known at compile-time. So, this approach to adding an exact induction variable does not work with uses of `*`.

5.1.2 Alternate Approaches to Parsing

Automaton-based parsers may be more attractive in total programming than recursive-descent parsers because the algorithm which runs on a particular input reliably consumes one character at a time, allowing for an exact induction variable rather than an overestimate. Although some steps of the process of creating the automaton may rely on overestimated induction variables, at least these steps are arguably a part of compilation rather than execution.

5.2 Regular Expressions

The regular expressions example demonstrates the value of automaton-based parsers in total programming. The example is a collection of four algorithms:

- Converting a regular expression to an NFA with λ-transitions
- Converting an NFA-λ to an NFA
- Converting an NFA to a DFA

- Executing a DFA on an input string

The last algorithm, executing a DFA on an input string, is the only one which is run for every input. The other algorithms are run only once for a particular regular expression. The last algorithm uses induction on the structure of the input string, and this induction variable is not an overestimate.

This is the DFA execution code:

```
type Trans = Map Char Nat
newtype DFA = DFA (Map Nat (Trans, Bool))

dfaAccept :: DFA -> Nat -> Bool
dfaAccept (DFA m) nat = case m ! nat of
  Nothing     -> False
  Just (_, r) -> r

dfaTrans :: DFA -> Nat -> Char -> Maybe Nat
dfaTrans (DFA m) nat x = case m ! nat of
  Nothing     -> Nothing
  Just (m, _) -> m ! x

run :: DFA -> [Char] -> Bool
run dfa = f 0
  where
    f cur [] = dfaAccept dfa cur
    f cur (x : xs) = case dfaTrans dfa cur x of
      Nothing   -> False
      Just next -> f next xs
```

The first algorithm uses induction on the structure of the regular expression. This induction variable is also exact.

The second algorithm takes the λ-closure of each state. At the top level, it uses induction on the list of states, which is exact. For each state, the λ-closure

algorithm uses induction on the number of states, which is inexact because each step adds *at least* one state to the closure, but possibly more. This algorithm does not combine indistinguishable states.

The third algorithm uses the subset construction to produce a DFA. Its implementation produces only those sets of states which are reachable from the start state. However, it uses induction on the number of subsets, so the induction variable is typically a gross overestimate.

Unlike the calculator example, however, both of these algorithms are at least using an induction variable which is an exact bound for some inputs.

5.3 Sudoku Solver

This example demonstrates a total backtracking algorithm which solves Sudoku puzzles [5]. Its input consists of one line per row of the puzzle, with columns separated by whitespace, and empty columns filled in with any non-numeric token. It reads this input from a file, making it trivially easy to parse with the function:

```
map (map read . words) . lines
```

This is actually even simpler than in standard Haskell, because `read` returns `Nothing` instead of causing a run-time error when it fails [15].

Once parsed, the puzzle is converted to an `Array (Array (Maybe Nat))`, to improve the cost of looking at all the cells in a particular column or box. The backtracking progresses recursively, using a list of all the empty cells in the grid as the induction variable.

All backtracking algorithms are fundamentally the same, but for a few details. In standard Haskell, the functional approach to backtracking is captured nicely by this class:

```
class Puzzle a where
  moves  :: a -> [a]
  solved :: a -> Bool
```

```
solve :: (Puzzle a, MonadPlus m) => a -> m a
solve p =
  if solved p
     then return p
     else msum (map solve (moves p))
```

This definition relies on `MonadPlus` [26]. It allows `solve` to return one solution, with `m=Maybe`, or all solutions, with `m=[]`. Unfortunately, this traditional backtracking approach is not valid in TFP, because `solve` recurses on `moves p`, which is not a substructure of the input, `p`. Using the Sudoku example as a template suggests this alternative:

```
class Puzzle a where
  steps  :: a -> [a -> [a]]
  solved :: a -> Bool

solve :: (Puzzle a, MonadPlus m) => a -> m a
solve p = f (steps p) p
  where
    f _      p
      | solved p = return p
    f []     _    = mzero
    f (x:xs) p    = msum (map (f xs) (x p))
```

Here, as in the Sudoku example, the steps which need to be taken to reach a solution are known. In Sudoku, the first step is to fill in the upper-left square, then the square to its right, and so on.

