PATH, a Program Transformation System for Haskell

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Mark Anders Tullsen

Dissertation Director: Paul Hudak

May 2002

Abstract

PATH, a Program Transformation System for Haskell

Mark Anders Tullsen
2002

PATH (Programmer Assistant for Transforming Haskell) is a user-directed program transformation system for Haskell. This dissertation describes PATH and the technical contributions made in its development.

PATH uses a new method for program transformation in which 1) total correctness is preserved, i.e., transformations can neither introduce nor eliminate non-termination; 2) infinite data structures and partial functions can be transformed; 3) generalization of programs can be done as well as specialization of programs; 4) neither an improvement nor an approximation relation is required to prove equivalence of programs—reasoning can be directly about program equivalence. Current methods (such as fold/unfold, expression procedures, and the tick calculus) all lack one or more of these features.

PATH uses a more expressive logic for proving equivalence of programs than previous transformation systems. A logic more general than two-level horn clauses (used in the CIP transformation system) is needed but the full generality of first order logic is not required. This logic used in PATH lends itself to the graphical manipulation of program derivations (i.e., proofs of program equivalence).

PATH incorporates a language extension which makes programs and derivations more generic: programs and derivations can be generic with respect to the length of tuples; i.e., a function can be written that works uniformly on 2-tuples, 3-tuples, and etc.

Acknowledgments

I wish to thank my advisor Paul Hudak for many years of constructive criticism, guidance, and encouragement. I also wish to thank the other readers of this dissertation: John Peterson, Zhong Shao, and Tim Sheard. To my wife, Teresa, and my children Andrew, Rachel, Zachary, and Jonathan: a heartfelt thanks for your support and patience while I have been working on this dissertation.

Soli Deo Gloria.

Contents

List of Figures

Chapter 1

Introduction

In order to automate and support software development via program transformation, I have designed and implemented a program transformation system for the pure functional language Haskell. This system is called PATH (Programmer Assistant for Transforming Haskell). In this dissertation, I describe PATH and the technical contributions made in its development. This chapter explains the need for program transformation, the obstacles to program transformation, and the design decisions made in developing PATH; lastly, this chapter gives an overview of the dissertation.

1.1 The Need for Program Transformation

Trade-offs between clarity and efficiency permeate the process of software development. To write software that is clear, and easily verified, is usually done at the expense of efficiency. To write efficient software is nearly always done at the expense of clarity. See Figure 1.1 for a Haskell program written for clarity and compare it to the program in Figure 1.2 which was written for efficiency: the functionality of the two programs is identical

```
wc h = do
       xs ← hGetContents h
       return (length xs, length (words xs))
```

Figure 1.1: Clear Code

```
wc h = wc h False 0 0
  where
  wc h inword cs ws =
    do
    eof ← hIsEOF h
    if eof then
       return (cs,ws)
     else
       do
       c ← hGetChar h
       if isSpace c then
         wc h False (cs+1) ws
        else
         if inword
           then wc h True (cs+1) ws
           else wc h True (cs+1) (ws+1)
```

Figure 1.2: Efficient Code

(count the characters and words in file), the code greatly differs.

This example demonstrates the general principle that a clear program is more easily seen to be correct, is faster to develop, and is easier to maintain. Likewise, an efficient program is usually less clear, is slower to develop, and is harder to maintain.

However, in software development we want clarity *and* efficiency. There are two major approaches to getting both. The first is the verification approach [22, 34]. In this approach, the specification is developed after the implementation. The specification is often some logic or specification language, usually non-executable; the implementation could be any language, often a procedural language. The disadvantage of this approach is that, because one starts with the implementation, the implementation may not meet the *intended* specification. (More often than not it will satisfy an incorrect specification—writing code that is both efficient and correct is difficult.)

The other major approach to getting both clarity and efficiency is the transformational approach [57]. In this approach, one *starts* with the specification. Then, by a sequence of correctness preserving transformations, it is transformed into a program of acceptable efficiency. This sequence of transformations is called the *program derivation.* Thus, one ends with not only an efficient program but also a proof (the derivation) that the implementation meets its specification. The advantage here is that there is no danger of a mismatch between the specification and the implementation.

The specification language can be a non-executable specification language or a functional language. The implementation language could be a functional language or a procedural language. The specification and implementation language could be the same language. The transformational approach has no inherent limits: one can take non-executable specifications to efficient algorithms, exponential algorithms to linear algorithms, and linear algorithms to logarithmic. Although the objective of program transformation is usually to make a program more efficient, program transformation can also support other tasks such as reverse engineering and re-factoring.

1.2 Obstacles to Program Transformation

The program transformation paradigm appears to be an effective alternative to the standard approach to software development:

- A executable specification can be generated rapidly (either as the first step or by refining a non-executable specification).
- Testing and requirements debugging can be done early in the software development process.

- Efficiency concerns do not affect the functional design.
- Rather than having one program which attempts to be both clear and efficient (where clarity usually defers to efficiency), we have two programs, guaranteed equivalent, one clear and one efficient.

So, why is program transformation not used in practice? First, there is the issue of tools:

- There are few tools for doing program transformation. Most which exist are research tools and not robust tools for languages used in practice.
- The tools that exist are in general hard to learn. They are primarily designed to be used by those that developed them—researchers and programming language experts. Generally they require an expertise and mathematical sophistication beyond that of a typical programmer.
- The tools that exist are in general hard to use. Most are based on a textual user interface.

Second, and more importantly, there are a number of problems, long recognized in the program transformation community, related to the intrinsic complexity of program derivations:

- Program derivations are large and complex. Thus, derivations are tedious to construct. It can be simpler to write an implementation from scratch even if a specification is at hand from which it could be derived.
- Program derivations are difficult to comprehend. One cannot easily understand the derivations others have constructed; this makes re-use and modification difficult.

- Program derivations are fragile with respect to changes in the specification. Requirements change and so do specifications. When the specification changes, the derivation can break beyond repair or require a large effort to repair. Although it is unrealistic to think that the specification can change without requiring changes to the derivation, we would like small changes in the specification to require proportionally small changes in the derivation.

Good tools are needed to make program transformation a feasible method of program development, the easier to use the better; but the next generation of program transformation systems also needs to be much better at dealing with the fundamental problem: reducing and managing the complexity of derivations. This thesis describes a number of contributions which may not appear to be closely related, but each contribution is aimed at this goal: reducing and managing the complexity of derivations.

1.3 The PATH Program Transformation System

The design space for a program transformation system is extremely large: Should the system be user-directed or fully-automated? What language, or languages, should it transform? Should it be, or work very similar to, a theorem-prover? Should a meta-language describe transformations or can transformations be done via a graphical user interface? Should it allow for incorrect transformations? And etc. This section describes the design decisions that have defined PATH.

1.3.1 User-Directed

PATH is user-directed, not fully automated. Much work in program transformation is on fully automatic methods, such as the work in partial evaluation [43], or the work on very highly optimizing compilers [62]. These methods generally give constant time speed ups, but user-directed methods are more powerful: algorithmic changes can be made that change the complexity class of the algorithm, e.g., exponential algorithms can be transformed into logarithmic algorithms. Also, user-directed methods are more general: the program can be restructured or made more general (re-factored) and not just made more efficient.

Fully automated methods are like a double-edged sword, powerful but hard to control: There is little control over the meta-program which transforms the program and there is no feedback except the resulting program. And as automated methods become more sophisticated, the harder they are to understand and use.

PATH is designed on the premise that although fully automated methods are useful for automating many simple transformations, they should be used as a supplement to, and not a replacement for, a user-directed system.

1.3.2 Aimed at Practitioners

Some do not understand why a program transformation system is required: a theorem prover in which one embeds the semantics of the language can be used to prove equivalences of programs. PATH does not adopt this approach because it requires understanding of a theorem prover, its logic, and its meta-programming language; it requires sophisticated knowledge of programming language semantics, either operational or denotational. The goal is for PATH to be usable by a novice functional programmer. Thus, no knowledge of logic or domain theory is required; no sophistication in mathematical reasoning is required.

1.3.3 Totally Correct

One of the most popular methods of program transformation is the fold/unfold method of Burstall and Darlington [15]. It is a simple, intuitive, and powerful approach but unfortunately it does not preserve total correctness: non-termination could inadvertently be introduced into the program[1]. In the PATH system, preserving total correctness is considered essential. This decision is motivated by these factors:

- PATH is user-directed and designed for programmers. Thus, the programmer should have confidence that he is not introducing non-termination into his program due to his inexperience. The programmer should not be required to produce proofs of termination (to guarantee total correctness).
- PATH should be scalable to large programs. Small programs transformed with fold/unfold can often be seen to terminate by inspection, but this is not the case with larger programs. The programmer should be able to transform large programs without concern that non-termination might be introduced in some obscure corner of the program.

1.3.4 Designed for Changing Specifications

Specifications change in the real world; often they change *after* the implementation has been developed. To ensure that a new implementation is correct with respect to a revised specification we must derive a new implementation from the new specification. The original derivation may be able to be re-used to some degree. In order to support changing specifications, PATH has been designed to maximize the re-use of previous derivations.

[1] If the language was strict, one would need to worry about *removing* non-termination from the program.

This is done in two ways: derivations are made generic and derivations are made manipulable.

Derivations (and programs) are generic.

The key to writing robust software is abstraction, but genericity is often needed to use abstraction: to write a generic sort routine, genericity over types is needed (i.e., polymorphism); to write a generic fold (a function inductively defined over an inductive data structure), genericity over *type constructors* (e.g., `List` and `Tree` where `List a` and `Tree a` are types) is needed—this is often called polytypism.

In program transformation the need for genericity is amplified. For example, in a monomorphic language, one cannot write a polymorphic `sort` but must write `sortInt`, `sortFloat`, and etc. One will have laws about `sortInt` and `sortFloat` instead of just one law about a generic `sort`; also, one must transform `sortInt` and `sortFloat` separately, even if the program derivations are identical. So, the ability to write a generic function, `sort`, reduces not only program size, but also the number of laws and the length of program derivations.

Consequently, the program transformation community—notably the Squiggol (or Bird-Meertens Formalism) community [11, 48, 49]—has been working to make programs more generic: not just polymorphic, but polytypic [41, 42, 46, 47]. However, the genericity provided by polymorphism and polytypism is still not adequate to achieve certain abstractions; another form of genericity is often needed—genericity over the length of tuples. Chapter 8 describes this form of genericity and how it can be achieved in a typed language.

Derivations are manipulable.

Historically, user-directed program transformation systems have worked as follows: the current state of the program (or part of it) can be viewed by the user and the user gives commands for applying transformation rules which change the program. The sequence of commands (the derivation) is usually stored for replay but it is implicit and is not displayable in a understandable form, only the current program is displayable.

The PATH approach is as follows: the user sees a program derivation, he changes it by applying a transformation rule which is added to the derivation. The original program, the final program, and the steps to transform the former into the latter are all in view.

There are two advantages to making derivations explicit: first, they become easier to understand because the user is accustomed to reading them, derivations are what he is manipulating; second, a visual representation of a program derivation allows it to be adapted more easily to a changing specification.

1.3.5 Simple

Whenever possible, PATH is made as simple as possible. The main contributions in this dissertation, described in Chapters 4, 5, and 8, were motivated by the desire to simplify the system as much as possible. I have attempted to make PATH simple to use and simple in theory: a meta-language is not used or required; the smallest set of primitive laws has been chosen; program derivations are based on the simplest logic possible; etc.

The following two design choices were motivated by this desiderata of simplicity.

A single language is transformed.

The seminal CIP system [8, 9] used a wide-spectrum language CIP-L which had three levels: specification, functional, and procedural. It was effectively three languages, a specification language, a functional language, and a procedural language, although they shared a common syntax. The idea was to start by writing programs at the specification level, transform them to the functional level where many transformations would be done, and then, if necessary, the functional program would be transformed into a procedural program for further optimization.

Although this is a very general approach, it is complex. There is a separate set of language constructs and a corresponding catalog of laws for each of the three levels. The user must use what amounts to three languages. PATH takes a more minimalist approach. All transformations in PATH are done on a single language PATH-L, a purely functional language similar to Haskell [38, 60]. This gains us much in simplicity but little is lost in generality:

- Although PATH-L has no non-deterministic or non-executable constructs as the specification sub-language of CIP-L does, PATH-L can express specification-like algorithms using standard features of a lazy functional language [85].

- PATH-L allows for writing procedural code, but it does so without sacrificing the semantically clean framework of a purely functional language. PATH-L accomplishes this in the same manner as Haskell, by using a monad for performing IO operations [84].

The language transformed is not Haskell.

The goal of PATH is to transform Haskell, but it achieves this goal indirectly: first, Haskell programs are translated into the PATH-L language (described in Chapter 2), then PATH-L

programs are transformed, and lastly, PATH-L programs are translated back into Haskell.

PATH-L is similar to Haskell: it is a statically typed, non-strict, purely functional language. The differences between Haskell and PATH-L consist in (1) a number of syntactic differences, e.g., recursion is explicit in PATH, and (2) some semantic extensions: the addition of unlifted tuples and the addition of tuples which are generic over the length of the tuple.

Using a language similar to, but not identical to, Haskell, is done for two reasons: first, the description of the system, the language, and its laws can be done more clearly in PATH-L without the unnecessary syntactic sugar of Haskell; the second reason is that in PATH-L laws are more easily expressed and derivations are more easily done.

1.4 Overview of the Dissertation

The rest of this dissertation describes the PATH program transformation system and the technical contributions made in the system. The dissertation is structured as follows:

- Chapter 2 introduces the language that is being transformed: a Haskell-like functional language.
- Chapter 3 discusses the two major approaches to program transformation, the generative set approach and the schematic approach. The advantages and disadvantages of these two incompatible approaches are discussed.
- Chapter 4 demonstrates how the two approaches to program transformation can be integrated. In particular, it is shown how the schematic approach can achieve the expressiveness of a powerful generative set approach if the right set of primitive laws is chosen.

- Chapter 5 discusses the underlying logic used in PATH: The form of a transformation law is explained (i.e., the formulas in the logic) as is the form of a program derivation (i.e., the proofs in the logic). The primitive laws are explained.
- Chapter 6 discusses the user interface aspects of PATH.
- Chapter 7 presents a number of examples of program derivations done in PATH.
- Chapter 8 discusses a new form of genericity, genericity over the length of tuples. It is explained why this form of genericity is useful in a program transformation system and how it can be achieved in a typed language.
- Chapter 9 summarizes the contributions, discusses related work, and points out future directions.
- Appendix A contains the predefined definitions in the PATH language.
- Appendix B contains the primitive transformation rules and laws.
- Appendix C contains a catalog of derived transformation laws along with their derivations.

Chapter 2

The PATH Language, PATH-L

In this chapter, the PATH-L[1] language used in PATH is described. PATH-L is similar to Haskell: it is a statically typed, non-strict, purely functional programming language. The following sections describe the syntax and semantics of PATH-L. The last section explains why PATH-L, and not Haskell, is used as the transformation language in PATH.

2.1 Syntax

The syntax of the terms and types of PATH-L is in Figure 2.1. It is a typed lambda calculus with products (tuples), sums, a fix point operator, and integers. It can be viewed as a desugared Haskell.

The terms of the language are described by the syntactic class *e*. Functions, or lambda abstractions, are written as "$v:t \mapsto e$" (without a leading lambda) where t is the type of the variable v. The v can be replaced by a tuple pattern, p. Tuples are written with angle brackets rather than parentheses as in Haskell. PATH-L has a more general way of projecting

[1] PATH-L for "PATH Language."

e	::=	v	variables
	|	$p{:}t \mapsto e$	abstraction
	|	$e_1\, e_2$	application
	|	$\langle e_1, e_2, ..., e_n \rangle$	constructor for n-tuples
	|	$e.m_n$	destructor for n-tuples ($1 \leq m \leq n$)
	|	$\texttt{In}.m_n$	constructors for n-sums ($1 \leq m \leq n$)
	|	$\texttt{case}\ e$	destructor for sums
	|	μ	fixed point operator
	|	m	integer constant
	|	$\texttt{prim}$	integer primitive
	|	$\perp_t$	the undefined value
p	::=	v	variables
	|	$\langle p_1, p_2, ... \rangle$	tuple patterns
t	::=	a	type variables
	|	$t_1 \rightarrow t_2$	functions
	|	$\times \langle t_1, t_2, ... \rangle$	tuple type (products)
	|	$+\langle t_1, t_2, ... \rangle$	sum type
	|	Int	integer type
m, n	::=	{natural numbers}	

Figure 2.1: Syntax of PATH-L

from tuples: if e is an n-tuple, then $e.m_n$ is the m-th element of e. For instance,

```
⟨x1,x2,x3⟩.2₃ = x2
```

The following program is ill-typed because a projection for a 4-tuple is being applied to a 3-tuple[2]:

```
⟨x1,x2,x3⟩.2₄
```

PATH-L has sums: the constructors being of the form `In.`m_n (the m-th constructor for an n-sum)[3], the destructor for sums is `case`. The `case` expression is different from that in Haskell in these ways: 1) the order of arguments: alternatives come first, the sum comes second; 2) the alternatives are written as a tuple of functions; and 3) no nested patterns are allowed. An example of a `case` reduction is as follows:

```
case ⟨e1,e2,e3⟩ (In.2₃ x)  ⇒  e2 x
```

PATH-L has an explicit fixed-point operator μ. It has integers and numerous strict primitive operators (denoted by the meta-variable *prim*). It also has an explicit undefined element, $\perp_t$, which corresponds to a non-terminating program of type t.

The types of the language are described by the syntactic class t. A type can be a function $t_1 \rightarrow t_2$, a product $\times\langle t_1, t_2, ...\rangle$, a sum $+\langle t_1, t_2, ...\rangle$, or an integer type. Discussing the type system of PATH-L will be postponed until Chapter 8, in which an extension of the language described here is discussed.

[2] A "projection" (such as 2_3) is said to be "applied" to a tuple using the "." operator.

[3] The reason for this odd syntax for constructors will be seen in Chapter 8.

Syntactic Conventions. A number of syntactic conventions will be used henceforth. The type annotations are often dropped, the projection m_n is sometimes written as just m, and the constructor $\texttt{In}.m_n$ is sometimes written as just $\texttt{In}.m$. Function application is left-associative:

$$abcd \equiv (((ab)c)d)$$

Lambda abstractions and μ's extend as far as possible to the right. So, we have this equivalence:

$$\mu x \mapsto y \mapsto z \mapsto e \equiv \mu(x \mapsto (y \mapsto (z \mapsto e)))$$

Function composition, written "$f \circ g$", is defined as "$f \circ g = x \mapsto f(gx)$".

As in Haskell, the variable "_" will sometimes be used to bind an unused variable.

Syntactic Sugar. There are also the `let` and `letrec` constructs which are merely syntactic sugar:

$$\texttt{let p:t=e in y} \quad \equiv \quad \texttt{(p:t} \mapsto \texttt{y) e}$$

$$\begin{array}{l} \texttt{letrec } x_1\texttt{:}t_1 \texttt{ = } e_1\texttt{; ...; } x_n\texttt{:}t_n \texttt{ = } e_n \texttt{ in m} \\ \equiv \\ \texttt{let } \langle x_1\texttt{,...,}x_n\rangle\texttt{=}\mu\langle x_1\texttt{,...,}x_n\rangle\texttt{:}\times\langle t_1\texttt{,...,}t_n\rangle \mapsto \langle e_1\texttt{,...,}e_n\rangle \texttt{ in m} \end{array}$$

Also, in let expressions $fx = e$ is syntactic sugar for $f = x \mapsto e$. Some laws are more clearly written using μ and other laws are more clearly written using `letrec`, the two notations will be used interchangeably.

We also have

$$\texttt{if b then t else f} \quad \equiv \quad \texttt{if b t f}$$

$$
\begin{array}{rcll}
c & ::= & p : t \mapsto e & \\
 & | & \langle e_1, e_2, \ldots \rangle & \\
 & | & \texttt{In}.m_n\ e & \\
 & | & m & \text{(integer)} \\
 & | & \bot_t &
\end{array}
$$

Figure 2.2: Canonical Forms

Appendix A contains the PATH-L prelude: a list of the predefined values used in programs and derivations. The `data` declarations for `List`, `Bool`, etc. in the prelude define sum types (possibly recursive) and induce some syntactic sugar. For instance the `Either` type,

```
data Either a b = Left a | Right b
```

induces the following syntactic sugar

```
Left  ≡ In.1₂
Right ≡ In.2₂

case ⟨Left x: e₁, Right x: e₂⟩  ≡  case ⟨x ↦ e₁,x ↦ e₂⟩
```

Note that `Left` is not a primitive, it is syntactic sugar for a sum. Nullary constructors are treated specially, as in the `List` type:

```
data List a = Nil | Cons ×⟨a,List a⟩

Nil  ≡  In.1₂ ⟨⟩
Cons ≡  In.2₂

case ⟨Nil: e₁, Cons⟨x,y⟩: e₂⟩  ≡  case ⟨⟨⟩ ↦ e₁ ,⟨x,y⟩ ↦ e₂⟩
```

No nested patterns are allowed in the sugared `case` expressions.

2.2 Semantics

The semantics of PATH-L is given operationally; we say $e \Downarrow c$ to signify that the well-typed closed expression e evaluates to the canonical form c. The notation $e \Downarrow$ signifies that there

$$
\begin{aligned}
(\mathtt{p} \mapsto \mathtt{e_1})\ \mathtt{e_2} &\Rightarrow \mathtt{e_1}\{\mathtt{e_2}/\mathtt{p}\} \\
\langle \mathtt{e_1},\ldots,\mathtt{e_n}\rangle.\mathtt{j_n} &\Rightarrow \mathtt{e_j} \\
\mathtt{case}\ \langle \mathtt{e_1},\mathtt{e_2},\ldots,\mathtt{e_n}\rangle\ (\mathtt{In.i_n}\ \mathtt{x}) &\Rightarrow \mathtt{e_i}\ \mathtt{x} \\
\mu \mathtt{p} \mapsto \mathtt{f} &\Rightarrow \mathtt{f}\{\mu \mathtt{p} \mapsto \mathtt{f}\ /\ \mathtt{p}\} \\
\mathit{prim}\langle \mathtt{c_1},\ldots,\mathtt{c_n}\rangle &\Rightarrow [\![\mathit{prim}\langle \mathtt{c_1},\ldots,\mathtt{c_n}\rangle]\!]
\end{aligned}
$$

Figure 2.3: Reduction Rules

---

exists a c such that $e \Downarrow c$. Figure 2.2 defines canonical forms; c ranges over canonical forms. The evaluation relation is defined using the notion of a reduction context [20]. A context, C, is an expression with one or more holes, [], embedded in it. $C[e]$ is the context C with its holes filled with the expression e. A reduction context, R, is a context with a single hole which indicates the place at which the next expression is to be reduced. The reduction contexts of PATH-L are defined inductively as follows:

$$
\begin{array}{rcll}
R & = & [] & \text{(hole)} \\
 & | & R\,e & \\
 & | & R.m_n & \\
 & | & \mathtt{case}\ e\ R & \\
 & | & \mu\, R & \\
 & | & \mathit{prim}\,\langle c_1, c_2, ..., R, e_1, e_2, ...\rangle &
\end{array}
$$

Evaluation does not occur inside functions, inside tuples, or in the arguments of constructors. The one step reduction relation, $\Rightarrow$, is the least relation satisfying the reduction rules (given in Figure 2.3) and the following rule (i.e., it is closed under reduction contexts):

$$R[e_1] \Rightarrow R[e_2] \quad \text{if} \quad e_1 \Rightarrow e_2$$

Multi-step reduction, $\Rightarrow^*$, is the transitive, reflexive closure of $\Rightarrow$. Evaluation, $\Downarrow$, is defined as follows: $e \Downarrow c$ if and only if $e \Rightarrow^* c$. The notation $e\{x/p\}$ used in Figure 2.3 signifies capture free substitution of x for p in the expression e. Since p could be a tuple pattern, the

notion of substitution is extended as follows:

$$e\{x/\langle p_1, p_2, ..., p_n\rangle\} = e\{x.1_n/p_1\}\{x.2_n/p_2\}...\{x.n_n/p_n\}$$

$$e\{x/v\} = \text{capture free substitution of } x \text{ for } v \text{ in } e$$

Haskell is lazy, or call-by-need, but the operational semantics given here is call-by-name. There is no need to add the extra complexity of call-by-need [2] because the theory of program equivalence used in PATH is call-by-name. PATH uses call-by-name because it is more expressive than call-by-need: it allows for both removing and introducing the sharing of computation in transformations.

Supporting a call-by-value functional language such as ML[4] [50] could be easily done by small changes to the semantics and the transformation laws.

2.3 PATH-L vs. Haskell

Instead of Haskell, PATH-L is used as the transformation language. Although PATH-L can almost be viewed as a desugared Haskell, a few changes were considered essential:

1. In PATH-L recursion is explicit (using either μ or `letrec`). In Haskell, recursion is implicit: any definition can be recursive or be mutually recursive with any other definition in the program. Making recursion explicit makes laws about recursive functions simpler and more concise. E.g., we have the law

   μ`F = `μ`(F∘F)`

   rather than the law

   ```
   let f = F f in e  =  let f = F(F f) in e
   ```

[4] Though supporting the impure features of ML would not be as straightforward.

2. PATH-L has unlifted products, for which it uses $\langle e_1, e_2, ... \rangle$ to distinguish them from Haskell's lifted products $(e_1, e_2, ...)$. Haskell only has lifted products. Unlifted products enjoy many more laws than lifted products and there is no loss of expressiveness as a lifted product can be had by simply wrapping a constructor around a unlifted product.

3. PATH-L extends Haskell with genericity over the length of both tuples and sums. The current `case` form is based, not on Haskell, but on this extension which will be introduced in Chapter 8.

4. PATH-L has a simple `case` construct rather than complex pattern-matching facilities. This makes the language and laws simpler[5], although at the expense of certain programs which are more easily expressed using pattern-matching. Rather than adding pattern-matching to PATH-L, I believe that a better approach is to add "first class patterns" [77]: extending PATH-L to use first class patterns is deferred to future work.

Other differences between Haskell and PATH-L are merely cosmetic[6]:

- Putting the alternatives first and the sum argument second in `case`. This allows for a higher order programming style as the second argument, of sum type, can often be left implicit.

- PATH-L uses "$p \mapsto e$" rather than Haskell's "`\p->e`" notation for functions.

Though the meaning of PATH could be revised to be "Programmer Assistant for Transforming a *Haskell-like-language*," the goal is really to transform Haskell, but I believe this

[5] Refer to the chapter on pattern matching in [21] for the complexity that is added to a transformation system by the existence of Haskell-like pattern-matching.

[6] These differences exist merely because the author desired to experiment with alternative syntax.

is easier done transforming an *extension* of the Haskell-language. (Differences 2 and 3 are extensions of Haskell; when first class patterns are added, difference 4 will be an extension.)

Chapter 3

Approaches to Program Transformation

Using laws about language primitives and laws derived from the operational semantics, the following transformation can be done:

```
  length(Cons⟨a,Cons⟨b,xs⟩⟩)
=                                  {def. length; reduce case}
  1 + length(Cons⟨b,xs⟩)
=                                  {def. length; reduce case}
  1 + (1 + length xs)
=                                        {associativity of +}
  (1 + 1) + length xs
=                                                  {def. +}
  2 + length xs
```

(The definition of `length` is in the PATH-L prelude, cf. Appendix A.) However, there are many transformations between recursive programs which cannot be performed using these primitive laws, e.g., "`map f ∘ map g`" cannot be transformed into "`map (f ∘ g)`". In order to do such transformations, more powerful laws or methods are needed. There is no lack of such methods: the problem is to choose which of a number of methods is most appropriate for the PATH system.

There are numerous methods for transforming functional programming languages. In their survey paper [56], Partsch and Steinbrueggen classify various methods for program trans-

formation into two basic approaches: (1) the generative set approach, which is based on a small set of simple rules which in combination are very expressive and (2) the schematic approach which is based on using a large catalog of laws, each performing a significant transformation. Fold/unfold [15] and expression procedures [69] are examples of the former. The Bird-Meertens Formalism (or Squiggol) [11, 48, 49] is an example of the latter. In this chapter I will discuss these two basic approaches to program transformation and compare them.

3.1 The Generative Set Approach

This section discusses various methods which take the generative set approach. The methods discussed here transform functional programs written as a set of recursive equations. So, the examples in this section are written as recursion equations but with a syntax otherwise the same as PATH-L.

3.1.1 Fold/Unfold

One of the most well known methods of program transformation is Burstall and Darlington's *fold/unfold* [15]. This methodology is extremely effective at a broad range of program transformations. It is based on six rules: (1) *unfold*: the unfolding of function calls by replacing the call with the body of the function where actual parameters are substituted for formal parameters; (2) *laws*: the use of laws about the primitives of the language; (3) *instantiation*: adding an "instance" of a function definition in which a parameter is replaced by a constant or pattern on both sides of the definition; (4) *fold*: the replacement of an expression by a function call when the function's body can be instantiated to the given

```
        ones  = Cons⟨1,ones⟩

        map f = case ⟨Nil       : Nil
                     ,Cons⟨x,xs⟩: Cons⟨f x, map f xs⟩
                     ⟩

(1)     twos = map plus1 ones
     ⇒                                                        {unfold ones}
(2)     twos = map plus1 (Cons⟨1,ones⟩)
     ⇒                                                         {unfold map}
(3)     twos = case ⟨Nil       : Nil
                    ,Cons⟨x,xs⟩: Cons⟨plus1 x, map plus1 xs⟩
                    ⟩
                    (Cons⟨1,ones⟩)
     ⇒                                                        {case reduce}
(4)     twos = Cons⟨plus1 1, map plus1 ones⟩)
     ⇒                                                              {laws}
(5)     twos = Cons⟨2, map plus1 ones⟩
     ⇒                                                              {fold}
(6)     twos = Cons⟨2, twos⟩
```

Figure 3.1: The "`twos`" Derivation Using Fold/Unfold

expression with suitable actual parameters—this fold can be done with any *previous* definition of the function; (5) *definition*: the addition of a new function definition; and (6) *abstraction*: the introduction of a `where` clause. Fold/unfold was intended to transform recursively defined functions but can also transform recursively defined data structures (for languages which allow for them).

An example of a fold/unfold derivation is in Figure 3.1. The definitions of `ones` and `map` are fixed and the definition of `twos` is transformed. The original program is on line 1. The definition of `ones` is unfolded to get the program on line 2. The next steps unfold `map`, evaluate the `case`, and apply a primitive law, giving the program on line 5. Note here that "`map plus1 ones`" is a previous definition of `twos` (from line 1); a fold can now be performed: the expression is replaced by `twos`, giving the program on line 6. This derivation has removed an intermediate data structure from the definition of `twos`.

The fold rule is what gives the method its power, but it is also the rule that makes the

method unsafe[1]. For example, consider the program

```
twos = map plus1 ones
```

Since the expression "`map plus1 ones`" is an instance of the right-hand-side of `twos`, it can be replaced with `twos`, yielding

```
twos = twos
```

which results in a non-terminating definition for `twos`. Although this example is simplistic, similar situations can arise in more subtle contexts, and thus non-termination can be introduced inadvertently.

In addition to this problem with partial correctness, fold/unfold has a significant inconvenience in practice: a history must be kept of all versions of the program as it is being transformed (or the user must specify which versions to keep). This history is essential because previous definitions of functions are used to give folding its power.

3.1.2 Totally Correct Fold/Unfold

One way to understand the partial correctness of fold/unfold is to characterize it denotationally. The essence of fold/unfold is captured by this equation[2]:

$$M \sqsupseteq \mu F \quad \text{if } M = F\,M$$

That is, we start with a program M and transform it until we see the "old definition" M in "$F\,M$". Then we replace M with a call to the current function (by replacing $F\,M$ with "$\mu x \mapsto F\,x$", or just "μF"). So, this equation states that if M is a fixed-point of F, then μF

[1] The instantiation rule of Burstall and Darlington is also unsafe, but it can be easily corrected by adding a strictness condition [65, 66].

[2] Where $N \sqsubseteq M$ signifies that N approximates M or N is less defined than M; the non-terminating program $\perp$ is less defined than all programs.

(the least fixed point of F) is less-defined than M. So, if we replace M with μF we may have a less-defined program: μF may fail to terminate where M terminates.

Several approaches have been proposed to solve the problem of partial correctness. One is to suitably constrain the use of fold, as proposed by Kott [44]. Unfortunately, Kott's method sufficiently constrains the form of program derivations, primitive laws, and function definitions so as to make his method unusable in practice [21, 68].

Another approach is to provide a separate proof of termination. Equationally,

$$M = \mu F \quad \text{if } M = F\,M, \ \text{total}(\mu F)$$

I.e., a proof is added that μF is never undefined. The disadvantages of this approach are one, infinite data structures and partial functions cannot be transformed (as neither can be proved total) and two, proving the totality of functions can be tedious or difficult.

Yet another approach is the tick algebra of Sands [68] which uses an improvement relation between programs, $\trianglerighteq$. His method can be loosely characterized as follows:

$$M = \mu F \quad \text{if } {}^{\surd}M \trianglerighteq F\,({}^{\surd}M)$$

The $^{\surd}$ is a tick which represents a computation step. This method is the most general way of ensuring the correctness of fold/unfold, but showing improvement can be onerous due to the manipulation of ticks involved.

A simpler approach which is similar to the tick algebra in ensuring improvement is expression procedures. This approach is described in the following section.

3.1.3 Expression Procedures

Motivated by the problems with fold/unfold, Scherlis proposed *Expression Procedures* (EPs) [69, 70]. (More recently Sands [66] extended this work to a higher-order non-strict

language.) Scherlis's key innovation was a new procedure definition mechanism in which the left hand side of an expression procedure definition can be an arbitrary expression: thus the name "expression procedure". In addition to laws about primitive functions and an *instantiation* rule (as in fold/unfold), three rules are used to transform programs: *abstraction*, which introduces new procedures; *composition*, which introduces new expression procedures; and *application*, which replaces a procedure call or expression procedure call with its definition (like the *unfold* rule).

