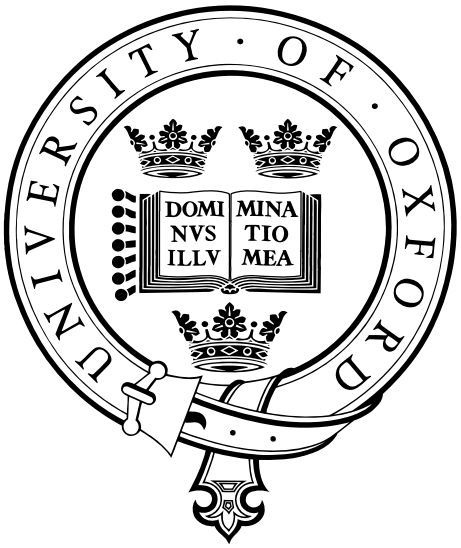

A Functional and Monadic Proof Assistant for Streams

by

Daniel W. H. James

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

supervised by

Ralf Hinze

Oxford University Computing Laboratory

2007–8

Abstract

Streams, which are infinite sequences of elements, are defined by a coinductive datatype and operations on streams are corecursive programs. Equations that define streams, under light restrictions, have a unique solution. This property gives rise to a succinct proof technique for proving equality between streams. This project presents the discussion and implementation of a proof assistant that supports this proof method. The tool is implemented in the purely functional language Haskell and makes extensive use of monads. The emphasis of the project is placed on simplicity, clarity and terseness.

Acknowledgements

Thanks go to: Ralf Hinze and Andres Löh for their tool lhs2TEX, which has been invaluable in the typesetting of this thesis; Richard Stallman and the maintainers of GNU Emacs; Dr. Stewart Adams OBE et. al. for their 1961 discovery of Ibuprofen; Whittard of Chelsea and the coffee growers and pickers of the world (notably Guatemala, Kenya, India and Nicaragua); Alfonso Bialetti for invention of the Moka Express; the Keble Library of Keble College, the Vere Harmsworth Library of the Rothermere American Institute, the Taylor Institution Library and the Radcliffe Science Library; the kitchen and hall staff of Keble College for breakfast, lunch and dinner.

Contents

Chapter 1

Introduction and Background

1.1 Introduction

This master's project presents the implementation of a proof assistant. The proofs that it is purposed for are equalities between *streams*. The project idea was borne out of Hinze's paper "Functional Pearl: Streams and Unique Fixed Points" [4]. The project's product, a proof tool, serves as a complement to this paper. Note that it is labelled as a 'proof assistant' rather than an 'automatic prover', the distinction being that the tool requires active human interaction to direct the proof search. The latter was soon found to be a goal that would cost much in the way of complexity and intricacy.

The meta-purpose of the project was a personal exploration of *theorem-proving*. There exist a number of general-purpose theorem-provers, some which are highly performant, as well as many other specialized tools. This project has different intentions — the guiding principle is *simplicity*. The program code presented in this thesis has been crafted to maximize clarity and readability; many of the features are suprisingly succicint and this terseness has been achieved, to a great degree, by the nature of the implementation language. My quest for elegance is supported by an emphasis on a functional and monadic implementation — optimized performance is superfluous within the bounds of this project. The starting point for the implementation of this project was Bird's functional calculator, presented in chapter 12 of "Introduction to Functional Programming using Haskell" [1].

1.2 Streams

A stream is an infinite sequence of elements. The following is a definition in the *lazy* functional language Haskell [8].

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}\ \alpha = \mathit{Cons}\{\mathit{head} :: \alpha, \mathit{tail} :: \mathit{Stream}\ \alpha\}\\
&\textbf{infixr}\ 5\ \prec\\
&(\prec) :: \alpha \to \mathit{Stream}\ \alpha \to \mathit{Stream}\ \alpha\\
&a \prec s = \mathit{Cons}\ a\ s
\end{aligned}$$

As well as the datatype *Stream*, we have defined an infix operator ($\prec$) so as to provide syntactic sugar for constructing streams.

The type of streams, *Stream* α, is a container type. Like Haskell's list datatype $[\alpha]$, it contains values of type α. While a list *can* be infinite in a lazy language such as Haskell, it is more widely viewed as a finite datatype. Streams however are most definitely infinite. Note that there is just a single data constructor (*Cons*) and no base constructor, so there is no means by which to terminate a stream and therefore all stream values are necessarily infinite objects.

We can delve briefly into the theory behind this. In contrast to the list datatype, which is an inductive type, a stream is a *coinductive* type. The semantics of the list datatype are given by an *initial algebra* and analogously the semantics of the stream datatype are given by a *final coalgebra*. The parallels continue from the category-theoretic world: *fold* the *catamorphism* of the initial algebra for lists, is mirrored by the *anamorphism* of the final coalegbra for streams, *unfold*. Just as Hutton points out the use of fold and its universal property as a proof method [5], Gibbons and Hutton show the use of unfold and its universal property as a high-level proof method for corecursive programs [3].

The following is an extremely simple stream which serves simply to demonstrate the construction of a stream — an infinite object.

$$\begin{aligned} &\mathit{zeros} :: \mathit{Stream}\ \mathit{Int} \\ &\mathit{zeros} = 0 \prec \mathit{zeros} \end{aligned}$$

The stream *zeros* is the infinite sequence of the number zero. This is an example of what we call a *constant* stream, a stream s such that $\mathit{tail}\ s = s$.

Suppose that we have a function of the type $(\alpha_1 \to ... \to \alpha_n \to \beta)$ and we wish to *lift* this function to streams. This lifting operator is familiar to functional programmers, again in the context of lists, and is commonly called *zipWith*. Thus the type of a *zipWith* for streams would be,

$$(\alpha_1 \to ... \to \alpha_n \to \beta) \to (\mathit{Stream}\ \alpha_1 \to ... \to \mathit{Stream}\ \alpha_n \to \mathit{Stream}\ \beta)$$

In Haskell *zipWith* is in fact a family of higher-order functions, one for each n. For $n = 0, 1, 2$ these are given specific names: *repeat*, *map* and *zip* respectively. Here is the first,

$$\begin{aligned} &\mathit{repeat} :: \alpha \to \mathit{Stream}\ \alpha \\ &\mathit{repeat}\ a = a \prec \mathit{repeat}\ a \end{aligned}$$

The function *repeat* can also be called a *parametrized stream*. Given a value, *repeat* constructs an infinite sequence of that value — a constant stream. As pointed out previously, $\mathit{tail}\ (\mathit{repeat}\ k) = \mathit{repeat}\ k$ for all k. With *repeat* we can now redefine *zeros* more succinctly as $\mathit{zeros} = \mathit{repeat}\ 0$.

The following are the functions *map* and *zip*, as defined for streams. They are indeed similar in definition to their list counterparts and are referred to as *stream operators*.

$$
\begin{aligned}
&map :: (\alpha \to \beta) \to (Stream\ \alpha \to Stream\ \beta)\\
&map\ f\ s = f\ (head\ s) \prec map\ f\ (tail\ s)\\
&zip :: (\alpha \to \beta \to \gamma) \to (Stream\ \alpha \to Stream\ \beta \to Stream\ \gamma)\\
&zip\ f\ s\ t = f\ (head\ s)\ (head\ t) \prec zip\ f\ (tail\ s)\ (tail\ t)
\end{aligned}
$$

Suppose that we have the streams *ones* and *twos*, defined as *repeat* 1 and *repeat* 2 respectively. It seems natural then to perform arithmetic operations such as addition on numeric streams like these. Addition, multiplication and other arithmetic operators are performed element-wise on streams, thus we can use the *zipWith* family of functions to lift arithmetic on numeric values to arithmetic on streams of numeric values. In Haskell we can make the datatype *Stream* an instance of the numeric type class *Num* with the following implementation[1].

$$
\begin{aligned}
&\textbf{instance}\ (Num\ \alpha) \Rightarrow Num\ (Stream\ \alpha)\ \textbf{where}\\
&\quad (+) \quad\quad = zip\ (+)\\
&\quad (-) \quad\quad = zip\ (-)\\
&\quad (*) \quad\quad = zip\ (*)\\
&\quad negate \quad = map\ negate\\
&\quad fromInteger = repeat \circ fromInteger
\end{aligned}
$$

Type classes [13, 2] allow a controlled 'brand' of *ad-hoc polymorphism* in the form of *overloading* [11]. Given this instance declaration, the addition operator (+) can be used on all numeric types, including streams.

As an aside, the type we originally gave to the stream *zeros* was *Stream Int*, however using the full power of Haskell and its type class mechanism, we can generalize this and give *zeros* the type $Num\ \alpha \Rightarrow Stream\ \alpha$. This means that for any type α that is an instance of the numeric type class *Num*, *zeros* has type $Stream\ \alpha$. The integer type *Int* is a numeric type and an instance of the *Num* type class, however the integer constant 0 in the definition is also in the set of natural, rational, real and complex numbers. The Haskell language allows not only the overloading of functions, but also the syntactic use of integer constants for any datatype which is an instance of the class *Num*. This is demonstrated with the following definition of the stream of natural numbers,

$$nat = 0 \prec nat + 1$$

The integer constant 1 in the expression $nat + 1$, abbreviates the *fromInteger* method from the *Num* type class, so that the above definition is in fact equivalent to,

$$nat = 0 \prec zip\ (+)\ nat\ (repeat\ 1)$$

Note that in the definition of the $(\prec)$ operator, it is declared to be right associative with a precedence level of 5; the precedence level of addition and multiplication is 6 and 7 respectively, with a higher level corresponding to a higher precedence. Therefore $0 \prec nat + 1$ is in fact $0 \prec (nat + 1)$ when all the parenthesises are added in.

[1] The methods *abs* and *signum* have been omitted

It is interesting to compare the previous definition of the natural numbers as a stream to its definition as a recurrence,

$$a_0 = 0 \quad a_{n+1} = a_n + 1$$

Recurrences of this form are easily expressed in a single stream equation using the $(\prec)$ operator. Again, here is a recurrence for the factorials,

$$a_0 = 1 \quad a_{n+1} = (n+1) * a_n$$

and the same sequence defined as a stream,

$$fac = 1 \prec (nat + 1) * fac$$

This relationship between recurrences and streams can be formalized. A sequence defined by, $a_0 = k$ and $a_{n+1} = f(a_n)$ becomes $a = k \prec f\ a$. A reference to the indexing variable, as in the definition of the factorials, become the natural numbers stream.

The Fibonacci numbers are a slightly more complicated numeric sequence, defined by the following recurrence,

$$a_0 = 0 \quad a_1 = 1 \quad a_{n+2} = a_{n+1} + a_n$$

which translates into the following stream equation,

$$fib = 0 \prec 1 \prec fib + tail\ fib$$

The first *two* values of the sequence are declared and successive values depend on the previous *two* values. This stream equation, while correct, is defined in terms of its tail, *tail fib*, which is less than desirable for reasons that will be elucidated on later. No matter however, as it is simple to eliminate this call to *tail* by splitting the equation into two and employing mutual recursion as follows,

$$\begin{aligned} fib &= 0 \prec fib' \\ fib' &= 1 \prec fib + fib' \end{aligned}$$

With the following calculation, these two equations can be recombined into one,

$$\begin{aligned} & fib' \\ = & \quad \{ \text{ definition of } fib' \ \} \\ & 1 \prec fib + fib' \\ = & \quad \{ \ 1 = 1 + 0 \ \} \\ & (1 + 0) \prec fib + fib' \\ = & \quad \{ \text{ distribute } \prec \text{ over } + \ \} \\ & (1 \prec fib) + (0 \prec fib') \\ = & \quad \{ \text{ definition of } fib \ \} \\ & (1 \prec fib) + fib \end{aligned}$$

giving

$$fib = 0 \prec fib + (1 \prec fib)$$

There is a another function called *iterate* that makes an important addition to our list of basic stream functions: *repeat*, *map* and *zip*. Given a function f of type $(\alpha \to \alpha)$ that transforms elements of type α, the function *iterate* builds a stream of repeated applications of f to an initial value a. The definition is as follows,

$$\begin{aligned} &iterate :: (\alpha \to \alpha) \to (\alpha \to Stream\ \alpha) \\ &iterate\ f\ a = a \prec iterate\ f\ (f\ a) \end{aligned}$$

Using *iterate* we can give an alternative definition of the stream of natural numbers, which is *iterate* (+1) 0, where (+1) is the function that increments a numeric value. The function *iterate* also generalizes the function *repeat*. The expression *repeat* 0 is the stream of zeros, however so is *iterate id* 0, where *id* is the identity function. This gives us our first hints of motivation, for proving equality between streams.

A final operator to add to our basic list is stream *interleaving*.

$$\begin{aligned} &\textbf{infixr}\ 5\ \curlyvee \\ &(\curlyvee) :: Stream\ \alpha \to Stream\ \alpha \to Stream\ \alpha \\ &s \curlyvee t = head\ s \prec t \curlyvee tail\ s \end{aligned}$$

This operator is neither commutative or associative. Like the stream cons operator $(\prec)$, stream interleave is right associative and has the same level of precedence.

Using interleaving we can give an alternative definition of the natural numbers. Our first definition *nat*, was based on Peano axioms. The following definition is based on the binary number system.

$$bin = 0 \prec 2 * bin + 1 \curlyvee 2 * bin + 2$$

The interleaving operator captures a further style of recurrence equation. Recurrences of the following form,

$$a_0 = k \quad a_{2n+1} = f(a_n) \quad a_{2n+2} = g(a_n)$$

translate into $a = k \prec f\ a \curlyvee g\ a$.

We have given two very different equations for what we believe to be the exact same stream. By definition the starting elements of both *nat* and *bin* are the same, but we cannot simply examine the remaining elements as we are dealing with infinite, not finite, sequences.

It is indeed possible to define a corecursive function of type $Stream\ \tau$ that does not actually produce a stream. The equations $s = tail\ s$ and $s = head\ s \prec tail\ s$ are two such examples. Operationally speaking, these definitions loop in Haskell and do so because they are not *productive*. As noted previously, $tail = s$ iff s is a constant

stream. Therefore s can be instantiated to any constant stream, of which there are infinitely many. The second equation is even more general, in fact the most general as this equation holds for all streams.

If one restricts the syntactic form of these definitions to ensure that they are unambiguous — that they possess *unique solutions* — then this restriction of uniqueness will pay dividends when it comes to proving that two streams are equal. Quite simply, uniqueness can be exploited to the extent that, if one can show that two streams satisfy the same equation, then they must be equal as this common equation has one and only one solution. Unique equations are of the following form,

$$x\ x_1 \ldots x_n = h \prec t$$

for $n \geq 0$, where x is an identifier of type *Stream* τ when $n = 0$ and x is a function of return type *Stream* τ when $n > 0$. Expression h is of type τ; t is an expression of type *Stream* τ that can contain recursive calls to x, however neither *head* or *tail* may be applied to x. Well-typed applications of *head* and *tail* to arguments x_i are permitted.

With the exception of our first definition of the Fibonacci stream, all of the stream equations thus far satisfy these restrictions. There are other streams where their natural definition does not follow the form of $x = h \prec t$. The following fractal sequence is a good example:

$$\mathit{frac} = \mathit{nat} \curlyvee \mathit{frac}$$

This is a sequence that contains itself in a sub-sequence. Here are the first 20 value of the stream *frac*:

$$0\ \mathbf{0}\ 1\ \mathbf{0}\ 2\ \mathbf{1}\ 3\ \mathbf{0}\ 4\ \mathbf{2}\ 5\ \mathbf{1}\ 6\ \mathbf{3}\ 7\ \mathbf{0}\ 8\ \mathbf{4}\ 9\ \mathbf{2}$$

Note that the odd indicies are the natural numbers and the even indicies are the stream itself! The use of a previously defined stream is legitimate and in this case it is what makes this definition productive. Using the definition of interleaving and natural numbers, this definition can be rewritten to a less elegant version that does satisfy the strict syntactic restrictions:

$$\mathit{frac} = 0 \prec \mathit{frac} \curlyvee \mathit{nat} + 1$$

A proof utilizing uniqueness can be formalized as follows. Let $s = \phi\ s$ be a valid stream equation that observes the above restrictions and let its unique solution be *fix* ϕ. Uniqueness is expressed in these terms by the following universal property:

$$\mathit{fix}\ \phi = s \quad \Longleftrightarrow \quad \phi\ s = s$$

This states that *fix* ϕ is a solution of $x = \phi\ x$ and that any solution s of $x = \phi\ x$ is equal to *fix* ϕ. To prove that $s = t$, where s is the unique solution of $x = \phi\ x$, *fix* ϕ, it is sufficient to prove that $\phi\ t = t$. The following is an illustratively succinct proof that

repeat $k = \mathit{iterate}\ \mathit{id}\ k$:

$$\begin{array}{ll} & \mathit{iterate}\ \mathit{id}\ k \\ = & \quad \{\ \text{definition of iterate}\ \} \\ & k \prec \mathit{iterate}\ \mathit{id}\ (\mathit{id}\ k) \\ = & \quad \{\ \text{identity}\ \} \\ & k \prec \mathit{iterate}\ \mathit{id}\ k \end{array}$$

Thus *iterate id k* equals the unique solution of $x = k \prec x$, which is by definition *repeat k*.