Note that `steps` is only called once, at the beginning of the solving process. Indeed, `steps` is only intended to be called on the initial state of the puzzle. It would be better to use a separate type for the initial state of the puzzle, in order to clarify and enforce this distinction, but it would then be necessary for `Puzzle` to have two parameters, which standard Haskell does not allow.

The recursion is then performed on the list returned by `steps`. So, the maximum depth of the backtracking is a finite constant known before the recursion begins. This is a significant limitation on what algorithms may be implemented.

Now, here is a demonstration of what a Sudoku solver implemented under this type class would look like, with some details omitted:

```
data Sudoku = ...

-- Length of one side.
size :: Sudoku -> Nat

-- Find all numbers that can't be put at a location.
conflicts :: Sudoku -> Nat -> Nat -> [Nat]

-- Put a number at a location.
fill :: Sudoku -> Nat -> Nat -> Nat -> Sudoku



instance Puzzle Sudoku where
  steps p = do
    i <- [1 .. size p]
    j <- [1 .. size p]
    return (makeStep i j)
  solved p = ...

-- Put all possible numbers at a location.
-- Note that \\ is set difference.
makeStep :: Nat -> Nat -> Sudoku -> [Sudoku]
makeStep i j p =
  map (fill p i j)
```

```
([1 .. size p] \\ conflicts p i j)
```

In Sudoku particularly, a solution can only be reached after all the steps are taken. However, to remain comparable in power to the traditional `Puzzle` class, the total one here does not assume that all steps must be taken to reach a solution.

In other backtracking problems, it may be less obvious what the steps are, but it may be simple to generate the moves as a tree, in which case a depth-first search of the tree, which performs induction on the structure of the tree, can obviously be written.

5.4 A* Search

This example uses the A* search algorithm to roll a glued-together pair of six-sided dice around in a rectangular grid, avoiding obstacles while reaching a target destination and orientation. The heuristic used is the Manhattan distance from the current position to the target. [6]

The example uses induction on the number of states. This is a reasonable approach, even though it is generally a large overestimate, because the states are so interconnected there will be few unreachable states at any point in the algorithm.

5.5 Longest Common Subsequence

This example demonstrates a common problem which has a dynamic programming solution. Many dynamic programming algorithms can be modeled as filling in the cells of a table, with each cell having dependencies on some cells which were filled in previously. Some final cell contains the solution. The longest common subsequence problem is no exception.

The implementation uses a pair of induction variables: the two input strings. As the algorithm builds up one row of the two-dimensional table, it takes char-

acters from one of the strings. When it begins a new row, it starts over on that string while taking one character from the other string.

The implementation of the algorithm was done without much consideration of generalizations for dynamic programming. However, the tediousness of building up a table cell by cell, and properly looking backward at the correct dependencies, suggests that a generalization would be welcome.

Here is the code for the core algorithm. Notice that the function `f` has three parameters grouped together into a tuple. This reduces the number of permutations considered by the recursion analysis, which improves compilation time.

```
-- each char of a is a row
-- each char of b is a col
table :: (Eq a)
      => [a] -> [a] -> Maybe (Array (Array Nat))
table a b =
  listArray $ f a b
    ( 1
    , catMaybes [listArray $ map (const 0) [0 .. bn]]
    , [0]
    )
  where
    bn = length b
    -- End of the last row
    f (_ : []) [] (_, rows, cols) =
      listArray cols `mCons` rows
    -- End of a row other than the last
    f (_ : as') [] (_, rows, cols) =
    f as' b (1, (listArray cols `mCons` rows), [0])
    -- Any other cell
    f as@(ca : as') (cb : bs')
```

```
    (i, rows@(pr : _), cols@(pc : _))
    | ca == cb =
      f as bs' ((i + 1), rows, (prc + 1 : cols))
    | otherwise =
      f as bs' ((i + 1), rows, (max prc pc : cols))
    where
      fix  = maybe 0 id
      prc  = fix (pr ! (bn - i))
      prpc = fix (pr ! (bn - (i - 1)))
f _ _ (_, rows, cols) =
  listArray cols `mCons` rows --[]
```

One possible generalization would accept a directed acyclic graph as input, with an edge being a dependency. This generalization might be excessive and cumbersome in the many cases where the dependencies can be represented as static positional relationships within a table.