The *composition* rule allows us to add an expression procedure to a list of recursion equations. Given the definition

```
f = F f
```

and if `C` is a strict context, an expression procedure can be added giving

```
f = F f
C f =ep= C(F f)
```

The second line is an expression procedure: the left hand side is not just a function symbol applied to variables and patterns, it is an arbitrary expression. (`=` is used for a regular definition and `=ep=` for an expression procedure definition.)

In Figure 3.2, expression procedures are used to, again, transform the definition of `twos`. Given the definition of `ones` and `map`, we transform the definition of `twos`. We can view this transformation as creating a new version of the recursive definition of `ones` which is specialized to the strict context "`map plus1 []`". The first step is to use the *composition* rule to introduce an expression procedure for this context, by filling in the hole, `[]`, with each side of the definition of `ones` giving the following expression procedure in step 2:

```
map plus1 ones =ep= map plus1(Cons⟨1,ones⟩)
```

```
      ones = Cons⟨1,ones⟩

      map f = case ⟨Nil       : Nil
                   ,Cons⟨x,xs⟩: Cons⟨f x, map f xs⟩
                   ⟩
```

```
(1)   twos = map plus1 ones
      ⇒                                                          {compose}
(2)   twos = map plus1 ones
      map plus1 ones =ep= map plus1 (Cons⟨1,ones⟩)
      ⇒                                                        {unfold map}
(3)   twos = map plus1 ones
      map plus1 ones =ep=
        case ⟨Nil       : Nil
             ,Cons⟨x,xs⟩: Cons⟨plus1 x, map plus1 xs⟩
             ⟩
             (Cons⟨1,ones⟩)
      ⇒                                                      {case reduce}
(4)   twos = map plus1 ones
      map plus1 ones =ep= Cons⟨plus1 1, map plus1 ones⟩
      ⇒                                                           {laws}
(5)   twos = map plus1 ones
      map plus1 ones =ep= Cons⟨2, map plus1 ones⟩
      ⇒                                                      {abstraction}
(6)   twos = map plus1 ones
      map plus1 ones =ep= twos'
      twos' = Cons⟨2, map plus1 ones⟩
      ⇒                                                          {apply}
(7)   twos = map plus1 ones
      map plus1 ones =ep= twos'
      twos' = Cons⟨2, twos'⟩
      ⇒                                                          {apply}
(8)   twos = twos'
      twos' = Cons⟨2, twos'⟩
```

Figure 3.2: The "`twos`" Derivation Using Expression Procedures

The next steps unfold `map`, evaluate the `case`, and apply a primitive law, giving the program in step 5. The expression procedure "`map plus1 ones`" now occurs recursively in its own definition. Next, a new function definition is introduced using the *abstraction* rule giving the program in step 6. Finally, we use the *application* rule to apply the expression procedure (replacing the left side with the right side) in the definition of `twos'` giving the program in step 7, where the intermediate data structure has been eliminated. Now the expression procedure is also applied in the definition of `twos` giving the program in step 8.

For the derivation to be correct, it would need to be confirmed that the context "`map plus1 []`" is strict. Showing that this context is strict cannot be done with expression procedures but the strictness condition can often be satisfied by syntactic inspection of the context.

On the one hand, expression procedures are strictly less powerful than fold/unfold (they can be simulated by fold/unfold); however, in practice, the great majority of fold/unfold transformations can be done as well by expression procedures. I am not aware of any *useful*[3] and *total correctness preserving* fold/unfold transformations which cannot be done by expression procedures either directly or indirectly (by finding a common "ancestor" from which to derive the two programs we wish to show equivalent). It should be noted that it is rather unfair to compare the expressiveness of a totally correct method with the expressiveness of a partially correct method because the partially correct method can derive anything (fold/unfold can show that any program is equivalent to $\perp$).

On the other hand, expression procedures have two key advantages over fold/unfold: (1) each of the transformation rules preserves total correctness, and (2) no history needs to be maintained, as all needed information is embedded in the expression procedures; and when compared to various methods of ensuring total correctness in fold/unfold, expression

[3] An example of a non-useful transformation is as follows [15, 88]: "`f x = 0`" can be transformed to "`f x = if x == 0 then 0 else f (x-1)`" in unfold/fold, but the reverse transformation cannot be done. Expression procedures cannot transform in either direction.

procedures are both easier to use and more expressive:

- Fold/unfold followed by a proof of termination: Expression procedures are simpler as they need no separate proof of termination. They are more expressive as they can transform infinite data structures and partial functions.

- Fold/unfold augmented with Sands tick calculus: Expression procedures derivations are simpler as there is not the added complication of ticks (and laws for manipulating them). Expression procedures also appear to be more expressive than Sands's original tick calculus[4].

Besides the technical improvements, in practice expression procedures have a simpler and more intuitive method of program derivation: with fold/unfold, the ability to add a new "eureka" definition to a program is essential; but with expression procedures, the analogous operation is selecting a recursive function and some context in which to specialize it. Thus, entering eureka definitions by hand is replaced by selecting contexts in the program.

3.1.4 The Reversibility Problem

Although expression procedures are an improvement over fold/unfold, they have one significant shortcoming: it is easy to specialize a function, but it is not always possible to generalize a function. This problem, shared with fold/unfold, comes about because the transformation rules are not reversible: in particular the *composition* and *application* rules are inherently one-directional[5]. (In fold/unfold the fold rule is inherently one-directional;

[4]Sands's original tick calculus couldn't prove the correctness of expression procedures: this seems to have been the motivation for his paper on expression procedures [66].

[5]Instantiation is only used in one direction in EPs and fold/unfold but the reverse of this rule could be added to the system.

it can fold using any *previous* definition of the function—the reverse of this would involve knowing the *future* definition of the function: a bit awkward!)

Let $P_1 \Rightarrow^{ep} P_2$ signify that the program P_2 can be derived from P_1 using some sequence of expression procedure rules. The relation $\Rightarrow^{ep}$ is not symmetric, nor is $\Rightarrow^{fu}$, the comparable derives relation for fold/unfold. Even when both $P_1 \Rightarrow^{ep} P_2$ and $P_2 \Rightarrow^{ep} P_1$, the derivation associated with $P_1 \Rightarrow^{ep} P_2$ may give no insight into how to find a derivation for $P_2 \Rightarrow^{ep} P_1$. For instance, given this definition of `map`:

```
map f = case ⟨Nil       : Nil
             ,Cons⟨y,ys⟩: Cons⟨f y, map f ys⟩
             ⟩
```

it is easy to go from

```
g = map plus1
```

to

```
g = case ⟨Nil       : Nil
         ,Cons⟨y,ys⟩: Cons⟨plus1 y, g ys⟩
         ⟩
```

using expression procedures but it is not possible to derive the first program from the second. This is not surprising because expression procedures were designed for *specialization* not generalization of functions.

Reversibility is important for two reasons: First, adding reversibility makes the system more expressive: as in the previous example, we often want to make programs shorter or more modular. Also, even when a more efficient program is wanted, we sometimes need to make it less efficient before making it more efficient (such transformations are impossible with a method—such as expression procedures—in which every transformation

step preserves or increases some measure of efficiency). Secondly, reversibility is important because the system becomes simpler if each rule is reversible: the user can learn one law and use it in two directions.

To get reversibility, a rule could be added such as this: "if $P_2 \Rightarrow P_1$ then $P_1 \Rightarrow P_2$." Burstall and Darlington added such a rule, called *redefinition*, to fold/unfold to get around this problem. The disadvantage of this approach is that if one has P_1 and wants to transform it, one needs to know the end result, P_2 before beginning—it cannot be derived directly or incrementally from P_1; also, the addition of this ad hoc rule makes the system more complex[6]. Instead of adding a rule, it would be preferable to modify the rules to make them all reversible. The following chapter shows how to do exactly that.

3.1.5 Summary of Generative Set Methods

So, fold/unfold is simple, intuitive, and powerful but lacks total correctness and requires a history of transformations. The expression procedure method is nearly as simple and powerful but preserves total correctness and requires no transformation history.

Were it not for the reversibility problem just discussed, expression procedures would be considered an excellent method for doing transformations in the PATH system.

However, there is one additional problem with the generative set methods: as they use a fixed set of rules there is no ability to abstract over common sequences of transformation steps and add more powerful rules to the system. Rather, the user is always transforming at the level of *composition*, *laws*, and *application* (or with fold/unfold: *unfold*, *laws*, *fold*). In contrast, the schematic approach allows the user to build up a useful catalog of

[6] In a partially correct method such as fold/unfold, the addition of *redefinition* compounds the correctness problem. Without redefinition, we have $\forall p.\, p \Rightarrow^{fu} \perp$ (any program p can be transformed into the undefined program), but with redefinition, we have $\forall p_1, p_2.\, p_1 \Rightarrow^{fu} p_2$ (that is, any program can be transformed into any other program).

transformation laws which can be re-used. This approach is discussed in the following section.

3.2 The Schematic Approach

The second major approach to program transformation is the schematic, or catalog, approach. In this approach, all transformation rules are expressed by laws about program schemes [39]. For instance, a law could be of the following form (where p_i are program schemes, $=$ is program equivalence, and $\Rightarrow$ is implication):

$$\forall x_1, x_2, \ldots . \; p_1 = p_2 \;\Rightarrow\; p_3 = p_4$$

Although one starts with a primitive catalog of laws, this catalog can be extended by adding laws derived by the user. All laws are symmetric and can be applied in either direction, thus there is no problem with reversibility. (Thus, in this approach $=$ is used for derives rather than the asymmetric symbol $\Rightarrow$[7].) Also, there is no intrinsic problem with partial correctness as long as all primitive laws preserve total correctness. However, as will be seen, the schematic approach has disadvantages of its own.

3.2.1 Large Catalog

The seminal Munich CIP system is an example of the schematic approach, it uses a large catalog of laws to reason about recursive programs. Refer to Figure 3.3 for an example of how the derivation of `twos` might be done with this approach. Here, recursion is explicit. The variable `ones` is replaced with its definition giving the program in line 2. Now, the law *Map-Inf-List*

[7] Which is now used for implication.

```
        ones = μones ↦ Cons⟨1,ones⟩

(1)     map plus1 ones
     =                                                        {def. ones}
(2)     map plus1 (μones ↦ Cons⟨1,ones⟩)
     =                                                    {Map-Inf-List}
(3)     (μtwos ↦ Cons⟨plus1 1,twos⟩)
     =                                                     {def. plus1}
(4)     (μtwos ↦ Cons⟨2,twos⟩)
```

Figure 3.3: The "twos" Derivation Using the Large Catalog Approach

$$\forall f,x.\ \texttt{map}\ f\ (\mu a \mapsto \texttt{Cons}\langle x,a\rangle) = \mu a \mapsto \texttt{Cons}\langle f\,x,a\rangle$$

is applied to this program giving the program in line 3 which is then simplified to the program in line 4. We have a short and elegant derivation; the derivation is reversible as every law is reversible; and the result of this derivation is a new law which could be added to the catalog of laws.

However, there is one problem: where did the *Map-Inf-List* law come from? If it is not a primitive law, it may not be derivable from the primitive laws; thus, the disadvantage of the large catalog approach: its expressiveness totally depends on the primitive laws in the catalog. Many systems using this approach have dozens of primitive laws about recursive programs [9, 35, 39]. Although it has numerous primitive laws, the CIP system adds the ability to use fold/unfold with a proof of termination. Although this addition is understandable—the expressiveness of the primitive laws is difficult to quantify but fold/unfold has proved to be extremely expressive in practice—it is unfortunate because it brings the disadvantages of fold/unfold (with a proof of termination) wherever fold/unfold is used: (1) partial functions and infinite data structures cannot be transformed, (2) a transformation history is required, and (3) derivations are not reversible.

Besides the potential lack of expressiveness, this approach lacks the simplicity and intuitiveness of fold/unfold (or expression procedures): With fold/unfold, one does not need

to search for applicable laws because the strategy for transforming recursive functions is virtually always the same: unfold the definition, simplify until there is an opportunity to do the folding, and do the fold step. So, one can always do basic transformations at the simplest level without any dependence on a catalog of laws.

3.2.2 Squiggol

Another example of the schematic approach to program transformation is the Squiggol method, otherwise known as BMF (Bird-Meertens Formalism); Bird and Meertens [12, 13, 48] were the pioneers of this approach which focuses on deriving programs by calculation. This approach is characterized by the use of a small set of recursion schemes (instead of general recursion) and a corresponding set of fusion laws for reasoning about these recursion schemes [11, 49].

Much of this work has focused on programming with total functions and giving the user recursion schemes which are primitive recursive (such as catamorphisms, which perform their computation inductively over some finite data type). By restricting the programs that can be written to total functions the laws become simpler: there are no strictness side-conditions. However, Meijer et al. [49] have shown how the approach can be extended to allow for partial functions and infinite data structures, although at some loss of simplicity.

Two of the primary recursion schemes used are catamorphisms and anamorphisms (or just `cata` and `ana`). Definitions of these are in Figure 3.4. Catamorphisms allow for defining functions defined inductively over lists. Anamorphisms are dual to catamorphisms, they allow for defining functions which construct lists by repeatedly "decomposing" a base value into either nothing (giving `Nil`) or a pair of values (giving the head of the list and a new base value). Catamorphisms and anamorphisms can be defined for other recursive data types besides lists.

```
cata = μcata ↦ ⟨f,b⟩ ↦ case ⟨Nil       : b
                            ,Cons⟨y,ys⟩: f⟨y, cata ⟨f,b⟩ ys⟩
                            ⟩

ana = μana ↦ f ↦ case ⟨Nothing  : Nil
                      ,Just⟨a,b⟩: Cons⟨a,ana f b⟩
                      ⟩
               ∘ f
```

Figure 3.4: Definition of Catamorphism (`cata`) and Anamorphism (`ana`)

```
(1)    map plus1 (ana (x ↦ Just⟨1,x⟩) ⟨⟩)
     =                                                          {Map-Ana}
(2)    ana (case⟨Nothing: Nothing, Just⟨a,b⟩: Just⟨plus1 a,b⟩⟩
            ∘ (x ↦ Just⟨1,x⟩))
           ⟨⟩
     =                                                           {def. ∘}
(3)    ana (x ↦ case⟨Nothing: Nothing, Just⟨a,b⟩: Just⟨plus1 a,b⟩⟩
                    (Just⟨1,x⟩))
           ⟨⟩
     =                                                  {case reduce}
(4)    ana (x ↦ Just⟨plus1 1,x⟩)
           ⟨⟩
     =                                                   {def. plus1}
(5)    ana (x ↦ Just⟨2,x⟩) ⟨⟩
```

Figure 3.5: The "`twos`" Derivation Using Squiggol

```
∀f,g,b.
    map f (ana g b)
  =
    ana (case ⟨Nothing: Nothing, Just⟨a,b⟩: Just⟨f a,b⟩⟩ ∘ g) b
```

Figure 3.6: The derived law *Map-Ana*

In Figure 3.5 is the derivation of `twos` again, but here the derivation differs significantly from the previous derivations of `twos`. In the first place, an infinite list cannot be directly written using a fixed-point operator but is constructed using `ana`. The infinite list of ones is now written as follows:

```
ana (x ↦ Just⟨1,x⟩) ⟨⟩
```

The first step is to apply the law *Map-Ana* (cf. Figure 3.6), giving program 2. Then in the following three steps the program is simplified by inlining $\circ$, doing case reduction, and applying a primitive. One might ask where *Map-Ana* comes from. Can it be derived from the primitive laws? In contrast to the large catalog approach, the answer is a definite yes. In Squiggol, there are laws for doing structural induction over each data type and universal properties (or fusion laws) for each recursion scheme (giving a computational induction principle). It is safe to say that the primitive laws are sufficiently expressive for programs written using the recursion schemes.

So, using Squiggol, there is no problem with the expressiveness of the primitive laws. The problem is in the expressiveness of the recursion schemes themselves. They can express a great many functions but not every algorithm for those functions is expressible. Although many programs can be written quite clearly using these recursion schemes and many transformations can be done elegantly, giving away general recursion is a tough price to pay.

3.2.3 Theorem Proving

A third approach would be to use a theorem prover (e.g., [58]) into which the semantics is embedded to prove equivalences of programs. Although research in theorem proving is generally outside the boundaries of research in program transformation, theorem proving is

often used to prove properties of programs and it is similar in goals to the work in program transformation.

A theorem proving approach has the following advantages: 1) it is the most general and expressive approach; 2) automatic proof search is available; and 3) strategies and tactics are extensible using a meta-language. But the disadvantages to using a theorem prover are the following: 1) the user needs to be expert in the theorem prover, its logic, and its meta-language; 2) the user needs to be expert in the semantics of the language and must usually reason using an approximation relation, $\sqsubseteq$, and be familiar with domain theory; and 3) most theorem provers provide a primitive interface.

So, a theorem prover seems to be more appropriate for the language designer than for the language user. Even were this approach taken, a front end would be wanted that hides the complexity, providing a specialized theorem prover for proving equivalences of Haskell programs. PATH would be an appropriate front end.

3.3 The Approaches Compared

3.3.1 Rules vs. Laws

In this chapter I have carefully discriminated between the terms *law* and *rule*. A *law* represents a program transformation that can be expressed logically or schematically, i.e., in a form such as the following:

$$\forall x_1, x_2, \ldots . \; p_3 = p_4$$

By definition a law is reversible. The generative set approach allows for laws about primitives but does not use laws to transform recursive programs. In the schematic approach,

every transformation is expressed as a law, and recursion must be explicit, otherwise no interesting transformations could be achieved.

A *rule*, in contrast, cannot in general be expressed as a *law*. A rule specifies a transformation but is more ad hoc. In the generative set approach, it is not a single rule but it is the set of rules in combination that gives a method its power. Rules in general will not be reversible.

In the schematic approach, laws are our currency. As a result we can abstract over transformations because we can develop new laws. We can make a system more powerful by adding new axioms (in the form of laws) to the system. This approach is like logic: If we want to extend a logic with another axiom, it is sufficient to prove the correctness of the new axiom in the underlying model without worrying about it conflicting with other axioms.

But in the generative set approach, we have both laws (about primitives) and rules (which give the method its power). We cannot abstract over the laws to get new laws (without some difficulty, cf. the following section) and we cannot abstract over the rules to get new rules. The rule set of a generative set approach is quite fragile: we cannot dispense with one without losing power, if we add a rule we need to show that the complete set of rules is still sound.

3.3.2 Laws and the Generative Set Approach

It is essential to note that the *laws* referred to in the discussions of fold/unfold and expression procedures are laws about the *primitives*, e.g., associativity of integer addition. One cannot use laws about the definitions of functions without risk of sacrificing correctness. For example, assume we have this lemma about the `power` function (which could be proved

with induction on the natural numbers):

```
power x m * power x n = power x (m+n)
```

We can use this lemma to perform the following EP derivation

```
  power ⟨x,n⟩ = if n == 0 then 1 else x * power ⟨x,n-1⟩
⇒                                                  {x = power⟨x,1⟩}
  power ⟨x,n⟩ = if n == 0 then 1 else power⟨x,1⟩ * power ⟨x,n-1⟩
⇒                                                      {power-Lemma}
  power ⟨x,n⟩ = if n == 0 then 1 else power⟨x,1+n-1⟩
⇒                                                  {arithmetic laws}
  power ⟨x,n⟩ = if n == 0 then 1 else power⟨x,n⟩
```

Thus we have transformed `power` into a function that is non-terminating on all inputs except zero. This is why laws are only allowed for the primitive operators. The problem in the above derivation is that a law about `power` is being used in the definition of `power`. This makes even expression procedures unsafe. If we want to extend EPs to use laws about program definitions, we need to ensure that laws about a definition `f` are not used to transform `f` or any definition that depends on `f`.

This problem does not arise in the schematic approach because the recursion is explicit: we would have the following definition of `power`

```
μpower ↦ ⟨x,n⟩ ↦ if n == 0  then 1 else x * power ⟨x, n-1⟩
```

and there is no way to apply laws about `power` inside the definition of `power` because the inner "`power`" is just a lambda bound variable.

Chin and Darlington [17] explain how to integrate laws, or schematic rules, into fold/unfold but totally ignore the correctness issue; though this is understandable since fold/unfold ignores the correctness issue.

Structural induction is a powerful proof principle which can be easily formulated in the schematic approach but cannot be done in fold/unfold or EPs. (In Scherlis's dissertation he

uses the associativity of list append—which can be proved with structural induction—in various derivations but conveniently list append is a primitive whose associativity is assumed.) So, could one add a proof rule for structural induction to a generative set approach such as expression procedures? One could use structural induction to prove things about definitions (as long as these laws are used safely, as noted above). But this use of structural induction happens outside of EPs: structural induction and EP derivations occur totally independently. Whether there is some way in which the expression rules and a structural induction rule could be integrated seems doubtful.

3.3.3 Summary

Neither the generative set nor the schematic approach is clearly better or more appropriate than the other. The generative set methods, such as fold/unfold or expression procedures, are simple and very expressive. Using expression procedures, partial functions and infinite data structures can be transformed correctly. However, these methods are asymmetric—they work well for specializing programs but not generalizing—and they do not allow for abstracting over transformation steps. The schematic approach is more concise and allows for the development of powerful laws which can represent major transformations; all transformation laws are symmetric. However, in this approach one either gives up general recursion (with an expressive set of primitive laws) or one has general recursion with a large catalog of laws, the expressiveness of these laws being unclear. Another disadvantage of this large catalog approach is that the user needs to search the catalog to find applicable laws.

The following chapter describes a way of integrating these two approaches to get the advantages of each.

Chapter 4

The PATH Approach

In this chapter I show how the essence of the expression procedure method can be distilled into one reversible rule called Fixed Point Fusion (Section 4.1). I then show the need for another law for reasoning about μ (Section 4.2). I give some examples of the expressiveness of these laws (Section 4.3). I then show how the expressiveness of expression procedures can be achieved using the schematic approach (Section 4.4) and discuss the advantages and limitations of this approach (Section 4.5).

4.1 From Expression Procedures to Fixed Point Fusion

As discussed in Section 3.1.4, expression procedures (EPs) lack a desirable property: reversibility. Could expression procedures be made reversible? The *composition* and *application* rules are inherently one-directional, but what if all the steps involved in a prototypical expression procedure transformation could be merged into one step? There are just four key steps (as seen in the example in Section 3.1.3):

1. the *introduction* of the expression procedure (*composition*),

2. the *transformation* of the body of the expression procedure,

3. the use of *abstraction* to capture the resulting recursion, and

4. *application* of the expression procedure.

These four steps can be merged as follows. We begin with a strict function `C` and a function definition `f = F f`. *Introduction* of the expression procedure gives

```
C f =ep= C(F f)
```

which is then *transformed* into the recursive expression procedure

```
C f =ep= G(C f)
```

for some `G`. After *abstraction* we arrive at

```
C f =ep= g
g = G(C f)
```

Finally, *application* of the expression procedure yields

```
C f =ep= g
g = G g
```

The above steps can be merged into one rule, (expressing the values of `f` and `g` as the fixed-points μ`F` and μ`G` respectively)

$$\texttt{C}(\mu\texttt{F}) \Rightarrow^{ep} \mu\texttt{G} \quad \text{if} \quad \forall\texttt{f}.\texttt{C(F f)} \Rightarrow^{ep} \texttt{G(C f)}, \texttt{C} \text{ strict}$$

The quantifier $\forall$`f` is used because no use is made of the definition of `f` in the transformation of the expression procedure body. So, this one rule replaces the three expression procedure rules—*composition*, *abstraction*, and *application*. We do not have reversibility yet, but if we replace $\Rightarrow^{ep}$ with $=$ in the above rule we would have the reversible law,

$$\texttt{C}(\mu\texttt{F}) = \mu\texttt{G} \quad \text{if} \quad \forall\texttt{f}.\texttt{C(F f)} = \texttt{G(C f)}, \texttt{C} \text{ strict}$$

a theorem of Stoy [75]. So, I join the company of many who have rediscovered or used this theorem [3, 28, 49]. Interestingly, it is a free theorem [82] of the fixed point operator μ. Also, μ, the *least* fixed point operator, is the only fixed point operator which satisfies this equation [28]. Its name, Fixed-Point Fusion (*FPF*), is taken from Meijer et al. [49] where the theorem is exploited considerably: most of their transformations are instances of this one general theorem.

Fixed-Point Fusion can be used in both directions:

$$\texttt{C}(\mu\texttt{F}) \Rightarrow \mu\texttt{G} \qquad \text{specialization (fusion)}$$
$$\mu\texttt{G} \Rightarrow \texttt{C}(\mu\texttt{F}) \qquad \text{generalization (fission)}$$

To do fusion, `C` and `F` are known, and `G` is desired; so the premise is proved by finding a derivation `C(F f)` $\Rightarrow$ `G(C f)`, `G` is discovered in so doing. To do fission, `G` is known, the user provides `C`, and `F` is desired; so the premise is proved by finding a derivation `G(C f)` $\Rightarrow$ `C(F f)`, `F` is discovered in so doing. (Had an extra "redefinition" rule been added to expression procedures, the user would also need to know the answer, `F`, before proceeding.)

This connection with expression procedures can give an intuition for *FPF*. Another intuition for *FPF* is provided in Figure 4.1. We start with `C(`μ`F)` in line 1 and in line 2 expand the μ`F`. In lines 3 and 4, `C` is moved past `F` using the premise. We eventually end with line 5; the strictness of `C` gives line 6; contracting the `G`'s gives line 7.

4.2 Fixed Point Expansion

With expression procedures the following transformation can be done

```
  f = F⟨f,f⟩
⇒                              {apply f}
  f = F⟨f,F⟨f,f⟩⟩
```

```
(1)     C(μF)
      =
(2)     C(F(F(F(...F(⊥)))))
      =
(3)     G(C(F(F(...F(⊥)))))
      =
(4)     G(G(C(F(...F(⊥)))))
      =
        ...
      =
(5)     G(G(G(...G(C(⊥)))))
      =
(6)     G(G(G(...G( ⊥ ))))
      =
(7)     μG
```

Figure 4.1: Intuition for Fixed Point Fusion (*FPF*)

which cannot be done with the *FPF* law. To see why we cannot accomplish this transformation using *FPF*: note two things

- We cannot change the value under a μ using the primitive laws.
- *FPF* is only applicable when we have a strict context and `F` is not strict.

A law which allows us to do the above transformation is Fixed Point Expansion (*FPE*)

$$\forall \mathtt{F}.\ \mu \mathtt{f} \mapsto \mathtt{F}\langle \mathtt{f}, \mathtt{f} \rangle = \mu \mathtt{f} \mapsto \mathtt{F}\langle \mathtt{f}, \mathtt{F}\langle \mathtt{f}, \mathtt{f} \rangle \rangle$$

which enables us to expand, or inline, the definition of a recursive definition inside itself; but this law also allows us to "reverse inline" recursive definitions by applying it right to left. Note that expression procedures cannot do this "reverse inline" transformation:

```
  f = F⟨f,F⟨f,f⟩⟩
⇒
  f = F⟨f,f⟩
```

To demonstrate that expression procedures cannot do this "reverse inline" transformation, I will show that fold/unfold (which is strictly more powerful than expression procedures)

cannot do the transformation "$\mu(F \circ F) \Rightarrow \mu F$": We assume that fold/unfold can be characterized by the following law (cf. Section 3.1.2)

$$M \Rightarrow \mu F \quad \text{if} \quad M \Rightarrow F\,M$$

To prove $\mu(F \circ F) \Rightarrow \mu F$, we must instantiate M with $F \circ F$, giving this:

$$\mu(F \circ F) \Rightarrow \mu F \quad \text{if} \quad \mu(F \circ F) \Rightarrow F\,(\mu(F \circ F))$$

But using laws we cannot satisfy the condition on the right: The only rule about recursive functions is "unfolding", i.e., $\mu(F \circ F) \Rightarrow (F \circ F)\,(\mu(F \circ F))$, and using this rule, $\mu(F \circ F)$ can only be transformed into a program with an even number of occurrences of F outside the μ. Thus, we see that $\mu(F \circ F) \Rightarrow \mu F$ cannot be done with fold/unfold.

4.3 Examples

Henceforth program derivations will be written in a form that is more like that used in the PATH system. The precise form of program derivations is treated in the next chapter, but here I discuss the conventions used in the derivations in this chapter. Derivation steps are written as a sequence of steps such as

```
  p1
= {r}
  p2
```

or

```
p1 ={r} p2
```

which signifies that `p1` is equivalent to `p2` by applying a law named `r` to some subexpression of `p1`. This notation is extended to allow for a law `r` that has *n* premises as follows: the derivation step

```
∀x,y,... .
  P1: p1=p2
; P2: p3=p4
; P3: p5=p6
; ...
⇒
   e1
 = {P1}
   e1'
 = {r}
   ...
 = {r}
   e2'
 = {red}
   e2
```

Figure 4.2: The Form of a Derivation

```
  p1
= {r
    d1
  ; ...
  ; dn
  }
  p2
```

signifies that p_1 is equivalent to p_2 by applying rule r to p_1 (or some sub-expression thereof) where d_1,...,d_n are the *n* derivations that prove the premises of r. A law may be of the form

```
∀x,y,... .
  p1=p2; p3=p4; p5=p6; ... ⇒ e1=e2
```

in which we have universal quantification and in which the equivalence "e_1=e_2" is conditional on the premises before the ⇒. A derivation, or proof, of such a law would look like the derivation in Figure 4.2, in which names are given to the premises (in order that they may be applied by name), and the "e_1=e_2" is replaced by a sequence of steps that prove this. The rule names inside the {}'s could be either the name of a premise (e.g., P_1) the name of a law proved elsewhere (e.g., r) or the name of some primitive rule (such as red—e.g., e_2'

```
(1)    map plus1 (μones ↦ Cons⟨1,ones⟩)
     = {FPF
          ∀ones'.
(2)         map plus1 (Cons⟨1,ones'⟩)
            =                                                    {def. map}
(3)         case ⟨Nil       : Nil
                 ,Cons⟨x,xs⟩: Cons⟨plus1 x, map plus1 xs⟩
                 ⟩
                 (Cons⟨1,ones'⟩)
            =                                                {case reduce}
(4)         Cons⟨plus1 1, map plus1 ones'⟩
            =                                                       {laws}
(5)         Cons⟨2, map plus1 ones'⟩
       }
(6)    μtwos ↦ Cons⟨2,twos⟩
```

Figure 4.3: The "twos" Derivation Using *FPF*

reduces to e_2). The rule {def. *v*} signifies the inlining of the prelude variable *v*. The rule {SS} (for Syntactic Sugar) signifies that the two programs are equivalent up to syntactic sugar. Rules are applied left to right but the prefix "R" before a rule name signifies that it is to be applied right to left. For further explanation of the primitive rules available in PATH, refer to Appendix B.

Also, the notation $C\{e_1=e_2\}$ is used as a shortcut for the law $C[e_1]=C[e_2]$, where C is any program context. This can be of great use when C is a large context.

4.3.1 The "twos" Derivation

The derivation of "twos" using *FPF* can be found in Figure 4.3. The original program, using the explicit fix point operator μ, is on line 1. As the function "map plus1" is strict, *FPF* can be applied here with the following instantiation of the free variables of *FPF* (although at first it is not known what G will be):

```
C = map plus1
F = ones ↦ Cons<1,ones>
```

```
G = twos ↦ Cons<2,twos>
```

To prove the premise of *FPF*, we start with the program on line 2, corresponding to "`C(F f)`". It is then transformed until we have brought the function "`map plus1`" against `ones′` in line 5 (corresponding to "`G(C f)`"). At this step, the premise is satisfied and we have discovered `G`, giving the result in line 6.

Note the similarity between this derivation and the expression procedure derivation shown in Section 3.1.3: applying *FPF* corresponds to introducing the expression procedure; transforming "`C(F f)`" in the premise corresponds to transforming the expression procedure definition; the end of the *FPF* premise derivation corresponds to the abstraction and apply steps. This derivation is also comparable to the fold/unfold derivation shown in Section 3.1.1: applying *FPF* corresponds to unfolding the definition of `ones`; transforming "`C(F f)`" in the premise corresponds to transforming the new definition of `twos`; the end of the *FPF* premise derivation corresponds to the fold step. By quantifying over `ones′` it is ensured that an unsafe fold step cannot be done.

The strictness condition for *FPF* is left out of this derivation in order to highlight the similarity to the expression procedure derivation. The following sub-derivation, for the premise "`C` $\perp = \perp$", would need to be added:

```
  map plus1 ⊥
=                                                        {def. map}
  case ⟨Nil       : Nil
       ,Cons⟨x,xs⟩: Cons⟨f x, map plus1 xs⟩
       ⟩ ⊥
=                                                    {case strict}
  ⊥
```

Note the advantages of this approach over expression procedures: the function `C` is shown to be strict *in the system* (whereas in expression procedures, it must be shown strict outside of

the system); also the derivation is reversible, one could start with "μ`twos`$\mapsto$`Cons<2,twos>`" and derive the original program from it.

Another advantage that can now be seen is that derivations are structured in a goal-directed fashion. Derivations are structured as 1) *goal*: a function and its context are specified; and 2) *sub-goal*: the derivation is developed which satisfies the sub-goal (thereby synthesizing the new definition). Besides clearly indicating the goal of each transformation, this allows all the sub-goals of an unreachable goal to be removed easily if the goal is removed. With fold/unfold and expression procedures the derivations can be much more unstructured.

