TOWARDS SAFER AND MORE EXPRESSIVE GENETIC PROGRAMMING THROUGH DEPENDENT TYPES

DONOVAN J CRICHTON

Submitted in partial fulfilment of the requirements of the degree of
Bachelor of Information Technology (Honours)
School of Information and Communication Technology
Griffith University

October 2019

Statement of Originality

This work has not previously been submitted for a degree or diploma in any university. To the best of myknowledge and belief, the dissertation contains no material previously published or written by another person except where due reference is made in the dissertation itself.

Signed: dcrichton

Date: 25/10/19

Donovan J Crichton: *Towards safer and more expressive Genetic Programming through Dependent Types*. Submitted in partial fulfilment of the requirements of the degree of Bachelor of Information Technology (Honours).

SUPERVISORS:

Andrew Lewis & David Billington

ABSTRACT

There is tension between expressive program representations, and verification with respect to functional correctness, in pure functional, strongly typed genetic programs. Well-typed embedded representations are difficult to manipulate, manually typed expressive representations require proofs of progress and preservation for verification. I examine existing approaches to the traversal and manipulation of generalised algebraic data type embedded languages and show that existing dependently typed zipper specifications do not hold when the generalised algebraic data type (GADT) is indexed over a universe of types. I define a new dependently typed zipper-like structure to traverse and manipulate GADT-embedded higher order abstract syntax trees. I also use this new structure to define a type-preserving crossover operation between these trees, and providing a working proof of concept in the Idris programming language. This allows for an approach to genetic programming in pure functional languages (that support dependent types) that combines the desirable traits of expressivity and functional verification.

CONTENTS

LIST OF FIGURES

LIST OF THEOREMS

LIST OF DEFINITIONS

LIST OF EXAMPLES

ACKNOWLEDGMENTS

I knew from the day I began my bachelor that I wanted to aim for Honours. The few professionals I had met in my working career came across as more knowledgeable than their pass-degree colleagues, and they would somehow grasp new or tricky concepts with an envious ease. What I did not appreciate was the sheer amount of effort and persistence a research project requires, and that such ease was likely borne from many thankless hours of intellectual and textual labour. I certainly would not have completed this thesis were it not for the patience and support of the following people:

- Dr Andrew Lewis

  My primary supervisor for this thesis. Andrew was kind enough to indulge a rogue honours student on his errant quest to implement all programs during his honours year in pure functional languages. Andrew was also kind enough to allow me the freedom to pursue an area of research rather tangential to his own expertise.

- Dr David Billington

  My co-supervisor for this thesis. David was always willing to chat about my work and discuss issues with notation, proofs, and logic.

- Dr Vladimir Estivil-Castro, Dr René Hexel, and the Griffith Machine Intelligence and Pattern Analysis Laboratory

  Vlad and René graciously provided me with a physical workspace throughout my honours candidature. This work would've been a great deal more difficult in the absence of a quiet place to work and colleagues who shared my enthusiasm for best practice in programming.

- The Brisbane Functional Programming Group

  I was lucky enough to live in one of the few Australian cities with active industry functional programming groups. The members of BFPG helped introduce me to the joys of functional programming and dependent types. In particular thank you to Isaac Elliot, George Wilson, and David Laing, for your willingness to help a floundering newbie who was mostly out of his depth.

- The members of the Discord Functional Programming server

  The friendly folk from the Discord Functional Programming server were always willing to answer any questions I had regarding Idris, Dependent Types, Type Theory or LaTeX. In particular I would like to thank the following users who were kind enough to share with me their names: Alex Gryzlov ("clayrat"), Ole Kristian Sandum ("Ole"), Edmund Noble ("edmundnoble"), Arvid Marx ("Univalence Axiom"). Along with the users who wished to remain anonymous, in particular "Melissa", "iitalics", and "user".

- My Friends

  While there are too many of my friends to name that have given me support and encouragement over the years of my honours thesis, four in particular have borne the weight of the various swings and roundabouts of elation and despondency over the years. Thank you Daniel, Ilona, Gonzalo and Nils.
  A special additional mention must be made for my good friend Fraser Tweedale, whose virtually *constant* passionate evangelising of functional programming over the many years I have known him finally encouraged me to investigate "this functional programming thing".