A table-based generalization would ideally work with any number of dimensions. To accomplish this within Haskell's type system and total programming, the table could be represented as a `Map [Nat] a`, where the table cells are identified by position. Induction across the table would use two variables: a `Nat` giving the position in one dimension and a `[Nat]` giving the size of all remaining dimensions. Another parameter of this recursive function would give the positions currently fixed in all previous dimensions.

5.6 Robot Simulator

This example simulates a simple robot: it occupies a single point, has perfect motor control, isn't damaged by running into walls, and has three perfect sonars. The program repeatedly performs trigonometric calculations of the robot's sonar observations, which the robot uses to determine its course. An animation of the robot's path is output in SVG [16].

The user specifies how many steps to simulate. The main loop thus uses induction on the number of simulated steps. If this simulator were embedded into a more full-featured application, it might instead be under the control of an event loop, which would trigger one simulated step each time a timer event is received.

The mathematical algorithms in this example frequently use division. In the total Prelude, the division operator returns a `Maybe` to encode the possibility of division by zero. In this example, that implementation of division is not a burden, because division by zero indicates a special case, a vertical line, which must be handled differently.

5.7 File Finder

This example finds files whose names contain a given substring. Using the library described in Section 4.7.5, this example was trivial.

5.8 Text Searcher

This example searches files for a target string. It can also be told to search standard input. Because of this, it operates in the bounded I/O paradigm, processing each input line by line.

Chapter 6

Lessons Learned

6.1 Programming Techniques

6.1.1 More and Simpler Algorithms

Decomposing a problem into smaller parts has always been considered good advice, but in the world of total programming it is practically mandatory. The more the programmer tries to accomplish with a single algorithm, the more difficult it is to make a good inductive argument about that algorithm's termination.

This principle was observed in the regular expressions example, where producing a DFA from a regular expression was done in many simple steps, some of which have better inductive arguments than the program as a whole.

A variation on this principle is that, when the recursion can be separated from the details of the problem domain, that separation should be made. This is also good advice in general, but in total programming, abstracting away the recursion from the problem domain can also make an inductive argument for the recursion easier to find. The potential for this kind of abstraction was seen in both the backtracking Sudoku solver and the dynamic programming LCS solver.

Finally, decomposing the problem may improve the exactness of an induction variable even if it remains an estimate. In the calculator example, a recursive descent parser, the induction variable was the length of the input. If the parser had been operating on higher-level tokens previously output by a lexical analyzer, the induction variable would have been a better estimate.

6.1.2 Resorting to an Inexact Induction Variable

My initial assumption was that I would be dealing with two kinds of recursive algorithms, those with exact and inexact induction variables, but I would argue that there are actually three kinds, the first two of which are generally acceptable:

- algorithms which always terminate when their induction variables are at their base cases,
- algorithms which, for some inputs, terminate when their induction variables are at their base cases (and for all other inputs terminate earlier), and
- algorithms which always terminate before their induction variables reach their base cases.

Algorithms of the second kind feel more dubious in some cases than others. For example, the `takeWhile` function is in this category, yet we would likely not consider writing it any way besides the obvious. Also in this category is my NFA to DFA conversion by the subset construction. Here, using the number of subsets of the NFA's states as the induction variable has that dubious feel. This is perhaps because that number is frequently a gross overestimate, but I suspect that it is also because, if Turner's rule is taken away, the number is no longer a necessary parameter of the recursion.

This reveals an assumption which we are prone to make: that a total algorithm is only written in "the best way" if there are no parameters which would

be unnecessary in the presence of general recursion. The truth of this assumption is yet to be seen, but it seems reasonable, since adding parameters to an algorithm also creates new ways for the program to be written incorrectly.

Another assumption is that such an extra parameter is more acceptable if it is exact. Yet, whether the extra parameter is supposedly exact or not, it seems that the real danger lies in the fact that the parameter's initial value must be the result of some calculation which is separate from the algorithm yet must produce a value consistent with what the algorithm will do. So, does the fact that such a value is an overestimate add any additional danger? One danger is that the meaning and purpose of that calculation may be less obvious, which is problematic for the long-term maintenance of a program.