4.3.2 Regarding Strictness Conditions

Two of the most useful laws, *FPF* and *Inst*, have strictness conditions. Can these conditions be avoided? There are three possibilities to eliminate the need for these: 1) the Squiggol approach where only total functions are allowed (thus, $\perp$ doesn't exist); or 2) these conditions are dropped and partially correct transformations are allowed; or 3) the strictness conditions are replaced by a totality condition on the result[1]. Neither of these methods was considered an option for PATH, and thus the strictness conditions remain. However, in actual use the strictness condition is very often satisfied automatically in PATH. Note that the contexts defined by *S*, an extension of reduction contexts *R*, are strict.

$$
\begin{array}{rcll}
S & = & [] & \text{(hole)} \\
 & | & Se & \\
 & | & S.m_n & \\
 & | & \texttt{case}\ \langle \texttt{nil}\colon e_1, \texttt{cons}\colon e_2 \rangle\ \mathrm{S} & \\
 & | & \mu S & \\
 & | & \texttt{prim}\ \langle e_1, e_2, ..., S, ..., e_n \rangle &
\end{array}
$$

[1] Refer to Section 5.3.4 for a further discussion of this option.

4.3.3 Introducing Mutual Recursion

The prototypical use of expression procedures (*composition*, *laws*, *abstraction*, and *application*) can obviously be done using *FPF*. Although the great majority of derivations using expression procedures do follow this pattern[2], what about the derivations which do not follow this pattern? What follows is an example of a derivation which does not follow the pattern but can be done easily with *FPF*.

Assuming that we can do the following derivations

$$\forall f,g.\ C(F\langle f,g\rangle) \Rightarrow A\ \langle C\ f,\ D\ g\rangle$$
$$\forall f,g.\ D(G\langle f,g\rangle) \Rightarrow B\ \langle C\ f,\ D\ g\rangle$$

we can do the expression procedure derivation seen in Figure 4.4. This derivation appears problematic to do with *FPF* because the uses of *composition*, *abstraction*, and *application* are completely intertwined. However, this derivation can be done as easily using *FPF* by explicitly representing the mutual recursion (see Figure 4.5).

4.4 Expression Procedures Equationally

The laws *FPF* and *FPE* appear to give us the expressiveness of expression procedures. But could the schematic approach, using these two laws about μ, accomplish *any* derivation possible with expression procedures? Section 4.1 showed how *FPF* can derive programs that would be done by the sequence of *composition*, *laws*, *abstraction*, and *application* in EPs. Section 4.2 showed how *FPE* can accomplish what is done by function *application* in expression procedures. And Section 4.3 gave a program derivation which was accomplished by a rather tangled ordering of the expression procedure rules: *composition*, *composition*,

[2]Likewise, *FPF* also captures the most common pattern in fold/unfold: unfold the definition one time, transform it, and then fold.

```
let
   f = F⟨f,g⟩
   g = G⟨f,g⟩
in
   C f  =ep= C(F⟨f,g⟩)
   D g  =ep= D(G⟨f,g⟩)
⇒                                                        {assumption}
   C f  =ep= A⟨C f, D g⟩
   D g  =ep= B⟨C f, D g⟩
⇒                                                   {abstract twice}
   C f =ep= f′
   D g =ep= g′
   f′  = A⟨C f, D g⟩
   g′  = B⟨C f, D g⟩
⇒                                                {apply "C f" twice}
   C f =ep= f′
   D g =ep= g′
   f′  = A⟨f′, D g⟩
   g′  = B⟨f′, D g⟩
⇒                                                {apply "D g" twice}
   C f =ep= f′
   D g =ep= g′
   f′  = A⟨f′,g′⟩
   g′  = B⟨f′,g′⟩
```

Figure 4.4: Introducing Mutual Recursion with Expression Procedures

```
let
  ⟨f,g⟩ = μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩,G⟨f,g⟩⟩
in
  ⟨C f, D g⟩
=                                                                  {R red}
  (⟨f,g⟩ ↦ ⟨C f, D g⟩) ⟨f,g⟩
= {FPF
    ∀f,g.
       (⟨f,g⟩ ↦ ⟨C f, D g⟩) ⟨F⟨f,g⟩,G⟨f,g⟩⟩
     =                                                               {red}
       ⟨C(F⟨f,g⟩), D(G⟨f,g⟩)⟩
     =                                                         {assumption}
       ⟨A⟨C f, D g⟩, B⟨C f, D g⟩⟩
     =                                                            {R red}
       (x ↦ ⟨A x, B x⟩) ⟨C f, D g⟩
     =                                                            {R red}
       (x ↦ ⟨A x, B x⟩) ((⟨f,g⟩ ↦ ⟨C f, D g⟩) ⟨f,g⟩)
  }
  μx ↦ ⟨A x,B x⟩
=                                                                   {eta}
  μ⟨f',g'⟩ ↦ ⟨A⟨f',g'⟩, B⟨f',g'⟩⟩
=                                                                    {SS}
  letrec f' = A⟨f',g'⟩ and g' = B⟨f',g'⟩ in ⟨f',g'⟩
```

Figure 4.5: Introducing Mutual Recursion with *FPF*

laws, *laws*, *abstraction*, *abstraction*, *application*, and *application*. Although many common EP derivations can be done using *FPF* and *FPE*, it is not clear whether, using these two laws about μ, one can do any derivation possible using EPs: i.e., derivations in which there may be arbitrary nesting and tangling of the rules. In this section, I demonstrate that PATH (with just two primitive laws about μ[3]) can give us the expressiveness of a powerful restriction of expression procedures.

4.4.1 Restricted Expression Procedures

PATH can do any expression procedure derivation which can be structured as a sequence of the following transformations:

- function *application* (replacing a function call by its definition)
- *laws* (application of primitive laws)
- *abstraction*
- a composition-laws-application transformation

PATH clearly allows for the first three transformations: function *application* (with *FPE*), *laws*, and *abstraction* (with reverse reduction). This section describes the fourth transformation and the next section proves that PATH can derive such transformations.

In the composition-laws-application transformation, the *composition* (creating new EPs) and *application* of EPs cannot be done in an arbitrary order. Such a transformation proceeds as follows: We start with this program

[3]As will be explained in the following chapter, the two primitive laws about μ are not *FPF* and *FPE* but *Scott-Induct* and *FPD*, from which *FPF* and *FPE* can be derived.

```
f = F⟨f,g⟩
g = G⟨f,g⟩
```

The definition `f` is a tuple of all the definitions which are to be composed over. The definition `g` is a tuple of all the other definitions in the program. Then we have a sequence of compositions over `f`, giving the EP program (where each of the C_i must be strict):

```
f = F⟨f,g⟩
g = G⟨f,g⟩
C1 f =ep= C1(F⟨f,g⟩)
C2 f =ep= C2(F⟨f,g⟩)
...
Cn f =ep= Cn(F⟨f,g⟩)
```

Then we transform the bodies of the EPs using primitive laws, *application* of `g` (but not *application* of `f`), and abstraction. In particular, EP *application* is not allowed. This gives the following program:

```
f = F⟨f,g⟩
g = G⟨f,g⟩
C1 f =ep= H1⟨f,g⟩
C2 f =ep= H2⟨f,g⟩
...
Cn f =ep= Hn⟨f,g⟩
```

Now a sequence of abstractions are made, resulting in the following:

```
f = F⟨f,g⟩
g = G⟨f,g⟩
C1 f =ep= h1
C2 f =ep= h2
...
Cn f =ep= hn

h1 = H1⟨f,g⟩
h2 = H2⟨f,g⟩
...
hn = Hn⟨f,g⟩
```

Now, the definitions of `f`, `g`, and h_i are re-written so that where we want to apply the EPs is made evident:

$$
\begin{aligned}
&f = F'\langle C_1\ f,\ C_2\ f,\ \ldots,\ C_n\ f,\ f,g\rangle \\
&g = G'\langle C_1\ f,\ C_2\ f,\ \ldots,\ C_n\ f,\ f,g\rangle \\
&\\
&h_1 = H_1'\langle C_1\ f,C_2\ f,\ldots,C_n\ f,\ f,g\rangle \\
&h_2 = H_2'\langle C_1\ f,C_2\ f,\ldots,C_n\ f,\ f,g\rangle \\
&\ldots \\
&h_n = H_n'\langle C_1\ f,C_2\ f,\ldots,C_n\ f,\ f,g\rangle
\end{aligned}
$$

Now, we perform a sequence of EP applies, giving the following (and dropping the EPs):

$$
\begin{aligned}
&f = F'\langle h_1,\ h_2,\ \ldots,\ h_n,\ f,g\rangle \\
&g = G'\langle h_1,\ h_2,\ \ldots,\ h_n,\ f,g\rangle \\
&\\
&h_1 = H_1'\langle h_1,h_2,\ldots,h_n,\ f,g\rangle \\
&h_2 = H_2'\langle h_1,h_2,\ldots,h_n,\ f,g\rangle \\
&\ldots \\
&h_n = H_n'\langle h_1,h_2,\ldots,h_n,\ f,g\rangle
\end{aligned}
$$

The above sequence of steps, a composition-laws-application transformation, constitutes the only way in which *composition* and EP *application* are allowed in the derivation. Although all useful EP derivations are structured similarly to this (first *composition*, then *laws* or other rules, *abstraction*, and lastly EP *application*)[4], what is being disallowed is the following two transformation rules in the transformation of the body of an EP: 1) the *application* of the function `f` (`f` being the definition "composed over") and 2) *application* of an EP to itself multiple times (it can only be done once, at the end), i.e., the results of an EP *application* cannot be transformed and be the subject of another EP *application*. As an example, the following derivation cannot be done using the restriction here:

[4]Without abstraction and EP *application* no gains could be made from the expression procedure created with *composition*.

```
        C f =ep= C(F f)
(1)  ⇒                                              {laws...}
        C f =ep= H1⟨f,C f,C f⟩
(2)  ⇒                                            {EP apply}
        C f =ep= H1⟨f,C f,H1⟨f,C f,C f⟩⟩
(3)  ⇒                                              {laws...}
        C f =ep= H2⟨f,C f⟩
(4)  ⇒                                  {abstract, EP apply}
        C f =ep= h
        h = H2⟨f,h⟩
```

This is because the result of applying an EP in step (2) is transformed and is the subject of a second EP *application* in step (4).

4.4.2 Restricted Expression Procedures Using PATH

This section shows how PATH can achieve the composition-laws-application transformation described in the previous section. That transformation corresponds directly to the law given in Figure 4.6. We can quantify over `f` as there is no dependence upon the definition of `f` anywhere in the derivation (this is because *application* of `f` is disallowed). The form of this law can be simplified to the law *Composition-Laws-Application* given in Figure 4.7. This is done by representing all the functions C_i as one function `C`, all the h_i functions as a tuple `h`, and introducing new definitions `F`, `G`, and `H` as follows:

```
C x = ⟨C1 x, C2 x, ..., Cn x⟩
h = H⟨h,f,g⟩

F⟨h,f,g⟩ = F'⟨h.1,h.2,...,h.n, f,g⟩
G⟨h,f,g⟩ = G'⟨h.1,h.2,...,h.n, f,g⟩
H⟨h,f,g⟩ = ⟨H1'⟨h.1,h.2,...,h.n, f,g⟩
           ,H2'⟨h.1,h.2,...,h.n, f,g⟩
           ...
           ,Hn'⟨h.1,h.2,...,h.n, f,g⟩
           ⟩
```

Note that `C` is strict iff each of the C_i is strict; for a proof of this refer to the laws *Tuple-Strict-Implies-Components-Strict* and *Components-Strict-Implies-Tuple-Strict* in Appendix

```
  C1 ⊥ = ⊥
; C2 ⊥ = ⊥
; ...
; Cn ⊥ = ⊥
; ∀f. letrec g = G'⟨C1 f, C2 f, ..., Cn f, f, g⟩
       in { C1(F⟨f,g⟩) = H1'⟨C1 f,C2 f,...,Cn f, f, g⟩ }
; ∀f. letrec g = G'⟨C1 f, C2 f, ..., Cn f, f, g⟩
       in { C2(F⟨f,g⟩) = H2'⟨C1 f,C2 f,...,Cn f, f, g⟩ }
; ...
; ∀f. letrec g = G'⟨C1 f, C2 f, ..., Cn f, f, g⟩
       in { Cn(F⟨f,g⟩) = Hn'⟨C1 f,C2 f,...,Cn f, f, g⟩ }
⇒
  letrec f = F'⟨C1 f, C2 f, ..., Cn f, f, g⟩
         g = G'⟨C1 f, C2 f, ..., Cn f, f, g⟩
  in g
 =
  letrec f  = F' ⟨h1,h2,...,hn, f, g⟩
         g  = G' ⟨h1,h2,...,hn, f, g⟩
         h1 = H1'⟨h1,h2,...,hn, f, g⟩
         h2 = H2'⟨h1,h2,...,hn, f, g⟩
         ...
         hn = Hn'⟨h1,h2,...,hn, f, g⟩
  in g
```

Figure 4.6: *Composition-Laws-Application-Expanded* Law

```
  C ⊥ = ⊥
; ∀f.
    letrec g = G⟨C f,f,g⟩ in { C(F⟨C f,f,g⟩) = H⟨C f,f,g⟩ }
⇒
   letrec g = G⟨C f,f,g⟩; f = F⟨C f,f,g⟩                in g
 = letrec g = G⟨h  ,f,g⟩; f = F⟨h  ,f,g⟩; h = H⟨h,f,g⟩ in g
```

Figure 4.7: *Composition-Laws-Application* Law

C. This is true because PATH-L has true products, in which $\bot = \langle \bot, \bot \rangle$.[5]

So, showing that PATH is as expressive as Restricted EPs is simply a matter of proving the law *Composition-Laws-Application*, this proof is in Figure 4.8; it relies on *Lemma-1*

```
∀C,F,G,H.
  C ⊥ = ⊥
; ∀f.  letrec g=G⟨g,C f⟩ in { C(F⟨f,g⟩) = H⟨g,C f⟩ }
⇒
  letrec g=G⟨g,C f⟩; f=F⟨f,g⟩ in g  =  letrec g=G⟨g,h⟩; h=H⟨g,h⟩ in g
```

which follows easily from the law *FPF-Ext*[6]:

```
∀C,F,G,H.
  C ⊥ = ⊥
; ∀f.  letrec g=G⟨f,g,C f⟩ in { C(F⟨f,g⟩) = H⟨f,g,C f⟩ }
⇒
  letrec f=F⟨f,g⟩; g=G⟨f,g,C f⟩ in ⟨f,g⟩
= letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in ⟨f,g⟩
```

Refer to the proof of *FPF-Ext* (and the laws it requires) in Appendix C.

In order to simplify the presentation, I have neglected to show how one would apply the *abstract* rule in the middle of a composition-laws-application transform. In expression procedures, one can abstract simultaneously over a common subexpression in the EP, the composed function, and the rest of the program (i.e., one can bring the subexpression out of the definitions of `f`, `g`, and `h` simultaneously). We can simulate this in PATH by creating identical abstractions in `F`, `G`, and `H` giving us

```
F⟨h,f,g⟩ =  let x=X⟨h,f,g⟩ in F′⟨h,f,g,x⟩
H⟨h,f,g⟩ =  let x=X⟨h,f,g⟩ in H′⟨h,f,g,x⟩
G⟨h,f,g⟩ =  let x=X⟨h,f,g⟩ in G′⟨h,f,g,x⟩
```

so that instead of using *Composition-Laws-Application* to get this program

[5] Haskell has lifted products for which this is not true.

[6] If we instantiate *FPF-Ext* to functions where `f` does not occur in the definition of `g` or `h` and we apply (`.2`) to each side, we get *Lemma-1*.

```
  P1: C ⊥ = ⊥
; P2: ∀f.
       letrec g = G⟨C f,f,g⟩ in { C(F⟨C f,f,g⟩) = H⟨C f,f,g⟩ }
⇒
  let
    D f = ⟨C f,f⟩
    F'⟨⟨a,b⟩,c⟩ = F⟨a,b,c⟩
    G'⟨⟨a,b⟩,c⟩ = G⟨a,b,c⟩
    H'⟨⟨a,b⟩,c⟩ = H⟨a,b,c⟩
  in
{
    letrec g = G⟨C f,f,g⟩; f = F⟨C f,f,g⟩
  =                                                          {def. D, G'}
    letrec g = G'⟨D f,g⟩;  f = F⟨C f,f,g⟩
  = {Lemma-1
       C ⊥ ={P1} ⊥
     ; ∀f.
        letrec g =G'⟨D f,g⟩ in
          {
            D(F⟨C f,f,g⟩)
          =                                                      {def. D}
            ⟨C(F⟨C f,f,g⟩), F⟨C f,f,g⟩⟩
          =                                                          {P2}
            ⟨H⟨C f,f,g⟩   , F⟨C f,f,g⟩⟩
          =                                                  {def. F',H'}
            ⟨H'⟨⟨C f,f⟩,g⟩, F'⟨⟨C f,f⟩,g⟩⟩
          =                                                      {def. D}
            ⟨H'⟨D f,g⟩, F'⟨D f,g⟩⟩
          }
    }
    letrec g = G'⟨i,g⟩; i = ⟨H'⟨i,g⟩,F'⟨i,g⟩⟩
  =                                                  {R Inline-Bndg}
    letrec g = G'⟨i,g⟩; i = ⟨h,F'⟨i,g⟩⟩; h = H'⟨i,g⟩
  =                                                  {R Inline-Bndg}
    letrec g = G'⟨i,g⟩; i = ⟨h,f⟩; h = H'⟨i,g⟩; f=F'⟨i,g⟩
  =                                                    {Inline-Bndg}
    letrec g = G'⟨⟨h,f⟩,g⟩; h = H'⟨⟨h,f⟩,g⟩; f=F'⟨⟨h,f⟩,g⟩
  =                                                  {def. F',G',H'}
    letrec g = G⟨h,f,g⟩   ; h = H⟨h,f,g⟩   ; f=F⟨h,f,g⟩
}
```

Figure 4.8: *Composition-Laws-Application* Proof

```
letrec g = G⟨h,f,g⟩; f = F⟨h,f,g⟩; h = H⟨h,f,g⟩ in g
```

we would get this program

```
letrec g = let x=X⟨h,f,g⟩ in G′⟨h,f,g,x⟩
     ; f = let x=X⟨h,f,g⟩ in F′⟨h,f,g,x⟩
     ; h = let x=X⟨h,f,g⟩ in H′⟨h,f,g,x⟩
in g
```

And this can be transformed to remove the duplicate abstractions:

```
letrec g = G′⟨h,f,g,x⟩
     ; f = F′⟨h,f,g,x⟩
     ; h = H′⟨h,f,g,x⟩
     ; x = X⟨h,f,g⟩
in g
```

Note that the majority of the proof that PATH can achieve the expressiveness of Restricted EPs is done in PATH itself, in the proofs of *Composition-Laws-Application* and *FPF-Ext.*

4.5 Evaluation of the PATH Approach

Although the restricted expression procedures seems to be a significant limitation to the form of possible derivations, I am aware of no EP derivation that is not already in the above restricted form (either from Scherlis [69, 70], Sands [66], or my own work). Thus, in practice, PATH appears to be as expressive as Expression Procedures (which appear to be as expressive in practice as fold/unfold used safely).

Many EP derivations which aren't in the restricted form could be transformed into the restricted form easily. The two rules that cannot be used inside an EP can often be moved outside the EP:

- An “apply f in EP” can often be moved before the composition. I.e., this EP derivation

```
  f = F f
⇒                              {compose}
  f = F f
  C f =ep= C(F f)
⇒                            {C ∘ F = G}
  C f =ep= G f
⇒                              {apply f}
  C f =ep= G (F f)
```

  can be transformed into this EP derivation

```
  f = F f
⇒                              {apply f}
  f = F (F f)
⇒                              {compose}
  C f =ep= C(F(F f))
⇒                            {C ∘ F = G}
  C f =ep= G (F f)
```

- An “apply EP in EP” can often be moved after the application. I.e., this EP derivation

```
  C f =ep= H(C f)
⇒                               {apply C f}
  C f =ep= H(H(C f))
⇒                     {abstract and apply}
  C f =ep= h
  h = H(H h)
```

  can be transformed into this EP derivation

```
  C f =ep= H(C f)
⇒                     {abstract and apply}
  C f =ep= h
  h = H h
⇒                                  {apply h}
  h = H(H h)
```

But unfortunately, it does not always appear possible to get around the restriction. Note the following law

```
∀C,F,H1,H2.
  ∀f. C(F f)                      = H1⟨f,C f,C f⟩
  ∀f. H1⟨f,C f,H1⟨f,C f,C f⟩⟩ = H2⟨f,C f⟩
⇒
  C(μF) = μh ↦ H2⟨μF,h⟩
```

It can be derived using EPs (cf. Section 4.4.1). Unfortunately, I have been unable to either prove it using PATH or to demonstrate that it cannot be proved using PATH. I conjecture that it cannot be proved using PATH. (And it would follow from this conjecture that restricted EPs are strictly less powerful than EPs.)

It would certainly be possible to extend PATH to come closer to or give the full power of EPs. Two possibilities are the following:

- Add an improvement relation, $\trianglerighteq$, such as that in Sands [67]. This approach is very expressive but unfortunately still not as expressive as expression procedures [66, 68]; to achieve the expressiveness of expression procedures, the improvement relation must be extended to a weighted improvement relation. This further complicates an already complex approach.

- Add another law to the system which would give us more expressiveness. One candidate for such a law would be the following:

```
    C ⊥ = ⊥
  ; ( ∀i : Nat . fi = F fi+1
      ⇒
      C(F f0) = H⟨fj,fk,...,C fj,C fk,...⟩
    )
⇒
    C(letrec f = F f in f)
  =
    letrec h = H⟨f,f,...,h,h,...⟩; f = F f in h
```

  This is similar in spirit to Sands's approach. It ensures that an improvement is being made and allows for the *application*, or inlining, of the function `f` in the "expression

procedure". Such a law, though more expressive than *FPF* and *FPE* is one directional in its nature: it is easy to use left to right but difficult to use right to left. Because PATH uses the schematic approach, adding a new primitive law such as this is as simple as adding it to the primitive rule catalog.

Either of these approaches could be used to increase the expressiveness of the PATH system. However, this has not been done in the current version of PATH as the need has not arisen for more expressiveness than that achieved with the primitives in PATH.

Although in some respects PATH appears less expressive than full EPs, in other respects it is more expressive: PATH has a law for structural induction on lists, *List-Induct*, which allows for many transformations impossible with EPs[7] and thanks to reversibility, PATH can do many transformations directly which cannot be done with EPs.

4.6 Conclusion

This chapter has expanded on the author's work in [79]. As discussed in Chapter 3, the schematic approach and the generative set approach each have their advantages and disadvantages. Some previous attempts have been made to integrate these two approaches:

- Extend fold/unfold with schematic rules: Chin and Darlington [17] added schematic rules to fold/unfold along with a method to generate new schematic rules using fold/unfold.

- Extend the schematic approach with fold/unfold: The CIP system takes this approach, laws can be derived using fold/unfold plus a proof of termination.

[7] See Section 3.3.2 for a discussion regarding the difficulty of extending generative set methods with laws such as *List-Induct*.

However, these attempts are primarily a *combination* of the approaches rather than an integration. The resulting systems are more complex and the disadvantages of fold/unfold are still present when fold/unfold is used. It would be preferable to have the advantages of *both* approaches—simplicity, expressiveness, symmetry, and abstraction over transformation rules—and the disadvantages of neither. This chapter has shown a better way to integrate the two approaches: a powerful generative set approach (expression procedures) is subsumed into the schematic approach. This gives a method with the following advantages:

- *Simple and intuitive*: There are only two primitive rules for reasoning about recursive definitions. *FPF* can be understood intuitively as a common pattern used in fold/unfold or expression procedures.
- *Expressive*: The method is strictly more expressive than "Restricted expression procedures" as presented here.
- *History independent*: As with expression procedures, no history is required.
- *Totally correct*: Total correctness is preserved without proof of termination. Thus, the method is able to correctly transform partial functions and infinite data structures.
- *Extensible*: New program equivalence laws can be derived. Note how this has been used to good effect in this chapter, where numerous additional laws about recursion have been developed.
- *Symmetric*: Programs can be generalized as well as specialized; derivations can be reversed.

In comparison to expression procedures, the advantages of the PATH method are the following:

- There is a symmetric derives relation; no ad hoc rules, such as *redefinition*, are needed to get reversibility. Thus the system is simpler than expression procedures would be with such an extra rule: there is *one* rule which the user uses to both specialize and generalize, rather than an extra rule added to a set of "one directional" rules. Also, the somewhat ad hoc *composition* and *application* rules are no longer needed, but are implicit in the *FPF* law.

- Transformations can be done directly which cannot be done directly with expression procedures. With expression procedures, there is sometimes a need to derive programs indirectly: e.g., in order to show that p_1 is equivalent to p_3, one must do the two derivations $p_1 \Rightarrow p_2$ and $p_3 \Rightarrow p_2$.

- The base language does not need to be extended with expression procedures. Although expression procedures would not need to be implemented in the language (they are removed in the final program), a semantics would need to be given to expression procedure definitions.

- Derivations are structured in a goal-directed fashion. It is clear what the goal is, what the sub-goals are, and where the sub-derivations are that support that goal.

PATH has achieved a number of advantages over methods such as fold/unfold and expression procedures but there is still the issue of expressiveness, intrinsic to the schematic approach:

- Fold/unfold, although partially-correct, can do transformations that neither PATH nor expression procedures can do and can do so with the fewest restrictions on the form of derivations.

- Expression procedures appear to be strictly more expressive than the restricted expression procedures which PATH can do. Clarifying the difference in expressiveness here is an area for further research.

However, it should be noted that PATH, with structural induction, can do transformations that fold/unfold and expression procedures cannot.

Some areas for future work are the following: determining transformations that require the extra expressiveness of full EPs; clarifying the expressiveness of restricted expression procedures in a more satisfying manner; and finding the simplest way to make PATH as expressive as full expression procedures.

Chapter 5

A Logic for Program Transformation

Previously, program derivations have been presented somewhat informally; but in this chapter the exact syntax and semantics of laws and program derivations will be elucidated.

With the generative set approach there is no logic, only a set of rules by which closed programs are transformed into closed programs; but with the schematic approach, we are proving *laws* about programs. What logic should be used in PATH? Pepper [59] describes the logic which was developed for use in the CIP transformation system [8, 9] and argues that a logic for program transformation should be simpler than that needed in a general purpose theorem prover. Although agreeing with Pepper's argument, I have developed a different logic for PATH—one that improves on the infelicities of the CIP logic. The goals for the design of the PATH logic were the following:

- It should be at least as expressive as the CIP logic.
- It should be able to express parametricity laws.
- It should be as simple as possible.

t	::=	$\alpha \mid t_1 \to t_2 \mid \times\langle t_1, t_2, ...\rangle \mid +\langle t_1, t_2, ...\rangle \mid \text{Int}$	types
e	::=	$v \mid v{:}t \mapsto e \mid e_1\, e_2 \mid ...$	expressions
f'	::=	$v \mid v{:}t \mapsto f' \mid f'_1\, f'_2 \mid ... \mid \{e_1 = e_2\}$	expr. equivalence
p'	::=	$v \mid v{:}t \mapsto p' \mid p'_1\, p'_2 \mid ... \mid \{p'_1 = \{j\}\ p'_2\}$	proof of expr. equivalence
f	::=	$\forall v_1{:}t_1, v_2{:}t_2, ...\ .\ [f_1; f_2; ...] \Rightarrow f'$	formula
p	::=	$\forall v_1{:}t_1, v_2{:}t_2, ...\ .\ [r_1{:}f_1; r_2{:}f_2; ...] \Rightarrow p'$	proof / derivation
j	::=	$j' \mid \text{R}\ j' \mid ?$	justifications
j'	::=	$\text{red} \mid \text{eta} \mid r\,\langle e_1, e_2, ...\rangle\,[p_1; p_2; ...]$	rules

$r \in$ names of rules (premises & known laws)
$v \in$ names of variables

Figure 5.1: The PATH Logic

- Proofs in the logic (i.e., program derivations) should be small, easy to read, and lend themselves to a graphical display.

A note on the terminology used in this chapter: An *expression* is a term of the PATH-L language. A *formula* is a syntactically valid statement of program equivalence. A *law* is a formula that is semantically sound: this includes both primitive laws and derived laws. A *primitive rule* is like a primitive law but a rule uses meta-notation (it cannot be expressed as a formula). A *derivation* for a law is a proof that the law is valid. (I will use *proof* and *derivation* interchangeably.)

5.1 The Syntax of Formulas and Proofs

The syntax of formulas and proofs in PATH is given in Figure 5.1. The syntactic categories e and t are expressions and types as given in Chapter 2. The syntactic category f' is a statement of program equivalence, it allows for writing "contextual equivalences", i.e., embedding equivalences, `{`e_1`=`e_2`}` inside program contexts, allowing one to write `C{`e_1`=`e_2`}` as a shortcut for `C[`e_1`]=C[`e_2`]`. The syntactic category p' is a proof of an expression equivalence f'. A proof of the expression equivalence `C{`e_1`=`e_2`}` might have this form:

```
C{e1 ={red} {e3 ={red} e2}}
```

that is, $\texttt{e}_1$ reduces to $\texttt{e}_3$ and $\texttt{e}_3$ reduces to $\texttt{e}_2$. The syntactic categories f and p correspond to formulas and proofs thereof. A formula consists of a sequence of typed quantified variables, a sequence of premises "f_1; f_2;..." (each of which is a formula), followed by the conclusion, which is a statement of program equivalence. The form of a proof p is similar except that the premises are named and there is a proof of expression equivalence, f', to the right of the $\Rightarrow$.

Each step, $\{p'_1 = \{j\}\ p'_2\}$, in a proof of expression equivalence must be given some justification j. A justification can be a j', which is the name of a primitive rule—i.e., *red* or *eta*—or the application of another rule (either a premise or known law). When one applies a rule, values for each of the quantified variables must be provided, $\langle e_1, e_2, ... \rangle$, and proofs for each of the rule's premises must be provided: $[p_1; p_2; ...]$. The justification {R j} allows for the rule to be applied right to left instead of left to right[1]. The justification {?} is not essential to the logic but is an essential part of the user interface, it is a placeholder for an unproved part of the proof.

All laws in PATH follow a particular variable convention which is rather extreme, but simple: No bound variable (with `let` or $\mapsto$) can be free in any quantified variable. So, for instance, if `F` is a quantified variable, we have the following step which is always valid, we need not say that `x` is free in `F`:

```
(x ↦ F x) y ={red} F y
```

Note the *FPE* law for expanding out a recursive definition:

```
∀F. μf ↦ F⟨f,f⟩ = μf ↦ F⟨f,F⟨f,f⟩⟩
```

[1] "R" signifies reverse.

```
∀C,D,F,G.
  C ⊥ = D ⊥
; ∀x,y . {C x = D y} ⇒ {C(F x) = D(G y)}
⇒
  C(μF) = D(μG)
```

Figure 5.2: *FPI* Law

The function `F` here cannot have a free `f`, this is why we use $\texttt{F}\langle\texttt{f},\texttt{f}\rangle$ and not "`F f`": if *FPE* was written thus

```
∀F. μf ↦ F f = μf ↦ F(F f)
```

we would have a law of less generality, in this case *every* occurrence of `f` would be replaced by "`F f`". In the original *FPE* we have two occurrences of `f`: one to represent the occurrences of `f` that are unchanged and the other to represent the occurrences of `f` that are expanded. This convention can make certain laws, such as *FPE*, more onerous to write, but it makes laws clearer and greatly simplifies performing derivations: one need not be concerned with conditions such as "`v` not free in `E`." Also, this variable convention means that there is no need for meta-notation for substitution in PATH laws.

Figure 5.2 gives an example of a PATH law, *FPI* (Fixed Point Induction). Figure 5.3 gives an example of a derivation that proves the law *FPF* using the law *FPI*. In order to make derivations more readable than this example, derivations in this dissertation are generally written less formally. They are simplified in the following ways:

- Brackets are dropped.
- Empty lists of premises and empty lists of sub-derivations are dropped.
- Types are dropped when not essential.
- Instantiation lists are dropped.

```
∀C,F,G.
[ P1: {C ⊥ = ⊥}
; P2: ∀x. {C (F x) = G (C x)}
]
⇒
  { C (μ F)
  ={FPI ⟨C,id,F,G⟩
    [ { C ⊥ ={P1} ⊥ ={R red} id ⊥ }
    ; ∀x, y.
      [P3: {C x = id y}] ⇒
      { C (F x)
      ={P2 ⟨x⟩}
        G { C x ={P3} id y ={red} y }
      ={R red}
        id (G y)
      }
    ]}
    id (μ G)
  ={red}
    μ G
  }
```

Figure 5.3: Full Derivation of *FPF*

- The simple law *GC* is sometimes applied implicitly.
- A sequence of `{red}` steps is sometimes written as a single `{red*}` step.
- The exact place where the rule is applied is not manifest but the whole program is repeated on each line. (Although making it less clear what part of the program has been changed, this makes it easier to see the complete program which is available for transformation.)

To see what the derivation of *FPF* looks like in the easier to read form, refer to its derivation in Appendix C.

All formulas and proofs must be well-typed to be valid. But it is not enough for a syntactically valid proof to be well-typed, it must also be valid in the sense that every step, $\{e_1 =\{r\}\ e_2\}$, in the derivation must be a valid application of the rule `r`. A precise definition

of what it means for a derivation to be valid is the subject of the following section.