- My mother, Debra

  For her patience and support over the years of study.

To the myriad of friends, colleagues, chatroom users and chance encounters that I haven't had the chance to mention here, thank you as well. All the small pieces count.

— Donovan J. Crichton

1 INTRODUCTION

Genetic Programming (GP) is a form of supervised learning that results in an expression, program, or function that may then be executed by a hosting programming language. Genetic Programming offers a human-readable solution to supervised learning problems that offer insight and explanation in a way that may be desirable to sensitive domains. Functional Programming (FP) is a programming paradigm that is concerned with producing programs that are easier to formally reason about than their imperative or object oriented counterparts. There is an infamous anecdote that functional programmers often use to illustrate concepts of Monadic IO, the dangers of side effects, and the desirability of type safety. This anecdote is usually given in the form of the following imperative function.

```
void do_something(){
  // do something useful
  fire_missiles(); // accidentally inserted by a tired programmer
  // return useful result
}
```

In short, functional programs (in languages that support referential transparency) won't fire the missiles.

We can imagine then a situation where the tired programmer is replaced with a genetic program. We would like similar guarantees that this genetic program will not also fire the missiles accidentally. This is *particularly* important due to the manner in which genetic programs perform supervised learning. A GP will generate a candidate program and then test that program *by executing it*. If such a program contains a call to fire the missiles, the missiles will then be fired. We concern ourselves here with ways to achieve genetic programming so that no missiles may be fired, namely through pure functional genetic programs.

In particular we investigate well-typed language representations and their manipulations through the use of dependent types. We seek to perform a type preserving crossover operation, one of the fundamental operators in the genetic programming process, such that any modifications to our programs will never result in "missiles firing", or ill-typed expressions occuring.

Research in the field is limited. Pure functional genetic programmers not only need a knowledge of two distinct fields of computer science, but in many popular functional languages they require a knowledge of type theory as well. This is a result of an aggressive erasure of type information at run-time by such languages. Dependently typed languages have more flexible type erasure conditions and allow us to leverage the existing type systems for our program representations.

There is a tension in existing pure functional genetic programming research. Genetic Programs with expressive type systems are often hand-crafted and lacking proofs of progress and preservation, or implemented in imperative languages. Genetic Programs that are formally verified are rare, and tend to have simple type systems.

By combining the best of both approaches we aim to advance the current state of the art in formally verified genetic programming. Such systems could then be used in domains where type correctness and verification is of vital importance. For example, in defence, security, health services, or finance.

1.1 Contributions

In particular I make the following contributions in this thesis:

- A counter example of the Hamana and Fiore's zipper over simple inductive families that cannot be type-checked in common dependently typed languages.
- An alternative zipper-like data structure that allows traversal and manipulation of generalised algebraic data type (GADT) embedded higher order abstract syntax (HOAS) trees.
- A type correct and type preserving crossover operation for GADT-embedded HOAS expressions.

These contributions allow progress towards a complete functionally verified genetic program.

1.2 Thesis Structure

BACKGROUND AND LITERATURE REVIEW In Chapter 2 we cover the necessary background in dependent types, well-typed language embeddings and the current state of pure functional genetic programs. We also highlight the areas for improvement in the existing literature.

TRAVERSAL OVER HOAS TREES In Chapter 3 we review the difficulties in traversing and manipulating GADT-embedded HOAS trees. We then go on to examine existing techniques for alleviating these difficulties. In particular we examine the specification for a zipper over inductive families. We find that such a specification does not hold in our given use case, and go on to define a structure that allows us to traverse and manipulate our expression trees.

VERIFIED CROSSOVER OVER PHOAS TREES In Chapter 4 we show how we may use the solution given for tree traversal in Chapter 3 to define a type correct crossover operation on GADT-embedded HOAS trees. We also provide a proof of concept implementation in the Idris programming language.

CONCLUSION We summarise the main contributions of the thesis and discuss areas for further work.

2 BACKGROUND AND LITERATURE REVIEW

2.1 Dependent Types

2.1.1 Algebraic Data Types

We say that types are *inhabited* by their values. While there is not one universally agreed upon definition of an algebraic data type (ADT), the term *algebraic* usually refers to an algebra which forms based on the number of inhabitants of a type. We go a step further here and define algebraic data types in a manner enabling relatively straightforward proofs of inequality between different constructor tags. First we require a concept for a type that has no inhabitants, which we denote by $\perp$. If we consider types as sets of values, then the logical value associated with the $\perp$ type is $\emptyset$. We now give definition 1 for algebraic data types.