6.1.3 Relying on Lazy Evaluation

Turner mentions that an advantage of total programming is that the evaluation strategy used by the language implementation has no effect on the semantics of the language. This point is true, but not especially useful in practice, except in that it makes proofs about a program easier to concoct. Programs which are efficient under lazy evaluation may exhibit horrendous performance under strict evaluation, and vice-versa. Thus, a programmer needs to be able to rely on the presence of a particular evaluation strategy.

One possible benefit of this transparency is that both strategies could be implemented for a single language specification. Although a program would probably only be meant to run on one strategy or the other, this at least reduces the number of unique specifications which need to be published.

It must also be noted that the presence of codata demands some kind of lazy evaluation scheme, and if that scheme is going to be present for codata, it is difficult to see why it should not be the scheme of the implementation generally.

Also, even though total functions must terminate, they may not always do so in a reasonable amount of time. So, the performance implications of the evaluation strategy remain just as relevant for total programs as for traditional

programs.

6.2 Numeric Type Classes

The numeric type class heirarchy in Haskell is centered around integers. Clear evidence of this can be seen in the `Num` class, the root of the heirarchy, which has the method `fromInteger`. With such a heirarchy in place, some operations on natural numbers are likely to have senseless definitions.

Given the pervasiveness of natural numbers in computer programs, I would suggest that perhaps any language might be better off with a heirarchy focused on providing sensible operations for natural numbers.

The heirarchy as it is now is shown in Figure 6.1 [15]. An alternate heirarchy which would improve the standing of natural numbers is shown in Figure 6.2. It has the advantage that the existing numeric typeclasses are not significantly disturbed. Specifically:

- A standard class retains its conceptual meaning.
- Every parent class of a standard class is retained, although the inheritance might occur via new layers.
- Every type constrained by a standard class may still be constrained in the same way, although a more general constraint may have become possible.

The class for natural numbers is called `Natugral` by analogy to the word `Integral`, to avoid clashing with the `Natural` type which is presumed to exist.

Some changes would be made which this diagram does not reveal:

- `Positive` would have a method `fromNatural` which would be applied to "integer" literals instead of `fromInteger`. This wouldn't prevent "negative literals," because those are actually applications of the syntactically special unary minus, which is implemented by `negate` in the `Num` class. So, the inferred class constraints would differ, as they should, between positive and negative literals.

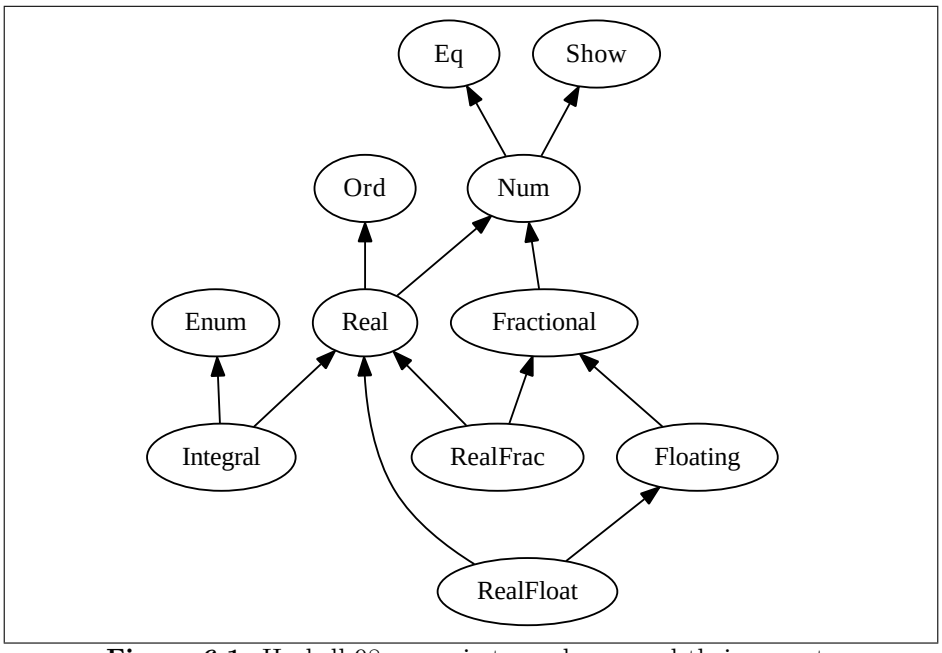

Figure 6.1: Haskell 98 numeric type classes and their parents

- `+`, `*`, and binary `-` would move from `Num` up to `Positive`.
- `quotRem`, `divMod`, their four derived methods, and `toInteger` would move from `Integer` up to `Natugral`.
- `toRational` would move from `Real` up to `RealPos`.