5.2 From Proofs to Laws

I have described the syntax of the PATH logic, but to use the language of logic, What are the "inference rules" for constructing new theorems from known theorems? Usually the answer to this question would be a set of natural deduction style rules. However, PATH takes a slightly different perspective on this: We start with trivial *derivations* and build larger *derivations* from them. That is, we start with a trivial derivation of the form

$\forall x_1, x_2, x_3, \ldots$. [p_1: f_1, p_2: f_2, ...] $\Rightarrow$ e

(where e is just an expression, i.e., a derivation without any {e_1={r}e_2} steps) and then we create larger derivations by modifying them in such ways as

- Apply a premise somewhere in the consequent.
- Add a premise.
- Remove quantified variables by instantiating them.
- Join derivations: e.g., $\forall a.[f_1] \Rightarrow q_1$ and $\forall b.[f_2] \Rightarrow q_2$ are joined into $\forall a, b.[f_1, f_2] \Rightarrow \langle q_1, q_2 \rangle$.

So, we always have a valid derivation of some program equivalence and we want to extend it till it is a valid derivation of what we want. The rules for modifying derivations are straightforward (and done automatically by PATH—cf. Chapter 6). I will not discuss the rules for modifying derivations but I will specify in this section what it means for a derivation to be valid. All rules for modifying PATH derivations preserve validity.

A derivation p is valid if it proves a formula f. The notation $A;\Gamma \vdash p \hookrightarrow f$ signifies that the derivation p proves the formula f given the laws in the law environment A and the types in the type environment Γ. This notation is overloaded to also work on expression equivalence and proofs thereof (i.e., syntactic classes f' and p'). The proves relation (on f and p) is defined in terms of the proves relation on f' and p':

$$\frac{A,r_1{:}f_1,\ldots,r_m{:}f_m\ ;\ \Gamma,v_1{:}t_1,\ldots,v_n{:}t_n \vdash p' \hookrightarrow q}{A;\Gamma \vdash (\forall v_1{:}t_1,\ldots,v_n{:}t_n\,.\,[r_1{:}f_1,\ldots,r_m{:}f_m] \Rightarrow p') \hookrightarrow (\forall v_1{:}t_1,\ldots,v_n{:}t_n\,.\,[f_1,\ldots,f_m] \Rightarrow q)}$$

(The variable q is used for elements of the syntactic class f'.) The relation $\hookrightarrow$ on f' and p' is defined inductively on the structure of the proofs and is given in Figure 5.4; it uses the `valid` relation which is defined by the rules in Figure 5.5 and the following axioms:

```
valid(A, Γ, (p ↦ e₁) e₂                 , {red}, e₁{e₂/p}            )
valid(A, Γ, ⟨e₁,...,eₙ⟩.jₙ              , {red}, eⱼ                  )
valid(A, Γ, (case ⟨e₁,...,eₙ⟩ (In.iₙ x), {red}, (eᵢ x)               )
valid(A, Γ, (μp ↦ f)                    , {red}, f{μp ↦ f / p}       )
valid(A, Γ, (prim⟨c₁,...,cₙ⟩)           , {red}, [[prim⟨c₁,...,cₙ⟩]] )
valid(A, Γ, (⊥ e)                       , {red}, ⊥                   )
valid(A, Γ, (⊥.mₙ)                      , {red}, ⊥                   )
valid(A, Γ, (case e ⊥)                  , {red}, ⊥                   )
valid(A, Γ, (prim⟨e₁,...,⊥,...,eₙ⟩)     , {red}, ⊥                   )
valid(A, Γ, (μ ⊥)                       , {red}, ⊥                   )
```

The `valid` relation uses the relation $\Gamma \vdash e :: \tau$ which signifies that e has type τ in type environment Γ. The equality in the rules is alpha-equivalence. The functions `first` and `final` extract expressions (syntactic class e) from expression equivalences (syntactic class f'). Their definitions are as follows:

```
first( {e₁ = e₂} )    = e₁
first( e₁ e₂ )        = (first e₁) (first e₂)
first( p ↦ e )        = p ↦ first e
first( ⟨e₁,e₂,...⟩ ) = ⟨first e₁,first e₂, ...⟩
...
```

$$\frac{A;\ \Gamma, x:\tau \vdash\ e\ \hookrightarrow\ q}{A;\ \Gamma \vdash\ (x:\tau \mapsto e)\ \hookrightarrow\ (x:\tau \mapsto q)} \qquad \frac{A;\ \Gamma \vdash\ e_1\ \hookrightarrow\ q_1,\quad A;\ \Gamma \vdash\ e_2\ \hookrightarrow\ q_2}{A;\ \Gamma \vdash\ e_1\, e_2\ \hookrightarrow\ q_1\, q_2}$$

$$\frac{A;\ \Gamma \vdash\ e_i\ \hookrightarrow\ q_i}{A;\ \Gamma \vdash\ \langle e_1, e_2, ..., e_n\rangle\ \hookrightarrow\ \langle q_1, q_2, ..., q_n\rangle} \qquad \frac{A;\ \Gamma \vdash\ e\ \hookrightarrow\ q}{A;\ \Gamma \vdash\ e.m_n\ \hookrightarrow\ q.m_n}$$

$$\frac{A;\ \Gamma \vdash\ e\ \hookrightarrow\ q}{A;\ \Gamma \vdash\ \text{case}\ e\ \hookrightarrow\ \text{case}\ q} \qquad \frac{e \in \{\text{In}.m_n, \mu, m, \texttt{prim}, \bot\}}{A;\ \Gamma \vdash\ e\ \hookrightarrow\ e}$$

$$\frac{A;\ \Gamma \vdash\ e_1\ \hookrightarrow\ q_1,\quad A;\ \Gamma \vdash\ e_2\ \hookrightarrow\ q_2,\quad \text{valid}(A, \Gamma, \text{final}(q_1), r, \text{first}(q_2))}{A;\ \Gamma \vdash\ \{e_1 = \{r\}\, e_2\}\ \hookrightarrow\ \{\text{first}(q_1) = \text{final}(q_2)\}}$$

Figure 5.4: The $\hookrightarrow$ (Proves) Relation

$$\frac{\text{valid}(A, \Gamma, e_2, \{j\}, e_1)}{\text{valid}(A, \Gamma, e_1, \{R\, j\}, e_2)} \qquad \frac{\Gamma \vdash e :: t_1 \to t_2,\ \ x \text{ not free in } e}{\text{valid}(A, \Gamma, e, \{eta\}, x \mapsto e\, x)}$$

$$\frac{\Gamma \vdash e :: +\langle t_1, ..., t_n\rangle}{\text{valid}(A, \Gamma, e, \{eta\}, \text{case}\, \langle \text{In}.1_n, ..., \text{In}.n_n\rangle\, e)} \qquad \frac{\Gamma \vdash e :: \times\langle t_1, ..., t_n\rangle}{\text{valid}(A, \Gamma, e, \{eta\}, \langle e.1_n, ..., e.n_n\rangle)}$$

$$\frac{\begin{array}{l} A, r:l;\ \Gamma\ \vdash\ p_j\ \hookrightarrow\ f_j\{i_1/x_1, ..., i_n/x_n\} \\ q' = q\{i_1/x_1, ..., i_n/x_n\} \\ e_1 = \text{first}(q') \\ e_2 = \text{final}(q') \\ l = \forall v_1 : t_1, ..., v_n : t_n\, .\, [f_1, ..., f_m]\, q) \end{array}}{\text{valid}((A, r:l), \Gamma, e_1, \{r\, \langle i_1, ..., i_n\rangle\, [p_1, ..., p_m]\}, e_2)}$$

Figure 5.5: The "valid" Relation

```
final( {e1 = e2} )    = e2
final( e1 e2 )        = (final e1) (final e2)
final( p ↦ e )        = p ↦ final e
final( ⟨e1,e2,...⟩ ) = ⟨final e1,final e2, ...⟩
...
```

Note that conjectures, `{?}`, are not used in the definition of `valid`, this is because derivations with conjectures are not valid.

5.3 The Design of the Logic

In this section I will describe why the logic has been designed as it is and compare it to other logics.

5.3.1 More expressive than the CIP logic

In [59], Pepper describes the logic used in the CIP transformation system. The CIP logic is basically two-level Horn Clause Logic. I.e., formulas are of the following form,

$$\begin{array}{l} \quad [[p, p, ...] \vdash p, [p, p, ...] \vdash p, ...] \\ \models \\ \quad [p, p, ...] \vdash p \end{array}$$

where p is comparable to a statement of program equivalence (f' in Figure 5.1). In CIP there are no explicit quantifiers, all quantifiers are implicit at the top level. This might suggest that one could not express *FPF*, which contains a nested quantifier:

$$\begin{array}{l} \forall C, F, G. \\ \quad C \perp = \perp \\ \; ; \; \forall x.\; C(F\ x) = G(C\ x) \\ \; \Rightarrow \\ \quad C(\mu F) = \mu G \end{array}$$

But CIP could express *FPF* as follows

$$\begin{array}{l} \quad c \perp = \perp \;,\; c(f\ \widehat{x}) = g(c\ \widehat{x}) \\ \models \\ \quad c(\mu f) = \mu g \end{array}$$

using a special class of "indeterminate" variables ($\widehat{x}$, $\widehat{y}$, ...) which allow it to handle one nested quantifier (but no more). So, CIP allows up to two levels of implication and up to two levels of quantifiers, while PATH allows for arbitrary nesting of implications and quantifiers.

By allowing for arbitrary nesting of implications and quantifiers, PATH has gained two advantages over the CIP logic: first, it becomes simpler as there are not two kinds of "implication" (one for each of the two levels allowed) and there are not two kinds of variables (quantified variables and indeterminate variables); second, it can express laws that cannot be expressed in the CIP logic (laws that are "third order"[2] and higher). Are such "higher order" laws necessary? They are necessary to meet the goal of being able to express all parametricity laws: parametricity theorems for n-th order functions are laws of order $n-1$. Although such higher order laws don't occur often[3], they easily arise when dealing with functions of high order. For instance, if we have the following function,

```
app :: ×⟨a→b,a⟩ → b
app ⟨f,x⟩ = f x
```

and convert it to Continuation Passing Style (CPS) [64] we get this:

```
appk :: ×⟨×⟨a, b→c⟩ → d, (×⟨a,b→c⟩ → d) → e⟩ → e

appk = let f1(f,k) =
         (let f2(x,k2) = (let r x = k2 x in f(x,r)) in k f2)
       in f1
```

Now, the fourth order function `appk` satisfies the third order law in Figure 5.6 (which is the free theorem for `appk`). Granted, this example may appear contrived and the law complex, but the law is valid and high order laws do arise naturally when high order code is involved (such as when using CPS).

5.3.2 Simpler than CIP Logic

In CIP, a number of syntactic predicates are also used in the premises of a law; these predicates are as follows:

[2]The order of a law being the depth of nested implications in it.

[3]No examples in this dissertation use more than two levels of implications or quantifiers.

```
∀a,b,c,d,e,f,f′,k,k′.
   ∀x,k2,k2′.
     ∀y. c (k2 y) = k2′ (b y)
    ⇒
     d (f (x, k2)) = f′ (a x, k2′)
 ; ∀g,g′.
     (∀x,k2,k2′.
       ∀y. c (k2 y) = k2′ (b y)
      ⇒
       d (g (x, k2)) = g′ (a x, k2′)
     )
    ⇒
     e (k g) = k′ g′
 ⇒
   e (appk (f, k)) =  appk (f′, k′)
```

Figure 5.6: Parametricity Theorem for `appk`

1. $m = \texttt{Type}[E]$ (The term E has type m.)

2. $\texttt{New}[v]$ (The variable v does not occur in any of the terms in the given law.)

3. $\texttt{Occurs}[v,E]$, $\texttt{NotOccurs}[v,E]$ (The variable v occurs in program E, does not occur in program E.)

4. $F = \texttt{Declaration}[f]$ (The identifier f has the term F as the right-hand side of its declaration.)

In PATH none of these syntactic predicates are necessary: Predicate 1 is not necessary because PATH is a typed logic working on a typed language, any typing constraints are implicit in the types of the quantified variables. Predicates 2 and 3 are not necessary because of the variable convention in PATH. Predicate 4 becomes unnecessary due to both the variable convention and the use of an explicit fix point operator in path. E.g., in CIP, one would use a side condition such as

```
F[f]=Declaration[f]
```

in order to refer to the defining equation for a recursive f. In PATH, one simply can write laws about the recursive function "μ`f` $\mapsto$ `F f`" without referring to a notion of declaration.

5.3.3 Making the Logic as Simple As Possible

Note the syntactic category f of formulas in Figure 5.1. This corresponds to first order logic without negation ($\neg$) and disjunction ($\vee$) but with implication ($\Rightarrow$) and conjunction (here ";"). For the purposes of program *transformation*, negation is not needed as we do not want to prove programs unequal, only equal. Disjunction is also dropped: laws of the form $f \Rightarrow (e_1 = e_2 \vee e_3 = e_4)$ are not directly useful (we need to know which of the equivalences is valid); a law of the form $(e_1 = e_2 \vee e_3 = e_4) \Rightarrow e_5 = e_6$ is easily replaced by the equivalent two laws: $e_1 = e_2 \Rightarrow e_5 = e_6$ and $e_3 = e_4 \Rightarrow e_5 = e_6$.

There are additional syntactic restrictions imposed by the definition of f: No quantifiers are allowed except at the outermost position in a formula and conjunction (using ";") is only allowed in the antecedent of the $\Rightarrow$. The reason for these syntactic restrictions (no expressiveness is lost by them[4]) is to simplify the *application* of laws. Since the consequent is always an expression equivalence (f'), a law is always applied in the same manner: giving an instantiation of the quantified variables and a sequence of sub-derivations, one for each premise. (Compare this to the situation where conjunctions or general formulas could be in the consequent, one would need different ways to apply different forms of laws.)

As a result of the above restrictions, the form of derivations (or proofs, p) become simple and directly follow the form of the formula to be proved. The syntactic form of formulas and proofs have been made nearly identical. Another result is that derivations become

[4]The formula $a \Rightarrow \forall x.b$ can be reformulated as $\forall x.a \Rightarrow b$ (x not free in a). The formula $a \Rightarrow (b \wedge c)$ could be reformulated as the two formulas $a \Rightarrow b$, $a \Rightarrow c$. Alternatively, one could reformulate $a \Rightarrow (e_1 = e_2 \wedge e_3 = e_4)$ as $a \Rightarrow \langle \{e_1 = e_2\}, \{e_3 = e_4\} \rangle$.

straightforward to read and write. Note also that when a law that has premises is applied the proof of the premises is proved right where the law is applied and is not distant in the derivation.

The equality relation between programs is transitive, reflexive, and compatible (that is, for all C, $e_1 = e_2 \Rightarrow C[e_1] = C[e_2]$); but the user doesn't need to think about or use such laws: these are implicit in how the user manipulates and constructs derivations, described in Chapter 6.

5.3.4 Predicates

From Figure 5.1 it can be seen that there is only one predicate in the PATH logic, $=$. This is in contrast to the five predicates in the CIP logic:

1. `Equivalent`$[e_1, e_2]$ (i.e., $e_1 = e_2$)
2. `Descendant`$[e_1, e_2]$
3. `Determinate`$[e]$
4. `LessDefined`$[e_1, e_2]$ (Or, $e_1 \sqsubseteq e_2$)
5. `Defined`$[e]$

The first is simply the program equivalence used in PATH. The second and third are useful in the CIP system because of its non-deterministic constructs but are not applicable to PATH-L as it has lacks non-deterministic constructs. The fourth and fifth allow for reasoning about the denotational meanings of programs [71, 28], i.e., `LessDefined`$[e_1, e_2]$ states that the domain theoretic meaning of e_1 is less defined than, or approximates, the meaning of e_2. (Or we could formulate `LessDefined` operationally: `LessDefined`$[e_1, e_2]$ if

for all C, $C[e_1] \Downarrow \Rightarrow C[e_2] \Downarrow$.) In CIP, the predicate `Defined`$[e]$ is overloaded: if e is in a flat domain, it signifies that $e \neq \bot$, if f is a function, `Defined`$[f]$ signifies $\forall x$.`Defined[`$f\,x$`]`, i.e., that f is total. For the purpose of clarity in what follows, I will use `Total` to refer to the second use of `Defined`.

These last two semantic predicates allow for reasoning which PATH cannot do: PATH only allows for reasoning about program equivalence. PATH certainly *could* be extended to allow for additional semantic relations besides program equivalence; however, the argument against this is the same as the argument in Section 4.5 for not adding Sands's improvement relation: it makes the system more complex and in practice the extra expressiveness has not been needed: the primitive laws, which use only equivalence, have been sufficient to achieve all desired transformations to which PATH has so far been applied to. The goal of reasoning about either `LessDefined` or $\trianglerighteq$ (improvement) is to eventually prove programs *equivalent*, and if proving two programs equivalent can be done directly, so much the better.

PATH has demonstrated a surprising expressiveness for a system that has only two primitive laws about μ and uses only program equivalence. This would not have been discovered had I not stuck to my original design principles (one being that PATH should not require knowledge of domain theory).

CIP can make the statement `Defined`$[e]$ (i.e., $e \neq \bot$). PATH can express that a program is undefined, but because it does not have inequality it cannot state that a program is *not* undefined. (CIP does not have inequality either, just the `Defined` predicate.) This lack of an $e \neq \bot$ predicate in PATH has pervasively influenced PATH: Many of the laws in the CIP system are littered with `Defined` predicates[5]; on the contrary, many of the laws in PATH are littered with strictness conditions. For example, note *FPF*:

`∀C,F,G. [ C ⊥ = ⊥ ; ∀x. C(F x) = G(C x) ] ⇒ C(`μ`F) = `μ`G`

[5] And it needs numerous laws in order to be able to prove `Defined`$[e]$.

One might have had this law instead

$$\forall C,F,G.\ [\ \texttt{Total}(\mu G)\ ;\ \forall x.\ C(F\ x) = G(C\ x)\ ] \Rightarrow C(\mu F) = \mu G$$

where a strictness condition on the context is replaced with a `Total` condition on the result. The strictness condition seems preferable for two reasons: one, a strictness condition is often simple to prove (totality of functions may be much harder to prove); two, reasoning about infinite data structures and partial functions (neither of which is "`Total`") is possible. However, there may be cases where one would want to use *FPF* in a non-strict context and the alternative *FPF* law would be what is needed. Determining what expressiveness might be gained with the alternative *FPF* is a matter of further research.

So, to summarize the differences between PATH and CIP: the logic of PATH is more expressive than the *logic* of CIP (i.e., it can express formulas that CIP cannot). It is also simpler because it does not require ad hoc syntactic predicates. However, PATH is less expressive than CIP with respect to the available semantic predicates: PATH has $=$ but does not have $\sqsubseteq$ and `Defined`. However, there is nothing (but the desire for simplicity) that would keep PATH from being extended with these predicates.

Extending the system with the `Defined` predicate may not gain us as much as we think: Pepper in [59] makes the statement "`Defined` ... [is] less amenable to treatment within the framework ... Therefore one often uses corresponding syntactic predicates ... to guarantee the desired semantic properties ... Defined [is frequently guaranteed] by the absence of recursion/iteration."

5.3.5 Semantics of Expression Equivalence ($=$)

What precisely is the meaning of the predicate $=$? It corresponds to observational equivalence at base types. More formally, we say two programs are equivalent ($e_1 = e_2$) if for all

contexts C where $C[e_1]$ and $C[e_2]$ are of `Int` type, $C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$. The before-mentioned laws are sound (but not complete) with respect to observational equivalence.

As noted in Chapter 2, PATH uses a call-by-name equivalence rather than call-by-need. This is because call-by-name is more expressive: it allows for both removing and introducing the sharing of computation in transformations. The semantics and the primitive rules could be easily adapted (as in [2, 52]) to support call-by-need; however, if call-by-need was needed, it would be preferable to extend PATH to allow for multiple program relations such as $=_{name}$ (equivalent under call-by-name), $\leq_{name}$ (equivalent under call-by-name with less sharing), $=_{need}$ (equivalent under call-by-need, i.e., the intersection of $\leq_{name}$ and $\geq_{name}$) so as to allow for both the greater expressiveness of call-by-name and the ability to reason about sharing in call-by-need. However, under call-by-need, many of the derived laws would no longer be derivable from the primitive laws but would need to be primitive laws themselves (e.g., *GC*, *Inline-Bndg*, *Inline-Self*, *Inline-Body*, etc.). This results because under call-by-need the reduction rule

```
(p ↦ e1) e2 ={red} e1{e2/p}
```

becomes the less general rule (where `c` is a canonical form):

```
(p ↦ e1) c ={red} e1{c/p}
```

5.4 Primitive Rules

This section describes the primitive rules in the PATH logic. Note that these are *rules*, not *laws*, they cannot be expressed as formulas in PATH but require meta-notation, such as that needed for substitution or "...". Note also that these rules do not follow the PATH variable convention. First, there are the reduction rules which correspond to the operational semantics of PATH-L (the variable `c` is for canonical forms):

$$(p \mapsto e_1)\ e_2 =\{red\}\ e_1\{e_2/p\}$$

$$\langle e_1,\ldots,e_n\rangle.j_n =\{red\}\ e_j$$

$$\texttt{case}\ \langle e_1,\ldots,e_n\rangle\ (\texttt{In}.i_n\ x) =\{red\}\ e_i\ x$$

$$\mu p \mapsto f =\{red\}\ f\{\mu p \mapsto f\ /\ p\}$$

$$\mathit{prim}\langle c_1,\ldots,c_n\rangle =\{red\}\ [\![\mathit{prim}\langle c_1,\ldots,c_n\rangle]\!]$$

In order to satisfy various strictness conditions, we need facts about strictness, thus there are the following rules about the strictness of the basic language constructs (note that *prim* represents integer primitives):

$$\begin{aligned}
\bot\ e &=\{red\}\ \bot\\
\bot.m_n &=\{red\}\ \bot\\
\texttt{case}\ e\ \bot &=\{red\}\ \bot\\
\mathit{prim}\langle e_1,\ldots,\bot,\ldots,e_n\rangle &=\{red\}\ \bot\\
\mu\ \bot &=\{red\}\ \bot
\end{aligned}$$

These are the minimal set of rules about strictness. From these are derived other laws about strictness, e.g., the law *Case-Strict*:

$$\forall E.\ \texttt{case}\ \bot\ E = \bot$$

The primitive rule {red} is used for both reduction and strictness properties. Note that these rules for strictness are similar to the definition of reduction contexts (cf. Section 2.2). It follows from these laws that reduction contexts are strict. Generally types have been dropped for clarity, but the rules for type directed expansion give an example where the types are essential:

$$\begin{aligned}
&\forall x{:}a\to b && .\ x =\{eta\}\ v \mapsto x\ v && (v \text{ not free in } x)\\
&\forall x{:}\times\langle t_1,t_2,\ldots,t_n\rangle && .\ x =\{eta\}\ \langle x.1_n,\ x.2_n,\ \ldots\ ,\ x.n_n\rangle &\\
&\forall x{:}+\langle t_1,t_2,\ldots,t_n\rangle && .\ x =\{eta\}\ \texttt{case}\ \langle \texttt{In}.1_n,\texttt{In}.2_n,\ldots,\texttt{In}.n_n\rangle\ x &
\end{aligned}$$

These allow for expansion or contraction of expressions based on their type. All three of these rules will be referred to as *eta*. There is also the *letrec* rule:

```
  letrec f1=F1;f2=F2;...;fn=Fn; g1=G1;g2=G2;...;gm=Gm in M
={letrec}
  letrec f1=F1;f2=F2;...;fn=Fn; ⟨g1,g2,...,gm⟩=⟨G1,G2,...,Gm⟩ in M
```

This rule allows for laws about `letrec` to be more general than they first appear. For instance, note the law *Inline-Bndg*:

```
∀C,F,G. letrec f=F⟨f,g⟩; g=G⟨f,{ f=F⟨f,g⟩ },g⟩ in C⟨f,g⟩
```

It appears that this only applies when a `letrec` has exactly two bindings. But this law is actually more general as it allows the inlining of a binding in a `letrec` with any number of bindings. This is true because by using *letrec*, the `g` binding can represent *all* the bindings except the `f` binding to be inlined. In most derivations the *letrec* is done implicitly when applying these laws. I have not treated this rule as one of the key laws (or rules) about μ because every instance of it can be derived using *FPF*. See Appendix C for a derivation of the law *Letrec-Equiv*:

```
∀F,G1,G2.
  letrec f=F⟨f,g1,g2⟩;⟨g1,g2⟩=⟨G1⟨f,g1,g2⟩,G2⟨f,g1,g2⟩⟩ in ⟨f,g1,g2⟩
=
  letrec f=F⟨f,g1,g2⟩; g1=G1⟨f,g1,g2⟩; g2=G2⟨f,g1,g2⟩ in ⟨f,g1,g2⟩
```

This is *letrec* for the case when $n = 1$ and $m = 2$. (Similar derivations could be done for any n and m.) So, the *letrec* rule does not add more expressiveness to PATH, it just makes all the other rules about `letrec` more easily applied.

There are a few "rules" used in derivations which are not actual rules or laws in PATH because they are effectively "no-ops". For instance, there is `{rename}` which allows us to change variable names, e.g.,

```
x ↦ F x ={rename} y ↦ F y
```

and there is `{SS}` (Syntactic Sugar) which allows for conversion between syntactic representations, e.g.,

```
let x=e in C x ={SS} (x ↦ C x) e
```

and there is also `{def.`*v*`}` which inlines the definition of *v* from the PATH-L prelude. These are effectively "no-ops" because in PATH none of these rules change the program: programs equivalent up to renaming are considered equal, programs equivalent up to syntactic sugar are considered equal[6], and prelude variables are treated as equivalent to their definitions, so there is no need to inline prelude variables. In addition, programs equivalent up to re-ordering of bindings in a `letrec` are considered equal.

5.5 Primitive Laws

Besides the primitive rules just described, PATH has five primitive laws. I.e., the rule catalog starts out with these laws. These are *laws*: they can be expressed in the PATH logic but cannot be derived from the primitive rules. Before describing these laws, I will explain a notation used in some of these laws; the notation $\langle^{\mathtt{i}}$ `C[i]`$\rangle$ represents an n-tuple, the meaning of which is as follows:

$\langle^{\mathtt{i}}$ `C[i]`$\rangle$ = $\langle$`C[1`$_{\mathtt{n}}$`],C[2`$_{\mathtt{n}}$`],...,C[n`$_{\mathtt{n}}$`]`$\rangle$

The reader could view n-tuples as meta-notation but in fact it will be shown in Chapter 8 how n-tuples can be a standard construct in the language. The law, *Inst* (Instantiation),

```
∀F,H,X.
  H ⊥ = ⊥
 ⇒
  H(case ⟨i y ↦ F.i y⟩ X) = case ⟨i y ↦ H(F.i y)⟩ X
```

[6]The various syntactic equivalences are listed in Appendix B.

is used to move a strict function (`H` $\perp = \perp$) into the branches of a `case`. This is a useful law which happens to be the free theorem [82] for `case`. A rule similar to *Inst* is also part of both fold/unfold and expression procedures but is only used left-to-right in those systems.

Although I focused on the laws *FPF* and *FPE* in Chapter 4, they are not primitive laws in PATH. The two primitive laws of PATH for reasoning about μ are *FPI* (Fixed Point Induction) and *FPD* (Fixed Point Duplication). The *FPI* law

```
∀C,D,F,G.
  C ⊥ = D ⊥
; ∀x,y . {C x = D y} ⇒ {C(F x) = D(G y)}
⇒
  C(μF) = D(μG)
```

gives us a form of fixed point induction, or Scott induction, for PATH (thus the name). *FPI* can be proved by fixed point induction with the predicate $P(x,y) = Cx = Dy$. This law, like *FPF*, is a free theorem for μ (*FPF* is the free theorem for μ generated from a binary relation, *FPI* is the free theorem generated from a ternary relation). *FPI* is more general than *FPF*: it can prove two programs equivalent even when the recursion does not appear in a strict context (*FPF* is basically *FPI* with `D` instantiated to `id`). The second primitive law for μ is *FPD*,

```
∀F,G.
  letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in f
=
  letrec f=(letrec g=G⟨f,g⟩ in F⟨f,g⟩) in f
```

which allows for mutual recursion to be expressed as two fixed points [10, 51, 87]. This is also known as Bekic's Theorem, proofs can be found in Bekic [10] and Winskel [87].

There is a law for induction on lists, *List-Induct*,

```
∀ C : List a → b, D : List a → b, xs : List a.
  C ⊥ = D ⊥
; C Nil = D Nil
; ∀x,xs. C xs = D xs ⇒ C( Cons⟨x,xs⟩ ) = D( Cons⟨x,xs⟩ )
⇒
  C xs = D xs
```

and a law for eta expansion of n-tuples, *N-Tuple-Eta*:

$$\forall x{:}\times a \;\; . \;\; x = \langle^{i}\; x.i \rangle$$

The reader may refer to Appendix B for a listing of all the primitive rules and laws. No mention has been made of laws for the primitive functions as they are of little theoretical interest. However, a practical transformation system needs a large number of laws about the primitive functions (e.g., associativity of integer addition).

Chapter 6

The PATH User Interface

In the previous chapter I discussed the PATH logic; in this chapter I hope to demonstrate its aptness for displaying and manipulating derivations.

One could type in the derivation of FPF (as written in Figure 5.3) and ask PATH to verify it; PATH will either print the law which it proves (in this case *FPF*) or return an error message stating where it is wrong. As useful as this may be, we would rather construct a proof which is guaranteed to be valid. This chapter describes the user interface to PATH which helps the user to create valid proofs.

6.1 The User Interface—Overview

The PATH user interface is implemented inside of the Emacs text editor using various Emacs Lisp functions and an external Haskell program. The user chooses functions to be performed or a law to be applied using a pop-up menu. Emacs, via Lisp functions, annotates the derivation and invokes a Haskell program that processes the annotation and returns a new derivation.

The Haskell program that does the bulk of the work has three main entry points, each one inputs a derivation:

get-formula Outputs the formula that the derivation derives or outputs an error message if it is not a valid derivation.

simplify Removes conjectures as follows: if any conjecture in the derivation can be satisfied by a primitive rule or by a premise, the premise is replaced with the corresponding rule.

meta-eval Processes an annotated derivation and applies the indicated rule, law, or meta-program to the selected sub-derivation.

The PATH user interface is just an Emacs mode in which certain functions and menus are enabled; this mode knows nothing of the syntax of derivations. The user works by selecting sub-derivations and choosing functions from menus. The user selects sub-derivations or expressions by moving the cursor to the end of the desired expression[1].

The functions available to the user via Emacs menus are the following:

get-formula Displays the formula derived by the derivation in a second window.

simplify Simplifies the derivation as described above. This is rarely used as all other functions that change the derivation automatically call **simplify**.

apply-rule Applies one of the primitive rules (*red*, *eta*, *R eta*) to the selection.

apply-law Applies a law from the rule catalog (*Inst*, *FPF*, *FPI*, *FPD*, ...) to the selection.

apply-premise Applies a premise from the derivation to the selection.

[1]This could require inserting parentheses when the desired expression is inside a larger expression; but in most cases, just a single mouse click is needed to select a region.

apply-mp Applies one of three meta-programs (*norm*, *eval*, *eval'*) to the selection. (These meta-programs are described in Section 6.3.)

meta-eval Applies the Haskell **meta-eval** function to the annotated derivation. This requires that the user inserts the annotations by hand. This is needed at times, particularly when the *R red* rule is applied (the new program must be entered).

meta-eval-no-simplify Calls meta-eval but does not simplify the derivation.

The next section provides further details by showing how to create a derivation using PATH.

6.2 Deriving *FPF* using PATH

Here I will show how we can create the derivation of *FPF* in Figure 5.3. We enter the trivial derivation in Figure 6.1 as a starting point. To ensure that no errors have been made in the entry of this derivation, we press get-formula and PATH displays in a separate window the law that this derivation derives:

```
∀C,F,G.
[ {C ⊥ = ⊥}
; ∀x. {C (F x) = G (C x)}
] ⇒
  C (μF)
```

This might look odd because there is a program, not an equality, in the consequent. However, this is correct because an expression context without any $\{p_1{=}p_2\}$ holes is also a valid expression equivalence: $C(\mu F)$ just corresponds to the equality $C(\mu F){=}C(\mu F)$. Note that the law is virtually identical to the derivation at this point. The next step is to move the cursor to the end of expression $C(\mu F)$ (i.e., selecting it) and picks FPI (from a pop-up menu) which results in the derivation in Figure 6.2. (In this sequence of figures underlines will represent the selected expression; the boxed text represents the action applied to the

selection; and the character '+' will mark the lines in the subsequent derivation that were changed or new as a result of the last operation.)

PATH has successfully matched against *FPI* and applied this law to the program, giving us the program `x(`μ`y)`. It has filled in the instantiation $\langle$`C,x,F,y`$\rangle$ and given us templates for all sub-derivations required for *FPI*. If necessary, PATH changes the names of variables (quantified, let, or lambda bound) to avoid variable conflict problems.

PATH must return a valid derivation, one with the result of the application in the program; but since the result of applying *FPI* is not known, PATH has created two unknown variables, `x` and `y`, and brought them into the derivation using the `ulet` (unknown let) construct, which is a placeholder for an unknown value. These unknowns will be either filled in by the user or filled in by PATH automatically. Note the `={?}` in the templates for the sub-derivations, this kind of derivation step is called a conjecture and acts as a placeholder for an unfinished derivation.

Now at this point, the derivation is syntactically correct, but it is not valid (in the sense that it derives a law) due to the presence of conjectures and unknowns, so if we press get-formula, we get

```
Error: bad step:
   C ⊥ ={?} x ⊥
conjecture present
```

We know what `x` should be—it should be the empty context—but since *FPI* expects a function here, we can use the identity function, "`z`$\mapsto$`z`". So, we fill in the value, giving the derivation in Figure 6.3. The `ulet` was changed into an `mlet`, an `mlet` (for meta-let) is like a `let` but it exists only in the program derivation, not in the program. The program being transformed is the program with all `mlet`'s expanded out. Thus, we cannot inline a variable

bound by `mlet`, it *is* the value bound by `mlet`. We have specified `x`, but we do not need to know the value of `y` before proceeding, we will determine `y` as we go[2].