Definition 1 (A general definition of an algebraic data type)**.**

Let X denote a set of symbols (or "tags") of cardinally $n \geq 1$.
Let Y denote a set of typed values.
Then an algebraic data type $\mathcal{T}$ may be formed from the following:
$\mathcal{T} = \forall x.(x, y) | x \in X \land y \in Y$

We give the name *constructors*, sometimes *data constructors* or *value constructors* to individual elements of $\mathcal{T}$. As seen above each $t \in \mathcal{T}$ is an ordered pair with the first element containing a tag from X and the second element containing a typed value from Y. The $\forall$ quantification in the definition of $\mathcal{T}$ shows that we must pair one of every tag from X with some element of Y. Definition 2 shows us a concrete example for an algebraic data type representing Boolean values. The type has two values as shown.

Definition 2 (A definition of the **Bool** ADT)**.**

$$X = \{\text{True}, \text{False}\}$$
$$Y = \{\emptyset : \perp\}$$
$$\mathbb{B} = \{(\text{True}, \emptyset : \perp), (\text{False}, \emptyset : \perp)\}$$

Here we see that the type of Boolean values is a set of two symbols (True and False), where each symbol is associated with no additional information. By the definition of $\mathcal{T}$ we see that a specific tag may only appear once in an algebraic data type, regardless of what information it may be associated with. This allows us to prove by definition that False $\neq$ True for type $\mathbb{B}$.

Throughout this document we define data types through type rules. These are inference-like rules that can be read in a similar way. Informally a rule contains a number of judgements of the form $\Gamma \vdash \mu$. Judgements above the line are premises of the rule, with a single conclusion below the line. If all the premises hold, then the conclusion must hold. We may read the expression $\Gamma \vdash \mu$ as "If we may derive some expression μ from a global (typing) context Γ then the following (below the line) must hold". We split the rule into two parts, once for the definition of the type itself, and then a rule for every value that we may construct. We see in definition 3 that we may always derive the Bool type regardless of the existing typing context. We then see that we have two values which also may be derived regardless of context: True, which results in a value of Bool and False which also results in the same. Informally, the type Bool does not depend on any other types existing in the global context for its definition, and likewise the two values of Bool do not depend on any other types or values for their definitions.

Definition 3 (A definition of boolean values using typing rules)**.**

$$\frac{}{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}}$$

Values:

$$\frac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{False} : \mathsf{Bool}}$$

We can also define inductive algebraic data types. Consider the inductive definition of natural numbers. Here in definition 4 we have a base case Z and an inductive case S(k). Once again Nat does not require any types or values derivable from our context in order to be defined as a type. The base case Z also requires no types or values to exist in context Γ. However, the inductive case requires that we may derive an existing value k of type Nat.

Definition 4 (A definition of natural numbers using typing rules)**.**

$$\frac{}{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}}$$

Values:

$$\frac{}{\Gamma \vdash \mathsf{Z} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash \mathsf{k} : \mathsf{Nat}}{\Gamma \vdash \mathsf{S}(\mathsf{k}) : \mathsf{Nat}}$$

Algebraic data types carry a restriction with respect to induction. All inductive terms present in the definition of the type must be parameterised by the variables that exist in the definition of the *type* (not the definition of the values). Consider definition 5 of the type of singly-linked-lists. We cannot ask that the Cons constructor depends on a $\mathsf{xs} : \mathsf{List}\ \mathsf{B}$ as a premise when we have only introduced the variable A in the type. We also cannot specify that the Cons case requires a concretely typed list, for example: $\mathsf{xs} : \mathsf{List}\ \mathsf{Nat}$.

Definition 5 (The inductive singly-linked list type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{List}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Nil} : \mathsf{List}\ \mathsf{A}} \qquad \frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{List}\ \mathsf{A}}{\Gamma \vdash \mathsf{Cons}(\mathsf{x}, \mathsf{xs}) : \mathsf{List}\ \mathsf{A}}$$

Along with $\bot$, we also define our universe of types as Type. This universe forms an infinite tower of types, such that $\mathsf{Type} : \mathsf{Type}_1$ and $\mathsf{Type}_1 : \mathsf{Type}_2$ and so on. We give the type rules for common types used in this document in definitions 6, 7, and 8. It is the sum and product types (so named for the number of inhabitants) particularly that give rise to the concept of an algebra of types.