This first example was an easy case as only one of the terms s and t was given as a fixed point. In the more complex case where both s and t are fixed points, $\mathit{fix}\ \phi$ and $\mathit{fix}\ \psi$ respectively, there are four basic equations that are sufficient to prove that $s = t$: $\phi\ (\psi\ s) = \psi\ s$, $\psi\ s = s$ and the analogous $\psi\ (\phi\ t) = \phi\ t$, $\phi\ t = t$. The sufficiency of the first two are shown by the following proof.

$$\begin{array}{ll} & \phi\ (\psi\ s) = \psi\ s \\ \Leftrightarrow & \quad \{\ \text{universal property}\ \} \\ & \mathit{fix}\ \phi = \psi\ s \\ \Leftrightarrow & \quad \{\ s = \mathit{fix}\ \phi\ \} \\ & s = \psi\ s \\ \Leftrightarrow & \quad \{\ \text{universal property}\ \} \\ & s = \mathit{fix}\ \psi \\ \Leftrightarrow & \quad \{\ t = \mathit{fix}\ \psi\ \} \\ & s = t \end{array}$$

Establishing any of these four equations may be tricky or even unattainable; these equations are expressing that one term is a fixpoint of the defining function of the other term. For all but the simplest stream equations, this immediate similarity is unlikely. An alternative approach would be to progress from both s and t to meet at a common function χ, with a unique fixed point. The proof is therefore in two halves, one to show that $s = \chi\ s$ and the other to show that $t = \chi\ t$. Relying on the fact that χ has a unique fixed point — $x = \chi\ x$ has a unique solution — these two halves can be joined to show that $s = t$. Such a proof would mimic the following skeleton.

$$\begin{array}{ll} & s \\ = & \quad \{\ \mathit{why?}\ \} \\ & \chi\ s \\ \subset & \quad \{\ x = \chi\ x \text{ has a unique solution}\ \} \\ & \chi\ t \\ = & \quad \{\ \mathit{why?}\ \} \\ & t \end{array}$$

The function χ is a product of the proof itself; it is discovered during the process and cannot be systematically predicted *a priori*. The symbol $\subset$ denotes the *link* in this *linked proof*.

In their paper "Proof methods for structured corecursive programs" [3], Gibbons and Hutton explore a number of proof methods. These include the use of the *unfold* operator along with its *universal* property, to conduct proofs in a style similar to the *calculational* linked proofs presented here. The comparison is worthwhile making, as we are interested in assuring ourselves that linked proofs are not only a good proof method, but that they are at least superior in some sense to competing methods.

The presentation given by Gibbons and Hutton uses lists (which can be finite) not streams, so the definitions and proofs given here have been recast into the setting of streams. The following is the definition of *unfold*,

$$\begin{aligned} &\mathit{unfold} :: (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to \mathit{Stream}\ \beta \\ &\mathit{unfold}\ f\ g\ s = f\ s \prec \mathit{unfold}\ f\ g\ (g\ s) \end{aligned}$$

and its universal property,

$$f = \mathit{unfold}\ g\ h \quad \Longleftrightarrow \quad \forall x.\ f\ x = g\ x \prec f\ (h\ x)$$

The first restriction of this proof methods is that corecursive programs must be written in terms of *unfold* so that the universal property can be employed. Previously we gave a linked proof of the equality $\mathit{repeat}\ k = \mathit{iterate}\ \mathit{id}\ k$. Given a new definition of *iterate* as $\mathit{iterate}\ f\ s = \mathit{unfold}\ \mathit{id}\ f\ s$ here is a proof for comparison.

$$\begin{aligned} & \mathit{repeat} = \mathit{iterate}\ \mathit{id} \\ \Leftrightarrow \quad & \{\ \text{definition of iterate}\ \} \\ & \mathit{repeat} = \mathit{unfold}\ \mathit{id}\ \mathit{id} \\ \Leftrightarrow \quad & \{\ \text{universal property}\ \} \\ & \forall x.\ \mathit{repeat}\ x = \mathit{id}\ x \prec \mathit{repeat}\ (\mathit{id}\ x) \\ \Leftrightarrow \quad & \{\ \text{simplification}\ \} \\ & \forall x. \mathit{repeat}\ x = x : \mathit{repeat}\ x \\ \Leftrightarrow \quad & \{\ \text{definition of repeat}\ \} \\ & \mathit{true} \end{aligned}$$

The second proof for comparison is the proof of the *iterate* fusion law:

$$\mathit{map}\ h \circ \mathit{iterate}\ f_1 = \mathit{iterate}\ f_2 \circ h \iff h \circ f_1 = f_2 \circ h$$

which is in fact the *free theorem* [12] of the type $(\alpha \to \alpha) \to (\alpha \to \mathit{Stream}\ \alpha)$, the type

of *iterate*. The first proof of this law is using the linked proof style:

$$
\begin{array}{ll}
 & \mathit{map}\ h\ (\mathit{iterate}\ f_1\ a) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
 & h\ (\mathit{head}\ (\mathit{iterate}\ f_1\ a)) \prec \mathit{map}\ h\ (\mathit{tail}\ (\mathit{iterate}\ f_1\ a)) \\
= & \quad \{\ \text{definition of } \mathit{iterate}, \text{ simplify } \mathit{head} \text{ and } \mathit{tail}\ \} \\
 & h\ a \prec \mathit{map}\ h\ (\mathit{iterate}\ f_1\ (f_1\ a)) \\
\subset & \quad \{\ x\ a = h\ a \prec x\ (f_1\ a) \text{ has a unique solution}\ \} \\
 & h\ a \prec \mathit{iterate}\ f_2\ (h\ (f_1\ a)) \\
= & \quad \{\ \text{assumption: } h \circ f_1 = f_2 \circ h\ \} \\
 & h\ a \prec \mathit{iterate}\ f_2\ (f_2\ (h\ a)) \\
= & \quad \{\ \text{definition of iterate}\ \} \\
 & \mathit{iterate}\ f_2\ (h\ a)
\end{array}
$$

To prove the same law using *unfold* and its universal property, we must first prove two auxiliary fusion laws. The first shows that the composition of an *unfold* and another function can be fused into a single *unfold*.

$$
\begin{array}{ll}
 & \mathit{unfold}\ f\ g_1 \circ h = \mathit{unfold}\ (f \circ h)\ g_2 \\
\Leftrightarrow & \quad \{\ \text{universal property}\ \} \\
 & \forall x.\ \mathit{unfold}\ f\ g_1\ (h\ x) = f\ (h\ x) \prec \mathit{unfold}\ f\ g_1\ (h\ (g_2\ x)) \\
\Leftrightarrow & \quad \{\ \text{definition of } \mathit{unfold}\ \} \\
 & \forall x.\ f\ (h\ x) \prec \mathit{unfold}\ f\ g_1\ (g_1\ (h\ x)) = f\ (h\ x) \prec \mathit{unfold}\ f\ g_1\ (h\ (g_2\ x)) \\
\Leftarrow & \quad \{\ \text{extensionality}\ \} \\
 & g_1 \circ h = h \circ g_2
\end{array}
$$

The second fusion law states that the composition of a *map* and an *unfold* can again be fused into a single *unfold*.

$$
\begin{array}{ll}
 & \mathit{map}\ f \circ \mathit{unfold}\ g\ h = \mathit{unfold}\ (f \circ g)\ h \\
\Leftrightarrow & \quad \{\ \text{universal property}\ \} \\
 & \forall x.\ \mathit{map}\ f\ (\mathit{unfold}\ g\ h\ x) = f\ (g\ x) \prec \mathit{map}\ f\ (\mathit{unfold}\ g\ h\ (h\ x)) \\
\Leftrightarrow & \quad \{\ \text{definition of } \mathit{map}\ \} \\
 & \forall x.\ f\ (\mathit{head}\ (\mathit{unfold}\ g\ h\ x)) \prec \mathit{map}\ f\ (\mathit{tail}\ (\mathit{unfold}\ g\ h\ x)) \\
 & \quad = f\ (g\ x) \prec \mathit{map}\ f\ (\mathit{unfold}\ g\ h\ (h\ x)) \\
\Leftrightarrow & \quad \{\ \text{definition of } \mathit{unfold}, \text{ simplify } \mathit{head} \text{ and } \mathit{tail}\ \} \\
 & \forall x.\ f\ (g\ x) \prec \mathit{map}\ (\mathit{unfold}\ g\ h\ (h\ x)) = f\ (g\ x) \prec \mathit{map}\ f\ (\mathit{unfold}\ g\ h\ (h\ x)) \\
\Leftrightarrow & \quad \{\ \text{reflexivity}\ \} \\
 & \mathit{true}
\end{array}
$$

Finally, using these two fusion laws we can prove the iterate fusion law,

$$
\begin{aligned}
& \mathit{iterate}\ f_2 \circ h \\
= \quad & \{\ \text{definition of } \mathit{iterate}\ \} \\
& \mathit{unfold}\ \mathit{id}\ f_2 \circ h \\
= \quad & \{\ \text{fusion (1)}\ \} \\
& \mathit{unfold}\ (\mathit{id} \circ h)\ f_1 \\
= \quad & \{\ \text{identity}\ \} \\
& \mathit{unfold}\ (h \circ \mathit{id})\ f_1 \\
= \quad & \{\ \text{fusion (2)}\ \} \\
& \mathit{map}\ h \circ \mathit{unfold}\ \mathit{id}\ f_1 \\
= \quad & \{\ \text{definition of iterate}\ \} \\
& \mathit{map}\ h \circ \mathit{iterate}\ f_1
\end{aligned}
$$

While this approach is appealing in the respect that the heavy lifting is accomplished with commons tools based upon *unfold*, this very same characteristic is constraining.

Chapter 2

Implementation

2.1 Expressions

The concept of streams has been introduced using Haskell however we will not use the full Haskell language for representing and manipulating streams. We will in fact restrict ourselves to a stripped down language that can express integer streams. This presents a domain that is simplistic enough but with a more than sufficient supply of interesting streams[1].

2.1.1 Representation of integer streams

The following datatype defines the abstract syntax that will be used to represent a stream expression.

```
type Ident = String
infixr 5 :≺:, :ϒ:
infixr 8 :^:
data Expr = Lit Int
          | Var Ident
          | App Ident [Expr]
          | Summ [Expr]
          | Prod [Expr]
          | Expr :^: Expr
          | Expr :≺: Expr
          | Expr :ϒ: Expr
            deriving (Eq, Ord)
```

The first line declares a *type synonym*, stating that an identifier is a string. The type *Expr* has eight data-constructors for constructing: literals, variables, function applications, sums, products, exponentials, cons'ed streams and interleaved streams. The deriving

[1] `http://www.research.att.com/~njas/sequences/`

clause at the end of the datatype declaration states which (language standard) type classes we would like to be automatically instantiated. Here the datatype *Expr* is given an instance for the equality type class *Eq* and the ordering type class *Ord*. The type class *Eq* defines two operators, ($\equiv$) and ($\not\equiv$); *Ord* defines a variety of comparison operators including ($\leqslant$). Ordering on abstract datatypes such as *Expr* is defined by the syntactic order of the data-constructors, thus $Lit\ 0 < Var$ `"x"`. The necessity of equality is obvious, but ordering is required so that collections of expressions, such as the arguments to *Summ* and *Prod*, can be ordered and compared.

The are several points worth making about the *Expr* data type. Addition and multiplication are commutative and associative operators, therefore instead of being binary operators, *Summ* and *Prod* hold their operands in a list. The summands and factors are to be kept in sorted order so that comparison is simplified and furthermore, literals will conveniently be at the head of the list. Subtraction is not a primitive in this abstract syntax, instead an expression such as $a-b$ would be $Summ\ [Var$ `"a"`$, Prod\ [Lit\ (-1), Var$ `"b"`$]]$. Expressivity is traded here for simplicity; the trade-off is worthwhile as the abstract syntax is an internal representation. A final point of interest is the representation of function application. An alternative implementation for the *App* data-constructor is *App Expr Expr*. The implementation given means that the function name is immediately accessible and the arguments are collected together so that they can be counted and ranged over.

2.1.2 Canonical form

Just as a human would want to kept their proofs tidy by eliminating redundancies, here we also want to keep our integer streams in a simple canonical form. The following functions do this by partially evaluating constants and applying basic arithmetic laws. Literals and variables are the base cases of the canonical form. A function application is in the canonical form if all of its arguments are. To construct a *Summ* in the canonical form we use the following function; it sums a list of expressions in canonical form and returns a single expression representing that sum:

$$
\begin{aligned}
&summ :: [Expr] \to Expr \\
&summ\ [\,] \ = zero \\
&summ\ [e] = e \\
&summ\ es\ = \\
&\quad \textbf{case}\ peval\ (mergeLists\ [summands\ e \mid e \leftarrow es])\ \textbf{of} \\
&\qquad [x] \to x \\
&\qquad xs \quad \to Summ\ xs \\
&\quad \textbf{where}\ peval\ (Lit\ m : Lit\ n : es) = peval\ (Lit\ (m+n) : es) \\
&\qquad\qquad peval\ (Lit\ 0 : es) \qquad\quad = es \\
&\qquad\qquad peval\ es \qquad\qquad\quad\ = es
\end{aligned}
$$

This definition illustrates a number of language features in Haskell. The first is pattern matching. The function *summ* is defined in three parts, the first case for the empty list,

the second case for the singleton list and the third as a catch all case. Pattern matching is again used in the explicit case statement as well as the implicit case analysis of the definition of the local function *peval*. The implementation of *summ* also features a *list comprehension*: $[summands\ e \mid e \leftarrow es]$, the list of *summands* of e, where e is drawn from the list es. The function *mergeLists* takes a list of sorted lists and merges them into a single sorted list. The function *peval* partially evaluates the merged summands by summing the literals. The function *summands*, used in the definition of *summ* is given here:

$$
\begin{aligned}
&summands :: Expr \to [Expr] \\
&summands\ (Lit\ 0) \quad\ = [\,] \\
&summands\ (Summ\ ts) = ts \\
&summands\ t \qquad\quad = [t]
\end{aligned}
$$

An analogous function can be defined for the multiplication of a list of expressions. The following function follows a very similar form to *summ*,

$$
\begin{aligned}
&prod :: [Expr] \to Expr \\
&prod\ [\,] \ = one \\
&prod\ [f] = f \\
&prod\ fs \\
&\quad \mid zero \in fs' = zero \\
&\quad \mid otherwise = \\
&\quad\ \textbf{case}\ peval\ fs'\ \textbf{of} \\
&\qquad [x] \to x \\
&\qquad xs \ \to Prod\ xs \\
&\ \ \textbf{where}\ fs' = mergeLists\ [factors\ f \mid f \leftarrow fs] \\
&\qquad\quad peval\ (Lit\ m : Lit\ n : fs) = peval\ (Lit\ (m * n) : fs) \\
&\qquad\quad peval\ (Lit\ 1 : fs) \qquad\ = fs \\
&\qquad\quad peval\ fs \qquad\qquad\ = fs
\end{aligned}
$$

This definition illustrates another Haskell language feature, namely *guards*. The first guard, $\mid zero \in fs'$, states that if $Lit\ 0$ is in the merged list of factors fs', then the whole product is zero. The 'where' clause introduces local definitions, similar to the **let** ... **in** expressions in Haskell and other functional languages. The auxiliary function *factors* mirrors the definition of *summands* given above.

The third function that constructs expressions in a canonical form is the operator $(\hat{}\hat{}\hat{})$,

$$
\begin{aligned}
&\textbf{infixr}\ 8\ \hat{}\hat{}\hat{} \\
&(\hat{}\hat{}\hat{}) :: Expr \to Expr \to Expr \\
&Lit\ m\ \hat{}\hat{}\hat{}\ Lit\ n \\
&\quad \mid n \geqslant 0 \quad\ = Lit\ (m \uparrow n) \\
&e\ \hat{}\hat{}\hat{}\ Lit\ n \\
&\quad \mid n \equiv 0 \quad\ = one
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Lit}\ n \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} e \\
&\quad \mid n \equiv 1 \qquad = \mathit{one} \\
&e \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} \mathit{Lit}\ n \\
&\quad \mid n \equiv 1 \qquad = e \\
&(\mathit{Prod}\ \mathit{fs}) \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} e = \mathit{prod}\ [f \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} e \mid f \leftarrow \mathit{fs}] \\
&(e :\!\wedge\!:\ a) \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} b \ = e \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} \mathit{prod}\ [a, b] \\
&a \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} b \qquad\quad = a :\!\wedge\!:\ b
\end{aligned}
$$

This definition again makes heavy use of pattern matching and guards to give a terse definition that includes a handful of arithmetic laws. The final two functions for constructing expressions in canonical form are $(\prec), (\curlyvee) :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}$, however we will omit these definitions as they are trivial.