Since `x` is known, we see two opportunities for simplification, we reduce the applications of `x`: select "`x(`μ`y)`" and press normalize then select "`x` $\perp$" and press normalize. These actions result in the derivation in Figure 6.4.

Applying normalize to "`x` $\perp$" caused the derivation to go "backward" (i.e., right to left). A rule can be applied to any "apply point." An apply point is a point in one of the following four regions: the start of the derivation, the end of the derivation, the right of a conjecture, and the left of a conjecture. When a rule is applied to the start of the derivation or the right of a conjecture, the rule is applied and then reversed. Thus, the user can work from whatever direction is easiest, not necessarily from left to right.

The normalize function invokes a built-in PATH meta-program[3]: it applies the rule *red* as many times as possible (even under lambdas, but it does not reduce μ redexes).

After "`x` $\perp$" was replaced by "$\perp$ `={R red} x` $\perp$", PATH detected that the conjecture "`C` $\perp$ `={?}` $\perp$" was satisfied by the premise labeled P_1 and changed the conjecture to P_1. Any conjecture that can be replaced by a premise or primitive rule is automatically removed.

So, the first premise of *FPI* is satisfied, let's turn our focus to the second conjecture. The "`C(F a)`" matches the left hand side of premise P_2, so we select it and press P_2, giving the derivation in Figure 6.5.

Now we see an opportunity to apply the premise P_3, so with two mouse clicks we have the derivation in Figure 6.6. Just as program equalities can be nested inside expressions, so also program derivations can be nested inside expressions. Note how this is used: above the

[2] In this case we know what `y` should be, but often one does not know the program one is aiming for. I will leave `y` unspecified for illustrative purposes.

[3] A meta-program here being a program (implemented in Haskell) that takes a derivation and transforms it into another derivation.

```
∀C,F,G.
[ P1: {C ⊥ = ⊥}
; P2: ∀x. {C (F x) = G (C x)}
] ⇒
  C(μF)                    FPI
```

Figure 6.1: Deriving *FPF* (1)

conjecture, we have G{ C a ={P_3} x b}, the "right side" of this derivation corresponds to the program "G(x b)", so, what we are left to prove is "G(x b)=x(y b)". We immediately see two opportunities for reductions (applying x). Doing that we get the derivation in Figure 6.7.

Now we are left with the conjecture "G b={?}y b", and it is easy to see what we need to do: we give the unknown y the value G and press simplify (simplify is done automatically after other actions: it replaces conjectures with applicable rules) and we get the complete derivation for *FPF* in Figure 6.8. To confirm this we press get-formula, giving

```
∀C,F,G.
[ {C ⊥ = ⊥}
; ∀x. {C (F x) = G (C x)}
] ⇒
  {C (μF) = μG}
```

Note that the `mlet`'s have disappeared. One thing to note about this derivation is that after the initial derivation was entered virtually no text was entered (just the text "`z↦z`" and the text "`G`"). Many derivations require little textual entry but others require more.

6.3 Meta-programs in PATH

Although it is a user-directed system, PATH should still automate as much of the tedious work of program transformation as possible. This is done primarily through the use of

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
+  ulet y = ? in
+  ulet x = ? in          edit
    { C (μF)
+   ={FPI ⟨C,x,F,y⟩
+      [ { C ⊥ ={?} x ⊥ }
+      ; ∀a,b.
+        [P3: {C a = x b}] ⇒
+        { C (F a) ={?} x (y b) }
+      ]}
+      x (μy)
+    }
```

Figure 6.2: Deriving *FPF* (2)

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
   ulet y = ? in
+  mlet x = z ↦ z in
    { C (μF)
    ={FPI ⟨C,x,F,y⟩
      [ { C ⊥ ={?} x ⊥ }        normalize
      ; ∀a,b.
        [P3: {C a = x b}] ⇒
        { C (F a) ={?} x (y b) }
      ]}
      x (μy)                    normalize
    }
```

Figure 6.3: Deriving *FPF* (3)

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
   ulet y = ? in
   mlet x = z ↦ z in
    { C (μF)
    ={FPI ⟨C,x,F,y⟩
+     [ { C ⊥ ={P1} ⊥ ={R red} x ⊥ }
      ; ∀a,b.
        [P3: {C a = x b}] ⇒
        { C (F a) ={?} x (y b) }   P2
      ]}
      x (μy)
+   ={red}
+     μy
    }
```

Figure 6.4: Deriving *FPF* (4)

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
   ulet y = ? in
   mlet x = z ↦ z in
    { C (μF)
    ={FPI ⟨C,x,F,y⟩
      [ { C ⊥ ={P1} ⊥ ={R red} x ⊥ }
      ; ∀a,b.
        [P3: {C a = x b}] ⇒
+       { C (F a) ={P2 ⟨a⟩} G (C a) ={?} x (y b) }   P3
      ]}
      x (μy)
    ={red}
      μy
    }
```

Figure 6.5: Deriving *FPF* (5)

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
   ulet y = ? in
   mlet x = z ↦ z in
    { C (μF)
    ={FPI ⟨C,x,F,y⟩
      [ { C ⊥ ={P1} ⊥ ={R red} x ⊥ }
      ; ∀a,b.
        [P3: {C a = x b}] ⇒
        { C (F a)
        ={P2 ⟨a⟩}
+         G { C a ={P3} x b }    red
        ={?}
          x (y b)                red
        }
      ]}
      x (μy)
    ={red}
      μy
    }
```

Figure 6.6: Deriving *FPF* (6)

```
   ∀C,F,G.
   [ P1: {C ⊥ = ⊥}
   ; P2: ∀x. {C (F x) = G (C x)}
   ] ⇒
   ulet y = ? in        edit  simplify
   mlet x = z ↦ z in
    { C (μF)
    ={FPI ⟨C,x,F,y⟩
      [ { C ⊥ ={P1} ⊥ ={R red} x ⊥ }
      ; ∀a,b.
        [P3: {C a = x b}] ⇒
        { C (F a)
        ={P2 ⟨a⟩}
+         G { C a ={P3} x b ={red} b }
        ={?}
+         y b
+       ={R red}
          x (y b)
        }
      ]}
      x (μy)
    ={red}
      μy
    }
```

Figure 6.7: Deriving *FPF* (7)

```
    ∀C,F,G.
    [ P1: {C ⊥ = ⊥}
    ; P2: ∀x. {C (F x) = G (C x)}
    ] ⇒
+   mlet y = G in
    mlet x = z ↦ z in
     { C (μF)
     ={FPI ⟨C,x,F,y⟩
       [ { C ⊥ ={P1} ⊥ ={R red} x ⊥ }
       ; ∀a,b.
         [P3: {C a = x b}] ⇒
         { C (F a)
         ={P2 ⟨a⟩}
           G { C a ={P3} x b ={red} b }
         ={R red}
           x (y b)
         }
       ]}
       x (μy)
     ={red}
       μy
     }
```

Figure 6.8: Deriving *FPF* (8)

meta-programs (a meta-program is a built-in program that applies multiple rules to a derivation based on some strategy). PATH currently has three built-in meta-programs: *normalize*, *eval*, and *eval'*. These meta-programs can be applied to any selection in the derivation.

Normalize applies the rule *red* as many times as possible with a leftmost-outermost strategy until no more redexes are left. It performs reductions even under lambdas. In order to avoid non-termination when applied to programs with μ, it does not reduce μ redexes.

The second meta-program is *eval*: it applies *red* at reduction contexts until a canonical form is reached. Unlike the standard evaluator, it need not be applied to a closed expression but it allows for free variables in the term it is applied to. The third meta-program, *eval'*, is a variant of *eval* that stops after a set number of steps (this keeps PATH from going into an infinite loop when the user tries to evaluate a non-terminating program). Using *eval*, PATH can be used as a simple evaluator. E.g., if we apply *eval* to this program

```
plus (succ (succ 3)) (succ 5)
```

we get the following derivation:

```
{ plus { succ (succ 3)
       ={red}
         plus 1 { succ 3 ={red} plus 1 3 ={red} 4 }
       ={red}
         5
       } { succ 5 ={red} plus 1 5 ={red} 6 }
={red}
  11
}
```

The resulting derivation gives us a complete execution trace of the evaluation. Viewing such a trace can be useful in understanding the order of evaluation in PATH-L. In the next version of PATH I plan to allow for selective displaying and hiding of parts of the program or of the derivation. Together with the *eval* meta-program, this would allow PATH to be used as an effective debugger for functional languages: we apply *eval* but only display the final result, then the user can explore whatever parts of the execution trace he chooses.

Although these three meta-programs have been found extremely useful, it would certainly be useful to allow for others. It is planned to allow the user to specify meta-programs by providing a set of laws (laws without premises) which will be applied with a left-most outermost strategy until none can be applied to the program. A further extension would be to allow the user to write their own strategies.

6.4 Dealing with Changes in Specifications

What does the user do when the program changes after we have already transformed it? It is not to be expected that the old derivation will automatically work on the new program.

But it might be hoped that small changes in the program will result in small changes to the derivation. With the approach where the derivation is stored as a script of commands, the best we can do is replay the script until it breaks and then continue by hand from there; trying to figure out how and when the original script can be re-used is very difficult. In the PATH approach, there seems to be more promise of reusing derivations because derivations are displayable and manipulable.

Here is an example that gives a sketch of how one might proceed when the original program (or specification) changes. Let's assume a program derivation starts as follows

```
{
  ⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs⟩
={R red}
  { xs ↦ ⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs⟩
  = {Cata-Merge-2}
    cata ⟨⟨y,z⟩ ↦ ⟨f₁⟨y,z.1⟩, f₂⟨y,z.2⟩⟩, ⟨b₁,b₂⟩⟩
  } xs
=
  ...
}
```

and that the initial program is changed to be the following program:

```
⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs, cata⟨f₃,b₃⟩ xs⟩
```

What we can do is add this new program to the top of the derivation using a conjecture, giving this derivation:

```
{
  ⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs, cata⟨f₃,b₃⟩ xs⟩
={?}
  ⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs⟩
={R red}
  { xs ↦ ⟨cata⟨f₁,b₁⟩ xs, cata⟨f₂,b₂⟩ xs⟩
  ={Cata-Merge-2}
    cata ⟨⟨y,z⟩ ↦ ⟨f₁⟨y,z.1⟩, f₂⟨y,z.2⟩⟩, ⟨b₁,b₂⟩⟩
  } xs
=
  ...
}
```

The objective is to remove the conjecture. We first "push" the conjecture through the first {R red} step giving the new derivation:

```
{
  ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
={R red}
  { xs ↦ ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
  ={?}
    (xs ↦ ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs⟩
  ={Cata-Merge-2}
    cata ⟨⟨y,z⟩ ↦ ⟨f1⟨y,z.1⟩, f2⟨y,z.2⟩⟩, ⟨b1,b2⟩⟩
  } xs
=
  ...
}
```

Now we see that we cannot push the conjecture through the {Cata-Merge-2} unless *Cata-Merge-2* could be applied to the expression above the conjecture. Since *Cata-Merge-2* merges a 2-tuple of cata applies, we need to generalize *Cata-Merge-2* to *Cata-Merge-N*[4], which merges an n-tuple of cata applies. If we do this generalization and apply the more general law, we get the following derivation

```
{
  ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
={R red}
  { xs ↦ ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
  ={?}
    (xs ↦ ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs⟩
  ={Cata-Merge-N}
    cata ⟨⟨y,z⟩ ↦ ⟨f1⟨y,z.1⟩, f2⟨y,z.2⟩⟩, ⟨b1,b2⟩⟩
  } xs
=
  ...
}
```

where nothing has changed except the rule name, but we have a more general derivation. Now we can push the conjecture through the {Cata-Merge-N} step giving this derivation:

[4]See Cata-Merge in Appendix C.

```
{
  ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
={R red}
  { xs ↦ ⟨cata⟨f1,b1⟩ xs, cata⟨f2,b2⟩ xs, cata⟨f3,b3⟩ xs⟩
  ={Cata-Merge-N}
    cata ⟨ ⟨y,z⟩ ↦ ⟨i  f1⟨y,z.1⟩, f2⟨y,z.2⟩, f3⟨y,z.3⟩⟩
         , ⟨b1,b2,b3⟩
         ⟩
  } xs
={?}
  ...
}
```

Modifying the derivation proceeds until we have removed all conjectures. If the next step in the derivation does not depend on the structure of the argument to `cata`, then pushing the conjecture through that step is trivial. So, we don't necessarily have to modify the complete derivation. There is no guarantee how easy it will be to completely develop a new derivation, but at least there is a chance to re-use a large portion of the original derivation.

6.5 The Advantages of Manipulable Derivations

In Section 1.3.4, it was mentioned that previous transformation systems allow the user to view the current state of the transformed program but the derivation itself is hidden from the user, possibly stored as a script of commands. Hopefully some of the advantages of manipulating *derivations*, not programs, can seen from the above examples. Here the advantages will be elucidated.

Derivations are more robust.

Having manipulable derivations makes PATH more robust in the face of changing specifications: As seen in Section 6.4, having a derivation that we can see and change allows us to modify it to work on a new specification.

Many aspects become visible.

Besides the fact that the derivation itself becomes visible, many things are now visible to the user which otherwise would not be:

- All the valid premises are kept in view.
- All unfinished derivations are kept in view.
- The goal of the transformation is kept in view.

Derivations can be developed non-linearly.

The user is no longer stuck developing derivations linearly, from left-to-right, but has far more freedom to explore and to transform in any order desired:

- One can transform either left to right or right to left.
- One can transform in the presence of unknown variables.
- An exploratory style is easily accommodated: if it is seen that the application of a law is not working (i.e., its premises cannot be derived), it is trivial to know exactly where to "back up" to: we just delete the application (and all its sub-derivations).

For example, another approach one could take to the derivation in Section 6.2 would have been to start with the derivation

```
∀C,F,G.
[P1: {C ⊥ = ⊥};
 P2: ∀x. {C (F x) = G (C x)}]
⇒
 C(μF)
={?}
 μG
```

apply *FPI*, fill in the unknowns with "z↦z" and G, and then work from both sides in. The original approach was taken to illustrate the typical case in which the goal is unknown.

One can load a program into PATH[5] and start transforming it: if the transformation is stuck at some point, one could just add a conjecture there and continue. One could later come back to the conjecture and prove it; or maybe one may want to add a premise to the derivation which corresponds to this conjecture. The point is that one can derive programs in any order one wants.

6.6 Conclusion

I hope this chapter has given some evidence that the PATH logic does indeed lend itself to the graphical display and editing of program derivations. There are numerous directions in which the work described here could be improved upon:

- Programs that Transform Derivations
  - A more sophisticated proof search would be desirable: PATH should replace {?} with valid proofs when possible or indicate that no proof is possible when that can be determined.
- Meta-programs
  - User-defined meta-programs could be added to PATH.
  - An extension of the PATH logic which would make derivations more robust is to allow not just for the application of rules in derivations but the application of

[5]Note: a program is just a special case of a derivation, one in which there are no quantifiers, no premises, and no rule applications.

meta-programs (i.e., instead of the meta-program being executed statically to create a derivation, it would be executed dynamically—whenever the program changes to which it is applied).

- User Interface. PATH could be extended to take more advantage of a visual interface to derivations:
  - Selectively display and hide parts of the program or derivation.
  - Control the layout of the derivation.
  - Navigate the program in a manner similar to an outline editor.
  - Give visual feedback when and where laws would be applicable.

Chapter 7

Applications of PATH

This chapter gives some examples of the derivations possible in the PATH system.

7.1 Filter-Iterate

An example from Sands [66], used to demonstrate expression procedures, is the *Filter-Iterate* law:

```
   p ↦ f ↦ x ↦ filter p (iterate f x)
=
   μg ↦ p ↦ f ↦ x ↦  if p x
                     then Cons⟨x, g p f (f x)⟩
                     else g p f (f x)
```

It can be derived simply using one application of *FPF*, see Figure 7.1. This is the most common method of transformation (and of using *FPF*): we have a recursive function (`iterate` here) and some context in which to specialize it; by specializing it, we hope to create a single recursive definition which performs fewer steps. The strictness condition is handled primarily by a series of reductions. The main premise of *FPF* is accomplished also by

```
  p ↦ f ↦ x ↦ filter p (iterate f x)
= {FPF
    p ↦ f ↦ x ↦ filter p (⊥ f x) =
  =                                                                  {red*}
    p ↦ f ↦ x ↦ ⊥
  =                                        {R Func-Bot,R Func-Bot,R Func-Bot}
    ⊥
  ;
  ∀iterate.
    p ↦ f ↦ x ↦ filter p ((f ↦ x ↦ Cons⟨x, iterate f (f x)⟩) f x)
  =                                                                   {red}
    p ↦ f ↦ x ↦ filter p (Cons⟨x, iterate f (f x)⟩)
  =                                                                  {red*}
    p ↦ f ↦ x ↦ if p x
                then Cons⟨x,filter p (iterate f (f x))⟩
                else filter p (iterate f (f x))
  =                                                                {R red}
    p ↦ f ↦ x ↦ if p x
                then Cons⟨x,(p ↦ f ↦ x ↦ filter p (iterate f x)) p f (f x)⟩
                else (p ↦ f ↦ x ↦ filter p (iterate f x)) p f (f x)
  }
  μg ↦ p ↦ f ↦ x ↦  if p x
                    then Cons⟨x, g p f (f x)⟩
                    else g p f (f x)
```

Figure 7.1: Derivation of *Filter-Iterate*

a series of reductions until we see two instances of the original "`filter p (iterate f x)`". So, we need to abstract, `{R red}`, in order to get the exact context we need. The end result of this transformation is that we have performed deforestation: the intermediate list which was generated by iterate has disappeared.

There is a certain convention used whenever possible in PATH: All rules and laws are written so that the right hand side is more efficient than the left hand side. This makes it easier to remember which direction to apply a law. Also, as a result of this, proofs are most easily achieved left to right (or top down): if the law is making a program more efficient, then the derivation will most likely have more reduction steps than reverse reduction steps; the former are easier to do than the latter as reverse reduction steps require user input (many terms may reduce to a particular term).

7.2 Map-Iterate

Here is a law, *Map-Iterate,* that has been used as an example of the usefulness of co-inductive proofs [63, 26]:

```
∀f,x.   map f (iterate f x) = iterate f (f x)
```

Here it can be derived simply with just two applications of *FPF*, the second in the reverse direction: see Figure 7.2. The whole derivation is bracketed by an abstraction over `x` and an application to `x`. This is required to create a context "`x↦map f ([] f x)`" which would be sufficiently general enough (we do not want to specialize where `x` is constant, we want a function where it is a parameter) In both applications of *FPF*, the whole derivation proceeded by reductions except for one step in which we need to give PATH some assistance by entering a Reverse reduction step. It is often difficult to work through a reverse application of *FPF*, but in performing this derivation, we started with the incomplete derivation

```
∀f,x.
  map f (iterate f x)
=                                                              {R red}
  (x ↦ map f (iterate f x))  x
= {FPF
    x ↦ map f (⊥ f x) ={red*} ⊥
  ;
  ∀iterate.
    x ↦ map f ((f ↦ x ↦ Cons⟨x, iterate f (f x)⟩) f x)
  =                                                              {red}
    x ↦ map f (Cons⟨x, iterate f (f x)⟩)
  =                                                              {red}
    x ↦ Cons⟨f x, map f (iterate f (f x))⟩
  =                                                              {R red}
    x ↦ Cons⟨f x, (x ↦ map f (iterate f x)) (f x)⟩
  }
  (μ g ↦ x ↦ Cons⟨f x, g (f x)⟩) x
= {R FPF
    x ↦⊥ f (f x) ={red*} ⊥
  ;
  ∀iterate.
    x ↦ (f ↦ x ↦ Cons⟨x, iterate f (f x)⟩ ) f (f x)
  =                                                              {red}
    x ↦ Cons⟨f x, iterate f (f (f x))⟩
  =                                                              {R red}
    x ↦ Cons⟨f x, (x ↦ iterate f (f x)) (f x)⟩
  }
  (x ↦ iterate f (f x)) x
=                                                                {red}
  iterate f (f x)
```

Figure 7.2: Derivation of *Map-Iterate*

```
∀f,x.
  map f (iterate f x)
=                                  {?}
  iterate f (f x)
```

and then applied *FPF* directly to the top program and the bottom program and attempted (successfully) to derive a common program in the middle.

7.3 Tupling

Here is the standard, two pass, program to compute the average of a list (`sum` and `length` defined in the prelude):

```
ys↦divide⟨sum ys, length ys⟩
```

Let's derive a program which computes this result in one traversal of the list. The first thing to do is expose the function which returns the sum and length of the list. This is done with the following two transformation steps:

```
  ys ↦ divide⟨sum ys, length ys⟩
=                                                                  {R red}
  ys ↦ let ⟨s,l⟩ = ⟨sum ys, length ys⟩ in divide⟨s,l⟩
=                                                                  {R red}
  ys ↦ let ⟨s,l⟩ = (ys ↦ ⟨sum ys, length ys⟩) ys in divide⟨s,l⟩
```

Now the function which we want to transform into a one-pass recursive function has been exposed:

```
ys↦⟨sum ys, length ys⟩
```

FPF is not directly applicable to this program because neither `sum` nor `length` is in a strict context[1]. However, the context in which *both* of these recursive functions occurs is strict:

[1]Thus, it appears that expression procedures cannot do this derivation because it requires a strict context and can only specialize one function at a time.

```
  ys ↦ ⟨⊥ ys, ⊥ ys⟩
=                                  {red,red}
  ys ↦ ⟨⊥, ⊥⟩
=                                {R Prod-Bot}
  ys ↦ ⊥
=                                {R Func-Bot}
  ⊥
```

So, if we could merge these two recursive calls into one function then we could apply *FPF*. This strategy is encapsulated in the *FPF-N* law (refer to its derivation in Appendix C):

```
∀C,F,G.
  C ⊥ = ⊥
; ∀x . C⟨i F.i x.i⟩ = G(C x)
⇒
  C⟨i μ(F.i)⟩ = μG
```

Or to instantiate this general form to the case of a 2-tuple:

```
∀C,F₁,F₂,G.
  C ⊥ = ⊥
; ∀x₁,x₂. C⟨F₁ x₁, F₂ x₂⟩ = G(C⟨x₁,x₂⟩)
⇒
  C⟨μ F₁, μ F₂⟩ = μG
```

So now we can use this law, *FPF2*, to achieve tupling: see Figure 7.3. The key step is the application of the *Abides* law which converts a tuple of `case`'s into a single `case`. This strategy of combining results together to eliminate multiple traversals is called (unsurprisingly) "tupling" [37].

7.4 Mix

The law *Mix* (here the `letrec` version)

```
∀F,M.
  letrec f   = F⟨– f⟩ in ⟨– f⟩
=
  letrec f.i = F⟨j f.(M.i.j)⟩ in f
```

```
  ys ↦ ⟨sum ys, length ys⟩
=
  {FPF-2
    ys ↦ ⟨⊥ ys, ⊥ ys⟩ ={...} ⊥
  ;
    ∀sum,length.
      ys ↦ ⟨case ⟨Nil: 0, Cons⟨x,xs⟩: x + sum xs⟩    ys
           ,case ⟨Nil: 0, Cons⟨x,xs⟩: 1 + length xs⟩ ys⟩
    =                                                          {R Abides}
      ys ↦ case ⟨Nil:        ⟨0,0⟩
                ,Cons⟨x,xs⟩: ⟨x + sum xs, 1 + length xs⟩
                ⟩ ys
    =                                                          {R eta}
      case ⟨Nil:        ⟨0,0⟩
           ,Cons⟨x,xs⟩: ⟨x + sum xs, 1 + length xs⟩
           ⟩
    =                                                          {R red}
      case ⟨Nil:        ⟨0,0⟩
           ,Cons⟨x,xs⟩: let ⟨s,l⟩ = ⟨sum xs, length xs⟩
                        in  ⟨x + s, 1 + l⟩
           ⟩
    =                                                          {R red}
      case ⟨Nil:        ⟨0,0⟩
           ,Cons⟨x,xs⟩: let ⟨s,l⟩ = (ys ↦ ⟨sum ys,length ys⟩) xs
                        in  ⟨x + s, 1 + l⟩
           ⟩
  }
  μsumlen ↦ case ⟨Nil:        ⟨0,0⟩
                 ,Cons⟨x,xs⟩: let ⟨s,l⟩ = sumlen xs in ⟨x+s,1+l⟩
                 ⟩
```

Figure 7.3: Tupling Derivation

can be a bit difficult to understand; viewed from the right to left perspective, this law allows for the coalescing of multiple recursive definitions which are identical up to names of functions . As an example, refer to the following derivation which uses *Mix-Letrec*, a law derived from *Mix*:

```
  letrec f = F⟨f,f,f⟩ in f
=                                                                  {Mix-Letrec}
  letrec ⟨f1,f2⟩.i = F⟨j ⟨f1,f2⟩.(M.i.j)⟩ in ⟨f1,f2⟩.j
=                                                                      {letrec}
  letrec f1 = F⟨j ⟨f1,f2⟩.(M.1.j)⟩
       ; f2 = F⟨j ⟨f1,f2⟩.(M.2.j)⟩
  in ⟨f1,f2⟩.j
=                                                                    {eta,eta}
  letrec f1 = F⟨⟨f1,f2⟩.(M.1.1),⟨f1,f2⟩.(M.1.2),⟨f1,f2⟩.(M.1.3)⟩
       ; f2 = F⟨⟨f1,f2⟩.(M.2.1),⟨f1,f2⟩.(M.2.2),⟨f1,f2⟩.(M.2.3)⟩
  in ⟨f1,f2⟩.j
```

We start with a recursive definition of `f` in the first line, in the last line we have two mutually recursive functions. `M` could be any two dimensional matrix containing the projections $\{1_2, 2_2\}$ and `j` can be any projection. So, by choosing the appropriate value of `M`, we can choose between f_1 and f_2 in each of the arguments to `F`. So, for example, we could instantiate `M` and `j` in order to derive the following law:

```
  letrec f = F⟨f,f,f⟩ in f
=
  letrec f1 = F⟨f1,f2,f2⟩
       ; f2 = F⟨f2,f2,f1⟩
  in f1
```

The law *Mix* is as expressive as Ariola and Blom's "copy" rule [1]. They need additional meta-notation and meta-concepts to express the rule, but here we have a simple rule, written without meta-notation. However, I do require n-tuples used in full generality. The non-letrec version of *Mix* is

$$\forall F, M.\quad \langle - \ \mu f \mapsto F\langle - \ f\rangle\rangle \ = \ \mu f \mapsto \langle^{i}\ F\langle^{j}\ f.(M.i.j)\rangle\rangle$$

and its derivation is one of the shortest in this dissertation:

```
∀F,M.
  ⟨i μf ↦ F⟨j f⟩⟩
= {FPF
    ⟨i ⊥⟩
   =                                {R Prod-Bot}
    ⊥
  ;
    ∀f.
      ⟨i F⟨j f⟩⟩
    =                                      {R red}
      ⟨i F⟨j ⟨i f⟩.(M.i.j)⟩⟩
  }
  μf ↦ ⟨i F⟨j f.(M.i.j)⟩⟩
```

7.5 Assertions

In his thesis [69] Scherlis noted that expression procedures allow us to specialize recursive functions in a syntactic context, but do not allow us to specialize functions based on non-syntactic information. For instance, expression procedures can specialize `f` in the syntactic context "`f x 0`" but cannot take advantage of "`x > y`" in the specialization of "`f x y`". To take advantage of the non-syntactic information available, Scherlis extended his system to support "qualified expression procedures." A qualified expression procedure looks like this

```
{p} e1 =ep= e2
```

in which `p` is a boolean valued expression similar to a pre-condition. In the transformation of the definition, we can assume `p` is true; the qualified expression procedure may only be applied where the qualifier is true.

Due to the non-strict semantics of PATH-L and the schematic approach to expression procedures, we can achieve the power of qualified expression procedures without adding ad

Introducing/Eliminating Assertions

```
e = assert True e

if p then {a = assert p a} else b

if p then a else {b = assert (not p) b}
```

Manipulating Assertions

```
assert(if p then p2 else p3)
  {if p then a else b = if p then assert p2 a else assert p3 b}

C ⊥ = ⊥  ⇒  assert p (C e) = C(assert p e)

assert p (assert q e) = assert q (assert p e)
```

Using Assertions

```
assert (e1==e2) C{e1 = e2}

assert p (C{ if p then a else b = a })

assert p (C{ if not p then a else b = b })
```

Figure 7.4: Laws Regarding `assert`

hoc constructs to the language or adding additional primitive laws. We define an `assert` function as follows:

```
assert p e = if p then e else ⊥
```

With `assert` and some simple laws we can derive about it, we get the power of qualified expression procedures. We can change `e` to "`assert p e`" where we know p is true and where we have "`assert p e`", we can transform `e` with the knowledge that `p` is true. Since the context "`assert p []`" is strict, we can specialize function calls in this context. Some easily proved laws regarding assertions are in Figure 7.4.

Let the function `check` be defined thus:

```
check p x = assert (p x) x
```

The law *Precondition* allows us to prove and use invariant properties of recursive functions:

```
∀F,F′,P.
  P ⊥ = ⊥
; ∀f.  {F f = F′ f} ∘ check P
; ∀f.  F {f = f ∘ check P} ∘ check P
⇒
  {μF = μF′} ∘ check P
```

The invariant is the (strict) predicate `P`. What this law is saying is this: if the invariant allows us to transform "`F f`" into "`F′ f`", and if the invariant is preserved across all recursive calls, then we can transform `μF` to `μF′` in contexts where the invariant is satisfied.

7.6 Conclusion

A number of the laws proved in PATH are listed in Figures 7.5 and 7.6. Derivations of these and other laws can be found in Appendix C. One derivation particularly worth noting is *FPF-Partial*. In its derivation a bit of insight is required to determine the context in which one ought to specialize.

As all the derivations presented in this chapter and in the Appendix fit on one page or less, does this imply that PATH will not scale? No. There is no reason to expect that PATH cannot do derivations of arbitrary length. However, one of the goals of PATH is to make derivations short: thus, we have laws that are very generic and laws which we can prove once and reuse multiple times. Just as functional programming languages scale to large programs even though most functions are small, I expect PATH to scale to large programs with most derivations being small (contingent of course on the user writing modular derivations).

Cata-Merge:

```
∀F,B. xs ↦ ⟨i cata ⟨F.i,B.i⟩ xs⟩ = cata ⟨⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩, B⟩
```

FPE:

```
∀F. μf ↦ F⟨f,f⟩ = μf ↦ F⟨f,F⟨f,f⟩⟩
```

FPF-Ext:

```
∀C,F,G,H.
  C ⊥ = ⊥
; ∀f.  letrec g=G⟨f,g,C f⟩ in { C(F⟨f,g⟩) = H⟨f,g,C f⟩ }
⇒
  letrec f=F⟨f,g⟩; g=G⟨f,g,C f⟩ in ⟨f,g⟩
= letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in ⟨f,g⟩
```

FPF-N:

```
∀C,F,G.
  C ⊥ = ⊥
; ∀x . C⟨i F.i x.i⟩ = G(C x)
⇒
  C⟨i μ(F.i)⟩ = μG
```

FPF-Partial:

```
∀C,F,G.
  C ⊥ = ⊥
; ∀f.  C(F⟨f,C f⟩)  =  G⟨f,C f⟩
⇒
  C(letrec f=F⟨f,C f⟩ in f)  =  letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in g
```

Inline-Self:

```
∀C,F,G. letrec f=F⟨f,{ f=F⟨f,f,g⟩ },g⟩; g=G⟨f,g⟩ in C⟨f,g⟩
```

Lambda-Mu-Switch:

```
∀F.  μf ↦ x ↦ F⟨f x,x⟩  =  x ↦ μf ↦ F⟨f,x⟩
```

Figure 7.5: Laws from the PATH Catalog

Letrec-Equiv:

```
∀F,G1,G2.
  letrec f=F⟨f,g1,g2⟩;⟨g1,g2⟩=⟨G1⟨f,g1,g2⟩,G2⟨f,g1,g2⟩⟩ in ⟨f,g1,g2⟩
=
  letrec f=F⟨f,g1,g2⟩; g1=G1⟨f,g1,g2⟩; g2=G2⟨f,g1,g2⟩ in ⟨f,g1,g2⟩
```

Letrec-Exp:

```
∀F,G,C.
 letrec f=F⟨f,g⟩; g=G⟨f                    ,f,g⟩ in ⟨f,g⟩
=
 letrec f=F⟨f,g⟩; g=G⟨letrec f=F⟨f,g⟩ in f,f,g⟩ in ⟨f,g⟩
```

Partial-Mu-Reduce:

```
∀F,G.  μ(F ∘ G) = F(μ(G ∘ F))
```

Split:

```
∀F.  μx ↦ ⟨i F.i x.i⟩  =  ⟨i μ F.i⟩
```

Trivial-Fusion:

```
∀F,H,I.
  H ⊥ = ⊥
; ∀x. I(H x) = x
⇒
  H(μF) = μg ↦ H(F(I g))
```

Tuple-Strict-Implies-Components-Strict:

```
∀F. { (x ↦ ⟨i F.i x⟩) ⊥ = ⊥ }  ⇒  ⟨i {F.i ⊥ = ⊥}⟩
```

Unused-Parameter-Elimination:

```
∀A,B,C,D,F.
  (μf ↦ ⟨x,y⟩ ↦  F⟨⟨i  f ⟨C.i x, D.i⟨f,x,y⟩⟩,x⟩) ⟨A,B⟩
=
  (μf ↦ x     ↦  F⟨⟨i  f (C.i x)⟩          ,x⟩) A
```

Figure 7.6: Laws from the PATH Catalog, continued

Chapter 8

Genericity with N-Tuples

This chapter describes an extension of PATH-L that adds n-tuples, a construct that allows for programs generic over the length of tuples. Section 8.1 explains why the genericity provided by n-tuples is needed. Section 8.2 describes the syntax and semantics of n-tuples. Section 8.3 returns to the examples in Section 8.1 and shows what programs, laws, and program derivations look like using n-tuples. Section 8.4 describes a higher order typed lambda calculus which gives us n-tuples. Section 8.5 then concludes.

In a number of the examples in this chapter, Haskell syntax will be used, not PATH-L syntax. Because these examples are directly from the Haskell prelude and libraries, I have thought it best to write them as Haskell, not convert to PATH-L syntax. I hope that with this forewarning, the reader will not be confused by the use of both Haskell and PATH-L in the following examples.

8.1 The Need for N-Tuples

An n-tuple is a tuple whose length is unknown. This section argues for the usefulness of n-tuples: similar to the genericity provided by polymorphism and polytypism (cf. Section 1.3.4), n-tuples result in more general programs (Section 8.1.1), more general laws about those programs (Section 8.1.2), and more general program derivations (Section 8.1.3).

8.1.1 More General Programs

The following family of `zip` functions are defined in the Haskell Prelude and Libraries[1]:

```
zip  :: ([a],[b]) → [(a,b)]
zip3 :: ([a],[b],[c]) → [(a,b,c)]
...
zip7 :: ([a],[b],[c],[d],[e],[f],[g]) → [(a,b,c,d,e,f,g)]
```

There is also the `zipWith` family of functions

```
zipWith  :: ((a,b)→c) → ([a],[b]) → [c]
zipWith3 :: ((a,b,c)→d) → ([a],[b],[c]) → [d]
...
zipWith7 :: ((a,b,c,d,e,f,g)→h) → ([a],[b],[c],[d],[e],[f],[g]) → [h]
```

and the `unzip` family of functions:

```
unzip  :: [(a,b)] → ([a],[b])
unzip3 :: [(a,b,c)] → ([a],[b],[c])
...
unzip7 :: [(a,b,c,d,e,f,g)] → ([a],[b],[c],[d],[e],[f],[g])
```

Although writing the `zip`, `zipWith`, and `unzip` families of functions is not difficult, it is tedious and error-prone. Clearly, it is preferable to abstract over these families and write one generic `zip`, one generic `zipWith`, and one generic `unzip`.

[1] Actually, it is their curried counterparts which are defined, but the uncurried versions of `zip` and `zipWith` are used here for illustrative purposes.

8.1.2 More General Laws

Note the free theorem [82] (or parametricity theorem) for `zip`:

```
let cross ⟨f,g⟩ ⟨x,y⟩ = ⟨f x,g y⟩ in {
  map(cross⟨f,g⟩) ∘ zip = zip ∘ cross⟨map f,map g⟩
}
```

The free theorem for `zip3` is

```
let cross3 ⟨f,g,h⟩ ⟨x,y,z⟩ = ⟨f x,g y,h z⟩ in {
  map(cross3⟨f,g,h⟩) ∘ zip3 = zip3 ∘ cross3⟨map f,map g,map h⟩
}
```

As before, to generate these laws is not difficult but tedious and error-prone. To formulate this *family* of laws yet another family of functions is needed:

```
cross, cross3, cross4, ...
```

Another family of laws are the laws for expansion of tuples:

```
∀x:×⟨a1,a2⟩    .  x = ⟨x.12,x.22⟩
∀x:×⟨a1,a2,a3⟩ .  x = ⟨x.13,x.23,x.33⟩
...
```

We would like to generalize over these *families* of laws. Having fewer and more generic laws is good in a program transformation system: there are fewer laws to learn, fewer laws to search, and program derivations are more robust (i.e., they are more likely to remain valid when applied to a modified input program).

8.1.3 More General Derivations

Program derivations of the following form (where the two "..." derivations are nearly identical) are common:

```
e ={eta} ⟨ e.1₂ ={r} ... ={r} e1
         , e.2₂ ={r} ... ={r} e2
         ⟩
```

This derivation gives the law

```
e = ⟨e1,e2⟩
```

When doing proofs or derivations informally we can do the derivation for the first case and then say "similarly `e.2₂ = e2`." However, in PATH this "similarly" step must be done without hand waving; we would also like to do this step without duplicating the derivation. How can this duplication be avoided? Note that in general the form of `⟨e1,e2⟩` is `⟨C[1₂],C[2₂]⟩`[2]. So, we would like to merge the two similar derivations

```
e.1₂ ={r} ... ={r} C[1₂]
e.2₂ ={r} ... ={r} C[2₂]
```

into a single derivation:

```
e.i ={r} ... ={r} C[i]
```

However, this cannot be done because the `i` in `e.i` must be a projection constant and cannot be a variable or expression (current type inference methods cannot handle such an extension). There are two ways of allowing for `i` to be a variable (in a typed language):

- The variable i could be a meta-variable, not a variable in the language. We move the problem to a meta-language or meta-logic. Then we could express the above law in the meta-language like this: $\forall n.\forall i < n$. e.i_n = C[i_n]. The disadvantage here is that a meta-language is needed to express program laws.

[2]Where `C[e]` represents a program context `C` with its holes filled by expression `e`.

- Increase the expressiveness of the type system to allow `i` to be a variable. Unfortunately, the only type systems which allow this expressiveness are dependent type systems, which do not have type inference.

The latter approach is what is taken here (without using dependent types).

8.2 N-tuples

To get n-tuples, a number of changes must be made to the PATH-L language as described in Chapter 2. In the syntax of terms, where before we had

$$| \; e.m_n$$

we now have

$$| \; e_1.e_2$$
$$| \; m_n$$
$$| \; \langle^{v} e \rangle$$

Projections become first class elements of the language. The last construct is an n-tuple, it abstracts over a projection variable *v* which can be used inside the body of the n-tuple. An n-tuple works much like a function parameterized over a projection (where "." is application) but the typing is different. We also have n-sums. Where before we had the various constructors for *n*-sums:

$$| \; \texttt{In}.m_n$$

we now have a tuple of all the constructors for *n*-sums

$$| \; \texttt{In}_n$$

To which we can apply a projection to get the *m*-th constructor:

$\texttt{In}_n\,.\,m_n$

To give some intuition for the semantics of n-tuples, note these equivalences:

$\langle^{\texttt{i}}\ \texttt{f x.i}\rangle = \langle\texttt{f x.1, f x.2, ..., f x.n}\rangle$
$\langle^{\texttt{i}}\ \langle\texttt{f,g}\rangle\texttt{.i}\ \langle\texttt{x,y}\rangle\texttt{.i}\rangle = \langle\texttt{f x, g y}\rangle$

In the latter, the tuples $\langle\texttt{f,g}\rangle$ and $\langle\texttt{x,y}\rangle$ are "zipped" together. The types, t, must become significantly more complex: where before we had

$\mid\ \times\langle t_1, t_2, ...\rangle$
$\mid\ +\langle t_1, t_2, ...\rangle$

we now have

$\mid\ \times t$
$\mid\ +t$
$\mid\ \langle t_1, t_2, ...\rangle$
$\mid\ t_1.t_2$
$\mid\ m_n$
$\mid\ \langle^{v:d}\, e\rangle$
$\mid\ v$

and a new syntactic category d (in the n-tuple construct) to represent "dimension variables":

d	::=	a	variables of dimension kind
	$\mid$	$n\texttt{d}$	dimension ($1 \leq n$)

The result is that we have tuples *and* n-tuples at the type level. We now have types such as $\times\alpha$; in this type, the type variable α must have a different kind, a kind that indicates that α is not a type but a tuple of types. The kind system will be discussed in Section 8.4.

8.3 Examples of N-Tuples

This section provides examples of the usefulness of n-tuples.

Syntactic Conventions.

Some syntactic shortcuts are used in the following: As previously, the "$:t$" is dropped in functions and m is put for the projection m_n. The following conventions are used for variables:i, j, k for projection variables (at both the term and type level); t, a, b, c for regular type variables; and I, J, K for type variables of dimension kind.

8.3.1 More General Programs

An uncurried `zip3` is as follows in Haskell:

```
zip3 :: ([a],[b],[c]) → [(a,b,c)]
zip3 (a:as,b:bs,c:cs) = (a,b,c) : zip3 (as,bs,cs)
zip3 _                = []
```

If Haskell had n-tuples[3], one could write a generic `zip` as follows:

```
zip :: ×⟨i:I [a.i]⟩ → [×a]
zip ⟨i x.i : xs.i⟩ = x : zip xs
zip _              = []
```

Note here that an n-tuple is part of a pattern. Note also that the type variable `a` in the above represents a tuple of types. A generic `zipWith` would be similar to the above. Haskell's `unzip3`

```
unzip3 :: [(a,b,c)] → ([a],[b],[c])
unzip3 ((a,b,c):xs) = let (as,bs,cs) = unzip xs in (a:as,b:bs,c:cs)
unzip3 []           = ([],[],[])
```

could be written generically as follows:

```
unzip :: [×a] → ×⟨i:I [a.i]⟩
unzip (x:xs) = ⟨i x.i : (unzip xs).i ⟩
unzip []     = ⟨– []⟩
```

[3]Or if PATH-L had patterns, which it doesn't have at the present.

8.3.2 More General Laws

The free theorem for `zip3` is

```
let cross3 ⟨f,g,h⟩ ⟨x,y,z⟩ = ⟨f x,g y,h z⟩ in {
  map(cross3⟨f,g,h⟩) ∘ zip3 = zip3 ∘ cross3⟨map f,map g,map h⟩
}
```

but it can be generalized to the free theorem for the generic `zip`:

```
let cross f x = ⟨ⁱ f.i x.i⟩ in {
  map(cross f) ∘ zip = zip ∘ cross⟨ⁱ map f.i⟩
}
```

And the laws for the expansion of tuples

```
∀x:×⟨a₁,a₂⟩     .  x = ⟨x.1₂,x.2₂⟩
∀x:×⟨a₁,a₂,a₃⟩  .  x = ⟨x.1₃,x.2₃,x.3₃⟩
...
```

can be generalized to the *N-Tuple-Eta* law:

```
∀x:×a   .  x = ⟨ⁱ x.i⟩
```

Note also the primitive law *Inst*, which is generic over sums of any length:

```
∀F,H,X.
  H ⊥ = ⊥
 ⇒
  H(case ⟨ⁱ y ↦ F.i y⟩ X) = case ⟨ⁱ y ↦ H(F.i y)⟩ X
```

Using n-tuples this law can be written as a law without meta-notation. The *eta* rule for sums,

```
∀x:+⟨t₁,t₂,...,tₙ⟩  . x ={eta} case ⟨In.1ₙ,In.2ₙ,...,In.nₙ⟩ x
```

can also now be written without meta-notation and with more concision:

```
∀x:+t . x ={eta} case In x
```

8.3.3 More General Derivations

This section shows how n-tuples can make derivations simpler and more generic. Previously, we had the following derivation, where the two "..." derivations are nearly identical:

```
e ={eta} ⟨e.1₂ ={r} ... ={r} C[1₂]
         ,e.2₂ ={r} ... ={r} C[2₂]
         ⟩
```

Using n-tuples we can merge these two sub-derivations into a single derivation:

```
e ={N-Tuple-Eta} ⟨ⁱ e.i ={r} ... ={r} C[i]⟩
```

So, we have a derivation that is both shorter and more generic, and there is no need for meta-notation.

Here is a law, *Abides*-2, which combines two `case` expressions in a 2-tuple, each `case` takes a 2-sum:

```
  case ⟨x ↦ ⟨F₁ x,G₁ x⟩, x ↦ ⟨F₂ x,G₂ x⟩⟩ e
=
  ⟨case ⟨F₁,F₂⟩ e, case ⟨G₁,G₂⟩ e⟩
```

Its derivation is in Figure 8.1. Now, here is a generic version of *Abides*-2, it combines *n* `case` expressions in an *n*-tuple, each `case` takes an *m*-sum:

```
case ⟨ⁱ y ↦ ⟨ʲ F.i.j y⟩⟩ x = ⟨ʲ case⟨ⁱ F.i.j⟩ x⟩
```

Its derivation, in Figure 8.2, corresponds directly to the non-generic derivation. The generic derivation is shorter: *Inst* is only applied once (not twice), *red* once (not four times), and "*R eta*" once (not four times). If we did the derivation in the reverse direction (from bottom to top), we would see another advantage to the generic approach: in the non-generic derivation, to do the `{red, 4 times}` step requires the input of four expressions[4] and all

[4]To do reverse reduction requires user input.

```
  case ⟨x ↦ ⟨F₁ x,G₁ x⟩, x ↦ ⟨F₂ x,G₂ x⟩⟩ e
=                                                                  {eta}
  ⟨(case ⟨x ↦ ⟨F₁ x,G₁ x⟩, x ↦ ⟨F₂ x,G₂ x⟩⟩ e).1
  ,(case ⟨x ↦ ⟨F₁ x,G₁ x⟩, x ↦ ⟨F₂ x,G₂ x⟩⟩ e).2
  ⟩
=                                                  {Inst, 2 times}
  ⟨(case ⟨x ↦ ⟨F₁ x,G₁ x⟩.1, x ↦ ⟨F₂ x,G₂ x⟩.1⟩ e)
  ,(case ⟨x ↦ ⟨F₁ x,G₁ x⟩.2, x ↦ ⟨F₂ x,G₂ x⟩.2⟩ e)
  ⟩
=                                                   {red, 4 times}
  ⟨(case ⟨x ↦ F₁ x, x ↦ F₂ x⟩ e)
  ,(case ⟨x ↦ G₁ x, x ↦ G₂ x⟩ e)
  ⟩
=                                                {R eta, 4 times}
  ⟨case ⟨F₁,F₂⟩ e, case ⟨G₁,G₂⟩ e⟩
```

Figure 8.1: Derivation of *Abides-2*

```
  case ⟨ⁱ y ↦ ⟨ʲ F.i.j y⟩⟩ x
=                                                {N-Tuple-Eta}
  ⟨ʲ (case ⟨ⁱ y ↦ ⟨ʲ F.i.j y⟩⟩ x).j⟩
=                                                       {Inst}
  ⟨ʲ  case ⟨ⁱ y ↦ ⟨ʲ F.i.j y⟩.j⟩ x⟩
=                                                        {red}
  ⟨ʲ  case ⟨ⁱ y ↦ F.i.j y⟩ x⟩
=                                              {R N-Tuple-Eta}
  ⟨ʲ  case ⟨ⁱ F.i.j⟩ x⟩
```

Figure 8.2: Derivation of *Abides*

four of these expressions must be entered properly in order to allow for `{Inst, 2 times}` in the next step and `{eta}` in the following step; in the generic derivation, only one reverse reduction step needs to be done.

8.3.4 Nested N-Tuples

Informal notations for representing vectors (or n-tuples) are generally ambiguous: e.g., one writes $f\overline{x}$ for the vector $\langle f\,x_1,...,f\,x_n\rangle$ but $g\,(f\,\overline{x})$ could signify either $\langle g(f\,x_1),...,g(f\,x_n)\rangle$ or $g\,\langle f\,x_1,...,f\,x_n\rangle$. These notations do not extend to nested vectors. With n-tuples one can easily manipulate nested n-tuples (i.e., matrices). For example, the application of a function to every element of a three-dimensional matrix is coded as follows (note that $\langle -\mathrm{x}\rangle$ is a tuple of identical elements):

```
map3Dmatrix :: (a→b)→×⟨–:I ×⟨–:J ×⟨–:K a⟩⟩⟩→×⟨–:I ×⟨–:J ×⟨–:K b⟩⟩⟩
map3Dmatrix = f↦m↦⟨i ⟨j ⟨k f m.i.j.k⟩⟩⟩
```

In the definition of `map3Dmatrix`, the expression $\langle^{\mathtt{i}}\,\langle^{\mathtt{j}}\,\langle^{\mathtt{k}}$ `f m.i.j.k`$\rangle\rangle\rangle$ is a 3-dimensional matrix where "`f m.i.j.k`" is the value of the `.i.j.k`-th element, which here is `f` applied to the corresponding value of the original matrix `m.i.j.k`. Matrix transposition is straightforward:

```
transpose :: ×⟨i:I ×⟨j:J a.i.j⟩⟩ → ×⟨j:J ×⟨i:I a.i.j⟩⟩
transpose x = ⟨j ⟨i x.i.j⟩⟩
```

The transpose is done by switching the subscripts of `x`. Note that the type variable `a` above is a *matrix* of types. An application of `transpose` would be reduced as follows:

```
        (transpose⟨⟨x₁,x₂⟩,⟨y₁,y₂⟩,⟨z₁,z₂⟩⟩) .2.3
={red} ⟨j ⟨i ⟨⟨x₁,x₂⟩,⟨y₁,y₂⟩,⟨z₁,z₂⟩⟩.i.j⟩⟩.2.3
={red}    ⟨i ⟨⟨x₁,x₂⟩,⟨y₁,y₂⟩,⟨z₁,z₂⟩⟩.i.2⟩   .3
={red}       ⟨⟨x₁,x₂⟩,⟨y₁,y₂⟩,⟨z₁,z₂⟩⟩.3.2
={red}                          ⟨z₁,z₂⟩   .2
={red}                              z₂
```

Note the various ways one can transform a two dimensional matrix:

$\langle^{i}\ \langle^{j}\ \mathrm{m.i.j}\rangle\rangle$	m
$\langle^{j}\ \langle^{i}\ \mathrm{m.i.j}\rangle\rangle$	the transpose of m
$\langle^{i}\ \langle^{j}\ \mathrm{f\ m.i.j}\rangle\rangle$	f applied to each element of m
$\langle^{i}\ \mathrm{f}\langle^{j}\ \mathrm{m.i.j}\rangle\rangle$	f applied to each "row" of m
$\langle^{j}\ \mathrm{f}\langle^{i}\ \mathrm{m.i.j}\rangle\rangle$	f applied to each "column" of m

Clearly this notation extends to matrices of higher dimensions. Some laws about the transpose function are as follows:

$$\mathtt{(transpose\ m).j.i} \;=\; \mathrm{m.i.j}$$

$$\mathtt{transpose(transpose\ m)} \;=\; \mathtt{m}$$

Here is a proof of the latter:

```
    transpose(transpose m)
=                                          {red}
    transpose⟨j ⟨i m.i.j⟩⟩
=                                          {red}
    ⟨l ⟨k ⟨j ⟨i m.i.j⟩⟩.k.l⟩⟩
=                                          {red}
    ⟨l ⟨k    ⟨i m.i.k⟩   .l⟩⟩
=                                          {red}
    ⟨l ⟨k       m.l.k      ⟩⟩
=                                {R N-Tuple-Eta}
    ⟨l          m.l         ⟩
=                                {R N-Tuple-Eta}
                m
```

8.3.5 Generic Catamorphisms

It is obvious that Haskell's zip family of functions could benefit from n-tuples; but interestingly, catamorphisms [47, 46, 49] can also benefit from n-tuples, giving catamorphisms

over mutually recursive data structures. Let's assume we have a fix point operator for types, also μ. A recursive type, μF, is the fixed point of a type functor F. The kind of μ is $(\star \to \star) \to \star$ (i.e., it takes a functor of kind $\star \to \star$ and returns a type). We have two polytypic primitives, $\texttt{in}_{\texttt{F}}$ and $\texttt{out}_{\texttt{F}}$, for explicitly getting values into and out of the recursive type. Their types are as follows:

$$\texttt{in}_{\texttt{F}} \;\texttt{::}\; \texttt{F}\,(\mu\texttt{F}) \to \mu\texttt{F}$$
$$\texttt{out}_{\texttt{F}} \;\texttt{::}\; \mu\texttt{F} \to \texttt{F}\,(\mu\texttt{F})$$

But we can extend these operators as follows: the kind of μ becomes the following

$$(\times\langle -^{:I}\star\rangle \to \times\langle -^{:I}\star\rangle) \to \times\langle -^{:I}\star\rangle$$

i.e., it takes a functor transforming I-tuples of types and returns an I-tuple of types. So, μF is a tuple of recursive types. The primitives in and out are also extended to be n-tuples of functions:

$$\texttt{in}_{\texttt{F}} \;\texttt{::}\; \times\langle^{\texttt{i:I}}\ (\texttt{F}\,(\mu\texttt{F}))\texttt{.i} \to (\mu\texttt{F})\texttt{.i}\rangle$$
$$\texttt{out}_{\texttt{F}} \;\texttt{::}\; \times\langle^{\texttt{i:I}}\ (\mu\texttt{F})\texttt{.i} \to (\texttt{F}\,(\mu\texttt{F}))\texttt{.i}\rangle$$

The polytypic function cata is also extended; compare the original and the n-tuple versions of cata[5]:

$$\texttt{cata}_{\texttt{F}} \;\texttt{::}\; (\texttt{F a} \to \texttt{a}) \to (\mu\texttt{F} \to \texttt{a}) \qquad \text{(original)}$$
$$\texttt{cata}_{\texttt{F}} \;\texttt{::}\; \times\langle^{\texttt{i:I}}\ (\texttt{F a})\texttt{.i} \to \texttt{a.i}\rangle \to \times\langle^{\texttt{i:I}}\ (\mu\texttt{F})\texttt{.i} \to \texttt{a.i}\rangle \qquad \text{(n-tuple)}$$

$$\texttt{cata}_{\texttt{F}}\ \phi = \mu\texttt{f} \mapsto \phi \circ (\texttt{F f}) \circ \texttt{out}_{\texttt{F}} \qquad \text{(original)}$$
$$\texttt{cata}_{\texttt{F}}\ \phi = \mu\texttt{f} \mapsto \langle^{\texttt{i}}\ \phi\texttt{.i} \circ (\texttt{F f})\texttt{.i} \circ \texttt{out}_{\texttt{F}}\texttt{.i}\rangle \qquad \text{(n-tuple)}$$

[5]This assumes that there is some form of polytypism—note the application of the functor F to a term.

So, $\texttt{cata}_{\texttt{F}}$ takes and returns an n-tuple of functions. All laws (such as cata-fusion) can now be generalized. Also, the standard functor laws for a functor F of kind $\star \to \star$

$$\begin{aligned} \texttt{id} &= \texttt{F id} \\ \text{F f} \circ \text{F g} &= \text{F (f} \circ \text{g)} \end{aligned}$$

can be generalized to functors of kind $\times\langle -{:^I}\star\rangle \to \times\langle -{:^J}\star\rangle$:

$$\begin{aligned} \langle -\ \texttt{id}\rangle &= \texttt{F}\ \langle -\ \texttt{id}\rangle \\ \langle \overset{\cdot}{\texttt{j}}\ \texttt{(F f).j} \circ \texttt{(F g).j}\rangle &= \texttt{F}\ \langle \overset{\cdot}{\texttt{i}}\ \texttt{f.i} \circ \texttt{g.i}\ \rangle \end{aligned}$$

The original functor laws can be derived from these by instantiating the n-tuples to 1-tuples and then making use of the isomorphism $\times\langle a\rangle \approx a$ (the bijections being $x \mapsto x.1_1$ and $x \mapsto \langle x\rangle$).

8.4 An Explicitly Typed Calculus with N-Tuples

So, n-tuples seem quite useful, but the difficulty is in developing a sound type system for them. This section presents a higher order, explicitly typed lambda calculus with n-tuples and n-sums. This calculus was introduced by the author in [78] as the "Zip Calculus." This calculus starts as F_ω—though in the form of a Pure Type System (PTS) [7, 61]. To this is added a construct for n-tuples and then n-sums are added. As the syntax of terms and types are very close (because tuples exist at the type level), the choice of a PTS seemed natural: in a PTS, terms, types, and kinds are all written in the same syntax. Also, the generality of a PTS makes for fewer typing rules. However, the generality of a PTS can make a type system harder to understand: it is difficult to know what is a valid term, type, and kind without understanding the type checking rules.

e	::=	v	variables
	|	$\lambda v{:}t.\,e$	abstraction
	|	$e_1 e_2$	application
	|	$\Pi v{:}t_1.\,t_2$	type of abstractions
	|	$\star$	type of types
	|	$\langle e_1, e_2, \ldots \rangle$	tuple
	|	m_n	projection ($1 \leq m \leq n$)
	|	$n\mathsf{d}$	dimension ($1 \leq n$)
	|	D	type of dimensions
	|	$+_d\, t$	sum type
	|	In_d	constructors for $+_d$
	|	case_d	destructor for $+_d$
i	::=	e	projections (of type $n\mathsf{d}$)
d	::=	e	dimensions (of type D)
t	::=	e	types and kinds (of type $\star$ or $\Box$)
m, n	::=	{natural numbers}	

Figure 8.3: Syntax

8.4.1 Syntax and Semantics

The syntax of the terms of the zip calculus is in Figure 8.3. The pseudo syntactic classes *i*, *d*, and *t* are used to provide intuition for what is enforced by the type system (but not by the syntax). The first five terms in Figure 8.3 correspond to F_ω, encoded as a PTS (although one needs to see the typing rules in the following section to get the full story). In a PTS, terms and types are merged into a single syntax. The correspondence between F_ω in the standard formulation and as a PTS is as follows:

standard	PTS	
$\lambda x{:}\alpha.\,e$	$\lambda x{:}\alpha.\,e$	value abstraction
$\Lambda\alpha.e$	$\lambda\alpha{:}\star.\,e$	type abstraction
$\alpha \to \beta$	$\Pi v{:}\alpha.\,\beta$ (v not free in β)	function type
$\forall\alpha.B$	$\Pi\alpha{:}\star.\,B$	quantification

So, lambda abstractions are used both for value abstractions and type abstractions; Π terms are used for the function type and quantification; $\star$ represents the type of types. (For a more leisurely introduction to Pure Type Systems, see [61].)

To this base are added the following: (1) Tuples which are no longer restricted to the term level but also exist at the type level. (2) Projection constants (m_n - get the m-th element of an n-tuple), their types (nd - dimensions, where $m_n : n$d; "d" here is the literal character), and "D" the type of these nd ("D" has a role analogous to $\star$). And (3) n-sums made via n-tuples: for n-sums ($+_{n\mathtt{d}}\langle t_1, ..., t_n\rangle$) the constructor family, $In_{n\mathtt{d}}$, is an n-tuple of constructors and the destructor $\mathtt{case}_{n\mathtt{d}}$ takes an n-tuple of functions.

Since one can write tuples of types, one must distinguish between $\langle t_1, t_2\rangle$ (a tuple of types, having kind $\Pi\_ : 2\mathtt{d} \rightarrow \star$)[6] and $\times_{2\mathtt{d}}\langle t_1, t_2\rangle$ (a type, i.e., something with kind $\star$).

A 3-tuple such as $\langle e_1, e_2, e_3\rangle$ is a function whose domain is the set $\{1_3, 2_3, 3_3\}$ (the projections with type 3d). To get the second element of a 3-tuple, one applies the 2_3 projection to it; thus "$\langle e_1, e_2, e_3\rangle\ 2_3$" reduces to e_2. The type of the tuple is a "dependent type" (a Π term): for instance, $\langle e_1, e_2, e_3\rangle$ has type "$\Pi i : 3\mathtt{d}\,.\,\langle E_1, E_2, E_3\rangle\ i$" where $e_i : E_i$. Genericity over tuple length is achieved because we can write functions such as "$\lambda d : D.\lambda i : d.e$" in which d can be any dimension (1d,2d,...). Although tuples are functions, the following syntactic sugar is used to syntactically distinguish tuple functions from standard functions:

$$\begin{array}{rcl} \langle^{\mathtt{i:I}}\ \mathtt{e}\rangle & \equiv & \lambda\mathtt{i:I.\ e} \\ \mathtt{e.i} & \equiv & \mathtt{e\ i} \\ \times_d \mathtt{t} & \equiv & \Pi\mathtt{i}{:}d\,\mathtt{.t\ i} \end{array}$$

Also, in what follows, $a \rightarrow b$ is used as syntactic sugar for $\Pi\_ : a.b$; in this case, a Π type corresponds to a normal function. Translating the (β reduce) law into the above syntactic sugar gives this law:

[6] Note that the variable "_" is used for unused variables.

$$\begin{array}{rcll} (\lambda v{:}t.e_1)\,e_2 & = & e_1\{e_2/v\} & (\beta \text{ reduce}) \\ \langle e_1,\ldots,e_n\rangle\, i_n & = & e_i & (\times \text{ reduce}) \\ \text{case}_d\ e\,(In_d.i\ e') & = & e.i\ e' & (+ \text{ reduce}) \end{array}$$

Figure 8.4: Reduction Rules

$$\langle^{i:d} e\rangle.j \quad = \quad e\{j/i\} \qquad \text{(n-tuple reduce)}$$

The semantics is given operationally, similarly to Section 2.2. Reduction contexts are defined inductively as follows:

$$\begin{array}{rcll} R & = & [] & \text{(hole)} \\ & | & R\,e & \\ & | & \texttt{case}\ e\ R & \end{array}$$

The one step reduction relation, $\Rightarrow$, is the least relation satisfying the reduction rules (given in Figure 8.4) and the following rule (i.e., it is closed under reduction contexts):

$$R[e_1] \Rightarrow R[e_2] \quad \text{if} \quad e_1 \Rightarrow e_2$$

Multi-step reduction, $\Rightarrow^*$, is the transitive, reflexive closure of $\Rightarrow$. Evaluation, $\Downarrow$, is defined as follows: $e \Downarrow c$ if and only if $e \Rightarrow^* c$.

8.4.2 The Type System

The terms of a PTS consist of the first four terms of Figure 8.3 (variables, lambda abstractions, applications, and Π terms) plus a set of constants, $\mathcal{C}$. The specification of a PTS is given by a triple ($\mathcal{S}$,$\mathcal{A}$,$\mathcal{R}$) where $\mathcal{S}$ is a subset of $\mathcal{C}$ called the sorts, $\mathcal{A}$ is a set of axioms of the form "$c : s$" where $c \in \mathcal{C}, s \in \mathcal{S}$, and $\mathcal{R}$ is a set of rules of the form $(s1,s2,s3)$ where $s1,s2,s3 \in \mathcal{S}$. The typing judgments for a PTS are as in Figure 8.5. In a PTS, the definition of $=_\beta$ in the judgment (conv) is beta-equivalence (alpha-equivalent terms are identified).

In the zip calculus, the set of sorts is $\mathcal{S} = \{1\mathtt{d}, 2\mathtt{d}, \ldots\} \cup \{\star, \Box, D\}$, the set of constants is $\mathcal{C} = \mathcal{S} \cup \{m_n | 1 \leq m \leq n\}$, and the axioms $\mathcal{A}$ and rules $\mathcal{R}$ are as follows:

$$\frac{\Gamma \vdash a : A, \quad \Gamma \vdash B : s, \quad A =_\beta B}{\Gamma \vdash a : B}\ (\text{conv}) \qquad \frac{c : s \in \mathcal{A}}{\vdash c : s}\ (\text{axiom})$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma \vdash b : B \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash b : B}\ (\text{weak})$$

$$\frac{\Gamma \vdash f : (\Pi x : A.B), \quad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B\{a/x\}}\ (\text{app}) \qquad \frac{\Gamma, x : A \vdash b : B, \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}\ (\text{lam})$$

$$\frac{\Gamma \vdash A : s, \quad \Gamma, x : A \vdash B : t, \quad (s,t,u) \in \mathcal{R}}{\Gamma \vdash (\Pi x : A.B) : u}\ (\text{pi})$$

Figure 8.5: Type Judgments for a Pure Type System

$$\frac{\forall j \in \{1..n\}.\, \Gamma \vdash a_j : A_j,\ \ \Gamma \vdash (\Pi i : n\mathtt{d}.\, \langle A_1, \ldots, A_n \rangle\, i) : t}{\Gamma \vdash \langle a_1, ..., a_n \rangle\ :\ \Pi i : n\mathtt{d}.\, \langle A_1, \ldots, A_n \rangle\, i}\ (\text{tuple})$$

Figure 8.6: Additional Type Judgments for the Zip Calculus

$\mathcal{A}$ axioms
$\star : \Box$
$m_n : n\mathtt{d}$
$n\mathtt{d} : \mathrm{D}$
$\mathrm{D} : \Box$

$\mathcal{R}$ rules	
$(\star, \star, \star)$	$\lambda v_e : t\,.\,e$
$(\Box, \star, \star)$	$\lambda v_t : T\,.\,e$
$(\Box, \Box, \Box)$	$\lambda v_t : T\,.\,t$
$(\mathrm{D}, \mathrm{D}, \star)$	$\lambda v_i : d\,.\,i$
$(\mathrm{D}, \star, \star)$	$\lambda v_i : d\,.\,e$
$(\mathrm{D}, \Box, \Box)$	$\lambda v_i : d\,.\,t$

The $\mathcal{R}$ rules, used in the (pi) rule, indicate what lambda abstractions are allowed (which is the same as saying which Π terms are well-typed). Here we have six $\mathcal{R}$ rules which correspond to the six allowed forms of lambda abstraction. The expression to the right of each rule is an intuitive representation of the type of lambda abstraction which the rule represents (e - terms, t - types, T - kinds, i - projections, d - dimensions, v_x - variable in class x). For instance, the $(\mathrm{D}, \mathrm{D}, \star)$ rule means that lambda abstractions are allowed of form $\lambda v_i : d\,.\,i$ where $d : \mathrm{D}$ (i.e., d is a dimension such as $3\mathtt{d}$) and thus v_i represents a projection such as 2_3 and the body i must have type D, and the type of the type of this whole lambda expression is $\star$.

In the zip calculus there is an additional term, $\langle e_1, e_2, ... \rangle$, which cannot be treated as a constant in a PTS (ignoring sums for the moment). The addition of this term requires two extensions to the PTS: one, an additional typing judgment (Figure 8.6) and two, the $=_\beta$ relation in the (conv) judgment must be extended to include not just (β reduce) but also ($\times$ reduce) and an additional law ($\times$ eta):

$$\langle e\,1_n, \ldots, e\,n_n \rangle \quad = \quad e \qquad \text{if } e :: \Pi i{:}n\mathtt{d}.A \qquad (\times \text{ eta})$$

To get generic sums, one needs only add $+$ as a constant and the following two primitives

$$\begin{array}{l} \mathtt{In} \quad :: \Pi \mathtt{I{:}D}.\ \Pi \mathtt{a{:}}\times_{\mathtt{I}}\langle -^{:\mathtt{I}} \star\rangle.\qquad \times_{\mathtt{I}}\langle \mathtt{i}^{:\mathtt{I}}\ \mathtt{a.i}\rightarrow +_{\mathtt{I}}\mathtt{a}\rangle \\ \mathtt{case} :: \Pi \mathtt{I{:}D}.\ \Pi \mathtt{a{:}}\times_{\mathtt{I}}\langle -^{:\mathtt{I}} \star\rangle.\ \Pi \mathtt{b{:}}\star.\ \times_{\mathtt{I}}\langle \mathtt{i}^{:\mathtt{I}}\ \mathtt{a.i}\rightarrow \mathtt{b}\rangle \rightarrow (+_{\mathtt{I}}\mathtt{a}\rightarrow \mathtt{b}) \end{array}$$

where `In` is a generic injection function: e.g., for the sum $+_{2\mathtt{d}}\langle a, b\rangle$ the two injection functions are "`(In 2d` $\langle a, b\rangle$`)`$.1_2$" and "`(In 2d` $\langle a, b\rangle$`)`$.2_2$".

8.4.3 Type Checking

There are numerous properties, such as subject reduction, which are true of Pure Type Systems in general [7]. There are also known type checking algorithms for certain subclasses of PTSs. Although the zip calculus is not a PTS, it is hoped that most results for PTSs will carry over to the "almost PTS" zip calculus.

A PTS is functional when the relations $\mathcal{A}$ and $\mathcal{R}$ are functions ($c : s_1 \in \mathcal{A}$ and $c : s_2 \in \mathcal{A}$ imply $s_1 = s_2$; $(s,t,u_1) \in \mathcal{R}$ and $(s,t,u_2) \in \mathcal{R}$ imply $u_1 = u_2$). In the zip calculus, $\mathcal{A}$ and $\mathcal{R}$ are functions. If a PTS is functional there is an efficient type-checking algorithm as given in Figure 8.7 (cf. [61] and [81]), where the type judgments of Figure 8.5 have been restructured to make them syntax-directed. The judgment (red) defines the relation "$\Gamma \vdash x :\twoheadrightarrow X$" and $\twoheadrightarrow_\beta$ is beta-reduction.

$$\frac{\Gamma \vdash f :\twoheadrightarrow (\Pi x : A.B), \quad \Gamma \vdash a : A', A =_\beta A'}{\Gamma \vdash f\, a : B\{a/x\}}\,(\text{app}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,(\text{var})$$

$$\frac{\Gamma, x : A \vdash b : B, \quad \Gamma \vdash (\Pi x : A.B) : t}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}\,(\text{lam}) \qquad \frac{c : s \in \mathcal{A}}{\vdash c : s}\,(\text{axiom})$$

$$\frac{\Gamma \vdash A :\twoheadrightarrow s, \quad \Gamma, x : A \vdash B :\twoheadrightarrow t, \quad (s,t,u) \in \mathcal{R}}{\Gamma \vdash (\Pi x : A.B) : u}\,(\text{pi}) \qquad \frac{\Gamma \vdash a : A, \quad A \twoheadrightarrow_\beta B}{\Gamma \vdash a :\twoheadrightarrow B}\,(\text{red})$$

Figure 8.7: Syntax Directed Type Judgments for a Functional PTS

$$\frac{\forall j \in \{1..n\}.\, \Gamma \vdash a_j :\twoheadrightarrow A_j, \;\; \Gamma \vdash (\Pi i : n\mathsf{d}.\, \langle A_1, \ldots, A_n \rangle\, i) : t}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle \;:\; \Pi i : n\mathsf{d}.\, \langle A_1, \ldots, A_n \rangle\, i}\,(\text{tuple1})$$

$$\frac{\forall j \in \{1..n\}.\, \Gamma \vdash a_j :\twoheadrightarrow A}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle \;:\; \Pi\_ : n\mathsf{d}.\, A}\,(\text{tuple2})$$

$$\frac{\Gamma \vdash f :\twoheadrightarrow C, \quad \Gamma \vdash a :\twoheadrightarrow A, C =_\eta \Pi x : A.B}{\Gamma \vdash f\, a : B\{a/x\}}\,(\text{app}')$$

$$\frac{\Gamma \vdash a : A, \quad A \twoheadrightarrow_{\beta\delta} B}{\Gamma \vdash a :\twoheadrightarrow B}\,(\text{red}')$$

Figure 8.8: Syntax Directed Type Judgments for the Zip Calculus

This algorithm can be modified as in Figure 8.8. The rules (tuple1) and (tuple2) replace (tuple) from Figure 8.6. The rules (app′) and (red′) replace the (app) and (red) judgments of Figure 8.7. Here $\twoheadrightarrow_{\beta\delta}$ is $\twoheadrightarrow_{\beta}$ extended with ($\times$ reduce) and $=_{\eta}$ is equality up to ($\times$ eta) convertibility. The reason for the change of (app) is because f may evaluate to

$$\langle \Pi x : a_1 . b_1, ..., \Pi x : a_n . b_n \rangle . i$$

and application should be valid when, for instance, this is equivalent to a type of the form

$$\Pi x : (\langle a_1, ..., a_n \rangle . i) \, . \, \langle b_1, ..., b_n \rangle . i$$

A proof of the soundness and completeness of this algorithm should be similar to that in [81].

8.5 Conclusion

8.5.1 Type Inference

The simply typed language PATH-L, given in Chapter 2, clearly allows for type inference. Does the zip calculus of the previous section allow for type inference? And would the zip calculus extended with μ and integer primitives allow for type inference? Currently, I have implemented an algorithm for PATH which infers most general types (up to $=_{\eta}$) for all the programs shown in this thesis, so I expect that it indeed can be shown to be sound and complete. An exposition of the algorithm and such proofs are left for future work.

8.5.2 Limitations

An n-tuple is similar to a heterogeneous array (or heterogeneous finite list); but although one can map over n-tuples, zip n-tuples together, and transpose nested n-tuples, one cannot *induct* over n-tuples. So, n-tuples are clearly limited in what they can express. As a result, one could not define the following functions in a Haskell extended with n-tuples (although they could be provided as primitives):