Definition 6 (The product type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{A} \times \mathsf{B} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{B}}{\Gamma \vdash (\mathsf{x}, \mathsf{y}) : \mathsf{A} \times \mathsf{B}}$$

Definition 7 (The sum type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{A} + \mathsf{B} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A}}{\Gamma \vdash \mathsf{Left}(\mathsf{x}) : \mathsf{A} + \mathsf{B}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{y} : \mathsf{B}}{\Gamma \vdash \mathsf{Right}(\mathsf{y}) : \mathsf{A} + \mathsf{B}}$$

Definition 8 (The function type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{A} \to \mathsf{B} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{y} : \mathsf{B}}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{A}.\, \mathsf{y} : \mathsf{A} \to \mathsf{B}}$$

2.1.2 Polymorphism

We give Wadler's definition of polymorphism as interpreted from Strachey (2000).

> "Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type. A typical example is overloaded multiplication: the same symbol may be used to denote multiplication of integers (as in 3×3) and multiplication of floating point values (as in 3.14×3.14).
>
> Parametric polymorphism occurs when a function is defined over a range of types, acting in the same manner for each type. A typical example is the **length** function, which acts in the same way on a list of integers and a list of floating point numbers." (Wadler and Blott, 1989, p. 1)

We denote an ad-hoc polymorphic constraint on a type with a side constraint on type rules. In function definitions we denote this ad-hoc polymorphic constraint through the use of the $\Rightarrow$ symbol. Consider examples 1 and 2 respectively.

Example 1 (The type rule for ad-hoc polymorphic addition)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash + : \mathsf{A} \to \mathsf{A} \to \mathsf{A}}\ \Gamma \vdash \mathsf{Num}\ \ \mathsf{A}$$

Example 2 (The function definition for ad-hoc polymorphic addition)**.**

$$(+) : \mathsf{Num}\ \mathsf{A} \Rightarrow \mathsf{A} \to \mathsf{A} \to \mathsf{A}$$
$$\mathsf{x{+}y} = \mathsf{x} + \mathsf{y}$$

2.1.3 Generalised Algebraic Data Types

Guarded recursive data types (Xi et al., 2003), first-class phantom types (Cheney and Hinze, 2003), dependent data types (Brady, 2017), inductive dependent data types (Pierce et al., 2010), and generalised algebraic data types (Peyton Jones et al., 2004). While there are some minor differences between these terms (some are specific to certain programming languages, and carry positivity restrictions), they all denote algebraic data types with the restriction on induction relaxed. In this document we will use the term most familiar to Haskell users - Generalised Algebraic Data Type (GADT), to refer to these structures.

If ADTs may be viewed as a set of values, then GADTs may be viewed as an indexed family of sets. A canonical example of a GADT is the Fin type. The type of finite sets (Bove and Dybjer, 2008). Fin is indexed by a natural number representing a k-ary number of elements strictly less than some upper bound n. Note the type index in definition 9, we now have a family of types indexed by a natural number *value*. So Fin 1 is an entirely separate type from Fin 2 and so on, for some k.

Definition 9 (*Fin*, the type of finite sets)**.**

$$\frac{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}}{\Gamma \vdash \mathsf{Fin\ Nat} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{n} : \mathsf{Nat}}{\Gamma \vdash \mathsf{FZ} : \mathsf{Fin\ S(n)}} \qquad \frac{\Gamma \vdash \mathsf{n} : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{k} : \mathsf{Fin\ n}}{\Gamma \vdash \mathsf{FS(k)} : \mathsf{Fin\ S(k)}}$$

2.1.4 Indices, Parameters and Higher Kinds

We saw in the definition of List that an ADT may have a type variable as a parameter. This is subject to the restriction that all inductive occurrences of List appearing as a premise in List values, are parameterised by at least one of the same variables appearing in the premise of the List type. We may say then that the type List is *higher-kinded* where kind is used in some languages (notably Haskell) to denote the type of types. We will denote higher kinded types where we are not concerned with their parameter(s) as a function between types as per example 3.