2.1.3 Higher-order traversal of stream expressions

Many operations on streams expression, particularly in the setting of a proof assistant, are in the form of a traversal. The most important of all the higher-order functions for traversing a data-structure is fold.

$$
\begin{aligned}
&\mathit{foldExpr}\ \mathit{summ}\ \mathit{prod}\ (\mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}})\ (\prec)\ (\curlyvee) = \mathit{fold} \\
&\quad \textbf{where}\ \mathit{fold}\ (\mathit{App}\ f\ \mathit{as}) = \mathit{App}\ f\ [\mathit{fold}\ a \mid a \leftarrow \mathit{as}] \\
&\qquad\qquad \mathit{fold}\ (\mathit{Summ}\ \mathit{ss}) = \mathit{summ}\ [\mathit{fold}\ s \mid s \leftarrow \mathit{ss}] \\
&\qquad\qquad \mathit{fold}\ (\mathit{Prod}\ \mathit{fs}) \ = \mathit{prod}\ [\mathit{fold}\ f \mid f \leftarrow \mathit{fs}] \\
&\qquad\qquad \mathit{fold}\ (a :\!\wedge\!:\ b) \ = (\mathit{fold}\ a) \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} (\mathit{fold}\ b) \\
&\qquad\qquad \mathit{fold}\ (h :\!\prec\!:\ t) \ = (\mathit{fold}\ h) \prec (\mathit{fold}\ t) \\
&\qquad\qquad \mathit{fold}\ (s :\!\curlyvee\!:\ t) \ = (\mathit{fold}\ s) \curlyvee (\mathit{fold}\ t) \\
&\qquad\qquad \mathit{fold}\ t \qquad\quad = t
\end{aligned}
$$

This version of fold for the *Expr* datatype is not a true fold as it will only return values of type *Expr*, rather than an arbitrary type. The function *foldExpr* will traverse the whole of a expression however it will not allow us to transform literals, variables or function applications. For the purpose of transforming an expression into the canonical form describe above, *foldExpr* is perfectly adequate.

$$
\begin{aligned}
&\mathit{reduce} :: \mathit{Expr} \to \mathit{Expr} \\
&\mathit{reduce} = \mathit{foldExpr}\ \mathit{summ}\ \mathit{prod}\ (\mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}})\ (\prec)\ (\curlyvee)
\end{aligned}
$$

The expression transformation *reduce* is our first operator for proof expressions. It will reduce expressions such as $2 * 0 \prec (s * 1) \mathbin{\hat{\wedge}\hat{\wedge}\hat{\wedge}} 1 + 1 + 0$, to $0 \prec s + 1$.

Haskell is a functional language that emphasises the use of higher-order functions such as folds and maps. These functions abstract over the details of a data-structure, so given implementations of *map* for lists, trees and even streams, one can apply *map f* to any of these data-structures. This idea taken further enters the realm of *generic programming*, where a function can be written once, but written in a generic way such that it works on any data-structure. However, unlike a plain parametrically polymorphic

function, a generic function exhibits type specific behaviour. One approach to generic programming (there are many) was proposed by Lämmel and Peyton-Jones and is called "Scrap Your Boilerplate"[6]. The following functions are not generic functions in the generic programming sense, they are for the *Expr* datatype only, however they imitate the style of traversal functions given in the "Scrap Your Boilerplate" library. The key characteristic of these traversal schemes is that they operate over one layer only — they apply a function only to the immediate children in the structure of a value. Such a single layered function can then be used to build a variety of recursive traversal schemes.

```
gmapExprQ :: (Expr → r) → Expr → [r]
gmapExprQ f expr =
  case expr of
    App _ xs → map f xs
    Summ xs → map f xs
    Prod xs  → map f xs
    x :∧: y  → [f x, f y]
    x :≺: y  → [f x, f y]
    x :ϒ: y  → [f x, f y]
    _        → []
```

The function *gmapExprQ* takes a query function of type $(Expr \rightarrow r)$ and applies it to all immediate sub-expressions, collecting the results. It is simply a query mapping function. A single layer traversal of a transformation function of type $(Expr \rightarrow Expr)$ is similar,

```
gmapExprT :: (Expr → Expr) → Expr → Expr
gmapExprT f expr =
  case expr of
    App g xs → App g (map f xs)
    Summ xs → Summ (map f xs)
    Prod xs  → Prod (map f xs)
    x :∧: y  → f x :∧: f y
    x :≺: y  → f x :≺: f y
    x :ϒ: y  → f x :ϒ: f y
    x        → x
```

Note that in the same way that *gmapExprQ* does not query literals and variables, the transformation function is not applied to those data constructors in *gmapExprT*. It is the sub-expressions not the expressions themselves that are being directly transformed.

Suppose that we want to rename some or all of the identifiers in an expression as well as substitute expressions for variables. The following function *mapIdent* takes a function that maps identifiers to expressions.

```
mapIdent :: (Ident → Expr) → Expr → Expr
mapIdent f e =
  case gmapExprT (mapIdent f) e of
```

```
(Var v)     → f v
(App v xs) →
  case f v of
    (Var v') → App v' xs
    _        → error "Invalid function"
x            → x
```

To rename identifier a to b, a mapping f would take `"a"` to *Var* `"b"`. A full expression substitution follows logically. Note that *gmapExprT* is employed in such a way as to produce a bottom-up traversal — substitutions occur at the leaves of the abstract syntax tree before those at the root. In the "Scrap Your Boilerplate" library bottom-up traversals are expressed using the *everywhere* combinator and top-down with *everywhere'*.

```
everywhere :: (∀a.Data a ⇒ a → a) → ∀a.Data a ⇒ a → a
everywhere f x = f (gmapT (everywhere f) x)
everywhere' :: (∀a.Data a ⇒ a → a) → ∀a.Data a ⇒ a → a
everywhere' f x = gmapT (everywhere' f) (f x)
```

2.1.4 Pretty-printing stream expressions

A string in Haskell is represented as list of characters. The complexity of string concatenation, $str_1 +\!\!+ str_2$, is the length of the first argument, str_1. Using concatenation to build up a large string from many parts can thus become very costly. Haskell provides the type class *Show* for representing values as strings. It obviates the time complexity of string concatenation with the type *ShowS*, a type synonym for a function that takes a string and appends it. Concatenation of many strings is then constant time, using function composition. The following is a listing of the *Show* type class and associated functions.

```
type ShowS = String → String
class Show a where
  showsPrec :: Int → a → ShowS
  show :: a → String
  ...
shows :: Show a ⇒ a → ShowS
showChar :: Char → ShowS
showString :: String → ShowS
showParen :: Bool → ShowS → ShowS
```

The class method *showsPrec* takes an additional integer argument in the range 0–11 that represents the surrounding precedence level. The helper function *shows* is equivalent to *showsPrec* 0 and the function *showParen* conditionally wraps a string in parenthesis.

The following is the instance of the *Show* type class for the *Expr* datatype. A sufficient instantiation defines the function *showsPrec* for each data-constructor. The definitions for $(:\wedge:)$ and $(:\Upsilon:)$ have been omitted as they follow similarly from $(:\prec:)$.

```
instance Show Expr where
  showsPrec d (Lit i)          = showsPrec d i
  showsPrec _ (Var v)          = showString v
  showsPrec _ (App f [])       = showString f
  showsPrec d (App f args)     = showParen (d > app_prec) $
    showString f ∘ showChar ' ' ∘ showOperands app_prec " " args
    where app_prec = 10
  showsPrec d (Summ sums) = showParen (d > 6) $ showSumm sums
  showsPrec d (Prod (Lit i : prods))
    | i ≡ −1 = showParen (d > neg_prec) $
               showChar '-' ∘ showProd prods
    | i < 0  = showParen (d > neg_prec) $
               showChar '-' ∘ showProd (Lit (abs i) : prods)
    where neg_prec = 6
  showsPrec d (Prod prods)   = showParen (d > 7) $ showProd prods
  showsPrec d (h :≺: t)      = showParen (d > cons_prec) $
    showsPrec (cons_prec + 1) h ∘
    showString " <: " ∘
    showsPrec cons_prec t
    where cons_prec = 5
  showsPrec d (a :^: b)      = ...
  showsPrec d (s :Υ: t)      = ...
```

The absence of a negation and subtraction operator from the abstract syntax datatype *Expr* means that pretty-printing sums and products is more complex. Note that the right associativity of the *cons* operator is encoded in the definition of *showsPrec*. This ensures that $a :\prec: (b :\prec: c)$ is printed as $a \prec b \prec c$. The same is done for the operators $(\hat{\ }\hat{\ }\hat{\ })$ and (Υ). The left associativity of addition and multiplication is irrelevant as sums and products are represented as lists. The arguments to a function application are also represented as a list. This common pattern is abstracted into the following auxiliary function,

```
showOperands :: Int → String → [Expr] → ShowS
showOperands _    _  [] = id
showOperands prec op xs =
  foldr1 (λs r → s ∘ showString op ∘ r) (map (showsPrec (prec + 1)) xs)
```

The function *showOperands* takes a precedence, a string to intersperse and a list of expressions. The expressions are pretty-printed and then assembled into a sequence, interspersed with the operator string. The function *foldr1* is a variant of the right

associative fold operator for lists; it requires a non-empty list. The following function is a simple instantiation of *showOperands*.

```
showProd [] = shows 1
showProd xs = showOperands 7 " * " xs
```

The lack of a primitive subtraction results in a more complex auxiliary function for pretty-printing sums. Subtraction is represented as multiplication by -1, however multiplication by any other negative integer is also a negated term. For aesthetic reasons the integer constant in a summ is printed at the end of the expression, but all other summands are printed in the order in which they appear.

```
showSumm []                 = shows 0
showSumm [x]                = showsPrec 7 x
showSumm (Lit i : xs)
  | i < 0                   = showSumm xs ∘
                              showString " - " ∘ shows (abs i)
  | otherwise               = showSumm xs ∘
                              showString " + " ∘ shows i
showSumm (x : xs@(y : ys)) = showsPrec 7 x ∘
  case y of
    Prod (Lit i : zs)
      | i ≡ −1              → showString " - " ∘
                              showSumm (Prod zs : ys)
      | i < 0               → showString " - " ∘
                              showSumm ((Prod (Lit (abs i) : zs)) : ys)
    _                       → showString " + " ∘ showSumm xs
```

The function *showSumm* is further illustration of the expressive power of pattern matching and guards. It features an *as-pattern*, which is of form *var*@*pat*, where *var* is the name bound to the value matched by the pattern *pat*. Here, the name *xs* refers to the tail of the input list and y and *ys* are the head and tail of this.

2.2 Type Checking

2.2.1 Introduction of Monads

Haskell is a pure functional language, as has been noted previously. The 'pure' label is key here. The implication is that Haskell has no concept of state, mutation or input/output — anything that manipulates the *Real–World*. If Haskell was left at this point it would be utterly useless as it would have no means by which to communicate with a program user. As Peyton-Jones has previously put it, the only way one would know that a purely functional program is running, is when the computer becomes hot. Referential transparency is what makes purity worthwhile. The guarantee that no matter when or how many times you apply a function to the same arguments, you will always receive the same answer in return. It is often very important to know that executing a computation will not mutate a shared piece of data — that it is free of *side-effects*. Nevertheless, input/output and managing state is a crucial requirement for real world programming.

To tackle what's known as the awkward squad (input/output, concurrency, exceptions etc.) Haskell uses a concept called monads. Monads are a means for structuring and sequencing computations. Consider the *Maybe* datatype, a standard Haskell datatype,

```
data Maybe a = Just a | Nothing
```

Suppose that we have three functions $f :: a \rightarrow Maybe\ b$, $g :: b \rightarrow Maybe\ c$, $h :: c \rightarrow Maybe\ d$. To sequence these functions we could write the following code,

```
case f a of
  Nothing → Nothing
  Just b  →
    case g b of
      Nothing → Nothing
      Just c  → h c
```

This is very inelegant as we are forced to deal with the two cases after each function application. If f, g and h were simply $f :: a \rightarrow b$, $g :: b \rightarrow c$ and $h :: c \rightarrow d$ then composition would be trivial: $h \circ g \circ f$.

A monad is primarily comprised on two functions, a function *return* that lifts a value into a monadic value and a function $(\gg\!=)$, pronounced 'bind', that combines monadic values with computations that produce monadic values. For generality Haskell implements the monad concept as a type class, given here:

```
class Monad m where
  (>>=)  :: m a → (a → m b) → m b
  return :: a → m a
```

The *Maybe* datatype is an instance of the *Monad* type class,

```
instance Monad Maybe where
  Nothing >>= f = Nothing
```

$$
\begin{aligned}
(\mathit{Just}\ x) \gg\!= f &= f\ x \\
\mathit{return}\ x \quad\ &= \mathit{Just}\ x
\end{aligned}
$$

We can now write the above composition of f, g and h as $\mathit{return}\ a \gg\!= f \gg\!= g \gg\!= h$ — much better! Haskell provides syntactic-sugar called 'do-notation' that imitates imperative code and is appropriate for sequencing more complex monadic computations.

$$
\begin{aligned}
&\mathit{action}\ a = \mathbf{do}\ b \leftarrow f\ a \\
&\qquad\qquad\quad\ c \leftarrow g\ b \\
&\qquad\qquad\quad\ d \leftarrow h\ c \\
&\qquad\qquad\quad\ \mathit{return}\ d
\end{aligned}
$$

There are are a number of datatypes that form useful monads. These are defined in the standard Haskell libraries. The first of these is *State*,

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\{\,\mathit{runState} :: (s \rightarrow (a, s))\,\} \\
&\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \mathbf{where} \\
&\quad \mathit{get} :: m\ s \\
&\quad \mathit{put} :: s \rightarrow m\ () \\
&\mathit{modify} :: \mathit{MonadState}\ s\ m \Rightarrow (s \rightarrow s) \rightarrow m\ ()
\end{aligned}
$$

A value of type $\mathit{State}\ s\ a$ is a computation that takes a state value of type s and returns a value of type a along with a new state. When state computations are sequenced together using the monad operator ($\gg\!=$), the state is 'threaded' through the program. The type class *MonadState*, of which the *State* datatype is an instance, defines a number of functions for working in the state monad. The function *get* retrieves the current state, *put* overwrites it and *modify* applies a function to it.

There are two specializations of the *State* monad, the *Reader* and *Writer* monads. A partial reproduction of the datatypes and associated type classes are given here.

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{Reader}\ r\ a = \mathit{State}\{\,\mathit{runReader} :: (r \rightarrow a)\,\} \\
&\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadReader}\ r\ m \mid m \rightarrow r\ \mathbf{where} \\
&\quad \mathit{ask} :: m\ r \\
&\quad \ldots \\
&\mathbf{newtype}\ \mathit{Writer}\ w\ a = \mathit{Writer}\{\,\mathit{runWriter} :: (a, w)\,\} \\
&\mathbf{class}\ (\mathit{Monoid}\ w, \mathit{Monad}\ m) \Rightarrow \mathit{MonadWriter}\ w\ m \mid m \rightarrow w\ \mathbf{where} \\
&\quad \mathit{tell} :: w \rightarrow m\ () \\
&\quad \ldots
\end{aligned}
$$

A reader captures the concept of a computation that reads from a shared environment, a value of type r, and returns a value of type a. The *ask* method of the *MonadReader* type class reads the environment value. A writer captures the concept of a computation that produces output (such as logging) that is secondary to the return value. The definition of the the *MonadWriter* type class requires that the type of the output is an instance of the *Monoid* type class. A monoid is a type that has an addition operator and an identity value. A list is a monoid, with concatenate as its operator and the empty list as the identity.

Monad Transformers

Monad transformers are a variation on regular monads. Suppose that your computation reads from a shared environment, but it also returns values of type *Maybe a*. Therefore your computation has the type $(r \rightarrow Maybe\ a)$. This problem is solved by using a monad transformer variant of the reader monad.

$$\textbf{newtype}\ ReaderT\ r\ m\ a = ReaderT\{\, runReaderT :: r \rightarrow m\ a \,\}$$

A computation of the form $(r \rightarrow Maybe\ a)$ can now be given the type *ReaderT r Maybe a*. The *ReaderT* datatype is also an instance of the *Monad* and *MonadReader* type classes.

$$\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ReaderT\ r\ m)\ \textbf{where}\ ...$$
$$\textbf{instance}\ Monad\ m \Rightarrow MonadReader\ r\ (ReaderT\ r\ m)\ \textbf{where}\ ...$$

It is possible to bolt together multiple monads to form a stack-like type. This gives rise to the idea of inner and outer monads. In the above example, *Maybe* is the inner monad. Suppose that we have a computation in the inner monad that we wish to sequence in the combined monad; the final type class of interest, *MonadTrans*, solves this problem.

$$\textbf{class}\ MonadTrans\ t\ \textbf{where}$$
$$lift :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a$$

The class method *lift*, lifts a computation in the inner monad to the combined monad. To continue with the running example, if we have a function f of type $b \rightarrow Maybe\ a$ which we wish to run in the monad *ReaderT r Maybe a* then $lift\ (f\ b)$ will return a reader computation that simply ignores the environment parameter.