These changes would leave the `Integral` class without any methods. However, it still serves the purpose of uniting together the `Natugral` and `Real` classes. The same would happen to the `Real` class, which only serves to unite the `RealPos` and `Num` classes. Both the `Integral` and `Real` classes would still be valuable for their ability to simplify constraints on other functions.

6.3 Walther Recursion

Another form of recursion, a possible alternative to primitive recursion, is Walther recursion [20]. To allow Walther recursion requires an analysis which

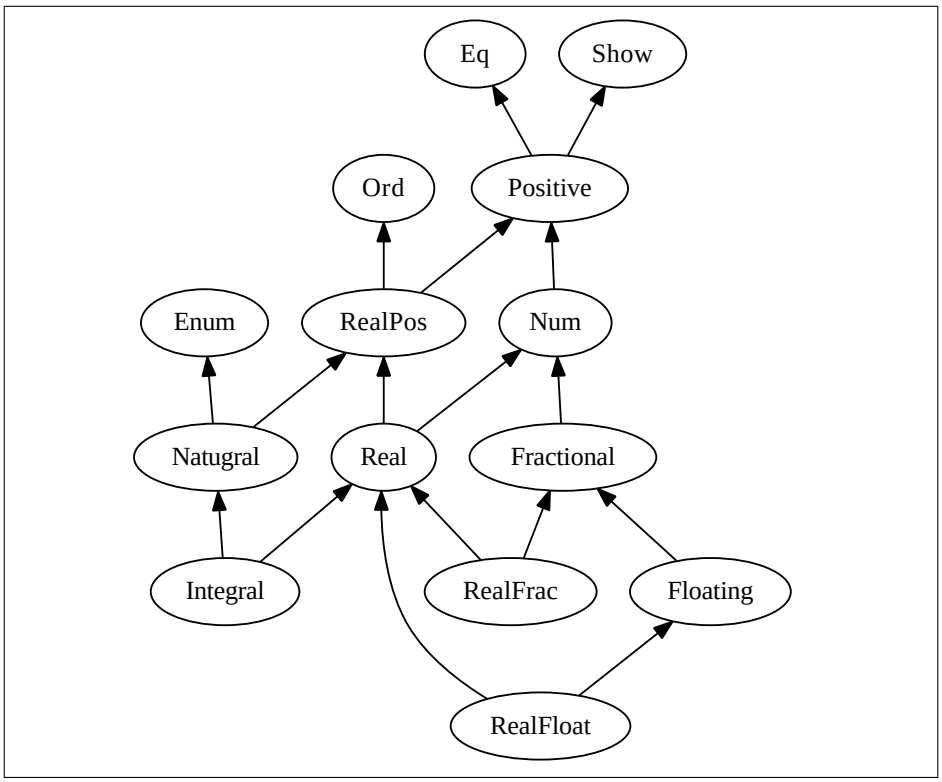

Figure 6.2: Proposed numeric type classes and their parents

creates and uses "reducer lemmas" such as $f(x_1, ..., x_n) < x_i$ and "conserver lemmas" such as $f(x_1, ..., x_n) \leq x_i$.

Both kinds of lemma say something about the structural relationship of a function's output to one of its inputs. A reducer lemma says that the output is a substructure of the input; a conserver lemma says that the output is either the input itself or a substructure of it.

The analysis uses these lemmas to determine that $e < x$ or $e \leq x$ for some variable x in an expression e. The lemmas for projection functions—or, in some languages, the variables bound by case analysis—are trivially known. The analysis of expressions to produce further lemmas enables the lemmas for other functions to be calculated. Then, the argument a to each recursive call can be checked against the lemmas to ensure that $a < p$, where p is the corresponding parameter.