Example 3 (Higher kinded representation of the *List* type)**.**

$$\frac{\Gamma \vdash \mathsf{List}\ : \mathsf{Type} \to \mathsf{Type}}{}$$

GADTs may have both type parameters and indices. We distinguish between the two by whether or not they adhere to the inductive restrictions for ordinary ADTs. Consider the type of length-indexed vectors in definition 10, here the A index can be considered an ordinary type parameter as all inductive Vect terms appearing as a premise in the values are indexed by the same A. However the vector is *indexed* by the Nat type, as the Nat terms appearing in the premises of the values are different to the premise in the type.

Definition 10 (The type of length indexed vectors)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Nat} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{n} : \mathsf{Nat}}{\Gamma \vdash \mathsf{Vect}\ \mathsf{n}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Nat} : \mathsf{Type}}{\Gamma \vdash \mathsf{Nil} : \mathsf{Vect}\ \mathsf{Z}\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Nat} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{k} : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{Vect}\ \mathsf{k}\ \mathsf{A}}{\Gamma \vdash \mathsf{Cons}(\mathsf{x}, \mathsf{xs}) : \mathsf{Vect}\ \mathsf{S}(\mathsf{k})\ \mathsf{A}}$$

2.1.5 Π Types and Function Families

We define the concept of a function family as a function from some specific type, to our universe of types. This allows us to define a function where the co-domain changes depending on the specific *value* given from the domain. We illustrated this through example 4. We denote function families with capital first letters to distinguish from ordinary functions.

Example 4 (A function family resulting in natural number or string codomains)**.**

$$\begin{aligned}
&\mathsf{NatOrString} : \mathsf{Bool} \to \mathsf{Type}\\
&\mathsf{NatOrString}(\mathsf{True}) = \mathsf{Nat}\\
&\mathsf{NatOrString}(\mathsf{False}) = \mathsf{String}
\end{aligned}$$

This in-effect is a type calculation function, where the return type is calculated from the value of the input. We define the Π in type definition 11 and give an example of its use on our function family $\mathsf{NatOrString}$ in example 5.

Definition 11 (The Π type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} \to \mathsf{Type}\ : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{A} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A}}{\Gamma \vdash \Pi\, \mathsf{x} : \mathsf{A}.\, \mathsf{B} : \mathsf{Type}}$$

Example 5 (A Π type mapping bools to $NatOrString$)**.**

$$\begin{aligned}
&\mathsf{natOrString} : \Pi\, \mathsf{x} : \mathsf{Bool}.\, \mathsf{NatOrString}\\
&\mathsf{natOrString}(\mathsf{True}) = 2\\
&\mathsf{natOrString}(\mathsf{False}) = \text{“Test”}
\end{aligned}$$

2.1.6 Σ Types

The Σ type denotes the dependent pair, where the type of the second element depends on the value of the first element. We give the type definition in 12 and examples in 6.

Definition 12 (The Σ type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma, \mathsf{x} : \mathsf{A} \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \Sigma\, \mathsf{A} : \mathsf{B}.\, \mathsf{a} : \mathsf{A}}$$

Example 6 (Various Σ of *List* values)**.**

$$\mathsf{f} : \Sigma\, \mathsf{x} : \mathsf{Type}\, .\, \mathsf{List}\ \mathsf{x}$$
$$\mathsf{f} = (\mathsf{Nat}, \mathsf{Cons}(1, \mathsf{Cons}(2, \mathsf{Cons}(3, \mathsf{Nil}))))$$

$$\mathsf{g} : \Sigma\, \mathsf{x} : \mathsf{Type}\, .\, \mathsf{List}\ \mathsf{x}$$
$$\mathsf{g} = (\mathsf{Bool}, \mathsf{Cons}(\mathsf{True}, \mathsf{Cons}(\mathsf{False}, \mathsf{Cons}(\mathsf{True}, \mathsf{Nil}))))$$

$$\mathsf{z} : \Sigma\, \mathsf{x} : \mathsf{Type}\, .\, \mathsf{List}\ \mathsf{x}$$
$$\mathsf{z} = (\mathsf{Type}\ , \mathsf{Cons}(\mathsf{Bool}, \mathsf{Cons}(\mathsf{Nat}, \mathsf{Cons}(\mathsf{String}, \mathsf{Nil}))))$$

2.1.7 Propositions as Types, Proofs as Programs

Wadler (2015) gives a well-written introduction to the notion of propositions as types. In short there exists a correspondence between logic and programming such that types in dependently typed programming languages may be interpreted as propositions in first order logic according to the following table.