2.2.2 Type Checking Algorithm

The typing algorithm implemented here is a *type reconstruction* algorithm that calculates the *principal type* of a stream expression. Given a stream expression it infers the types without any explicit type-annotations. There are four different types for integer streams,

$$\begin{aligned}\textbf{data}\ Ty = {} & TyInteger \\ | {} & TyStream \\ | {} & TyArrow\ Ty\ Ty \\ | {} & TyVar\ Int \\ & \textbf{deriving}\ (Eq)\end{aligned}$$

integers, streams, functions and type variables. The type checking procedure is not as trivial as one would initially expect, particularly as we have so few types. The cause is that the abstract syntax for streams has a limited amount of polymorphism, namely that an integer literal can be either a plain integer or a constant stream. Furthermore the arithmetic operators can be applied to both integers and streams.

Given a term t and a context Γ, the algorithm calculates a set of typing constraints. A term t is typeable iff the constraints are satisfiable. The *constraint set* C that is

produced, is a set of equations between type expressions — expressions of the datatype Ty. The solution to the constraint set is a substitution that unifies every equation in the set; a substitution σ unifies $S = T$ if $\sigma S \equiv \sigma T$.

This constraint-based approach to typing can be formalised by the *constraint typing relation* $\Gamma \vdash t : T \mid C$. This says that a term t in context Γ has type T when constraints C are satisfiable. The relation is defined by the following rules,

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \{\}}(\text{CT-Var}) \qquad \frac{\mathit{unique}(X)}{\Gamma \vdash n : X \mid \{\}}(\text{CT-Lit})$$

$$\frac{\Gamma \vdash t : T \mid C \quad C' = C \cup \{T = \text{Stream}\}}{\Gamma \vdash \texttt{head}\ t : \text{Int} \mid C'}(\text{CT-Head}) \qquad \frac{\Gamma \vdash t : T \mid C \quad C' = C \cup \{T = \text{Stream}\}}{\Gamma \vdash \texttt{tail}\ t : \text{Stream} \mid C'}(\text{CT-Tail})$$

$$\frac{\Gamma \vdash h : T_1 \mid C_1 \quad \Gamma \vdash t : T_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{T_1 = \text{Int}, T_2 = \text{Stream}\}}{\Gamma \vdash h \prec t : \text{Stream} \mid C'}(\text{CT-Cons})$$

$$\frac{f : T_1 \in \Gamma \quad \Gamma \vdash t : T_2 \mid C \quad \mathit{unique}(X) \quad C' = C \cup \{T_1 = T_2 \to X\}}{\Gamma \vdash f\ t : X \mid C'}(\text{CT-App})^*$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \quad \Gamma \vdash t_2 : T_2 \mid C_2 \quad \mathit{unique}(X) \quad C' = C_1 \cup C_2 \cup \{X = T_1, X = T_2\}}{\Gamma \vdash t_1 + t_2 : X \mid C'}(\text{ST-Summ})^*$$

New type variables must be unique; this condition is represented by $\mathit{unique}(X)$. Each instance of an integer literal is given a fresh type variable as its type can only be determined by the context in which it is used. The rules for function application and summation are given in a simplified form for notational convenience, however it should be clear as to how these rules would be extended to an arbitrary number of arguments. The rules for multiplication and exponentiation follow from the rule for summation, similarly the rule for interleaving follows from the rule for cons.

The implementation of the constraint-based type checking algorithm makes heavy use of monads and monad transformers. This allows all the minor details to be hidden away and results in a very clean implementation. The implementation uses all the monads we have seen so far: reader, writer, state and maybe. As the reader, writer and state monads are kindred monads, there is a datatype named *RWS* that combines these three into one. The following is the monad transformer variant of this datatype,

$$\textbf{newtype}\ \mathit{RWST}\ r\ w\ s\ m\ a = \mathit{RWST}\{\,\mathit{runRWST} :: (r \to s \to m\ (a, s, w))\,\}$$

To implement the type checker the reader monad is used for the typing context Γ that maps identifiers to types, the state monad is used to generate fresh type variables, the writer monad is used to gather the constraint set and the maybe monad is for when the typing algorithm might fail. This is all represented in the following type,

$$\textbf{type}\ Checker\ a = RWST\ (Map\ Ident\ Ty)\ [(Ty, Ty)]\ Int\ Maybe\ a$$

The type $Map\ Ident\ Ty$ is a finite map where values of type $Ident$ are keys and an equation in a constraint set is represented by a 2-tuple (Ty, Ty).

The generation of a fresh variable is accomplished with the following auxiliary function,

$$\begin{aligned}
&getUVar :: Checker\ Ty \\
&getUVar = \textbf{do}\ i \leftarrow get \\
&\qquad\qquad\quad modify\ (+1) \\
&\qquad\qquad\quad return\ (TyVar\ i)
\end{aligned}$$

The state monad holds an integer counter which is incremented by $getUVar$ each time a new type variable is created. The lookup of a type in the context is achieved with the following one-liner,

$$\begin{aligned}
&lookupType :: Ident \rightarrow Checker\ Ty \\
&lookupType\ v = ask \gg\!= M.lookup\ v
\end{aligned}$$

The class method $ask :: m\ r$ from the type class $MonadReader$ retrieves the context and is passed to the lookup function for the finite map, which has the following type signature.

$$lookup :: (Monad\ m, Ord\ k) \Rightarrow k \rightarrow Map\ k\ a \rightarrow m\ a$$

If the lookup fails, the function invokes the *fail* method of the *Monad* type class. For the *Maybe* monad, *fail* returns *Nothing*. The functions $getUVar$ and $lookupType$ are used to give a concrete implementation of the above typing rules. The condition $unique(X)$ becomes a call to $getUVar$ and similarly the condition $x : T \in \Gamma$ becomes a call to $lookupType$.

Each typing rule becomes a case in the function *tycheck*. This implementation follows the implemenation of a constraint-based type checker, given in Standard ML in the textbook "Types and Programming Languages" [9]. The two simple base cases are given here.

$$\begin{aligned}
&tycheck :: Expr \rightarrow Checker\ Ty \\
&tycheck\ (Lit\ _) = getUVar \\
&tycheck\ (Var\ v) = lookupType\ v
\end{aligned}$$

The functions *head* and *tail* are special cases, and thus have typing rules independent of the rule for general function application.

$$\begin{aligned}
tycheck\ (App\ \texttt{"head"}\ [x]) = \textbf{do}\ &ty \leftarrow tycheck\ x \\
&tell\ [(ty, TyStream)] \\
&return\ TyInteger
\end{aligned}$$

$$
\begin{aligned}
&\mathit{tycheck}\ (\mathit{App}\ \texttt{"head"}\ \_) &&= \mathit{lift}\ \mathit{Nothing}\\
&\mathit{tycheck}\ (\mathit{App}\ \texttt{"tail"}\ \mathit{xs}) &&= ...
\end{aligned}
$$

If the *head* function appears with anything other than a single argument, the algorithm signals failure. The implementation of the typing rule for *tail* has been omitted for brevity. Function application has the most complex typing rule:

$$
\begin{aligned}
&\mathit{tycheck}\ (\mathit{App}\ f\ [\,]) = \mathit{lookupType}\ f\\
&\mathit{tycheck}\ (\mathit{App}\ f\ \mathit{xs}) = \mathbf{do}\ \mathit{funTy} \leftarrow \mathit{lookupType}\ f\\
&\qquad\qquad\qquad\qquad\quad \mathit{tys} \leftarrow \mathit{mapM}\ \mathit{tycheck}\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad\quad \mathit{ty} \leftarrow \mathit{getUVar}\\
&\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{funTy}' = \mathit{foldr}\ \mathit{TyArrow}\ \mathit{ty}\ \mathit{tys}\\
&\qquad\qquad\qquad\qquad\quad \mathit{tell}\ [(\mathit{funTy}, \mathit{funTy}')]\\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ \mathit{ty}
\end{aligned}
$$

The implementation makes use of the monadic library function *mapM*,

$$\mathit{mapM} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow [a] \rightarrow m\ [b]$$

It is the monadic version of *map*; the monadic side-effects are sequenced from left to right along the list. In the usage within the type checker, mutations to the state and output to the writer may occur and if so, they will happen in the sequence dictated by the given list. The use of the right associative list fold function, *foldr TyArrow ty tys*, builds a curried function type where *ty* is the final return type.

The typing rules for addition and multiplication are identical. All the operands must be of the same type but it is not possible to determine that type from just one operand. Thus a fresh type variable must be given to the whole expression to unify all the operands.

$$
\begin{aligned}
&\mathit{tycheck}\ (\mathit{Summ}\ \mathit{xs}) = \mathbf{do}\ \mathit{ty} \leftarrow \mathit{getUVar}\\
&\qquad\qquad\qquad\qquad\quad \mathit{tys} \leftarrow \mathit{mapM}\ \mathit{tycheck}\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad\quad \mathit{tell}\ (\mathit{map}\ ((,)\ \mathit{ty})\ \mathit{tys})\\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ \mathit{ty}\\
&\mathit{tycheck}\ (\mathit{Prod}\ \mathit{xs}) = ...
\end{aligned}
$$

The monadic function *mapM* features again and the other item of note is the construction of the list of constraints. The expression $((,)\ ty)$ is the partial application of the pairing operator; this expression has the type $a \rightarrow (Ty, a)$.

The final rules are for exponentiation, cons and interleaving. The typing rule for exponentiation is a specific form of the rules for addition and multiplication.

$$
\begin{aligned}
&\mathit{tycheck}\ (x :\hat{}: y) = \mathbf{do}\ \mathit{ty} \leftarrow \mathit{getUVar}\\
&\qquad\qquad\qquad\qquad\quad \mathit{tyX} \leftarrow \mathit{tycheck}\ x\\
&\qquad\qquad\qquad\qquad\quad \mathit{tyY} \leftarrow \mathit{tycheck}\ y\\
&\qquad\qquad\qquad\qquad\quad \mathit{tell}\ [(\mathit{ty}, \mathit{tyX}), (\mathit{ty}, \mathit{tyY})]
\end{aligned}
$$

```
                                  return ty
tycheck (h :≺: t) = do tyH ← tycheck h
                       tyT ← tycheck t
                       tell [(tyH, TyInteger), (tyT, TyStream)]
                       return TyStream
tycheck (s :ϒ: t) = ...
```

Again the implementation for interleaving has been omitted as it follows the implementation of cons.

The function *tycheck* is a very good demonstration of the power of monads. The code for each typing rule has a surprisingly close correspondence to the mathematical definition given above. The writer monad even conceals the concatenation of constraints from recursive calls.

To execute the *tycheck* function, an initial typing context must be supplied. The following function *buildContext*, builds the map of identifiers to types in list form, which can then be turned into a *Map* data-structure.

```
buildContext :: Expr → [(Ident, Ty)]
buildContext = flip zip (map TyVar [0..]) ∘
               filter (λx → x ≢ "head" ∧ x ≢ "tail") ∘
               nub ∘
               sort ∘
               extractIdents
  where
    extractIdents (Var v)       = [v]
    extractIdents e@(App f _) = f : concat (gmapExprQ extractIdents e)
    extractIdents e             = concat (gmapExprQ extractIdents e)
```

The implementation of this function exemplifies the *pointfree* style of functional programming, where explicit references to values are minimized and functions are defined in terms of function composition. The implementation describes a flow of operations: the identifiers are extracted, then sorted, the duplicates are removed (using the function *nub*), as are `"head"` and `"tail"` and finally each identifier is paired with a type variable. The local function *extractIdents* traverses the input expression using the function *gmapExprQ*. The expression ([0..]) produces an infinite list of increasing integers starting at zero. By mapping the data-constructor *TyVar* over this infinite list we get a list of all the type variables. By zipping this list with the list of identifiers stripped from the input expression, we give each identifier a unique type variable. This is a common idiom is Haskell. As the language is lazy, we can describe excessively large or even infinite values with the intention of using only a part of them. The function *zip* terminates at the end of the shortest list, so it is of no concern that the other list is infinite in length. The function *flip* reverses the arguments of *zip* so that the zip can be incorporated in the function composition.

The output of the function *tycheck* is a list of pairs representing a constraint set. If this set can be unified then the initial expression is typeable. The process of unifying

the constraint set, produces a substitution which can be applied to the typing context to give each identifier a principal type.

```
unify :: [(Ty, Ty)] → [(Ident, Ty)] → Maybe [(Ident, Ty)]
unify [] ctx = Just ctx
unify ((s, t) : cs) ctx =
  if s ≡ t then
    unify cs ctx
  else if isTyVar s ∧ ¬ (s ∈ fv t) then
    unify (mapPair (substTy s t) cs) (mapSnd (substTy s t) ctx)
  else if isTyVar t ∧ ¬ (t ∈ fv s) then
    unify (mapPair (substTy t s) cs) (mapSnd (substTy t s) ctx)
  else
    case (s, t) of
      (TyArrow a b, TyArrow c d) → unify ([(a, c), (b, d)] ++ cs) ctx
      _                          → Nothing
  where
    fv (TyVar v)       = [v]
    fv (TyArrow a b) = fv a ++ fv b
    fv _               = []
    substTy s@(TyVar v) r ty =
      case ty of
        TyVar v'      → if v ≡ v' then r else ty
        TyArrow a b → TyArrow (substTy s r a) (substTy s r b)
        _             → ty
    mapPair f = map (λ(a, b) → (f a, f b))
    mapSnd f = map (λ(a, b) → (a, f b))
```

The local function *fv*, extracts a list of type variables from a type and the local function *substTy* takes a type variable to search for, a type to substitute and a type to modify. The second and third conditions which check if a type variable is present in the free variables of another type is called the *occurs check*. This is required to ensure that a cyclic substitution is not created, resulting in an infinite type. For example take the expression $f\ f$, with an initial context of $[(\texttt{"f"}, TyVar\ 0)]$. The type checking rules would produce the constraint set: $[(TyVar\ 0, TyArrow\ (TyVar\ 0)\ (TyVar\ 1))]$. The occur check would catch this case and prevent us from trying to construct an infinite type for f.

2.3 Substitutions and Matchings

We will now cover a core component of the proof assistant, the matching of one expression against another. Matching is required so that rules and laws can be applied to expressions. Two expressions can match if the have the same 'structure', even if they use different identifiers. For example the expression $f\ (head\ x) \prec map\ f\ (tail\ x)$ should match $g\ (head\ (repeat\ 0)) \prec map\ g\ (tail\ (repeat\ 0))$, as we can form a substitution that when applied to the first expression, results in the second expression. In this case the substitution would map the variable f to the variable g and the variable x to the expression $repeat\ 0$. Matching is straightforward for expressions that predominantly involve function applications, however it becomes more complex when dealing with arithmetic expressions, for example the matching of $x + 1$ with $a + b + 2$.

2.3.1 Substitutions

The type of a substitution, as discussed informally above, is a map from identifiers to expressions. The renaming of identifier x to identifier y, is a map from x to the expression $Var\ y$.

$$\textbf{type}\ Subst = Map\ Ident\ Expr$$

The data-structure *Map* is provided by the Haskell standard libraries and is implemented as a balanced binary tree. Therefore the lookup of an identifier in a substitution has logarithmic complexity. An alternative data-structure for the type *Subst* could have been $[(Ident, Expr)]$ however a lookup operation then has linear complexity.

The following function returns the expression that a given identifier is mapped to in a substitution.

$$binding :: Subst \rightarrow Ident \rightarrow Expr$$
$$binding\ s\ v = findWithDefault\ (Var\ v)\ v\ s$$

If an identifier v has no mapping then the expression $Var\ v$ is returned. The function *binding* is then used to apply a substitution to an expression.

$$applySubst :: Subst \rightarrow Expr \rightarrow Expr$$
$$applySubst\ s = mapIdent\ (binding\ s)$$

The higher-order function *mapIdent*, defined previously, is used here to implement the substitution.

Any computation that updates a substitution will be of the form $Subst \rightarrow ... \rightarrow Subst$ or $Subst \rightarrow ... \rightarrow (a, Subst)$, as a substitution is an object of state. In anticipation of this, the following type synonym is defined,

$$\textbf{type}\ SubstState\ a = StateT\ Subst\ Maybe\ a$$

The state monad is used to manage the 'mutation' of a substitution and the maybe monad is used to model failure, such as a conflicting substitution. This monadic approach will be particularly useful when implementing functions for matching.

The final operator needed is one to extend a substitution with an mapping,

```
extendSubstM :: (Ident, Expr) → SubstState ()
extendSubstM (v, Var v') | v ≡ v' = return ()
extendSubstM (v, e) = do
  s ← get
  case lookup v s of
    Just e'  → guard (e ≡ e')
    Nothing → modify (M.insert v e)
```

A mapping is ignored (*return* ()), if an identifier maps to a variable of the same name. If the identifier is already mapped to a different expression then the extension fails. The function *guard* implements this behaviour, however we will defer the discussion of this.