As described, the Walther recursion analysis operates on a first-order language which has monomorphic data constructors and in which each unique data constructor produces a value of a unique subtype of the algebraic data type defined partly in terms of that constructor. Consider this example from [20] of two subtraction functions:

```
data Nat = Zero | Succ Nat

minus :: Nat -> Nat -> Nat
minus  Zero     _         = Zero
minus  n        Zero      = n
minus  (Succ m) (Succ n) = minus m n

rminus :: Succ -> Succ -> Nat
rminus (Succ m) (Succ n) = minus m n
```

Note that $rminus(x_0, x_1) < x_0$, whereas $minus(x_0, x_1) \leq x_0$. McAllester and Arkoudas speculate that their analysis can be adapted to a language with polymorphic data constructors. Although the previous example looks almost like valid Haskell, the use of `Succ` as a type means that it is not. Those authors make no mention of languages, such as Haskell and others, which lack that subtyping. Turner considers the prospect of adapting Walther recursion to higher-order languages to be an "important challenge." [30]

6.3.1 Usefulness

Just a glance over the `Prelude` and `Data.List` modules reveals the potential lying in Walther recursion. Common functions, such as `-`, `div`, `sort`, `filter`, and `partition` would all bear interesting lemmas about the relationship between their inputs and outputs. Without Walther recursion, all of this readily-encapsulated functionality goes to waste much of the time.

A lack of Walther recursion has an even more severe impact on abstract data types. The goal of an abstract data type is to enforce a clear abstraction

boundary between the type's interface and implementation. One of the main motivations for pursuing this goal is that the implementation may be studiously maintaining invariants in its internal representation which might be violated by a user of the type, if that user has unfettered access to the implementation.

Without Walther recursion, abstract data types in TFP can never be the basis for recursion, since all recursion is structural, and the data's structure is hidden. Yet TFP and abstract data types both enhance the verifiability of programs, and both see this is a major goal. For TFP to inhibit the use of abstract data types is thus quite disappointing.

Chapter 7

Future Work

7.1 The Total Language

7.1.1 Walther Recursion and Beyond

Implementing Walther recursion analysis atop a rich language such as Haskell is a worthwhile next step. I also suggest one addition to that analysis. As written, the analysis automatically generates two kinds of lemmas: reducer and conserver. These lemmas relate information about the structural relationship between a function's parameters and its result.