Logic Term	Logic Symbol	Type
Implication	$p \Rightarrow q$	$\mathsf{p} \rightarrow \mathsf{q}$
Conjunction	$p \wedge q$	$\mathsf{p} \times \mathsf{q}$
Disjunction	$p \vee q$	$\mathsf{p} + \mathsf{q}$
Negation	$\neg\, p$	$\mathsf{p} \rightarrow \bot$
IFF/Eq	$p \equiv q,\ p \Leftrightarrow q$	$(\mathsf{q} \rightarrow \mathsf{p}) \times (\mathsf{p} \rightarrow \mathsf{q})$
Universal	$\forall\ x.\ P\ x$	$\Pi\, \mathsf{x} : \mathsf{Type}\, .\, \mathsf{P}$
Existential	$\exists\ x.\ P\ x$	$\Sigma\, \mathsf{x} : \mathsf{Type}\, .\, \mathsf{P}$

Types and terms share the same language in dependently typed programming languages, all terms may also be types. This includes functions. In order to avoid non-termination during the type checking process, these languages often introduce a totality and positivity clause. A function is total if it is defined on all inputs and is guaranteed to terminate in some finite time. Additionally any GADTS must be *strictly positive*.

That is the constructor values must not require a value of the type under definition (an inductive step) to the right of a function arrow.

2.2 Higher Order Abstract Syntax Embedding

(Pfenning and Elliott, 1988) give us *higher order abstract syntax* (HOAS), a technique for representing programs in an object language (the language we construct) where variable binding and reference management are taken care of by the meta language (the language in which we program). HOAS is well known for being attractive and simple to use, but awkward to manipulate (Washburn and Weirich, 2003; Meijer et al., 1991; Atkey et al., 2009). This is particularly true when we embed our HOAS object language within a generalised algebraic data type (a so-called deep embedding). Consider example 7, the simply typed lambda calculus represented as GADT-embedded HOAS.