The monadic function *extendSubstM* can be *run* with the following function, One of the auxiliary functions for the state transformer monad is $execStateT$,

```
execStateT :: Monad m ⇒ StateT s m a → s → m s
```

It runs a state monad computation for the purpose of retrieving the end state, in this case the final substitution.

```
extendSubst :: Subst → (Ident, Expr) → Maybe Subst
extendSubst s ie = execStateT (extendSubstM ie) s
```

2.3.2 Expression Matching

Expression matching is implemented in the function *matchM* along with several auxiliary functions. The implementation uses the combined state and maybe monad, defined for substitutions. The following are the two base cases,

```
matchM :: Expr → Expr → SubstState ()
matchM (Lit i1) (Lit i2) = guard (i1 ≡ i2)
matchM (Var i) x         = extendSubstM (i, x)
```

The function *guard* was also used in the implementation of *extendSubstM* to 'guard' against mapping an identifier to two different expressions. It is a library function defined as follows,

```
guard :: MonadPlus m ⇒ Bool → m ()
guard p = if p then return () else mzero
```

It makes use of *MonadPlus*, yet another monad related type class. Instances are monads that support choice and failure.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

The datatype *Maybe* is an instance of *MonadPlus*, with *mzero* = *Nothing* and *mplus* as a left-biased choice. Furthermore, a monad transformer is an instance of *MonadPlus* if its inner monad is an instance of *MonadPlus*. Therefore as *Maybe* is an instance of *MonadPlus*, so is *SubstState*. Thus the first case of *matchM* reads as: a literal matches a literal if the integers are equal, otherwise the whole match fails.

A function application matches another function application if the function names match and if they have the same number of arguments, all of which match. Matching two equal length lists of expressions can be accomplished with the monadic library function *zipWithM_*,

$$zipWithM\_ :: Monad\ m \Rightarrow (a \to b \to m\ c) \to [a] \to [b] \to m\ ()$$

This is the monadic version of the *zipWith* function for lists. The underscore in the name of the function is the Haskell nomenclature for monadic functions that discard the output, thus the true purpose being the side-effects rather than the final result. In this case, the desired effect of (*zipWithM_ matchM*) is an updated substitution.

Consider the matching of $f\ (head\ a)$ with $g\ (head\ b)$. Here f needs to be mapped to g and a to b. These two expressions should indeed match, however $f\ (head\ a)$ with $f\ (tail\ a)$ should not. A substitution mapping *head* to *tail* is not valid in our understanding of the system. We cannot simply require that to match, the function names should be equal, as this would disqualify our first example. The approach taken here is to make a syntactic distinction between *variable identifiers* and *constant identifiers*. An identifier string is taken to be a lower case letter followed by zero or more alphanumeric characters. Variable identifiers are a subset of strings, comprised of a lowercase letter followed by zero or more digits. Thus f, x and *a1* are all variable identifiers that can be identifiers in a substitution mapping and *head*, *nat* and *map* are all constant identifiers. This leads to the following implementation of function application matching.

```
matchM (App f xs) (App g ys) =
  guard (length xs ≡ length ys) ≫
  if all isDigit (tail f) then do
    extendSubstM (f, Var g)
    zipWithM_ matchM xs ys
  else if f ≡ g then
    zipWithM_ matchM xs ys
  else mzero
```

The function *guard* is used again to achieve conditional execution of the rest of the function. The operator (≫) is also featured and it features again in the following definitions,

$$\begin{aligned} matchM\ (a :\wedge: b)\ (c :\wedge: d)\ &= matchM\ a\ c \gg matchM\ b\ d \\ matchM\ (a :\prec: b)\ (c :\prec: d) &= matchM\ a\ c \gg matchM\ b\ d \\ matchM\ (a :\curlyvee: b)\ (c :\curlyvee: d) &= matchM\ a\ c \gg matchM\ b\ d \end{aligned}$$

It is an operator similar to the monad bind operator (≫=), except that it discards the value produced by the first monadic computation. It simply sequences two monadic actions, $(\gg) :: m\ a \to m\ b \to m\ b$.

The final task in expression matching is to match addition and multiplication expressions. Addition and multiplication are associative and commutative operators, which makes this task difficult. Furthermore, literals can be viewed in their atomic form or as a compound addition (or multiplication) of several literals. Take for example the matching of $x + 1$ with $y + 2$. One could take a restrictive approach, similar to the matching rule for function application,

$$
\begin{aligned}
&\mathit{matchM}\ (\mathit{Summ}\ \mathit{xs})\ (\mathit{Summ}\ \mathit{ys}) = \mathbf{do}\\
&\quad \mathit{guard}\ (\mathit{length}\ \mathit{xs} \equiv \mathit{length}\ \mathit{ys})\\
&\quad \mathit{zipWithM\_}\ \mathit{matchM}\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

This says that two sums will only match if they have an identical number of summands and the paired summands match. As long as the terms to be matched are in canonical form, this simplistic implementation will successfully match many expressions. It will not however match $x + 1$ with $y + 2$, as the variable x matches with the variable y, but not 1 with 2. Alternatively, the expression $y + 2$ can be viewed as $y + 1 + 1$ and now it is clear that $x + 1$ can be matched to this if we map x to $y + 1$ in the substitution.

Take as a further example, the matching of $x + 1$ with $y + z + 1$. Here the two expressions having differing numbers of summands. Despite this we can match x to $y + z$. The associativity of addition is what comes in to play. The situation is more complex when we involve more than one variable, such as the matching of $a + b$ with $c+d+1$. The commutativity of addition means that several substitutions can be formed: $a \to c$ and $b \to d+1$; $a \to d$ and $b \to c+1$; $a \to c+d$ and $b \to 1$; $a \to 1$ and $b \to c+d$.

The 'philosophy' of this project is to create a simple tool with a handful of tools for manipulation and proof. The aim is to be *sound* but not necessarily *complete*. The ideal is for proof commands to simply do-the-right-thing (most of the time).

For an addition (or multiplication) expression x to match addition (or multiplication) expression y, there must be at least as many operands in y. A variable in x can be mapped to several operands in y but it cannot be mapped to none. All the non-variable operands of x must be matched in y and what remains of y must be matched by the variables of x. This procedure is implemented by the following code,

$$
\begin{aligned}
&\mathit{matchM}\ (\mathit{Summ}\ \mathit{xs})\ (\mathit{Summ}\ \mathit{ys}) = \mathbf{do}\\
&\quad \mathit{guard}\ (\mathit{length}\ \mathit{xs} \leqslant \mathit{length}\ \mathit{ys})\\
&\quad \mathbf{let}\ (\mathit{vars}, \mathit{xs}') = \mathit{partition}\ \mathit{isVar}\ \mathit{xs}\\
&\quad \mathit{rest} \leftarrow \mathit{matchSummsM}\ \mathit{xs}'\ \mathit{ys}\\
&\quad \mathit{matchVarsM}\ \mathit{Summ}\ \mathit{vars}\ \mathit{rest}\\
&\mathit{matchM}\ (\mathit{Prod}\ \mathit{xs})\ (\mathit{Prod}\ \mathit{ys}) \quad = \mathbf{do}\\
&\quad \mathit{guard}\ (\mathit{length}\ \mathit{xs} \leqslant \mathit{length}\ \mathit{ys})\\
&\quad \mathbf{let}\ (\mathit{vars}, \mathit{xs}') = \mathit{partition}\ \mathit{isVar}\ \mathit{xs}\\
&\quad \mathit{rest} \leftarrow \mathit{matchProdsM}\ \mathit{xs}'\ \mathit{ys}\\
&\quad \mathit{matchVarsM}\ \mathit{Prod}\ \mathit{vars}\ \mathit{rest}
\end{aligned}
$$

This completes the function *matchM* and all other arguments will fail to match.

```
matchM _ _ = mzero
```

An addition (or multiplication) expression x to be matched is first split into its constituent variables and other terms using the function $partition :: (a \to Bool) \to [a] \to ([a], [a])$. To match a list of expressions against another we use the function *matchListM*.

```
matchListM :: [Expr] → [Expr] → SubstState [Expr]
matchListM []       ys = return ys
matchListM _        [] = mzero
matchListM (x : xs) ys = matchListM' ys >>= matchListM xs
  where matchListM' []       = mzero
        matchListM' (y : ys) = do
          subst ← get
          case execStateT (matchM x y) subst of
            Just subst' → put subst' >> return ys
            Nothing     → matchListM' ys >>= return ∘ (y:)
```

This matches all the expressions from a list xs against the list of expressions ys and returns the left over expressions from ys. The expressions in list xs are tried in order and while there may be a scenario where a different ordering will give a different result, we will ignore this case.

The function *matchListM* is used to define *matchSummsM* and *matchProdsM*. These functions are responsible for the partial matching of integer literals.

```
matchSummsM :: [Expr] → [Expr] → SubstState [Expr]
matchSummsM (Lit a : xs) e =
  case e of
    (Lit b : ys) → do
      guard (a ⩽ b)
      ys' ← matchSummsM xs ys
      return (Lit (b − a) : ys')
    _            → mzero
matchSummsM xs ys = matchListM xs ys
```

For sums, a literal matches any literal greater or equal to itself. For products, a literal matches any literal that is a multiple of it. In both cases the remainder is added to the return list.

```
matchProdsM :: [Expr] → [Expr] → SubstState [Expr]
matchProdsM (Lit a : xs) e =
  case e of
    (Lit b : ys) → do
      guard (b `mod` a ≡ 0)
      ys' ← matchProdsM xs ys
      return (Lit (b `div` a) : ys')
```

```
  _              → mzero
matchProdsM xs ys = matchListM xs ys
```

The final function *matchVarsM* matches the variables of the matching expression against the terms remaining of the expression to be matched.

```
matchVarsM :: ([Expr] → Expr) → [Expr] → [Expr] → SubstState ()
matchVarsM f xs ys = g xs (uncurry (++) (partition isVar ys))
  where
    g []          []       = return ()
    g _           []       = mzero
    g [Var v]     ys       = extendSubstM (v, f ys)
    g (Var v : xs) (y : ys) = extendSubstM (v, y) >> g xs ys
```

The first argument is the function to rebuild an expression from a list of expressions — either *Summ* or *Prod*. The function takes the list of variables and matches each to a single expression, until the final variable, which takes the remaining expressions to be matched against. Note that the list of expressions to be matched against is rearranged so that all the variables are at the start of the list. The expression (*uncurry* (++) (*partition isVar ys*)) separates out the variables and then concatenates them back onto the front of the list. This ensures that when $a + b + 1$ is matched against against $c + d + 2$, $a \to c$ and $b \to d + 1$.

The function *match* runs the *matchM* state computation with an initially empty substitution.

```
match :: Expr → Expr → Maybe Subst
match e1 e2 = execStateT (matchM e1 e2) empty
```

2.4 Expression Parsing

The parser for stream expressions is built using 'Parsec', which is a monadic parser combinator library for Haskell. It can be used to parse context-sensitive grammars that require infinite look-ahead, however it is better suited to LL(1) parsers — a predictive parser than reads the input from left to right and constructs a leftmost derivation using one token of look-ahead. The advantage of Parsec is that a parser can be constructed and used with the same programming language (Haskell) as is used in the rest of the program. A Parsec parser is a first-class value in Haskell, so it is no different to a regular datatype or a function; one could return a parser as the result of a function.

We will build up the complete parser by composing many smaller, specialized parsers, the first of which parses integer literals.

```
number = Parser Expr
number = do { ds ← many1 digit
            ; spaces
            ; return (Lit (read ds))
            } ⟨?⟩ "number"
```

The type *Parser* is a monad and the monadic bind operator ($\gg=$) sequences parsers — $(p \gg= f)$ first applies parser p then applies the function f to the result and finally applies the returned parser. The first line parses one or more digits, where *digit* :: *Parser Char* and *many1* :: *Parser a* → *Parser* [*a*]. The parser *spaces* :: *Parser* () eats up the trailing whitespace. The string of digits is then coerced to an integer with the function *read*. The final point of interest is the operator ⟨?⟩, which is the error combinator that attaches a string label to a parser to assist with error reporting.

The next parser is one that parses an identifier. An identifier is a lowercase letter followed by zero or more alphanumeric characters (including apostrophe).

```
identifier :: Parser String
identifier = do { x ← lower
                ; xs ← many (alphaNum ⟨|⟩ char '\'')
                ; spaces
                ; return (x : xs)
                } ⟨?⟩ "identifier"
```

The Parsec approach to parser construction is particularly elegant as program code ends up bearing a close resemblance to the formal grammar notation. The operator ⟨|⟩ is the predictive choice operator, which will only try the second parser if the first parser consumes no input. The following parser for variables is a further demonstration how parser actions are implemented in Parsec.

```
var :: Parser Expr
var = do { ident@(_ : xs) ← identifier
         ; if all isDigit xs then
```

```
      return (Var ident)
    else
      return (App ident [])
  } <?> "variable ident or constant ident"
```

As discussed in the context of expression matching, variable identifiers are a subset of identifiers, where the tail of the identifier is comprised of zero or more digits. To assist in the delineation of variable identifiers and constant identifiers, the former are constructed as expression variables and the latter are constructed as no argument function applications.

The parser for stream expressions is named *expr*, but first we shall define the parser for atomic expressions,

```
aexpr = number
      <|> var
      <|> between (char '(') (char ')' >> spaces) expr
      <?> "atomic expression"
```

The use of the predictive choice operator $\langle|\rangle$ is safe as the parser *number* starts with a digit, *var* starts with a lowercase letter and the final parser starts with a left parenthesis. The combinator *between* wraps the *expr* parser with parenthesises. The parser for function application is built on top of the parser for atomic expressions.

```
fexpr = do { ident@(_ : xs) <- identifier
           ; args <- many aexpr
           ; if null args && all isDigit xs then
               return (Var ident)
             else
               return (App ident args)
           }
      <|> aexpr
      <?> "function application"
```

Note that there is a resolvable conflict in the predictive choice – both sub-parsers will match an identifier. As per the semantics of the choice operator, if an identifier is found, the first parser will be applied. The zero or more arguments to a function must all be atomic expressions. If there are no arguments and the identifier is of the variable kind, a variable expression is returned, otherwise a function application expression is constructed.

The Parsec library contains a module for parsing expression grammars. The following is an operator table for the operators in the stream expression grammar,

```
table :: OperatorTable Char () Expr
table = [[binary "^"    (^^)                    AssocRight]
        ,[binary "*"    (\x y -> prod [x, y])   AssocLeft]
```

```
    , [prefix  "-"    (λx → prod [Lit (−1), x]),
      binary "+"      (λx y → summ [x, y])                    AssocLeft,
      binary "-"      (λx y → summ [x, prod [Lit (−1), y]]) AssocLeft]
    , [binary "<:"    (≺)                                     AssocRight,
      binary "\\/"    (ϒ)                                     AssocRight]]
  where binary op f assoc = Infix (do {string op; spaces; return f}) assoc
        prefix op f = Prefix (do {string op; return f})
```

The outer list is ordered by descending operator precedence. Each sub-list contains operators with identical precedence levels, but it is not required for them to have the same associativity. The exponentiation operator has the highest precedence and the cons and interleaving operators have the lowest. The local functions *binary* and *prefix* help construct values of the datatype *Operator*, where *Infix* and *Prefix* are data-constructors. Both of these constructors take a parser that returns a function, the function being the operational interpretation of the operator. The functions *summ*, *prod*, (^^^), (≺) and (ϒ) are used to ensure that the output of the parser is an expression already in canonical form.

```
expr :: Parser Expr
expr = buildExpressionParser table fexpr ⟨?⟩ "expression"
```

The final parser for stream expressions is constructed using the Parsec function *buildExpressionParser*. It builds the final expression parser for terms parsed by *fexpr*, with operators from *table*.

2.5 Expression Rewriting

2.5.1 Replacement

To be able to manipulate all or part of an expression we need the concept of sub-expressions, given by the following definition.

```
type SubExpr = (Location, Expr)
data Location = All | Pos Int Location
              deriving (Eq, Show)
```

The type *SubExpr* is the paring of an expression and its location within another expression. The location of a sub-expression within an expression x is recursively defined to be either the whole of x or a location within the i'th child of x.