Yet, for the purpose of deciding whether a recursive function terminates, this structural information is excessive. All that matters is that the *depth* of the output is a reduction or conservation of the *depth* of the input. This way of thinking suggests one a new kind of inference which may produce both reducer and conserver lemmas. Consider this function:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs) =
  if p x then x : filter p xs else filter p xs
```

It would be useful to infer that `filter` conserves the depth of its input.

This inference requires a third kind of lemma, which might perhaps be called an "expander" lemma. Such a lemma would indicate that the depth of a function's output is greater than the depth of its input by no more than some compile-time constant. Analogously, "reducer" lemmas would be enriched to indicate that the depth of the output is less than the depth of the input by at least some compile-time constant. Putting these lemmas together, as would happen regarding `(x : filter p xs)`, would result in the inference that that expression conserves the depth of the input list `(x : xs)`.

7.1.2 Identifying Induction Variables

A significant chunk of my implementation is devoted to finding the correct order in which to consider the parameters of a recursive function so that the induction argument for it becomes apparent. In much larger programs, the compile-time cost associated with my brute-force method of inference may become unacceptable. There are a couple of alternatives.

First, a smarter method may be used to determine the order of the parameters. At the very least, it is quite obvious when a parameter cannot possibly be useful toward the induction, such as when it is passed a value that bears no provable relationship to the corresponding input. Since the number of induction variables is unlikely to be too large, this alone may be a sufficient optimization.

An alternate method is to require that the programmer provide a manual annotation of the induction variables, and perhaps their ordering. It would not be especially burdensome to provide this information: the programmer has to be aware of it anyway, and the extra syntax could be as minimal as one special indicator character per parameter. However, in a language like Haskell, it is not clear where exactly such a syntactic indicator would fit, since type declarations are optional and parameters aren't necessarily bound to a simple name.

7.1.3 Natural Number-Typed Induction Variables

One minor but annoying issue faced by a Haskell programmer using induction over natural numbers is that the use of arithmetic operations and pattern-matching on the induction variable does not force its type to be a natural number type: the compiler can typically generalize these variables to any numeric type, or at least any integral type. But a variable that might not be a natural number cannot be used for an inductive argument, so such a function results in an error. The only presently-available solution is to artificially restrict the type of the variable with a declaration. There are many better solutions:

- Infer from the need to use the variable as an induction variable that it must have a natural number type. This could be difficult to infer under a more advanced analysis like Walther recursion, since it may not be known whether a function operating on numbers will later be used by the inductive arguments of other functions.
- Restrict `n+k` patterns to natural numbers.

Another problem with Haskell's `n+k` patterns is that {`0`, `n+1`} is not considered a complete set of patterns, because `n` might be negative. Of course, this problem could be solved. It would be simpler to solve if the second choice above, restricting such patterns to natural numbers, were also implemented.

7.1.4 List Literals and Comprehensions

It seems unfair to colists that lists get all the syntactic sugar, such as list literals and comprehensions. Such syntax could be generalized to both lists and colists. For list comprehensions, the generalization might as well extend to all types in `MonadPlus`.

7.2 Implementation of Type Classes

GHC's implementation of type classes causes some problems for my implementation which are described in Section 3.4.1. An industrial-strength implementation of TFP atop GHC would need to alter the way typeclasses are implemented. The simplest way to resolve the problems would be for any instance which uses default implementations of methods supplied by the class to emit a copy of the default implementation as a part of the instance. However, this is not feasible given GHC ability to compile each module separately.

7.3 Generalizing Over Codata Types

Working with codata is bound to generate massive amounts of boilerplate code unless more effective methods of generalizing codata are provided. The key problem is the syntactic rule that demands the presence of a coconstructor. An effective generalization would have to allow a variable to provably be a coconstructor. Two possible ways the type system could record this information are:

- A different $\rightarrow$ operator for coconstructor types.
- A different kind for coconstructor types.

7.4 The Total Prelude

Large portions of my total Prelude are simple borrowed functions from the standard Haskell Prelude, and many more are standard functions within a wrapper. For greater reliability, a total Prelude should implement as many functions as possible in a provably total way.

Chapter 8

Related Work

Some related systems exist, to which Turner alludes [30]. These systems implement dependently-typed languages, which in short have two critical qualities:

- Types may be parameterized by values
- In a function type, the result type may vary depending on the argument type.

One of the first uses of such a system one encounters is a list type which has the length of the list as a parameter of the type. Thus, operations requiring non-empty lists as input can enforce that constraint by their types.

I will discuss two dependently-typed languages here: Agda [22] and Epigram [21]. Both share many characteristics: they are purely functional and require explicit type declarations. They also both require that all functions be total, in a manner similar to Turner's TFP [30]. One purpose of this totality is to allow parts of the program to be safely executed at compile-time, in order to evaluate values which are parameters to types.

8.1 Agda