```
tupleToList         :: ×⟨− a⟩ → [a]
seqTupleL,seqTupleR :: Monad m => ×⟨i a.i →m b.i⟩ → ×a →m(×b)
```

However, if `seqTupleL` and `seqTupleR` were provided as primitives, we get a great deal of expressiveness (without the need to extend n-tuples to allow some form of induction):

- Each of these families of Haskell functions could be generalized to a generic function:

  ```
  zip, zip3, ...
  zipWith, zipWith3, ...
  unzip, unzip3, ...
  liftM1, liftM2, ...
  ```

- There are a number of list functions in Haskell that work "uniformly" on lists—they act on lists without permuting the elements or changing the length: `zip`, `zipWith`, `unzip`, `map`, `sequence`, `mapM`, `transpose`, `mapAccumL`, `mapAccumR`. We can write a tuple version of each of these.

Other functions cannot be given a type in the zip calculus. For instance, there is the `curry` family of functions but there is no way to give a type to a generic `curry`:

```
curry2 :: ((a,b)→c)    → (a→b→c)
curry3 :: ((a,b,c)→d) → (a→b→c→d)
```

8.5.3 Related Work

Polytypic programming [47, 46, 49] has similar goals to this work (e.g., PolyP [41] and Functorial ML [42]). (Intensional type analysis [30] is similar to polytypism, it gives the same expressiveness, and sometimes more, at the expense of a heavier notation.) However, n-tuples do not give us polytypism (nor does polytypism give us genericity over the length of tuples); these are orthogonal language extensions:

- Polytypism allows for generalizing over type constructors (e.g., List, Maybe, Tree), but does not allow for genericity over the length of tuples. E.g., polytypism generalizes over

  ```
  zipList2, zipMaybe2, zipTree2, ...
  ```

- N-tuples cannot generalize over type constructors, only over the length of tuples: E.g., n-tuples generalizes over

  ```
  zipList2, zipList3, zipList4, ...
  ```

As seen in Section 8.3.5, with *both* polytypism and n-tuples some very generic programs and laws can be written.

Currently, projections have their dimension embedded (e.g., the projection "1_3" has dimension (or type) "3d"); to allow for projections that are "polymorphic" over dimensions (e.g., projection 1 could be applied to a tuple of any size) would take us into the realm of extensible records [24, 54, 86]. N-tuples and extensible records appear to be orthogonal issues.

The following table is an attempt to summarize the differences between the genericity provided by n-tuples, polytypism, and extensible records. Each allows for abstracting over a different *x*:

Form of Genericity	Context	What is abstracted
n-tuples	e.x	$x \in \{1_n, 2_n, ..., n_n\}$
polytypism	x e	$x \in$ {mapList, mapMaybe, mapTree, ...}
extensible records	e.2_x	$x \in \{2d, 3d, 4d, ...\}$

With n-tuples, we can abstract over all the projections of a given arity; with polytypism, we can abstract over the functor of type "$(a \to b) \to (Fa \to Fb)$" for each type functor F (at least with one form of polytypism); with extensible records, we can abstract over the dimension of the projection (in other words, we can overload the 2 projection).

It would be a simple and obvious extension to allow for finite sets other than the naturals as projections, e.g., one could have strings as projections and finite sets of strings as dimensions.

Two approaches that achieve the same genericity as n-tuples are the following: First, one can forgo typed languages and use an untyped language to achieve this level of genericity: e.g., in Lisp a list can be used as an n-tuple. Second, a language with dependent types [4] could encode n-tuples (and more); though the disadvantages are that type checking is undecidable (not to mention the lack of type inference) and the types are more complex. N-tuples can be viewed as a way to add dependent types in a restricted way to a typed language.

Related also is Hoogendijk's thesis [36] in which he develops a notation (not a type system) for n-tuple valued functors for program calculation; his notation is variable free, categorical, and heavily overloaded.

Section 5.4 of Morrisett's thesis [53] presents an extension of his explicitly typed λ_i^{ML} calculus which is similar in power to n-tuples. This extension allows for lists at the type level and has an additional kind, k^*, for lists of types. Unfortunately no properties were proved for this extension. This extension allows for inductive definitions of types (over a

list of types). The zip calculus cannot do induction over tuples of types. However, the zip calculus provides a simpler and more elegant way—in my opinion—to express functions at the type level that are commonly needed: e.g., matrix transpose, zipping tuples and matrices, mapping functions across tuples and matrices, etc. Further exploring the relation between the extended λ_i^{ML} and the zip calculus is an area for future work.

The recent work by Shao et al. [72] introduces a complex and expressive type system for certified binaries. Their type system resembles the calculus of constructions [18] extended with induction definitions. It appears feasible that the zip calculus can be embedded into this system. Such an embedding would allow the zip calculus to inherit all the properties already proved for their calculus: subject reduction, strong normalization, confluence, etc. I plan to explore this in future work.

8.5.4 Summary

So, a new form of genericity has been developed for typed languages: genericity over the length of tuples. We have seen that this genericity gives us shorter programs, fewer and more general laws, and shorter derivations. Future work is to increase the expressiveness of n-tuples (so `seqTupleL`, `seqTupleR`, and `tupleToList` can be defined in the language and `curry` could be given a type). Although it is questionable whether such an extended system would allow for type-inference.

Chapter 9

Conclusion

9.1 Contributions

In this dissertation, I have made the following contributions:

- Described the PATH program transformation system.
- Integrated the two major approaches to transformation—the generative set approach and the schematic approach—by showing how we can achieve, using the schematic approach, the expressiveness of a powerful generative set approach, expression procedures.
- Began developing a catalog of useful laws, all of which can be derived from a small number of primitive rules and laws.
- Developed a logic for program transformation which is more expressive than two level horn clauses but less general than first order logic. Given some examples of how this logic lends itself to the graphical display of program derivations.

- Developed a new form of genericity for typed languages: genericity over the length of tuples. Developed an explicitly typed calculus for this.

9.2 Related Work

In the body of this work, PATH has been compared and contrasted in detail to previous transformation systems and methods. In this section, a very broad survey of the work being done in program transformation will be given. More detailed surveys of work in program transformation can be found in [19] and [56].

There are many similarities between theorem proving [58] and program transformation. A program transformation system can be viewed as a specialized theorem prover—a theorem prover specialized for proving equivalences of a particular programming language. I will not attempt to explore this connection further here but will proceed to survey the work in program transformation per se. Work in program transformation can be categorized into four broad categories: transformation systems, generative set methods, theories, and fully automatic meta-programs.

Transformation Systems. Numerous systems have been built for supporting user directed program transformation. One of the seminal systems is the Munich CIP project [8, 9]. This system was based on the schematic approach, but with extensions to support fold/unfold, a proof of termination being required to preserve correctness. This system used a wide-spectrum language which had constructs for non-executable descriptions, functional programming, and procedural programming. A recent system which builds on (and simplifies) that work is the Ultra system [55]; it transforms a subset of a Haskell-like language. Other transformation systems are Prospectra [35], Starship [21], and KIDS [73]. This is

but a small sampling of a large number of systems; Firth [21] gives an overview and comparison of many more transformation systems.

Generative Set Methods. One of the oldest and most well known methods for transforming functional programs is the fold/unfold approach of Burstall and Darlington [15]. In spite of its drawbacks, in particular its lack of total correctness, it is commonly used for informal reasoning about functional programs and is the basis of some program transformation systems. Scherlis's expression procedure method [69, 66] preserves total-correctness and does not require the keeping of a transformation history as fold/unfold does. Unfortunately this method has been largely ignored; this may be due to the lack of interest in total-correctness in the transformation community. The tick algebra of Sands [67, 68] is a very general and powerful approach. It has been extended to deal with sharing [52] and to reason about space usage [29].

These generative set methods are asymmetric. This asymmetry is what gives these methods their power and certainly is desired when we want to reason about things as improvement (as with Sands's Tick Algebra). However, the lack of symmetry can be undesirable, as explained in Section 3.1.4.

Theories for Program Transformation. In the generative set approaches, there is an emphasis on developing strategies and meta-programs for achieving transformation of programs, but in the schematic approaches, there is an emphasis on discovering new laws one can use to transform programs. This discovery and development of sets of laws for program transformation has been subject of much work. A important line of such work was pioneered by Bird and Meertens [12, 13, 48] and has been carried on by many others [11, 6, 23, 49, 25]. This approach to program transformation is called by many names: the calculational approach, the Bird-Meertens Formalism (BMF), and Squiggol (due to its

fondness for using new symbols—or squiggols). It is characterized by an emphasis on developing powerful laws by which one can conduct linear, equational proofs of program equivalence, without reverting to inductive proofs. Numerous laws, or theories, have been developed for lists and various recursion schemes.

There is a growing interest in using this calculational approach in the development of fully automated methods (e.g., deforestation [27, 76]) where previously generative set approaches were used [83, 16]. The advantages of using the calculational approach is that there is no need to keep a history or to determine where to perform folding.

The BMF approach has focused on using a small set of recursion schemes and disallowing general recursion. Thus, only total functions can be written—this makes reasoning simpler, not having to deal with non-termination. This work has been extended to deal with partial functions [49].

The disadvantage of BMF is that general recursion is disallowed. The motivation for the result in Chapter 4 was to maintain the elegance of the calculational approach while allowing for general recursion.

Fully Automatic Meta-Programs. In contrast to *user directed* program transformation, there are *fully automated* meta-programs in which major program transformations are accomplished using sophisticated algorithms. Some examples of these are deforestation [83, 16], partial evaluation [14, 43], tupling [37], super-compilation [80, 74], and etc. When the desired transformation is achieved, these methods can give dramatic results (though only constant time improvements for most of these methods). The drawback is when the desired transformation is not achieved: the meta-program rarely gives useful feedback and allows for little control over the transformation process.

I believe that the best use of meta-programs is in the context of a user directed trans-

formation system: The goal in PATH is not to achieve major transformations in a single application of one complex meta-program to the whole program, but rather it is to allow for simpler meta-programs to be applied—selectively and often—in conjunction with user-directed transformations.

9.3 Future Directions

There are numerous areas for future research related to this work.

Language Extensions. As explained in Chapter 8, genericity is useful to have in the language, but even more important in a transformation system. Adding polytypism [46, 47] to the language is the next step toward increasing genericity. How best to add polytypism to a typed language is still an area of active research [5, 40, 32, 33].

Currently PATH does not support Haskell's type classes. As type classes are implemented by dictionary passing, it would appear straightforward to support type classes by transforming dictionary passing code. However, as polytypism overlaps with type classes [31], another possibility is to support type classes using polytypism.

Haskell implementations allow for strictness annotations and perform some degree of strictness analysis. If PATH is to evolve into a industrial-quality tool, it also needs to support strictness annotations and be capable of strictness analysis at least as powerful as current compilers. How is this strictness information to be encoded and used in PATH? One approach to doing so is using the approach taken by Launchbury and Paterson [45].

One area in which PATH has totally avoided the reality of Haskell is in the area of pattern-matching. Haskell has numerous sophisticated pattern-matching constructs. These add to the complexity of Haskell and make transformation much more complex. In the Starship

system, great pains are taken to support transforming patterns at the source level [21] (this in a Haskell-like language with simpler pattern-matching constructs). Another approach is to use the author's work on first class patterns [77]: the complex pattern-matching of Haskell would be transformed into pattern-combinators which would then be the subject of transformation.

Multiple Program Relations. Currently PATH supports reasoning about only a single program relation: program equivalence. By supporting multiple program relations, PATH could add more precision to program derivations (at the expense of some complexity). For instance, to support reasoning about sharing, we would have two relations, $=_{name}$ and $=_{need}$, the latter being used for transformations which preserve sharing properties and the former being used for programs which may not preserve sharing [52]. Or, we could add a non-symmetric improvement relation, $\trianglerighteq$, for transformations which improve some measure of efficiency [68].

Improving Meta-Programs. Currently PATH has a few simple built in meta-programs. One area for further research here is in the continued development of such meta-programs as deforestation, partial evaluation, and tupling for use in PATH. A second area of research here is in extending PATH to allow for user-written meta-programs, as can be done in theorem provers. Also, techniques for proof search could be borrowed from the theorem proving community to be applied automatically in PATH.

GUI Design and Development. Currently PATH has a two-dimensional interface to the program derivation. It is certainly an improvement over older systems that interface to the user via textual commands and which have no user visible representation for the derivation (only for the final program). However, the next step would be to improve this interface to

one in which the user can 1) selectively display and hide parts of the program or derivation, 2) control the layout of the derivation, 3) navigate the program in a manner similar to an outline editor, and 4) be given visual feedback when and where laws would be applicable.

Appendix A

The PATH-L Prelude

```
data List a     = Nil | Cons ×⟨a,List a⟩
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Bool       = False | True


f ∘ g = x ↦ f(g x)

map = μmap ↦ f ↦ case ⟨Nil       : Nil
                      ,Cons⟨y,ys⟩: Cons⟨f y, map f ys⟩
                      ⟩

length  = μlength ↦ case ⟨Nil: 0, Cons⟨x,xs⟩: 1 + length xs⟩

sum     = μsum ↦ case ⟨Nil: 0, Cons⟨x,xs⟩: x + sum xs⟩

iterate = μiterate ↦ f ↦ x ↦ Cons⟨x, iterate f (f x)⟩

filter  = μfilter ↦ p ↦
            case⟨Nil       : Nil
                ,Cons⟨x,xs⟩: if p x
                             then Cons⟨x,filter p xs⟩
                             else filter p xs
                ⟩

cata = μcata ↦ ⟨f,b⟩ ↦ case ⟨Nil       : b
                            ,Cons⟨y,ys⟩: f⟨y, cata ⟨f,b⟩ ys⟩
                            ⟩

id = x ↦ x

if = b ↦ t ↦ f ↦  case ⟨True: t, False: f⟩ b
```

Appendix B

Primitive Rules & Laws

B.1 Syntactic Sugar

```
 let x:t=e in C x
={SS}
 (x:t ↦ C x) e

 letrec x₁:t₁ = e₁; ...; xₙ:tₙ = eₙ in C⟨x₁,...,xₙ⟩
={SS}
 let ⟨x₁,...,xₙ⟩=μ⟨x₁,...,xₙ⟩:×⟨t₁,...,tₙ⟩ ↦ ⟨e₁,...,eₙ⟩ in C⟨x₁,...,xₙ⟩

 let f x = F x in C f
={SS}
 let f = x ↦ F x in C f

 let f.i = F i in C f
={SS}
 let f = ⟨ⁱ F i⟩ in C f

 if p then t else f
={SS}
 if p t f
```

B.2 Primitive Rules

Reduction:

$$(p \mapsto e_1)\ e_2 =\{red\}\ e_1\{e_2/p\}$$

$$\langle e_1,\ldots,e_n\rangle.j_n =\{red\}\ e_j$$

$$\texttt{case}\ \langle e_1,\ldots,e_n\rangle\ (\texttt{In}.i_n\ x) =\{red\}\ e_i\ x$$

$$\mu p \mapsto f =\{red\}\ f\{\mu p \mapsto f\ /\ p\}$$

$$\mathit{prim}\langle c_1,\ldots,c_n\rangle =\{red\}\ [\![\mathit{prim}\langle c_1,\ldots,c_n\rangle]\!]$$

Strictness:

$$\begin{aligned}
\bot\ e &=\{red\}\ \bot\\
\bot.e &=\{red\}\ \bot\\
\texttt{case}\ e\ \bot &=\{red\}\ \bot\\
\mathit{prim}\langle e_1,\ldots,\bot,\ldots,e_n\rangle &=\{red\}\ \bot\\
\mu\ \bot &=\{red\}\ \bot
\end{aligned}$$

Eta:

$$\begin{aligned}
&\forall x{:}a\to b &&.\ x =\{eta\}\ v \mapsto x\ v && (v \text{ not free in } x)\\
&\forall x{:}\times\langle t_1,t_2,\ldots,t_n\rangle &&.\ x =\{eta\}\ \langle x.1_n,\ x.2_n,\ \ldots\ ,\ x.n_n\rangle\\
&\forall x{:}+\langle t_1,t_2,\ldots,t_n\rangle &&.\ x =\{eta\}\ \texttt{case}\ \langle \texttt{In}.1_n,\texttt{In}.2_n,\ldots,\texttt{In}.n_n\rangle\ x
\end{aligned}$$

Letrec Equivalence:

$$\begin{aligned}
&\texttt{letrec}\ f_1=F_1;f_2=F_2;\ldots;f_n=F_n;\ g_1=G_1;g_2=G_2;\ldots;g_m=G_m\ \texttt{in}\ M\\
=&\{\texttt{letrec}\}\\
&\texttt{letrec}\ f_1=F_1;f_2=F_2;\ldots;f_n=F_n;\ \langle g_1,g_2,\ldots,g_m\rangle=\langle G_1,G_2,\ldots,G_m\rangle\ \texttt{in}\ M
\end{aligned}$$

B.3 Law *FPD* (Fixed Point Duplication)

```
∀F,G.
  letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in f
=
  letrec f=(letrec g=G⟨f,g⟩ in F⟨f,g⟩) in f
```

Alternatively, using μ notation,

$$\forall F,G.\quad (\mu\langle f,g\rangle \mapsto \langle F\langle f,g\rangle, G\langle f,g\rangle\rangle).1 \;=\; \mu f \mapsto F\langle f, \mu g \mapsto G\langle f,g\rangle\rangle$$

B.4 Law *FPI* (Fixed Point Induction)

```
∀C,D,F,G.
  C ⊥ = D ⊥
; ∀x,y . {C x = D y} ⇒ {C(F x) = D(G y)}
⇒
  C(μF) = D(μG)
```

B.5 Law *Inst* (Instantiation)

```
∀F,H,X.
  H ⊥ = ⊥
 ⇒
  H(case ⟨i y ↦ F.i y⟩ X) = case ⟨i y ↦ H(F.i y)⟩ X
```

B.6 Law *List-Induct* (Structural Induction on Lists)

```
∀ C : List a → b, D : List a → b, xs : List a.
  C ⊥ = D ⊥
; C Nil = D Nil
; ∀x,xs. C xs = D xs ⇒ C( Cons⟨x,xs⟩ ) = D( Cons⟨x,xs⟩ )
⇒
  C xs = D xs
```

B.7 Law *N-Tuple-Eta*

```
∀x:×a  .  x = ⟨i x.i⟩
```

Appendix C

Derived Transformation Laws

This appendix contains a list of derived laws in PATH, along with their derivations.

C.1 *Abides*

$$\forall F.\ \ \mathtt{case}\ \langle^{i}\ y \mapsto \langle^{j}\ F.i.j\ y\rangle\rangle\ x = \langle^{j}\ \mathtt{case}\langle^{i}\ F.i.j\rangle\ x\rangle$$

Derivation:

$$\begin{array}{ll}
\forall F. & \\
\quad \mathtt{case}\ \langle^{i}\ y \mapsto \langle^{j}\ F.i.j\ y\rangle\rangle\ x & \\
\quad = & \{\text{N-Tuple-Eta}\} \\
\quad \langle^{j}\ (\mathtt{case}\ \langle^{i}\ y \mapsto \langle^{j}\ F.i.j\ y\rangle\rangle\ x).j\rangle & \\
\quad = & \{\text{Inst}\} \\
\quad \langle^{j}\ \ \mathtt{case}\ \langle^{i}\ y \mapsto \langle^{j}\ F.i.j\ y\rangle.j\rangle\ x\rangle & \\
\quad = & \{\text{red}\} \\
\quad \langle^{j}\ \ \mathtt{case}\ \langle^{i}\ y \mapsto F.i.j\ y\rangle\ x\rangle & \\
\quad = & \{\text{R N-Tuple-Eta}\} \\
\quad \langle^{j}\ \ \mathtt{case}\ \langle^{i}\ F.i.j\rangle\ x\rangle &
\end{array}$$

C.2 *Case-Strict*

```
∀E. case ⊥ E = ⊥
```

Derivation:

```
∀E.
  case ⊥ E
=                                  {Prod-Bot}
  case ⟨i ⊥⟩ E
=                                  {Func-Bot}
  case ⟨i x ↦ ⊥ ⟩ E
=                                     {R red}
  case ⟨i x ↦ ⊥ ⟨⟩⟩ E
=                                   {R Inst}
  ⊥ (case ⟨i x ↦ ⟨⟩⟩ E)
=                                      {red}
  ⊥
```

C.3 *Cata-Merge*

```
∀F,B. xs ↦ ⟨i cata ⟨F.i,B.i⟩ xs⟩ = cata ⟨⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩, B⟩
```

Derivation:

```
∀F,B.
  xs ↦ ⟨i cata ⟨F.i,B.i⟩ xs⟩
= {FPF-N
    xs ↦ ⟨i ⊥ ⟨F.i,B.i⟩ xs⟩
  =                                                              {red, red}
    xs ↦ ⟨i ⊥⟩
  =                                                {R Func-Bot, R Prod-Bot}
    ⊥
  ; ∀cata.
      xs ↦ ⟨i (⟨f,b⟩ ↦ case ⟨Nil       : b
                            ,Cons⟨y,ys⟩: f⟨y, cata ⟨f,b⟩ ys⟩
                            ⟩) ⟨F.i,B.i⟩ xs ⟩
    =                                                                {red}
      xs ↦ ⟨i (case ⟨Nil       : B.i
                    ,Cons⟨y,ys⟩: F.i⟨y, cata ⟨F.i,B.i⟩ ys⟩⟩ xs) ⟩
    =                                                               {Inst}
      xs ↦ case ⟨Nil       : ⟨i B.i⟩
                ,Cons⟨y,ys⟩: ⟨i F.i⟨y, cata ⟨F.i,B.i⟩ ys⟩⟩⟩ xs
    =                                                  {R N-Tuple-Eta}
      xs ↦ case ⟨Nil       : B
                ,Cons⟨y,ys⟩: ⟨i F.i⟨y, cata ⟨F.i,B.i⟩ ys⟩⟩⟩ xs
    =                                                            {R eta}
      case ⟨Nil       : B
           ,Cons⟨y,ys⟩: ⟨i F.i⟨y, cata ⟨F.i,B.i⟩ ys⟩⟩⟩
    =                                                            {R red}
      case ⟨Nil       : B
           ,Cons⟨y,ys⟩: ⟨i F.i⟨y, (⟨i cata ⟨F.i,B.i⟩ ys⟩.i⟩⟩⟩
    =                                                            {R red}
      case ⟨Nil       : B
           ,Cons⟨y,ys⟩: ⟨i F.i⟨y, ((xs ↦ ⟨i cata ⟨F.i,B.i⟩ xs⟩)ys).i⟩⟩⟩
 }
  μg ↦ case ⟨Nil: B, Cons⟨y,ys⟩: ⟨i F.i ⟨y, (g ys).i⟩⟩⟩
=                                                                {R red}
  μg ↦ case ⟨Nil: B, Cons⟨y,ys⟩: (⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩) ⟨y,g ys⟩⟩
=                                                                {R red}
  (⟨f,b⟩ ↦ μg ↦ case⟨Nil: b, Cons⟨y,ys⟩: f⟨y,g ys⟩⟩)
   ⟨⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩, B⟩
=                                                {Lambda-Mu-Switch}
  (μg ↦ ⟨f,b⟩ ↦ case⟨Nil: b, Cons⟨y,ys⟩: f⟨y,g⟨f,b⟩ ys⟩⟩)
   ⟨⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩, B⟩
=                                                          {def. cata}
  cata ⟨⟨y,z⟩ ↦ ⟨i F.i ⟨y,z.i⟩⟩, B⟩
```

C.4 *Components-Strict-Implies-Tuple-Strict*

$$\forall F.\ \langle^{i}\ \{F.i\ \bot = \bot\}\rangle \ \Rightarrow\ (x \mapsto \langle^{i}\ F.i\ x\rangle)\ \bot = \bot$$

Derivation:

```
∀F.
 P₁: ⟨ⁱ {F.i ⊥ = ⊥}⟩
⇒
  (x ↦ ⟨ⁱ F.i x⟩) ⊥
=                               {red}
  ⟨ⁱ F.i ⊥⟩
=                               {P₁}
  ⟨ⁱ ⊥⟩
=                        {R Prod-Bot}
  ⊥
```

C.5 *FPD’* (Fixed Point Duplication - Alternative)

$$\forall F.\ \mu f \mapsto F\langle f,f\rangle \ =\ \mu f \mapsto F\langle f,\mu f \mapsto F\langle f,f\rangle\rangle$$

Derivation:

```
∀F.
  μf ↦ F⟨f,f⟩
=                                                        {SS}
  letrec f = F⟨f,f⟩ in f
=                                                {Mix-Letrec}
  letrec f = F⟨f,f′⟩; f′ = F⟨f′,f′⟩ in f
=                                                       {FPD}
  letrec f = F⟨f,letrec f′ = F⟨f′,f′ ⟩ in f′⟩ in f
=                                                        {SS}
  μf ↦ F⟨f,μf ↦ F⟨f,f⟩⟩
```

C.6 *FPE* (Fixed Point Expansion)

```
∀F. μf ↦ F⟨f,f⟩ = μf ↦ F⟨f,F⟨f,f⟩⟩
```

Derivation:

```
∀F.
  μf ↦ F⟨f,f⟩
=                                                              {SS}
  letrec f=F⟨f,f⟩ in f
=                                                      {Mix-Letrec}
  letrec f=F⟨f,f'⟩; f'=F⟨f,f⟩ in f
=                                                              {FPD}
  letrec f=F⟨f,letrec f'=F⟨f,f⟩ in f'⟩ in f
=                                                     {Inline-Body}
  letrec f=F⟨f,letrec f'=F⟨f,f⟩ in F⟨f,f⟩⟩ in f
=                                                          {GC-let}
  letrec f=F⟨f,F⟨f,f⟩⟩ in f
=                                                              {SS}
  μf ↦ F⟨f,F⟨f,f⟩⟩
```

C.7 *FPF* (Fixed Point Fusion)

```
∀C,F,G.
   C ⊥ = ⊥
 ; ∀x. C(F x) = G(C x)
 ⇒
   C(μF) = μG
```

Derivation:

```
∀C,F,G.
   P1: C ⊥ = ⊥
 ; P2: ∀x. C(F x) = G(C x)
 ⇒
    C (μF)
  = {FPI
      C ⊥ ={P1} ⊥ ={R red} id ⊥
    ; ∀x,y.
         P3: C x = id y
       ⇒
          C (F x)
        =                {P2}
          G (C x)
        =                {P3}
          G (id y)
        =               {red}
          G y
        =             {R red}
          id(G y)
    }
=
  id(μG)
=                  {red}
  μG
```

C.8 *FPF-Ext* (Fixed Point Fusion - Extended)

```
∀C,F,G,H.
  C ⊥ = ⊥
; ∀f.  letrec g=G⟨f,g,C f⟩ in { C(F⟨f,g⟩) = H⟨f,g,C f⟩ }
⇒
  letrec f=F⟨f,g⟩; g=G⟨f,g,C f⟩ in ⟨f,g⟩
= letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in ⟨f,g⟩
```

Derivation:

```
∀C,F,G,H.
  P1: C ⊥ = ⊥
; P2: ∀f.  letrec g=G⟨f,g,C f⟩ in { C(F⟨f,g⟩) = H⟨f,g,C f⟩ }
⇒
  letrec f=F⟨f,g⟩; g=G⟨f,g,C f⟩ in ⟨f,g⟩
=
```

Zooming in on "`C f`" (rules may apply to the larger context):