To replace the sub-expression found at a location with an expression, we define the following function,

```
replace :: Expr → Location → Expr → Expr
replace _ All rexpr = rexpr
```

The replacement of location *All* in expression e with expression e', is simply e'. If the location is not *All* then we must recurse into the expression e.

```
replace (Lit _)  _ _ = error "Lit is atomic"
replace (Var _) _ _ = error "Var is atomic"
```

Literals and variables are atomic expressions and thus there are no sub-expressions that can be replaced. Function application, sums and products store their operands in a list. A location of the form (*Pos i l*) refers to the i'th index in the list of operands.

```
replace (App f args) loc rexpr =
  App f (replaceList loc rexpr args)
replace (Summ ss)    loc rexpr =
  Summ (replaceList loc rexpr ss)
replace (Prod fs)    loc rexpr =
  Prod (replaceList loc rexpr fs)
```

The auxiliary function *replaceList* is defined as follows,

```
replaceList :: Location → Expr → [Expr] → [Expr]
replaceList (Pos i loc) rexpr exprs = updateList i exprs
  where updateList 0 (x : xs) = replace x loc rexpr : xs
        updateList n (x : xs) = x : updateList (n − 1) xs
        updateList 0 [ ] = error "Invalid location"
```

Note that there is mutual recursion between *replace* and *replaceList*. For the binary operators, the location index 0 refers to the left operand and the index 1 refers to the right operand.

```
replace expr (Pos 0 loc) rexpr =
  case expr of
    a :^: b → r a :^: b
    h :≺: t → r h :≺: t
    s :Y: t → r s :Y: t
  where r x = replace x loc rexpr
replace expr (Pos 1 loc) rexpr =
  case expr of
    a :^: b → a :^: r b
    h :≺: t → h :≺: r t
    s :Y: t → s :Y: r t
  where r x = replace x loc rexpr
```

All other locations are invalid.

```
replace _ _ _ = error "Invalid location"
```

2.5.2 Expression Searching

Now that we have a datatype to describe the concept of a sub-expression, we can use this to this to write functions that will search for a desired sub-expression. Take for example the proof of $nat = 2 * nat \curlyvee 2 * nat + 1$. Starting with the right-hand side, we would like to expand the definition of *nat*. Rather than searching for complex sub-expressions, we are simply interested in constant identifiers. The function *findIdent* searches for an identifier in an expression and returns all the function application sub-expressions where the identifier is used.

```
findIdent :: Ident → Expr → [SubExpr]
findIdent f expr =
  (if isMatch expr then (:) (All, expr) else id) rest
  where isMatch (App f' _) | f ≡ f' = True
        isMatch _                   = False
        rest     = concatMap g $
                   zip [0..] (gmapExprQ (findIdent f) expr)
        g (i, x) = [(Pos i l, e) | (l, e) ← x]
```

The expression *zip* $[0..]$ (*gmapExprQ* (*findIdent f*) *expr*) finds all the matching sub-expressions in the children of *expr* and produces a list of type $[(Int, [SubExpr])]$. The local function *g* take a pair $(Int, [SubExpr])$ and updates all the locations. The function $concatMap :: (a \to [b]) \to [a] \to [b]$ applies this to all pairs and collects the results. The function *findIdent* can be used not only to find named streams such as *nat* and *fib*, it can also be used to find applications of functions such as *map* and *zip*. The sub-expression returned will be the whole function application.

To find a specific expression the function *findExpr* is used. Suppose that you want to find all exponentiation sub-expressions, then you could supply $x \uparrow y$ as the search expression. A more specific search for all squares would be $x \uparrow 2$.

```
findExpr :: Expr → Expr → [(Location, Subst)]
findExpr sExpr expr =
  case match sExpr expr of
    Just subst → (All, subst) : rest
    Nothing    → rest
  where rest     = concatMap f $
                   zip [0..] (gmapExprQ (findExpr sExpr) expr)
        f (i, x) = [(Pos i l, s) | (l, s) ← x]
```

The implementation of *findExpr* is very much similar to *findIdent*, however rather than returning a sub-expression, *findExpr* returns a location paired with a substitution. The actual sub-expression found is the result of applying the substitution to the search term.

Suppose that you wish to find a sub-expression that matches $x \uparrow 2 + 1$. If the sub-expression $y \uparrow 2 + y + 1$ is present, *findExpr* will not find a match as the search term does not match the whole sum. Rather than performing the match with *match*, instead we shall use *matchSummsM* so that we can find a match within a larger sum. The expression $x \uparrow 2 + 1$ matches $y \uparrow 2 + y + 1$ with the substitution $x \to y$ and the expression y left over.

```
findSumm :: Expr → Expr → [(Location, Subst, Expr)]
findSumm sExpr@(Summ xs) expr =
  case expr of
    (Summ ys) →
      case runStateT (matchSummsM xs ys) M.empty of
        Just (e, s) → (All, s, Summ e) : rest
        Nothing     → rest
    _             → rest
  where rest     = concatMap f $
                   zip [0..] (gmapExprQ (findSumm sExpr) expr)
        f (i, x) = [(Pos i l, s, e) | (l, s, e) ← x]
findSumm _ _ = error "Search expression must be a sum"
```

A similar definition is given for products in *findProd*. The functions *findSumm* and *findProd* return the left over expression as well as the location and substitution, as returned by *findExpr*.

2.5.3 Rewriting

The function *rewrite* takes a search expression, an expression to search, a replacement expression and a transformation function. It returns a list of expressions that are all the single-step rewrites of the original expression. One possible use case is the transformation

of sub-expressions that match the search expression. Here the search expression and the replacement expression are identical. Another use for the function *rewrite* is the application of a law. For this, the search expression is the left-hand side of the law, the replacement expression is the right-hand side of the law and the transformation function is the identity function. The implementation is as follows,

```
rewrite :: Expr → Expr → Expr → (Expr → Expr) → [Expr]
rewrite sExpr@(Summ _) expr rExpr trans =
  map
    (λ(loc, subst, rest) →
      replace expr loc (summ [trans (applySubst subst rExpr), rest]))
    (findSumm sExpr expr)
rewrite sExpr@(Prod _) expr rExpr trans =
  map
    (λ(loc, subst, rest) →
      replace expr loc (prod [trans (applySubst subst rExpr), rest]))
    (findProd sExpr expr)
rewrite sExpr expr rExpr trans =
  map
    (λ(loc, subst) →
      replace expr loc (trans (applySubst subst rExpr)))
    (findExpr sExpr expr)
```

The functions *findExpr*, *findSumm* and *findProd* are all used to find sub-expressions that match the search expression. The extracted substitution is applied to the replacement expression and the transformation is applied to the result. Finally, *replace* is used with the location information to build the rewritten expression.

Not all transformation functions are guaranteed to succeed. The function *rewriteM* is for use with transformation functions that have the type $Expr \to Maybe\ Expr$.

```
rewriteM :: Expr → Expr → Expr → (Expr → Maybe Expr) → [Expr]
rewriteM sExpr@(Summ _) expr rExpr trans =
  forMaybe (findSumm sExpr expr) $ λ(loc, subst, rest) →
    trans (applySubst subst rExpr) >>=
      return ∘ replace expr loc ∘ summ ∘ (:[rest])
rewriteM sExpr@(Prod _) expr rExpr trans =
  forMaybe (findProd sExpr expr) $ λ(loc, subst, rest) →
    trans (applySubst subst rExpr) >>=
      return ∘ replace expr loc ∘ prod ∘ (:[rest])
rewriteM sExpr expr rExpr trans =
  forMaybe (findExpr sExpr expr) $ λ(loc, subst) →
    trans (applySubst subst rExpr) >>= return ∘ replace expr loc
```

The function $forMaybe :: [a] \to (a \to Maybe\ b) \to [b]$ is the library function $mapMaybe :: (a \to Maybe\ b) \to [a] \to [b]$ with the arguments flipped. It is like the function map,

except that it throws away values that return *Nothing*. The expression (:[*rest*]) is the partial application of the cons operator. This returns a function that cons'es an expression onto the list [*rest*].

2.6 Simplifications and Transformations

2.6.1 Distribution of Expressions

The distribution of multiplication over addition is a common operation. It can be as simple as turning $2 * (x + y)$ into $2 * x + 2 * y$. However, an expression such as $(x^2 + 3 * x + 2) * (x + 4)$ is more complex. The function *mult* takes a list of expressions and 'multiplies them out' to form a sum.

$$
\begin{aligned}
&mult :: [Expr] \to Expr \\
&mult\ ts = summ'\ [prod'\ ts' \mid ts' \leftarrow distr\ [summands\ t \mid t \leftarrow ts]] \\
&\quad \textbf{where}\ prod'\ [x] = x \\
&\qquad\qquad prod'\ x\ \ \ = Prod\ x \\
&\qquad\qquad summ'\ [x] = x \\
&\qquad\qquad summ'\ x = Summ\ x
\end{aligned}
$$

For each expression the list of summands are collected. The list of lists of summands is distributed using the function *distr*, where each element of the first list is cons'ed onto every list of the distribution of the remaining lists.

$$
\begin{aligned}
&distr :: [[a]] \to [[a]] \\
&distr\ [\,] \qquad = [[\,]] \\
&distr\ (x : xs) = [a : as \mid a \leftarrow x, as \leftarrow distr\ xs]
\end{aligned}
$$

The result is a sum of products which is constructed back into an expression using the local functions $summ'$ and $prod'$.

Multiplication distributes over addition and furthermore the following abide laws for interleaving and cons show that the arithmetic operators, multiplication, addition and exponentiation all distribute over interleaving and cons.

$$(s_1 \curlyvee t_1) \oplus (s_2 \curlyvee t_2) \oplus \cdots \oplus (s_n \curlyvee t_n) = (s_1 \oplus s_2 \oplus \cdots \oplus s_n) \curlyvee (t_1 \oplus t_2 \oplus \cdots \oplus t_n)$$

$$(h_1 \prec t_1) \oplus (h_2 \prec t_2) \oplus \cdots \oplus (h_n \prec t_n) = (h_1 \oplus h_2 \oplus \cdots \oplus h_n) \prec (t_1 \oplus t_2 \oplus \cdots \oplus t_n)$$

The function *exprDistr* distributes exponentiation over cons and interleaving.

$$
\begin{aligned}
&exprDistr :: Expr \to Expr \to Expr \\
&exprDistr\ (h1 :\prec: t1)\ (h2 :\prec: t2) = (h1\ \hat{}\hat{}\hat{}\ h2) :\prec: (t1\ \hat{}\hat{}\hat{}\ t2) \\
&exprDistr\ n@(Lit\ \_)\ \ (h :\prec: t)\ \ \ \ = (n\ \hat{}\hat{}\hat{}\ h)\ \ \ \ :\prec: (n\ \hat{}\hat{}\hat{}\ t) \\
&exprDistr\ (h :\prec:\ \ t)\ \ n@(Lit\ \_)\ \ = (h\ \hat{}\hat{}\hat{}\ n)\ \ \ \ :\prec: (t\ \ \hat{}\hat{}\hat{}\ n) \\
&exprDistr\ (h1 :\curlyvee: t1)\ (h2 :\curlyvee: t2) = (h1\ \hat{}\hat{}\hat{}\ h2) :\curlyvee: (t1\ \hat{}\hat{}\hat{}\ t2) \\
&exprDistr\ n@(Lit\ \_)\ \ (h :\curlyvee: t)\ \ \ \ = (n\ \hat{}\hat{}\hat{}\ h)\ \ \ \ :\curlyvee: (n\ \hat{}\hat{}\hat{}\ t) \\
&exprDistr\ (h :\curlyvee: t)\ \ n@(Lit\ \_)\ \ = (h\ \hat{}\hat{}\hat{}\ n)\ \ \ \ :\curlyvee: (t\ \ \hat{}\hat{}\hat{}\ n) \\
&exprDistr\ x\ \ \ \ \ \ \ \ \ \ y\ \ \ \ \ \ \ \ \ \ = x\ \ \ \ \ \ \ \ \ \ \hat{}\hat{}\hat{}\ y
\end{aligned}
$$

Note that literals are being distributed as well. A literal n represents a constant stream and by the definition of *repeat*, $n = n \prec n$. Similarly for interleaving, $c \curlyvee c = c$ where c is a constant stream.

The following function *consolidate*, is abstracted for both sums and products. It takes a sum (or product) of cons'es (and interleaves) and using the abide law, compresses them to a cons (and interleave) of sums (or products).

$$
\begin{aligned}
&\mathit{consolidate} :: ([\mathit{Expr}] \to \mathit{Expr}) \to [\mathit{Expr}] \to [\mathit{Expr}] \\
&\mathit{consolidate}\ f\ ts = \\
&\quad (\textbf{if}\ \mathit{null}\ \mathit{conses}\ \textbf{then} \\
&\qquad \mathit{id} \\
&\quad \textbf{else let}\ (ls, rs) = \mathit{unzipWith}\ (\lambda(a :\prec: b) \to (a, b))\ \mathit{conses} \\
&\qquad\ \ \textbf{in}\ (:)\ (f\ ls :\prec: f\ rs)) \\
&\quad (\textbf{if}\ \mathit{null}\ \mathit{interleaves}\ \textbf{then} \\
&\qquad \mathit{ts}'' \\
&\quad \textbf{else let}\ (ls, rs) = \mathit{unzipWith}\ (\lambda(a :\curlyvee: b) \to (a, b))\ \mathit{interleaves} \\
&\qquad\ \ \textbf{in}\ (f\ ls :\curlyvee: f\ rs) : \mathit{ts}'') \\
&\quad \textbf{where} \\
&\qquad (\mathit{conses}, \mathit{ts}') \qquad\ = \mathit{partition}\ \mathit{isCons}\ \mathit{ts} \\
&\qquad (\mathit{interleaves}, \mathit{ts}'') = \mathit{partition}\ \mathit{isInterleave}\ \mathit{ts}'
\end{aligned}
$$

The function *unzipWith* separates out a list of binary operator expressions into a list of left and a list of right operands. It is defined as follows,

$$
\begin{aligned}
&\mathit{unzipWith} :: (a \to (b, c)) \to [a] \to ([b], [c]) \\
&\mathit{unzipWith}\ \_\ [\,] \qquad\ = ([\,], [\,]) \\
&\mathit{unzipWith}\ f\ (x : xs) = (a : as, b : bs) \\
&\quad \textbf{where}\ (a, b) \quad = f\ x \\
&\qquad\qquad\ (as, bs) = \mathit{unzipWith}\ f\ xs
\end{aligned}
$$

Once the cons and interleave sub-expressions have been consolidated the functions *summDistr* and *prodDistr* complete the process by distributing the literals over the compressed cons and interleave expressions.

$$
\begin{aligned}
&\mathit{summDistr} :: [\mathit{Expr}] \to \mathit{Expr} \\
&\mathit{summDistr}\ ((h :\prec: t) : s@(\_ :\curlyvee: \_) : n@(\mathit{Lit}\ \_) : xs) = \\
&\quad \mathit{summ}\ ((\mathit{summ}\ [n, h] :\prec: \mathit{summ}\ [n, t]) : s : xs) \\
&\mathit{summDistr}\ ((h :\prec: t) : n@(\mathit{Lit}\ \_) : xs) = \\
&\quad \mathit{summ}\ ((\mathit{summ}\ [n, h] :\prec: \mathit{summ}\ [n, t]) : xs) \\
&\mathit{summDistr}\ ((s :\curlyvee: t) : n@(\mathit{Lit}\ \_) : xs) = \\
&\quad \mathit{summ}\ ((\mathit{summ}\ [n, s] :\curlyvee: \mathit{summ}\ [n, t]) : xs) \\
&\mathit{summDistr}\ xs = \mathit{summ}\ xs
\end{aligned}
$$

The function *prodDistr* follows similarly. All the above functions can be brought together and composed, to form the function *distribute* that traverses an expression and performs all the individual distribution operations.

$$
\begin{aligned}
\mathit{distribute} = \mathit{foldExpr}\ &(\mathit{summDistr} \circ \mathit{consolidate}\ \mathit{summ}) \\
&(\mathit{mult} \circ \mathit{prodDistr} \circ \mathit{consolidate}\ \mathit{prod}) \\
&\mathit{exprDistr}\ (\prec)\ (\curlyvee)
\end{aligned}
$$

2.6.2 Compaction of Expressions

Suppose that we have an expression $x^2 + 2 * x + x + 1$. We would like to *compact* this expression to $x^2 + 3 * x + 1$. The function *scompact* takes a list of expressions, the summands of an addition, and summs together the common terms.

```
scompact :: [Expr] → Expr
scompact = summ ∘
           map melt ∘
           groupBy (λa b → snd a ≡ snd b) ∘
           sortBy (comparing snd) ∘
           map fsplit
  where melt []       = zero
        melt ((c, t) : cts) = prod [summ (c : map fst cts), t]
```

This function is again written in the pointfree style. The factors of the terms are first extracted, the pairs of factors and expressions are sorted by the expressions and then grouped, the groups are merged and the whole thing is summed back together. The function *fsplit* extracts the multiplicative factor out of an expression and returns a tuple of the literal representing the factor and the remaining expression. An expression with a factor that can be extracted is a *Prod* term in canonical form.

```
fsplit :: Expr → (Expr, Expr)
fsplit (Prod [n@(Lit _), t])   = (n, t)
fsplit (Prod (n@(Lit _) : ts)) = (n, Prod ts)
fsplit t                       = (one, t)
```

The first case is a special case of the second; it ensures that we don't return a singleton product. The sorting is achieved with the $sortBy :: (a \to a \to Ordering) \to [a] \to [a]$ function. The functional argument is the comparison function. The list that we are operating on is a list of tuples and we wish to sort on the second element of the tuple, the expression. This is achieved with the helper function $comparing :: Ord\ a \Rightarrow (b \to a) \to b \to b \to Ordering$. The grouping of the sorted list in performed with the $groupBy :: (a \to a \to Bool) \to [a] \to [[a]]$ function. Again we are grouping with respect to the second element in the tuple. The local function $melt :: [(Expr, Expr)] \to Expr$ sums the factors and multiplies the common expression by the result.