Norell states that explicit type declarations are advantageous because they allow the compiler to better help the programmer to implement a function [22]. This statement probably has some truth in conventional functional programming, and it has even more in a dependently-typed language, where types can reach new heights of richness and complication.

In pursuit of the goal of the compiler helping the programmer, an expression may be omitted, with a `?` in its place. The compiler will tell the programmer whatever useful things it knows about these blank parts of the program. The presence of a feature such as this suggests that, contrary to Turner's claim [30], it may be possible to use a language of Agda's complexity for teaching purposes, also this would require experimentation to discern.

8.2 Epigram

Epigram places great emphasis on its central paradigm of program-building, which is that each function is built up as a sequence of nested refinements, where a problem is altered and possibly broken down into multiple pieces which are then solved. This is reflected heavily in its syntax.

Like Agda, Epigram has a syntax for unimplemented subproblems. It also has an interactive editor which generates what code it can, helpfully highlights the subproblem being worked, and shows messages from the compiler. This tool might make Epigram even better suited to an educational setting than Agda.

One interesting aspect of Epigram is that, in the course of solving a problem recursively, the programmer must declare the induction variable recursively. In Section 7.1.2, I discuss both the desirability and difficulty of declaring induction variables in a language like Haskell; Epigram provides a model for accomplishing this.

One of Epigram's key goals is extensibility. The top-level structure in the definition of a function is a decomposition of the problem, and the two built-

in decompositions are case analysis and structural recursion. However, the programmer may create additional decompositions. This is similar in concept to the way functions in TFP may encapsulate inductive patterns which may be tedious to regularly reinvent.

8.3 Relationship to Walther Recursion

As McBride aptly describes it, "dependent types make more programs structurally recursive, because dependent types have more structure." [21] Walther recursion also makes more programs structurally recursive, compared to primitive TFP, but even Walther recursion is a subset of dependently-typed structural recursion. The lemmas which Walther recursion infers about the relationship of a function's input to its output are one particular kind of information which a dependent function type already conveys. Walther recursion does have the advantage that these lemmas can be inferred, whereas in both Agda and Epigram, those lemmas, being a part of the explicit type declarations, are not inferred.

Chapter 9

Conclusions

My works suggests that, with some improvements, TFP could become a practical way of writing programs. Potential improvements, which have been discussed in more detail earlier in this report, include:

- Improving the performance of the recursion analysis.
- Improving the error messages produced by the compiler.
- Finding a way for the corecursion analysis to admit cofunctions which have a polymorphically-generalized result type.
- Enhancing the definition of Walther recursion to work on a practical modern language such as Haskell.
- Extending Walther recursion with expander lemmas.

All of these improvements may be the subject of useful future work, after which I expect that some TFP implementation could become a practical platform for implementing large, reliable programs.

Bibliography

[1] Annotations. http://hackage.haskell.org/trac/ghc/wiki/Annotations.

[2] Data.Array. http://www.haskell.org/ghc/docs/6.12.2/html/libraries/array-0.3.0.0/Data-Array.html.

[3] Data.Map. http://www.haskell.org/ghc/docs/6.12.2/html/libraries/containers-0.3.0.0/Data-Map.html.

[4] The Mercury Language Reference Manual. http://www.mercury.csse.unimelb.edu.au/information/doc-release/mercury_ref/index.html.

[5] *Penpa Mix 2*. Nikoli Co., Ltd., Tokyo, Japan, 2005.

[6] Zack Butler. Artificial Intelligence class project. RIT course 4003-455, term 20073.

[7] Simon Peyton Jones, et. al. Basic Input/Output. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[8] Simon Peyton Jones, et. al. Character Utilities. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[9] Simon Peyton Jones, et. al. Declarations and Bindings. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[10] Simon Peyton Jones, et. al. Expressions. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[11] Simon Peyton Jones, et. al. *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[12] Simon Peyton Jones, et. al. List Utilities. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[13] Simon Peyton Jones, et. al. Modules. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[14] Simon Peyton Jones, et. al. Overview of Types and Classes. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[15] Simon Peyton Jones, et. al. The Standard Prelude. In *Haskell 98 Language and Libraries: the Revised Report*. 2002.

[16] Jon Ferraiolo and Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation, W3C, January 2003. http://www.w3.org/TR/2003/REC-SVG11-20030114/.

[17] Ralf Hinze. Functional Pearl: Streams and Unique Fixed Points. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 189–200, New York, NY, USA, 2008. ACM.

[18] Graham Hutton and Erik Meijer. Monadic Parser Combinators, 1996.

[19] Research Software Limited. *Miranda System Manual*. 1989.

[20] David McAllester and Kostas Arkoudas. Walther Recursion. In *Proceedings CADE 13, Springer LNCS*, pages 643–657. Springer-Verlag, 1996.

[21] Conor McBride. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming*. 2005.

[22] Ulf Norell. Dependently Typed Programming in Agda. In *Advanced Functional Programming*. 2009.

[23] Rinus Plasmeijer and Marko van Eekelen. *The Clean Language Report*. University of Nijmegen, November 2002.

[24] Colin Runciman. What About the Natural Numbers. *Computer Languages*, 14:181–191, 1989.

[25] Axel Schreiner. Language Processing. http://www.cs.rit.edu/ ats/fp-2009-1/04_bnf.pdf.

[26] Axel Schreiner. More about Monads. http://www.cs.rit.edu/ ats/fp-2009-1/06_more-monads.pdf.

[27] Alastair Telford and David Turner. Ensuring Streams Flow. In *Proc. 6 th AMAST*, pages 509–523. Springer, 1997.

[28] The GHC Team. GHC Language Features. In *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2*.

[29] The GHC Team. Rebindable syntax and the implicit Prelude import. In *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2*.

[30] D. A. Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.