```
   C f
 =                                                          {Letrec-Exp}
   C(letrec f=F⟨f,g⟩; g=G⟨f,g,C f⟩ in f)
 =                                                                  {FPD}
   C(letrec f=(letrec g = G⟨f,g,C f⟩ in F⟨f,g⟩)in f)
 =
   {FPF-Partial
     C ⊥ ={P1} ⊥
   ; ∀f'.
       C(letrec g = G⟨f',g,C f'⟩ in F⟨f',g⟩)
     =                                                      {Letrec-Ctxt}
       letrec g = G⟨f',g,C f'⟩ in C(F⟨f',g⟩)
     =                                                            {P2}
       letrec g = G⟨f',g,C f'⟩ in H⟨f',g,C f'⟩
   }
   letrec f=(letrec g=G⟨f,g,h⟩in F⟨f,g⟩)
        ; h=(letrec g=G⟨f,g,h⟩in H⟨f,g,h⟩) in h
 =                                                          {R FPD}
   letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩;
          h=(letrec g=G⟨f,g,h⟩in H⟨f,g,h⟩) in h
 =                                                    {R Letrec-Exp}
   letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in h

=
  letrec f=F⟨f,g⟩;
         g=G⟨f,g,letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in h⟩ in ⟨f,g⟩
=                                                                {R GC}
  letrec f=F⟨f,g⟩;
         g=G⟨f,g,letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in h⟩
         h=H⟨f,g,h⟩ in ⟨f,g⟩
=                                                    {R Letrec-Exp}
  letrec f=F⟨f,g⟩; g=G⟨f,g,h⟩; h=H⟨f,g,h⟩ in ⟨f,g⟩
```

C.9 *FPF-N* (Fixed Point Fusion - On N Mu's)

$$
\begin{array}{l}
\forall C,F,G. \\
\quad C\ \bot = \bot \\
;\ \forall x\ .\ C\langle^{i}\ F.i\ x.i\rangle = G(C\ x) \\
\Rightarrow \\
\quad C\langle^{i}\ \mu(F.i)\rangle = \mu G
\end{array}
$$

Derivation:

$$
\begin{array}{l}
\forall C,F,G. \\
\quad P_1\colon C\ \bot = \bot \\
;\ P_2\colon \forall x\ .\ C\langle^{i}\ F.i\ x.i\rangle = G(C\ x) \\
\Rightarrow \\
\quad C\langle^{i}\ \mu(F.i)\rangle \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\text{Split}\} \\
\quad C(\mu(x \mapsto \langle^{i}\ F.i\ x.i\rangle)) \\
= \{\text{FPF} \\
\qquad C\ \bot = \{P_1\}\ \bot \\
\quad ;\ \forall x.\ C(\langle^{i}\ F.i\ x.i\rangle)\ \ = \{P_2\}\ \ G(C\ x) \\
\quad \} \\
\quad \mu G
\end{array}
$$

C.10 *FPF-Partial* (Fixed Point Fusion - Partial)

```
∀C,F,G.
  C ⊥ = ⊥
; ∀f.  C(F⟨f,C f⟩)  =  G⟨f,C f⟩
⇒
  C(letrec f=F⟨f,C f⟩ in f)  =  letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in g
```

Derivation:

```
∀C,F,G.
  P₁: C ⊥ = ⊥
; P₂: ∀f.  C(F⟨f,C f⟩)  =  G⟨f,C f⟩
⇒
  mlet D x = ⟨x,C x⟩ in
  {
    C(letrec f=F⟨f,C f⟩ in f)
  =                                                        {def. D, R red}
    D(letrec f=F⟨f,C f⟩ in f) .2
  =
    {FPF
        D ⊥
      =                                                          {def. D}
        ⟨⊥,C ⊥⟩
      =                                                            {P₁}
        ⟨⊥,⊥⟩
      =                                                     {R Prod-Bot}
        ⊥
    ; ∀f.
        D(F⟨f,C f⟩)
      =                                                          {def. D}
        ⟨F⟨f,C f⟩ , C(F⟨f,C f⟩)⟩
      =                                                            {P₁}
        ⟨F⟨f,C f⟩ , G⟨f,C f⟩   ⟩
      =                                                          {def. D}
        ⟨F(D f)   , G(D f)     ⟩
    }
  =
    (μ h ↦ ⟨F h,G h⟩)       .2
  =                                                              {SS}
    (letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in ⟨f,g⟩).2
  =                                                       {Letrec-Ctxt}
    letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in ⟨f,g⟩.2
  =                                                            {red}
    letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in g
  }
```

C.11 *Func-Bot*

$\bot_{[a \to b]} = x{:}a \mapsto \bot_{[b]}$

Derivation:

$$
\begin{array}{ll}
 & \bot_{[a \to b]} \\
= & \qquad\qquad \{eta\} \\
 & x{:}a \mapsto \bot_{[a \to b]}\ x \\
= & \qquad\qquad \{red\} \\
 & x{:}a \mapsto \bot_{[b]}
\end{array}
$$

C.12 *GC* (Garbage Collect Letrec)

$\forall F,G,C.$
$\quad \texttt{letrec f = F f; g=G}\langle \texttt{f,g} \rangle \texttt{ in C f} \;=\; \texttt{letrec f = F f in C f}$

Derivation:

$$
\begin{array}{ll}
 & \forall F,G,C. \\
 & \quad \texttt{letrec f = F f; g=G}\langle \texttt{f,g} \rangle \texttt{ in C f} \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{FPD\} \\
 & \quad \texttt{letrec f = (letrec g=G}\langle \texttt{f,g} \rangle \texttt{ in F f) in C f} \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \{SS\} \\
 & \quad \texttt{letrec f = (let g=}\mu \texttt{g} \mapsto \texttt{G}\langle \texttt{f,g} \rangle \texttt{ in F f) in C f} \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{GC\text{-}let\} \\
 & \quad \texttt{letrec f = F f in C f}
\end{array}
$$

C.13 *GC-Let* (Garbage Collect Let)

$\forall X,M.\ \ \texttt{let x = X in M} \;=\; \texttt{M}$

Derivation:

$$
\begin{array}{ll}
 & \forall X,M. \\
 & \quad \texttt{let x = X in M} \\
= & \qquad\qquad\qquad\qquad\ \{SS\} \\
 & \quad (x \mapsto M)\ X \\
= & \qquad\qquad\qquad\qquad\ \{red\} \\
 & \quad M
\end{array}
$$

C.14 *Inline-Bndg*

```
∀C,F,G. letrec f=F⟨f,g⟩; g=G⟨f,{ f=F⟨f,g⟩ },g⟩ in C⟨f,g⟩
```

Derivation:

```
∀C,F,G.
  letrec f=F⟨f,g⟩; g=G⟨f,f,g⟩ in C⟨f,g⟩
=                                                          {R Letrec-Ctxt}
  C(letrec f=F⟨f,g⟩; g=G⟨f,f,g⟩ in ⟨f,g⟩)
=                                                                    {SS}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩, G⟨f,f,g⟩⟩)
=                                                                {R red}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩, G⟨f,⟨f,g⟩.1,g⟩⟩)
=                                                                  {FPE}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩, G⟨f,⟨F⟨f,g⟩, G⟨f,⟨f,g⟩.1,g⟩⟩.1,g⟩⟩)
=                                                                  {red}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩, G⟨f,F⟨f,g⟩,g⟩⟩)
=                                                                   {SS}
  C(letrec f=F⟨f,g⟩; g=G⟨f,F⟨f,g⟩,g⟩ in ⟨f,g⟩)
=                                                        {Letrec-Ctxt}
  letrec f=F⟨f,g⟩; g=G⟨f,F⟨f,g⟩,g⟩ in C⟨f,g⟩
```

C.15 *Inline-Body*

```
∀F,G,C. letrec f=F⟨f,g⟩; g=G⟨f,g⟩ in C⟨f,{ f=F⟨f,g⟩ },g⟩
```

Derivation:

```
∀F,G,C.
  mlet fg′ = μ⟨f,g⟩ ↦ ⟨F⟨f,g⟩,G⟨f,g⟩⟩ in
  {
    letrec f = F⟨f,g⟩; g=G⟨f,g⟩ in C⟨f,f,g⟩
  =                                                                  {SS}
    let ⟨f,g⟩=fg′ in C⟨f,f,g⟩
  =                                                              {R red}
    let ⟨f,g⟩=fg′ in C⟨f,⟨f,g⟩.1         ,g⟩
  =                                                      {Inline-Let}
    let ⟨f,g⟩=fg′ in C⟨f,  fg′.1         ,g⟩
  =                                                            {red}
    let ⟨f,g⟩=fg′ in C⟨f,⟨F fg′,G fg′⟩.1 ,g⟩
  =                                                            {red}
    let ⟨f,g⟩=fg′ in C⟨f,F fg′           ,g⟩
  =                                                    {R Inline-Let}
    let ⟨f,g⟩=fg′ in C⟨f,F⟨f,g⟩          ,g⟩
  =                                                             {SS}
    letrec f = F⟨f,g⟩; g=G⟨f,g⟩ in C⟨f,F⟨f,g⟩,g⟩
  }
```

C.16 *Inline-Let*

```
∀C,X.  let x=X in C⟨x,{x=X}⟩
```

Derivation:

```
∀C,X.
  let x=X in C⟨x,x⟩
=                                {SS}
  (x ↦  C⟨x,x⟩) X
=                               {red}
  C⟨X,X⟩
=                             {R red}
  (x ↦ C⟨x,X⟩) X
=                               {SS}
  let x=X in C⟨x,X⟩
```

C.17 *Inline-Self*

```
∀C,F,G. letrec f=F⟨f,{ f=F⟨f,f,g⟩ },g⟩; g=G⟨f,g⟩ in C⟨f,g⟩
```

Derivation:

```
∀C,F,G.
  letrec f=F⟨f,f,g⟩; g=G⟨f,g⟩ in C⟨f,g⟩
=                                                          {R Letrec-Ctxt}
  C(letrec f=F⟨f,f,g⟩; g=G⟨f,g⟩ in ⟨f,g⟩)
=                                                                    {SS}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,f,g⟩,G⟨f,g⟩⟩)
=                                                                 {R red}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,⟨f,g⟩.1,g⟩,G⟨f,g⟩⟩)
=                                                                   {FPE}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f,⟨F⟨f,⟨f,g⟩.1,g⟩,G⟨f,g⟩⟩.1,g⟩,G⟨f,g⟩⟩)
=                                                                   {red}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f, F⟨f,⟨f,g⟩.1,g⟩          ,g⟩,G⟨f,g⟩⟩)
=                                                                   {red}
  C(μ⟨f,g⟩ ↦ ⟨F⟨f, F⟨f,f      ,g⟩          ,g⟩,G⟨f,g⟩⟩)
=                                                                    {SS}
  C(letrec f=F⟨f,F⟨f,f,g⟩,g⟩; g=G⟨f,g⟩ in ⟨f,g⟩)
=                                                            {Letrec-Ctxt}
  letrec f=F⟨f,F⟨f,f,g⟩,g⟩; g=G⟨f,g⟩ in C⟨f,g⟩
```

C.18 *Lambda-Mu-Switch*

$$\forall F.\ \ \mu f \mapsto x \mapsto F\langle f\ x, x\rangle \ \ = \ \ x \mapsto \mu f \mapsto F\langle f, x\rangle$$

Derivation:

$$\begin{array}{l}
\forall F. \\
\quad \mu f \mapsto x \mapsto F\langle f\ x, x\rangle \\
= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\text{eta}\} \\
\quad y \mapsto \ (\mu f \mapsto x \mapsto F\langle f\ x, x\rangle)\ y \\
= \{\text{FPF} \\
\quad\ \ \bot\ y = \{\text{red}\}\ \bot \\
\quad ;\ \forall f'. \\
\qquad\ ((f \mapsto x \mapsto F\langle f\ x, x\rangle)\ f')\ y \\
\qquad = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\text{red}\} \\
\qquad\ (x \mapsto F\langle f'\ x, x\rangle)\ y \\
\qquad = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\text{red}\} \\
\qquad\ F\langle f'\ y, y\rangle \\
\quad \} \\
\quad y \mapsto \mu f \mapsto F\langle f, y\rangle
\end{array}$$

C.19 *Let-Ctxt*

```
∀C,D,X.   C(let x=X in D x)  =  let x=X in C(D x)
```

Derivation:

```
∀C,D,X.
  C(let x=X in D x)
=                                    {SS}
  C((x ↦ D x) X)
=                                  {red}
  C(D X)
=                                {R red}
  (x ↦ C(D x)) X
=                                    {SS}
  let x=X in C(D x)
```

C.20 *Letrec-Ctxt*

```
∀C,D,F.  C(letrec f = F f in D f)  =  letrec f = F f in C(D f)
```

Derivation:

```
∀C,D,F.
  C(letrec f = F f in D f)
=                                                {SS}
  C(let f = μf ↦ F f in D f)
=                                          {Let-Ctxt}
  let f = μf ↦ F f in C(D f)
=                                                {SS}
  letrec f = F f in C(D f)
```

C.21 *Letrec-Equiv*

```
∀F,G₁,G₂.
  letrec f=F⟨f,g₁,g₂⟩;⟨g₁,g₂⟩=⟨G₁⟨f,g₁,g₂⟩,G₂⟨f,g₁,g₂⟩⟩ in ⟨f,g₁,g₂⟩
=
  letrec f=F⟨f,g₁,g₂⟩; g₁=G₁⟨f,g₁,g₂⟩; g₂=G₂⟨f,g₁,g₂⟩ in ⟨f,g₁,g₂⟩
```

Derivation:

```
∀F,G₁,G₂.

mlet H⟨f,g⟩=⟨f,g.1,g.2⟩ in
{
  letrec f=F⟨f,g₁,g₂⟩;⟨g₁,g₂⟩=⟨G₁⟨f,g₁,g₂⟩,G₂⟨f,g₁,g₂⟩⟩ in ⟨f,g₁,g₂⟩
=                                                                    {SS}
  letrec f=F⟨f,g.1,g.2⟩; g=⟨G₁⟨f,g.1,g.2⟩,G₂⟨f,g.1,g.2⟩⟩ in ⟨f,g.1,g.2⟩
=                                                                {def. H}
  letrec f=F⟨f,g.1,g.2⟩; g=⟨G₁⟨f,g.1,g.2⟩,G₂⟨f,g.1,g.2⟩⟩ in H⟨f,g⟩
=                                                          {R Letrec-Ctxt}
  H(letrec f=F⟨f,g.1,g.2⟩; g=⟨G₁⟨f,g.1,g.2⟩,G₂⟨f,g.1,g.2⟩⟩ in ⟨f,g⟩
=                                                                    {SS}
  H(μ⟨f,g⟩ ↦ ⟨F⟨f,g.1,g.2⟩,⟨G₁⟨f,g.1,g.2⟩,G₂⟨f,g.1,g.2⟩⟩)
={FPF
      H ⊥
    =                                                          {Prod-Bot}
      H⟨⊥,⊥⟩
    =                                                            {def. H}
      ⟨⊥,⊥.1,⊥.2⟩
    =                                                           {red,red}
      ⟨⊥,⊥,⊥⟩
    =                                                        {R Prod-Bot}
      ⊥
  ;
    ∀⟨f,g⟩.
      H(⟨F⟨f,g.1,g.2⟩,⟨G₁⟨f,g.1,g.2⟩,G₂⟨f,g.1,g.2⟩⟩)
    =                                                            {def. H}
      ⟨F⟨f,g.1,g.2⟩, G₁⟨f,g.1,g.2⟩, G₂⟨f,g.1,g.2⟩⟩
    =                                                            {def. H}
      ⟨F(H⟨f,g⟩)   , G₁(H⟨f,g⟩)   , G₂(H⟨f,g⟩)   ⟩
 }
  μh ↦ ⟨F h, G₁ h, G₂ h⟩
=                                                                    {eta}
  μ⟨f,g₁,g₂⟩ ↦ ⟨F⟨f,g₁,g₂⟩,G₁⟨f,g₁,g₂⟩,G₂⟨f,g₁,g₂⟩⟩
=                                                                    {SS}
  letrec f=F⟨f,g₁,g₂⟩; g₁=G₁⟨f,g₁,g₂⟩; g₂=G₂⟨f,g₁,g₂⟩ in ⟨f,g₁,g₂⟩
}
```

C.22 *Letrec-Exp*

```
∀F,G,C.
 letrec f=F⟨f,g⟩; g=G⟨f                  ,f,g⟩ in ⟨f,g⟩
=
 letrec f=F⟨f,g⟩; g=G⟨letrec f=F⟨f,g⟩ in f,f,g⟩ in ⟨f,g⟩
```

Derivation:

```
∀F,G,C.
 letrec f=F⟨f,g⟩; g=G⟨f ,f,g⟩ in ⟨f,g⟩
=                                                                {Mix-Letrec}
 letrec f=F⟨f,g⟩; g=G⟨f',f,g⟩; f'=F⟨f',g⟩; g'=G⟨f',f',g'⟩ in ⟨f,g⟩
=                                                                      {GC}
 letrec f=F⟨f,g⟩; g=G⟨f',f,g⟩; f'=F⟨f',g⟩ in ⟨f,g⟩
=                                                                   {FPD}
 letrec f=F⟨f,g⟩; g=(letrec f'=F⟨f',g⟩ in G⟨f',f,g⟩) in ⟨f,g⟩
=                                                          {Letrec-Ctxt}
 letrec f=F⟨f,g⟩; g=G⟨letrec f'=F⟨f',g⟩ in f',f,g⟩ in ⟨f,g⟩
=                                                               {rename}
 letrec f=F⟨f,g⟩; g=G⟨letrec f=F⟨f,g⟩ in f,f,g⟩ in ⟨f,g⟩
```

C.23 *Mix*

$$\forall F,M.\quad \langle - \ \mu f \mapsto F\langle - \ f\rangle\rangle \ = \ \mu f \mapsto \langle^{i}\ F\langle^{j}\ f.(M.i.j)\rangle\rangle$$

Or, using letrec notation,

```
∀F,M.
  letrec f   = F⟨– f⟩ in ⟨– f⟩
=
  letrec f.i = F⟨ʲ f.(M.i.j)⟩ in f
```

Derivation:

```
∀F,M.
  ⟨ⁱ μf ↦ F⟨ʲ f⟩⟩
= {FPF
    ⟨ⁱ ⊥⟩
   =                              {R Prod-Bot}
    ⊥
  ;
    ∀f.
      ⟨ⁱ F⟨ʲ f⟩⟩
    =                                   {R red}
      ⟨ⁱ F⟨ʲ ⟨ⁱ f⟩.(M.i.j)⟩⟩
  }
  μf ↦ ⟨ⁱ F⟨ʲ f.(M.i.j)⟩⟩
```

C.24 *Mix-Letrec*

```
∀F,M,j.
  letrec f   = F⟨— f⟩ in f
=
  letrec f.i = F⟨j f.(M.i.j)⟩ in f.j
```

Derivation:

```
∀F,M,j.
  letrec f   = F⟨— f⟩ in f
=                                                    {R red}
  letrec f   = F⟨— f⟩ in ⟨— f⟩.j
=                                                {Letrec-Ctxt}
  (letrec f   = F⟨— f⟩ in ⟨— f⟩).j
=                                                      {Mix}
  (letrec f.i = F⟨j f.(M.i.j)⟩ in f).j
=                                            {R Letrec-Ctxt}
  letrec f.i = F⟨j f.(M.i.j)⟩ in f.j
```

C.25 *Partial-Mu-Reduce*

```
∀F,G.  μ(F ∘ G) = F(μ(G ∘ F))
```

Derivation:

```
∀F,G.
   μ (F ∘ G)
=                                                      {SS}
  letrec f = (F ∘ G) f in f
=                                                     {red}
  letrec f = F (G f) in f
=                                                   {R GC}
  letrec f = F (G f); g = G f in f
=                                             {R Inline-Bndg}
  letrec f = F g; g = G f in f
=                                               {Inline-Body}
  letrec f = F g; g = G f in F g
=                                             {R Letrec-Ctxt}
  F(letrec f = F g; g = G f in g)
=                                         {Inline-Bndg, GC}
  F(letrec g = G (F g) in g)
=                                                    {R red}
  F(letrec g = (G ∘ F) g in g)
=                                                      {SS}
  F(μ (G ∘ F))
```

C.26 *Prod-Bot*

$$\bot_{[\times \mathtt{a}]} = \langle^{\mathtt{i}}\ \bot_{[\mathtt{a}.\mathtt{i}]}\rangle$$

Derivation:

$$\begin{array}{ll}
\bot_{[\times \mathtt{a}]} & \\
= & \{\mathtt{R\ eta}\} \\
\langle^{\mathtt{i}}\ \bot_{[\times \mathtt{a}]}.\mathtt{i}\rangle & \\
= & \{\mathtt{red}\} \\
\langle^{\mathtt{i}}\ \bot_{[\mathtt{a}.\mathtt{i}]}\rangle &
\end{array}$$

C.27 *Split*

$$\forall \mathtt{F}.\ \ \mu \mathtt{x} \mapsto \langle^{\mathtt{i}}\ \mathtt{F.i\ x.i}\rangle\ \ =\ \ \langle^{\mathtt{i}}\ \mu\ \mathtt{F.i}\rangle$$

Derivation:

$$\begin{array}{ll}
\forall \mathtt{F}. & \\
\quad \mu \mathtt{x} \mapsto \langle^{\mathtt{i}}\ \mathtt{F.i\ x.i}\rangle & \\
\quad = & \{\mathtt{N\text{-}Tuple\text{-}Eta}\} \\
\quad \langle^{\mathtt{j}}\ \ (\mu(\mathtt{x} \mapsto \langle^{\mathtt{i}}\ \mathtt{F.i\ x.i}\rangle).\mathtt{j}\ \rangle & \\
\quad = \{\mathtt{FPF} & \\
\qquad \bot.\mathtt{j} =\{\mathtt{red}\}\ \bot & \\
\qquad ; & \\
\qquad\ \forall \mathtt{y}\ . & \\
\qquad\quad ((\mathtt{x} \mapsto \langle^{\mathtt{i}}\ \mathtt{F.i\ x.i}\rangle)\ \mathtt{y}).\mathtt{j} & \\
\qquad\quad = & \{\mathtt{red}\} \\
\qquad\quad \langle^{\mathtt{i}}\ \mathtt{F.i\ y.i}\rangle.\mathtt{j} & \\
\qquad\quad = & \{\mathtt{red}\} \\
\qquad\quad \mathtt{F.j\ y.j} & \\
\quad \} & \\
\quad \langle^{\mathtt{j}}\ \mu(\mathtt{x} \mapsto \mathtt{F.j\ x})\rangle & \\
\quad = & \{\mathtt{R\ eta}\} \\
\quad \langle^{\mathtt{j}}\ \mu\ \mathtt{F.j}\rangle &
\end{array}$$

C.28 *Trivial-Fusion*

```
∀F,H,I.
  H ⊥ = ⊥
; ∀x. I(H x) = x
⇒
  H(μF) = μg ↦ H(F(I g))
```

Derivation:

```
∀F,H,I.
  P1:  H ⊥ = ⊥
; P2:  ∀x. I(H x) = x
⇒
  H(μF)
= {FPF
       H ⊥
    =                        {P1}
       ⊥
  ;
    ∀f.
      H(F f)
    =                      {R P2}
      H(F(I(H f)))
  }
  μg ↦ H(F(I g))
```

C.29 *Tuple-Strict-Implies-Components-Strict*

```
∀F. { (x ↦ ⟨i F.i x⟩) ⊥ = ⊥ }  ⇒  ⟨i {F.i ⊥ = ⊥}⟩
```

Derivation:

```
∀F.
 P1: (x ↦ ⟨i F.i x⟩) ⊥ = ⊥
⇒
  ⟨i F.i ⊥⟩
=                            {R red}
  (x ↦ ⟨i F.i x⟩) ⊥
=                               {P1}
  ⊥
=                          {Prod-Bot}
  ⟨i ⊥⟩
```

C.30 *Unused-Parameter-Elimination*

```
∀A,B,C,D,F.
  (μf ↦ ⟨x,y⟩ ↦ F⟨⟨i  f ⟨C.i x, D.i⟨f,x,y⟩⟩,x⟩) ⟨A,B⟩
=
  (μf ↦ x     ↦ F⟨⟨i  f (C.i x)⟩           ,x⟩) A
```

Derivation:

```
∀A,B,C,D,F.
mlet fst = ⟨x,y⟩ ↦ x in
{
  (μf ↦ x     ↦ F⟨⟨i  f (C.i x)⟩         ,x⟩) A
=                                                                {R red}
  (μf ↦ x     ↦ F⟨⟨i  f (C.i x)⟩         ,x⟩) ⟨A,B⟩.1
=                                                                {R red}
  ((μf ↦ x     ↦ F⟨⟨i  f (C.i x)⟩        ,x⟩) . fst) ⟨A,B⟩
= { FPF
      ⊥ . fst
    =                                                {def. compose}
      x ↦ ⊥ (fst x)
    =                                                      {red}
      x ↦ ⊥
    =                                                  {R Func-Bot}
      ⊥
  ; ∀f.
      (x ↦ F⟨⟨i  f (C.i x)⟩         ,x⟩) . fst)
    =                                                      {red}
      z ↦  F⟨⟨i  f (C.i (fst z)⟩                , fst z⟩
    =                                                      {eta}
      ⟨x,y⟩ ↦ F⟨⟨i  f(C.i (fst ⟨x,y⟩)⟩          , fst ⟨x,y⟩⟩
    =                                                {red,red}
      ⟨x,y⟩ ↦ F⟨⟨i  f(C.i x)⟩                   , x⟩)
    =                                                    {R red}
      ⟨x,y⟩ ↦ F⟨⟨i  f(fst ⟨C.i x, D.i⟨f,x,y⟩⟩   , x⟩)
    =                                                    {R red}
      ⟨x,y⟩ ↦ F⟨⟨i  (f.fst) ⟨C.i x, D.i⟨f,x,y⟩⟩ , x⟩)
  }
  (μf ↦ ⟨x,y⟩ ↦ F⟨⟨i  f ⟨C.i x, D.i⟨f,x,y⟩⟩,x⟩) ⟨A,B⟩
}
```

Bibliography

[1] Z. M. Ariola and S. Blom. Cyclic lambda calculi. In *Theoretical Aspects of Computer Software*, pages 77–106, 1997. 116

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, San Francisco, California, Jan. 1995. 19, 84

[3] J. Arsac and Y. Kodratoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, Apr. 1982. 45

[4] L. Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999. 146

[5] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *Lecture Notes in Computer Science*, volume 1608, pages 28–115. Springer-Verlag, 1999. 153

[6] R. C. Backhouse. An exploration of the Bird-Meertens formalism. Computing Science Notes CS 8810, Department of Mathematics and Computing Science, University of Groningen, Groningen, NL, 1988. 151

[7] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992. 136, 141

[8] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. G. R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985. 10, 69, 150

[9] F. L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich project CIP. Volume II: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987. 10, 35, 69, 150

[10] H. Bekic. *Programming Languages and their Definition*, volume 177 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984. 88

[11] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, 1997. 8, 24, 36, 151

[12] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36, pages 5–42. Springer-Verlag, 1987. NATO ASI Series. 36, 151

[13] R. S. Bird and L. Meertens. Two exercises found in a book on algorithmics. In L. Meertens, editor, *Program Specification and Transformation*. North-Holland, 1987. 36, 151

[14] D. Bjorner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988. 152

[15] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977. 7, 24, 30, 151

[16] W. N. Chin. Safe fusion of functional expressions. *ACM LISP Pointers*, 5(1):11–20, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming. 152

[17] W. N. Chin and J. Darlington. Schematic rules within unfold/fold approach to program transformation. In *TENCON '89: Fourth IEEE Region 10 International Conference*, 1989. 41, 65

[18] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. 147

[19] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In L. Meertens, editor, *Program specification and transformation*, pages 165–195. North-Holland, 1987. 150

[20] M. Felleisen, D. Friedmand, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987. 18

[21] M. A. Firth. *A Fold/Unfold Transformation System for a Non-Strict Language*. PhD thesis, University of York, 1990. 20, 27, 150, 151, 154

[22] R. W. Floyd. Assigning meanings to programs. *Proceedings Symposium on Applied Mathematics*, 19:19–32, 1967. 2

[23] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992. 151

[24] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical report NOTTCS-TR-96-3, University of Nottingham, Languages and Programming Group, Department of Computer Science, Nottingham NG7 2RD, UK, Nov. 1996. 145

[25] J. Gibbons. An introduction to the Bird-Meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, 1994. 151

[26] J. Gibbons and G. Hutton. Proof methods for structured corecursive programs. In *Proceedings of the 1st Scottish Functional Programming Workshop, Stirling, Scotland*, 1999. 111

[27] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993. 152

[28] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 633–674. North-Holland, Amsterdam, 1990. 45, 81

[29] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In A. D. Gordon and A. M.Pitts, editors, *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999. 151

[30] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, Jan. 1995. 145

[31] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, Sept. 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28. 153

[32] R. Hinze. A new approach to generic functional programming. In T. W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, Jan. 2000. 153

[33] R. Hinze. Polytypic values possess polykinded types. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, July 2000. 153

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, Oct. 1969. 2

[35] B. Hoffman and B. Krieg-Bruckner. *Program Development by Specification and Transformation: the PROSPECTRA Methodology, Language Family, and System*. Lecture Notes in Computer Science. Springer-Verlag, New York, 1993. 35, 150

[36] P. F. Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Dept. of Math. and Computing Science, Eindhoven Univ. of Technology, 1997. 146

[37] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, pages 164–175. ACM Press, New York, 1996. 114, 152

[38] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992. 10

[39] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Inf.*, 11:31–55, 1978. 34, 35

[40] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000. 153

[41] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997. 8, 145

[42] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998. 8, 145

[43] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993. 6, 152

[44] L. Kott. Unfold/fold transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 412–433. CUP, 1985. 27

[45] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In H. R. Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218, Linköping, Sweden, Apr. 1996. Springer-Verlag. 153

[46] G. R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen, 1990. 8, 134, 145, 153

[47] G. R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990. 8, 134, 145, 153

[48] L. Meertens. Algorithmics - towards programming as a mathematical activity. In J. W. de Bakker, E. M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. CWI Monographs, volume 1. 8, 24, 36, 151

[49] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, June 1991. 8, 24, 36, 45, 134, 145, 151, 152

[50] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990. 19

[51] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996. 88

[52] A. K. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56. ACM Press, January 1999. 84, 151, 154

[53] J. G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon Univeristy, 1995. 146

[54] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995. 145

[55] H. Partsch, W. Schulte, and T. Vullinghs. System support for the interactive transformation of functional programs. In *International Workshop on Software Transformation Systems*, 1999. 150

[56] H. Partsch and R. Steinbrueggen. Program transformation systems. *ACM Computing Surveys*, 15:199–236, 1983. 23, 150

[57] H. A. Partsch. *Specification and transformation of programs: a formal approach to software development*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1990. 3

[58] L. C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge Univ. Press, 1987. 38, 150

[59] P. Pepper. A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, 9(3):221–262, Dec. 1987. 69, 77, 83

[60] S. Peyton Jones and J. Hughes. Report on the programing language Haskell 98. Technical Report YALEU/DCS/RR-1106, Yale University, Feb. 1999. 10

[61] S. Peyton Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997. 136, 138, 141

[62] S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP'96 — European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Linköping, Sweden, April 22–24, 1996. Springer-Verlag. 6

[63] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1995. 111

[64] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975. 78

[65] C. Runciman, M. Firth, and N. Jagger. Transformation in a non-strict language: An approach to instantiation. In K. Davis and R. J. M. Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989*, pages 133–141, London, UK, 1990. Springer-Verlag. British Computer Society Workshops in Computing Series. 26

[66] D. Sands. Higher-order expression procedures. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 1995. 26, 27, 31, 62, 64, 109, 151

[67] D. Sands. Proving the correctness of recursion-based automatic program transformations. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *Lecture Notes in Computer Science*, pages 681–695. Springer-Verlag, 1995. 64, 151

[68] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, Mar. 1996. 27, 64, 151, 154

[69] W. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, California, August 1980. Stanford Computer Science Report STAN-CS-80-818. 24, 27, 62, 117, 151

[70] W. Scherlis. Program improvement by internal specialization. In *Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, January 1981*, pages 41–49. ACM, 1981. 27, 62

[71] D. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*, 1970. 81

[72] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232. ACM, January 2002. 147

[73] D. Smith. KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990. 150

[74] M. Sørensen, R. Glück, and N. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K.*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, Apr. 1994. 152

[75] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977. 45

[76] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 306–313, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press. 152

[77] M. Tullsen. First class patterns. In E. Pontelli and V. S. Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Jan. 2000. 20, 154

[78] M. Tullsen. The zip calculus. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, July 2000. 136

[79] M. Tullsen and P. Hudak. Shifting expression procedures into reverse. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 95–104, San Antonio, Texas, Jan. 1999. 65

[80] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986. 152

[81] L. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for Pure Type Systems. In H. Barendregt and T. Nipkow, editors, *Proceedings of the International Workshop on Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 19–61, Nijmegen, The Netherlands, May 1994. Springer-Verlag. 141, 143

[82] P. Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'89, London, UK*, pages 347–359, New York, Sept. 1989. ACM Press. 45, 88, 125

[83] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. 152

[84] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 10

[85] P. L. Wadler. How to replace failure by a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, Sept. 1985. 10

[86] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. IEEE Symp. on Logic in Computer Science*, page 132, 1988. 145

[87] G. Winskel. *The formal semantics of programming languages*. MIT Press, 1993. 88

[88] H. Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, Jan. 1994. 30