The function *pcompact* does the same for products as *scompact* does for sums. Take the expression $2 * x^2 * x$ as an example. We would like to compact this to the expression $2 * x^3$.

```
pcompact :: [Expr] → Expr
pcompact = prod ∘
           map melt ∘
           groupBy (λa b → fst a ≡ fst b) ∘
           sortBy (comparing fst) ∘
```

$$
\begin{array}{l}
\quad\quad map\ esplit \\
\textbf{where}\ melt\ [\,] \quad\quad = one \\
\quad melt\ ((b, e) : bes) = b\ \hat{\wedge}\hat{\wedge}\hat{\wedge}\ summ\ (e : map\ snd\ bes)
\end{array}
$$

The function *esplit* extracts the exponent out of an expression; for all except an exponential, the exponent is 1.

$$
\begin{array}{l}
esplit :: Expr \to (Expr, Expr) \\
esplit\ (f\ :\hat{}:\ g) = (f, g) \\
esplit\ f \quad\quad\quad = (f, one)
\end{array}
$$

As with *scompact*, the list of pairs is sorted, grouped and then each group is merged and a product is constructed from the resulting list. The function *compact* uses *foldExpr* to apply *scompact* and *pcompact* to all sums and products in an expression.

$$compact = foldExpr\ scompact\ pcompact\ (\hat{}\hat{}\hat{})\ (\prec)\ (\curlyvee)$$

The opposition of *compact* is *expand*. In contrast to *compact*, *expand* is a very *targeted* function as it has the potential to significantly increase the size of an expression.

$$
\begin{array}{l}
expand :: Expr \to Expr \\
expand\ (Prod\ (Lit\ n : xs)) = summ\ (replicate\ n\ (prod\ xs)) \\
expand\ (e\ :\hat{}:\ Lit\ n) \quad\quad = prod\ (replicate\ n\ e) \\
expand\ e \quad\quad\quad\quad\quad\quad = e
\end{array}
$$

The function $replicate :: Int \to a \to [a]$ takes a value and constructs a list containing that value n times.

2.6.3 Extraction of Expressions

Another class of expression manipulation is the extraction of a common integer literal or variable from an expression. For example take the expression $2 * x + 4 * y$, there is a common integer factor and this can be rearranged into $2 * (x + 2 * y)$. These manipulations are well understood for the arithmetic operators of addition, multiplication and exponentiation, however extraction of literals also extends to the cons and interleave constructs. Let c be a constant stream, a stream defined by *repeat* k for some constant k, then the law $c \curlyvee c = c$ holds. The abide law for interleave is

$$(s_1 \oplus s_2) \curlyvee (t_1 \oplus t_2) = (s_1 \curlyvee t_1) \oplus (s_2 \curlyvee t_2)$$

Using these two laws we can extract a constant summand.

$$
\begin{array}{ll}
 & (s + 1) \curlyvee (t + 1) \\
= & \quad \{\ \text{abide law}\ \} \\
 & (s \curlyvee t) + (1 \curlyvee 1) \\
= & \quad \{\ \text{reduction}\ \} \\
 & (s \curlyvee t) + 1
\end{array}
$$

A similar abide law exists for cons,

$$(h_1 \oplus h_2) \prec (t_1 \oplus t_2) = (h_1 \prec t_1) \oplus (h_2 \prec t_2)$$

These laws along with the laws for addition, multiplication and exponentiation give rise to four operations for expression manipulation. The following subsections give their implementations.

Extract Integer Summand

The transformation *extractLitSummand* is implemented in terms of *extractLitSummand′* and *combineLitSummand*.

$$\begin{aligned}
&\mathit{extractLitSummand} :: \mathit{Expr} \to \mathit{Expr} \\
&\mathit{extractLitSummand} = \mathit{combineLitSummand} \circ \mathit{extractLitSummand}'
\end{aligned}$$

The function *extractLitSummand′* splits an expression into an integer paired with the remaining expression.

$$\begin{aligned}
&\mathit{extractLitSummand}' :: \mathit{Expr} \to (\mathit{Int}, \mathit{Expr}) \\
&\mathit{extractLitSummand}'\ (\mathit{Lit}\ i) &&= (i, \mathit{Lit}\ 0) \\
&\mathit{extractLitSummand}'\ (\mathit{Summ}\ (\mathit{Lit}\ i : xs)) &&= (i, \mathit{Summ}\ xs)
\end{aligned}$$

The definitions for literals and sums with literals, serve as base cases for the remaining definitions.

$$\begin{aligned}
&\mathit{extractLitSummand}'\ (\mathit{Summ}\ xs) = \\
&\quad \textbf{let}\ (is, xs') = \mathit{unzip}\ (\mathit{map}\ \mathit{extractLitSummand}'\ xs) \\
&\qquad\ \ i = \mathit{sum}\ is \\
&\quad \textbf{in if}\ i > 0\ \textbf{then} \\
&\qquad (i, \mathit{Summ}\ (xs')) \\
&\qquad \textbf{else} \\
&\qquad (0, \mathit{Summ}\ xs)
\end{aligned}$$

A sum with no literals can have a literal extracted if one or more of its summands has a literal to extract — the result is the sum.

$$\begin{aligned}
&\mathit{extractLitSummand}'\ (e1\ :\Upsilon:\ e2) = \\
&\quad \textbf{let}\ (i1, e1') = \mathit{extractLitSummand}'\ e1 \\
&\qquad\ \ (i2, e2') = \mathit{extractLitSummand}'\ e2 \\
&\qquad\ \ i\ \ \ \ \ \ \ = \mathit{min}\ i1\ i2 \\
&\quad \textbf{in if}\ i > 0\ \textbf{then} \\
&\qquad (i, \mathit{combineLitSummand}\ (i1 - i, e1') \\
&\qquad\quad :\Upsilon:\ \mathit{combineLitSummand}\ (i2 - i, e2')) \\
&\qquad \textbf{else} \\
&\qquad (0, e1\ :\Upsilon:\ e2)
\end{aligned}$$

$$
\begin{array}{l}
\mathit{extractLitSummand}'\ (\mathit{e1} :\prec: \mathit{e2}) = \\
\quad \textbf{let}\ (\mathit{i1}, \mathit{e1}') = \mathit{extractLitSummand}'\ \mathit{e1} \\
\qquad (\mathit{i2}, \mathit{e2}') = \mathit{extractLitSummand}'\ \mathit{e2} \\
\qquad i = \mathit{min}\ \mathit{i1}\ \mathit{i2} \\
\quad \textbf{in if}\ i > 0\ \textbf{then} \\
\qquad (i, \mathit{combineLitSummand}\ (\mathit{i1} - i, \mathit{e1}') \\
\qquad\quad :\prec: \mathit{combineLitSummand}\ (\mathit{i2} - i, \mathit{e2}')) \\
\qquad \textbf{else} \\
\qquad (0, \mathit{e1} :\prec: \mathit{e2})
\end{array}
$$

The result of extracting a literal from either a cons or an interleave term is the minimum of the literals extracted from the two operands. No other expressions allow a literal to be extracted,

$$\mathit{extractLitSummand}'\ \mathit{expr} = (0, \mathit{expr})$$

The function *combineLitSummand* adds a literal to an expression. The clauses for specialized cases ensure that the resulting expression remains in canonical form.

$$
\begin{array}{ll}
\mathit{combineLitSummand} :: (\mathit{Int}, \mathit{Expr}) \to \mathit{Expr} & \\
\mathit{combineLitSummand}\ (0, e) & = e \\
\mathit{combineLitSummand}\ (i, \mathit{Lit}\ j) & = \mathit{Lit}\ (i + j) \\
\mathit{combineLitSummand}\ (i, \mathit{Summ}\ (\mathit{Lit}\ j : \mathit{xs})) & = \mathit{Summ}\ (\mathit{Lit}\ (i + j) : \mathit{xs}) \\
\mathit{combineLitSummand}\ (i, \mathit{Summ}\ \mathit{xs}) & = \mathit{Summ}\ (\mathit{Lit}\ i : \mathit{xs}) \\
\mathit{combineLitSummand}\ (i, e) & = \mathit{summ}\ [\mathit{Lit}\ i, e]
\end{array}
$$

Extract Integer Factor

The function *extractLitFactor* is implemented in the same fashion as *extractLitSummand*. The factor is extracted and then combined again to form the transformed expression.

$$
\begin{array}{l}
\mathit{extractLitFactor} :: \mathit{Expr} \to \mathit{Expr} \\
\mathit{extractLitFactor} = \mathit{combineLitFactor} \circ \mathit{extractLitFactor}'
\end{array}
$$

For the function *extractLitFactor*′, the sole base case is the integer literal.

$$
\begin{array}{l}
\mathit{extractLitFactor}' :: \mathit{Expr} \to (\mathit{Int}, \mathit{Expr}) \\
\mathit{extractLitFactor}'\ (\mathit{Lit}\ i) = (i, \mathit{Lit}\ 1)
\end{array}
$$

A literal factor can be extracted from a sum, as in the example of $2 * x + 4 * y$, when all the summands have an integer factor and the collective greatest common divisor is greater than 1.

$$
\begin{array}{l}
\mathit{extractLitFactor}'\ (\mathit{Summ}\ \mathit{xs}) = \\
\quad \textbf{let}\ \mathit{ixs}' = \mathit{map}\ \mathit{extractLitFactor}'\ \mathit{xs}
\end{array}
$$

```
      i     = foldr (λa b → fst a `gcd` b) 0 ixs'
  in if i > 1 then
       (i, Summ (map (combineLitFactor ∘ first (`div` i)) ixs'))
     else
       (1, Summ xs)
```

The expression *foldr* $(\lambda a\ b \rightarrow fst\ a\ `gcd`\ b)\ 0\ ixs'$ computes the greatest common divisor of the factors extracted from each summand — note that $n\ `gcd`\ 0 = n$. To produce the remaining sum, the factors are divided by the common factor and *combineLitFactor* is used to merge it back together.

Extracting an integer factor from a product can be trivial if a literal is one of the factors, otherwise the non-literal factors are traversed to extract factors.

```
extractLitFactor' (Prod [Lit i, e])    = (i, e)
extractLitFactor' (Prod (Lit i : xs)) = (i, Prod xs)
extractLitFactor' (Prod xs)           =
  let (is, xs') = unzip (map extractLitFactor' xs)
      i         = product is
  in if i > 1 then
       (i, Prod (xs'))
     else
       (1, Prod xs)
```

Consider an expression of the form 2^{x+1}. This could be rearranged into the form $2 * 2^x$. One could also rearrange it to $4 * 2^{x-1}$, however we will draw the line at the first rearrangement.

```
extractLitFactor' (Lit i :^: y) = (i, Lit i ^^^ summ [Lit (-1), y])
```

Finally, just as the addition of a constant can be extracted from a cons or interleave expression, so can a constant factor. The factors extracted from the two operands must have a greatest common divisor greater than 1.

```
extractLitFactor' (e1 :ϒ: e2) =
  let (i1, e1') = extractLitFactor' e1
      (i2, e2') = extractLitFactor' e2
      i         = gcd i1 i2
  in if i > 1 then
       (i, combineLitFactor (i1 `div` i, e1')
          :ϒ: combineLitFactor (i2 `div` i, e2'))
     else
       (1, e1 :ϒ: e2)
extractLitFactor' (e1 :≺: e2) =
  let (i1, e1') = extractLitFactor' e1
      (i2, e2') = extractLitFactor' e2
```

$$
\begin{array}{l}
\quad i \quad = gcd\ i1\ i2 \\
\textbf{in if } i > 1 \textbf{ then} \\
\quad (i, combineLitFactor\ (i1\ `div`\ i, e1') \\
\quad\quad :\prec: combineLitFactor\ (i2\ `div`\ i, e2')) \\
\quad \textbf{else} \\
\quad\quad (1, e1 :\prec: e2)
\end{array}
$$

No other expressions can be transformed to extract a literal factor.

$$extractLitFactor'\ expr = (1, expr)$$

The function *combineLitFactor* inserts a factor back into an expression, in a similar fashion to *combineLitSummand*.

$$
\begin{array}{l}
combineLitFactor :: (Int, Expr) \to Expr \\
combineLitFactor\ (1, e) = e \\
combineLitFactor\ (i, Lit\ j) = Lit\ (i * j) \\
combineLitFactor\ (i, Prod\ (Lit\ j : xs)) = Prod\ (Lit\ (i * j) : xs) \\
combineLitFactor\ (i, Prod\ xs) = Prod\ (Lit\ i : xs) \\
combineLitFactor\ (i, Lit\ j :\wedge: y) \mid i \equiv j = Lit\ j\ \hat{}\hat{}\hat{}\ summ\ [Lit\ 1, y] \\
combineLitFactor\ (i, x) = prod\ [Lit\ i, x]
\end{array}
$$

Extract Variable Summand

The extraction of a variable from an expression is potentially more complex as there could be many variables to choose from. To simplify this, we will require that the user specifies which identifier they wish to extract.

$$
\begin{array}{l}
extractVarSummand' :: Ident \to Expr \to Maybe\ Expr \\
extractVarSummand'\ v\ (Var\ v') = \\
\quad \textbf{if } v \equiv v' \textbf{ then } Just\ (Lit\ 0) \textbf{ else } Nothing
\end{array}
$$

A variable summand can be extracted from a product such as $3 * x$. It becomes $x + 2 * x$.

$$
\begin{array}{l}
extractVarSummand'\ v\ (Prod\ [Lit\ i, Var\ v']) = \\
\quad \textbf{if } i > 1 \wedge v \equiv v' \textbf{ then} \\
\quad\quad Just\ (prod\ [Lit\ (i - 1), Var\ v']) \\
\quad \textbf{else } Nothing
\end{array}
$$

To extract a variable from a sum, we take the first summand from which the variable can be extracted.

$$
\begin{array}{l}
extractVarSummand'\ v\ (Summ\ xs) = \\
\quad mapFirst\ (extractVarSummand'\ v)\ xs \gg\!= return \circ summ
\end{array}
$$

The function *mapFirst* is defined as follows,

```
mapFirst :: (a → Maybe a) → [a] → Maybe [a]
mapFirst _ [] = Nothing
mapFirst f (x : xs) =
  case f x of
    Just x' → Just (x' : xs)
    Nothing → mapFirst f xs >>= return ∘ (x:)
```

The first element of the list for which the function is successful, is altered to the result of the function. For expressions such as $1 + x + 2 * x + x^2$, there are several alternatives for extracting the variable x, but this implementation will take only the first. To work around this limitation, a more specific invocation of the transformation must be made. There are no other expressions from which a variable summand can be extracted.

```
extractVarSummand' _ _ = Nothing


extractVarSummand :: Ident → Expr → Expr
extractVarSummand v expr =
  case extractVarSummand' v expr of
    Just expr' → summ [Var v, expr']
    Nothing → expr
```

Extract Variable Factor

As with the function *extractVarSummand*, *extractVarFactor* requires the user to specify the variable identifier to be extracted. The base case is as follows,

```
extractVarFactor' :: Ident → Expr → Maybe Expr
extractVarFactor' v (Var v') =
  if v ≡ v' then Just (Lit 1) else Nothing
```

Just as a literal can be extracted from an exponential, so can a variable. Consider the expression x^{y+1}, which can be transformed into $x * x^y$.

```
extractVarFactor' v (Var v' :^: e) =
  if v ≡ v' then
    Just (Var v' ^^^ summ [Lit (−1), e])
  else Nothing
```

As with the definition of *extractLitFactor*, a variable factor can be extracted from a sum if the factor can be extracted from all summands.

```
extractVarFactor' _ (Summ []) =
  Nothing
extractVarFactor' v (Summ xs) =
  mapM (extractVarFactor' v) xs >>= return ∘ summ
```

The function *mapFirst* is used again to extract a variable factor from a product and it has the same limitations as before.

```
extractVarFactor' v (Prod xs) =
  mapFirst (extractVarFactor' v) xs >>= return ∘ prod
```

There are no other expressions from which a variable factor can be extracted.

```
extractVarFactor' _ _ = Nothing


extractVarFactor :: Ident → Expr → Expr
extractVarFactor v expr =
  case extractVarFactor' v expr of
    Just expr' → prod [Var v, expr']
    Nothing → expr
```

2.6.4 Definitions and Unrolling Expressions

We have previously given definitions of stream equations for well known integer sequences such as *nat* and *fib*, as well as a number of stream operators such as *repeat*, *map* and *zip*. The datatype *Definition* will be used to represent defined stream equations within the system.

```
data Definition = Def { defLhs :: Expr, defRhs :: Expr }
                deriving (Eq)
```

The left-hand side defines the name and any parameters and the right-hand side defines the body of the definition. The following is an implementation of a parser for strings representing definitions.

```
def :: Parser Definition
def = do { lhs ← fun
         ; char '='; spaces
         ; rhs ← expr
         ; return (Def lhs rhs)
         } ⟨?⟩ "definition"
  where
    arg = do { x ← lower
             ; xs ← many digit
             ; spaces
             ; return (Var (x : xs))
             } ⟨?⟩ "function argument"
    fun = do { ident ← identifier
             ; args ← many arg
```

```
    ; return (App ident args)
    } ⟨?⟩ "function declaration"
```

The left-hand side of the definition is parsed by the specialized parsers *fun* and *arg*, defined locally.

A collection of definitions will be represented by a map data-structure, which maps identifiers to definitions. When a function application is encountered, the identifier can be used to retrieve the definition of the function being applied.

```
type Definitions = Map Ident Definition
```

There are a number of operations that require the lookup of stream definitions. The type *Env* is a monad specialized for that purpose.

```
type Env r = ReaderT Definitions Maybe r
```

This is a reader transformer monad with the maybe monad as the inner monad and the map of definitions as the shared environment. This describes computations that need to lookup definitions and can also fail.

Unrolling

The first operation that operates in the *Env* monad is the function *unrollExpr*. Given a function application, *unrollExpr* looks up the definition and substitutes in the body of the definition.

```
unrollExpr :: Expr → Env Expr
unrollExpr expr@(App f _) = do
  defs ← ask
  def ← M.lookup f defs
  subst ← lift $ match (defLhs def) expr
  return (applySubst subst (defRhs def))
```

The function *match* computes the substitution needed to replace the function application with the body of the definition. The function *unrollExpr* can fail by two different causes. The first is if the function application does not have a corresponding definition in the shared environment, so the *lookup* function fails; the second is if the match fails.

If an expression is not a function application we can still 'unroll' the expression. We simply extract the head and the tail of the expression and form a cons expression.

```
unrollExpr expr = do
  h ← getHead expr
  t ← getTail expr
  return (h ≺ t)
```

The operation of unrolling an expression into a cons expression with a head and tail, is often one of the first steps in a proof. Our intention is to discover a stream equation

with a unique solution. Therefore, transforming to a cons expression is more often than not the best way to start, followed by simplifying the head and rearranging the tail.

Take the example of proving that $nat = 2 * nat \curlyvee 2 * nat + 1$. Applying *unrollExpr* to $2 * nat \curlyvee 2 * nat + 1$ results in $2 * 0 \prec 2 * nat + 1 \curlyvee 2 * (nat + 1)$. The operations *reduce*, *distribute* and *extract* will complete the proof.

The function *getHead* used in the definition of *unrollExpr* is defined as follows,

$$\begin{aligned} &getHead :: Expr \to Env\ Expr \\ &getHead\ i@(Lit\ \_) = return\ i \end{aligned}$$

The head of a constant stream is the constant itself, therefore the head of a literal is the literal.

$$getHead\ v@(Var\ \_) = return\ (App\ \texttt{"head"}\ [v])$$

There is nothing to do with a variable other than construct an application of *head*.

$$\begin{aligned} &getHead\ (Summ\ xs) = mapM\ getHead\ xs \gg\!= return \circ Summ \\ &getHead\ (Prod\ xs) \ \ = mapM\ getHead\ xs \gg\!= return \circ Prod \end{aligned}$$

The arithmetic operators of addition, multiplication and exponentiation are applied element-wise, therefore the head of a sum, product or exponential is the head of the operands.

$$getHead\ (x :\!\wedge\!:\ y) = liftM2\ (:\!\wedge\!:)\ (getHead\ x)\ (getHead\ y)$$

The function $liftM2 :: Monad\ m \Rightarrow (a \to b \to c) \to m\ a \to m\ b \to m\ c$ lifts a binary function to a monadic one.

$$\begin{aligned} &getHead\ (x :\!\prec\!:\ y) = return\ x \\ &getHead\ (x :\!\curlyvee\!:\ y) = getHead\ x \end{aligned}$$

The head of a cons is naturally the first operand, and head of an interleave is the head of the first operand, by the definition of interleaving. Finally the head of a function application is the head of the unrolled function.

$$getHead\ e@(App\ \_\ \_) = unrollExpr\ e \gg\!= getHead$$

Note that *unrollExpr* and *getHead* are mutually recursive. The definition of *getTail* follows very similarly to *getHead*.

$$\begin{aligned} &getTail :: Expr \to Env\ Expr \\ &getTail\ i@(Lit\ \_) \ \ \ \ = return\ i \\ &getTail\ v@(Var\ \_) \ \ \ \ = return\ (App\ \texttt{"tail"}\ [v]) \\ &getTail\ (Summ\ xs) \ \ \ \ = mapM\ getTail\ xs \gg\!= return \circ Summ \\ &getTail\ (Prod\ xs) \ \ \ \ = mapM\ getTail\ xs \gg\!= return \circ Prod \\ &getTail\ (x :\!\wedge\!:\ y) \ \ \ \ = liftM2\ (:\!\wedge\!:)\ (getTail\ x)\ (getTail\ y) \end{aligned}$$

```
getTail (x :≺: y)       = return y
getTail (x :Υ: y)       = liftM2 (:Υ:) (return y) (getTail x)
getTail e@(App _ _) = unrollExpr e >>= getTail
```

The function *elimHeadAndTail* performs a top-down traversal of an expression to eliminate function applications of *head* and *tail*.

```
elimHeadAndTail :: Expr → Env Expr
elimHeadAndTail expr =
  case expr of
    (App "head" [a]) → getHead a
    (App "tail" [a]) → getTail a
    x                → gmapExprM elimHeadAndTail x
```

The function *gmapExprM* is the monadic version of *gmapExprT*

```
gmapExprM :: Monad m ⇒ (Expr → m Expr) → Expr → m Expr
gmapExprM f expr =
  case expr of
    App g xs → mapM f xs >>= return ∘ App g
    Summ xs → mapM f xs >>= return ∘ Summ
    Prod xs  → mapM f xs >>= return ∘ Prod
    x :^: y  → liftM2 (:^:) (f x) (f y)
    x :≺: y  → liftM2 (:≺:) (f x) (f y)
    x :Υ: y  → liftM2 (:Υ:) (f x) (f y)
    x        → return x
```

Unique Solution Verification

A procedure for verifying that a stream equation has a unique solution, as discussed in section 1.2, can be implemented using the *Env* monad. The following is not the only possible implementation, however this procedure will admit definitions such as $fib = 0 \prec 1 \prec tail\ fib + fib$ and $carry = 0\ \Upsilon\ carry + 1$.

```
verifyDefinition :: Definition → Env Bool
verifyDefinition (Def lhs rhs@(h :≺: t)) =
  let res1 = findExpr (App "head" [lhs]) rhs
      lhsTail = (App "tail" [lhs])
      res2 = findExpr lhsTail rhs
  in if null res1 then
       if null res2 then
         return True
       else do
         h' ← getHead t
```

```
        t' ← getTail t
        let res1 = findExpr (App "head" [lhsTail]) (h' :≺: t')
            res2 = findExpr (App "tail" [lhsTail]) (h' :≺: t')
        return (null res1 ∧ null res2)
    else
      return False
```

If the body of the definition is already in 'cons-form', then it is searched for applications of *head* or *tail* to a recursive call. This procedure will not permit any applications of head. It will permit an application of tail, so long as the tail of the body is itself can be a valid stream equation with no illegal applications of head or tail. This is equivalent to splitting the definition of *fib* into *fib* and *fib′*.

If the body of a definition is not a cons expression the functions *getHead* and *getTail* are used to rewrite it in that way.

```
verifyDefinition (Def lhs rhs) = do
  h ← getHead rhs
  t ← getTail rhs
  verifyDefinition (Def lhs (h :≺: t))
```

This implicitly requires that other streams used in the definition must be already defined.

2.7 User Interface

The user interface for the project has been built as a textual command line interface using Haskell bindings to the GNU Readline library, as well as further use of the Parsec parser library. It runs as a REPL (Read-Evaluate-Print loop). The program begins by accepting a stream equality from the user and then proceeds in a loop, allowing the user to input commands to manipulate the current expression. The available commands are as follows:

adddef This command parses a stream definition from the command line and prints out the current list of active definitions. The input definition is not verified at this time as it may depend on other definitions that have not yet been entered.

addlaw This command parses a law declaration from the command line and prints out the current list of active laws. A law is simply two expressions given as an equality. The datatype and parser for a law is

$$
\begin{aligned}
&\textbf{data}\ \mathit{Law} = \mathit{Law}\{\,\mathit{lawName} :: \mathit{String}, \mathit{lawLhs} :: \mathit{Expr}, \mathit{lawRhs} :: \mathit{Expr}\,\}\\
&\mathit{law} :: \mathit{Parser}\ \mathit{Law}\\
&\mathit{law} = \textbf{do}\ \{\,\mathit{name} \leftarrow \mathit{many1}\ (\mathit{satisfy}\ (\not\equiv \texttt{':'}))\\
&\qquad\quad ;\ \mathit{char}\ \texttt{':'}; \mathit{spaces};\\
&\qquad\quad ;\ \mathit{lhs} \leftarrow \mathit{expr}\\
&\qquad\quad ;\ \mathit{char}\ \texttt{'='}; \mathit{spaces}\\
&\qquad\quad ;\ \mathit{rhs} \leftarrow \mathit{expr}\\
&\qquad\quad ;\ \mathit{return}\ (\mathit{Law}\ \mathit{name}\ \mathit{lhs}\ \mathit{rhs})\\
&\qquad\quad \}\ \langle ? \rangle\ \texttt{"law"}
\end{aligned}
$$

reduce This command applies the *reduce* function to the whole of the current working expression and uses the *elimHeadAndTail* function to eliminate applications of *head* and *tail*.

distrib This command takes an expression as an optional argument, which is a search expression. The function *distribute* is applied to the whole of the current working expression, or if the optional argument is provided, a list of matching sub-expressions are retrieved and the user is given the choice of which specific application is to be used.

compact This command takes an expression as an optional argument, which is a search expression. The function *compact* is applied to the whole of the current working expression, or if the optional argument is provided, the command behaves in the same way as the **distrib** command.

expand This command requires an expression as an argument, which is the search expression for the sub-expression to which the function *expand* should be applied to.

extracts This command takes either an expression as an argument or an identifier and an expression as two arguments. In the case of the former, the function *extractLitSummand* is applied to sub-expressions that match the search expression argument. When an identifier is given as well, the function *extractVarSummand* is applied instead, using the given identifier as the variable to extract.

extractf This command functions in exactly the same way as the **extracts** function, except that it is for factors rather than summands.

unroll This command can be used in three different ways. With no arguments, the whole of the current working expression is unrolled. With an identifier as an argument, the user is given a choice about which applications of the identifier are to be unrolled. Finally if the command is given an expression, matching sub-expressions are to be unrolled.

applylaw This command takes a law name as input and searches for matches for both the left and right-hand sides of the law.

rewrite This final command is provided so that should any of the previous commands fail to do what the user requires, this command will allow an explicit rewrite to be made.

There are some commands that can potentially return multiple results. In the case where a search expression is provided, there can be multiple matches in the current working expression. The following function handles this,

```
applySearchTransformation :: (Expr → Expr) → Expr → Eval ()
applySearchTransformation f sExpr =
  gets current ≫= λexpr →
  let exprs′ = filter (≢ expr) (rewrite sExpr expr sExpr f) in
  case exprs′ of
    [] → outputLn "No change"
    [x] → updateAndPrintCurrent x
    xs → do x ← liftIO $ chooseExpr xs
            updateAndPrintCurrent x
```

The current working expression is retrieved. The function *rewrite* is used to find all matchings and return a list of new working expressions for each match, where the match has had the transformation applied. If there were no matches, then no change occurs. If there is a single match then that is taken as the new working expression. If there are multiple matches, then the function *chooseExpr* presents the user with a menu so that they can select the transformation result that they intended.

Chapter 3

Evaluation and Conclusions

3.1 Evaluation

One of the main simplifications made in the formulation of this project is the restriction of streams to integer streams. This decision was made largely because of the time constraints. Nevertheless, this restriction is not particularly limiting as much of the work on streams by Hinze and Rutten [4, 10] focuses on numeric streams (Rutten uses real numbers rather than integers).

Later sections of Hinze's paper explore the application of streams to *finite calculus*. This project did not explore this area, again due to time constraints. The project, as is, allows extensibility through definitions and laws that can be dynamically added into the running proof assistant. While this approach could be taken to tackle the application of streams to finite calculus, this is not ideal. The arithmetic operators of addition, multiplication and exponentiation as well as cons and interleave are given special treatment within the system. The operators in finite calculus of *finite difference* and *summation* cannot be added in with this same special status; furthermore, the handful of laws that pertain to these operators and their interactions with the others, cannot be assimilated into the simplifications such as *distribute*. This is the price paid for this particular instance of simplicity. The proof operator for distribution can be applied to many sorts of expressions and encompasses many specific laws of distribution, but this is a 'one tool for all' only as so far as the predefined operators. Altering proof operators, such as *distribute*, to be extensible is a significant adaptation.

In the early stages of the project a large amount of effort was expended in an attempt to make an 'automatic prover' rather than a 'proof assistant'. The initial assumptions were that the restrictions of integer streams and a specific proof method would make an automatic approach possible and preferable to an interactive one. The hope for a fully automatic, even limited, prover was abandoned. Many proofs given in Hinze's paper begin by unrolling definitions to reveal a cons expression with a head and tail. The proof then proceeds by simplifying the head and rearranging the tail to result in a recursion equation. The process of unrolling and simplifying is indeed a *mechanical* process that can be automated; the functions *unrollExpr*, *elimHeadAndTail* and *reduce*

do exactly this. The rearrangement of the tail to discover the recursion is the true work. Despite the knowledge that the starting expression must appear in some form in the tail, this remains a tough task. There were two alternatives, a brute force proof search or a goal-directed proof search. The former is not desirable due to the sheer quantity of choices presented by arithmetic operations. No progress was made with the later, so a fully automatic approach was discontinued.

The attempt at full automation was not the only part of what can be described as 'exploring the design space'. Early on in the project, the datatype for describing streams went through a number of iterations. These experimental variations explored alternative forms of representing function application and the specific treatment of arithmetic operators. This has resulted in large amounts of the code presented here being adapted and rewritten several times over.

The issues that were found to be challenging during the project were not the ones expected at the outset. Tasks such as verifying the uniqueness of stream equations and stream specific proof operations were the presumed substance of the project. As it transpired, the bulk of the project was dealing with arithmetic manipulation. The task of the proof tool essentially boiled down to an implementation of an arithmetic simplifier and manipulator.

The generality of matching addition and multiplication expressions is another shortcoming of the project. The commutativity and associativity of these operators means that there is the potential for a significant number of alternative matchings. The implementation presented makes an 'arbitrary' choice on how to match the variables within a sum or product. This certainly is a limitation and in some cases this will lead to a match erroneously failing. This is not as debilitating as it might first appear, for two reasons. Firstly, based on the proofs explored during the project, it appears to be uncommon to have multiple variables in a sum or product to be matched. Secondly, the tool provides a number of general commands for manipulation, which can applied to a whole expression or part of it. By using these to *preprocess* an expression, the matching can potentially be simplified.

3.2 Conclusions

This project has produced a specialized proof assistant that provides computer aid to proofs of stream equality. While it admits a number of simplifications, it can be seen as the basis for a more general theorem-prover for streams. The project has also served as a vehicle for first exploration into the topic of theorem proving and term rewriting. The assumptions made at the outset, about the nature of the project, have largely turned out to be false. The major challenges were related to arithmetic manipulation rather than anything specific to streams. The goal of a fully automatic prover turned out to be an especially hard task, again largely due to arithmetic.

From the perspective of programming languages, the project does present a strong case for the expressive power of pattern matching and other language features present in Haskell. Furthermore the power of monads, as a means of expressing stateful compu-

tations, is illustrated throughout the project and particularly so in the implementation of the type checker.

3.3 Further Work

A simple extension to the project, which would have been attempted given more time, is the addition of the finite difference and summation operators from finite calculus. There are a number of arithmetic laws for each of these operators, some of which would be naturally amalgamated into the functions such as *reduce* and *distribute*. Other possible extensions to the representation of expressions include generalising to real numbers to allow for division, or even to make streams fully polymorphic.

There are a number of improvements that can be made to the proof assistant. For example, instead of requiring that the user specify the law that they wish to employ, the proof assistant could search all the active laws for potential uses. This would help a user 'discover' the next step in the proof.

It would good to revisit the idea of a goal directed proof search for arithmetic proofs. One would need to answer the questions of how should a goal be represented and how a goal should be reached (uninformed search, informed search, planning).

At the beginning of the project, the use of 'Agda' was considered [7]. "Agda is a system for incrementally developing proofs and programs. Agda is also a functional language with dependent types." It is also written in Haskell, however at the time of the project, the support for co-inductive datatypes had not yet been implemented. It would be worthwhile revisiting this.

Bibliography

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, second edition, 1998.

[2] C. Hall et. al. Type classes in Haskell. In *European Symposium on Programming*, volume 788 of *LNCS*, pages 241–256. Springer-Verlag, April 1994.

[3] Jeremy Gibbons and Graham Hutton. Proof methods for structured corecusive programs. In *1st Scottish Functional Programming Workshop*, Stirling, Scotland, August 1999.

[4] Ralf Hinze. Functional pearl: Streams and unique fixed points. Submitted to ICFP'08, 2008.

[5] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[6] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.

[7] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[8] Simon Peyton Jones, editor. *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press, 2003.

[9] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[10] J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[11] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(39):11–49, April 2000.

[12] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, September 1989.

[13] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on PoPL*, Austin, Texas, January 1989. ACM Press.