Example 7 (A HOAS embedding of STLC in Haskell)**.**

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
import Data.Kind (Type)
data STLC :: Type -> Type where
  Var :: a -> STLC a
  Abs :: (STLC a -> STLC b) -> STLC (a -> b)
  App :: STLC (a -> b) -> STLC a -> STLC b
```

A HOAS embedding ensures that our object language is typed checked by our meta language. Ill typed terms simply will not compile. This, coupled with the notion of propositions as types, gives us a powerful ability to verify our programs with respect to functional correctness.

2.3 Genetic Programming

Genetic programming (GP) is a class of evolutionary algorithm from the broader field of evolutionary computation. This field examines meta-heuristic optimisation algorithms that have been inspired by ideas from biological evolution. Pioneered by Koza (Koza, 1992), genetic programming adapts the techniques used to simulate evolution in genetic algorithms to work on representations of executable functions. A genetic program, like other evolutionary algorithms, maintains a pool of candidate solutions (the *population*). Each member of this pool is evaluated and scored against a user-provided function (the *fitness function*) measuring the distance (in solution space) a particular candidate is from an ideal solution. Once each candidate has been evaluated, high-performing candidates are chosen to produce new offspring by exchanging feature information (called the *crossover* operation). The new offspring also have a small chance that a feature is randomly altered (the *mutation* operation). Ideally each successive generation should consist of individual candidate solutions that, when evaluated, give a closer, or more accurate solution to the problem than the previous generation. This process is repeated until a solution is found, or until a user-specified termination criteria is met.

Function Representation

Functions in genetic programming are commonly represented in parse tree form. This allowed for easy parsing and execution compared to the binary string representation commonly found in genetic algorithms (Koza, 1992, p. 63-77). An example of a parse tree can be found in figure 1.

$$f(x, y) = \neg x \vee (x \wedge y)$$

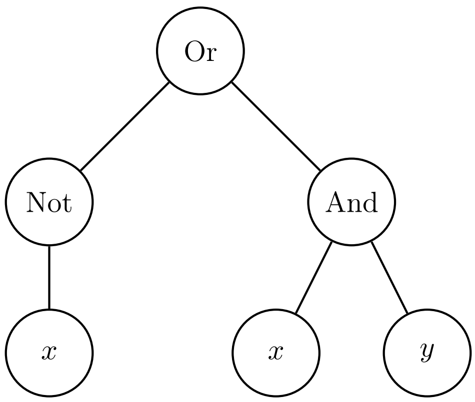


Figure 1: A parse tree

2.3.1 The Crossover Operation

The crossover operator takes two high performing trees and exchanges their sub-trees at a randomly determined point (Koza, 1992, p. 101-105) (Engelbrecht, 2007, p. 180). This facilitates the exchange of features between solution candidates. Crossover is the primary operator for most genetic program implementations. Figure 2 shows two trees $f(x, y) = \neg x \vee (x \vee y)$ and $g(x, y) = (x \vee \neg y) \vee (\neg y \wedge \neg x)$ (where red marks the sub-trees for which the roots were chosen at a random position for each function). The crossover function when applied to f and g yields a pair of new trees f' and g'.

2.4 Functional Genetic Programs

The literature on functional genetic programming is sparse, perhaps understandably so given the union of two distinct fields of computer science. It also has a disproportionate preoccupation with type systems. This is also understandable when popular pure functional languages such as Haskell erase types after compile time, leaving meta-programmers to come up with creative solutions to allow run-time reasoning over types.

Yu and Clack (1998) give us *PolyGP* a polymorphic genetic programming system in Haskell. This work is notable for being the first modern pure functional genetic programming system in the literature, along with the first to implement a manual type system to perform strongly typed genetic programming (only well-typed candidates are accepted into the population (Montana, 1995)). Yu and Clack give their abstract type syntax as per example 8.

Example 8 (Abstract type syntax for PolyGP)**.**

```
γ :: τ            -- built-in type
  | ν              -- type variable
  | σ1 → σ2      -- function type
  | [σ1]           -- lists parameterised by σ1
  | (σ1 → σ2) -- bracketed function type

τ :: Int | String | Bool | Generic_i
ν :: Dummy_i | Temporary_i
```

The interesting cases here are clearly the `Generic` in τ, and the entirety of ν. Yu and clack define a generic as a type variable which forms part of the signature of the desired function to be learned. If we wish to learn the function `length :: [a] -> Int` we represent it as `length :: [Generic₁] -> Int`. This is distinct from occurrences of polymorphism inside the function and terminal set (the sets of functions and values that our genetic program has to work with). Here the type system will instantiate `Dummy` variables to concrete types it may infer from the return type of a function. If a concrete type cannot be inferred the `Dummy` will be instantiated as a `Temporary` type (for delayed binding or application). Yu and Clack show here how we may represent parametric polymorphism in PolyGP, but give no mention of ad-hoc polymorphism. Yu and Clack also give no proofs of preservation or progress (Pierce and Benjamin, 2002, p. 95) for their type system.

(Diehl, 2011) shows a dependently typed crossover operation for a strongly typed stack-based genetic program (Perkis, 1994). Diehl's work is notable for being the first formally verified (with respect to functional correctness) genetic program via dependent types. Diehl first defines a language for the stack-based GP that we give in example 9.

Example 9 (The object language of Dhiel's verified stack-based GP).

```
data Word : Set where
  true not and : Word
```

Diehl then defines `Term` as an inductive structure indexed by two natural numbers, one for the length of the stack being consumed, and the other for the length of the stack being returned. We also note Diehl's use of the Σ type to represent operations where the exact value of one of the natural number indices cannot be known, including a type safe crossover that maintains the appropriate stack size invariants. Diehl's work is type safe and verified, however the object language is trivial and practically untyped. We also note that crossover on a stack of untyped terms is significantly closer to the definition of a genetic *algorithm* (Holland et al., 1992), rather than Koza's vision of a genetic program.

Our goal then is to take the most desirable traits from each of these approaches. We would like the expressiveness and parametric polymorphism shown by Yu and Clack, and the type safe crossover and verification properties that Diehl shows via dependent types.

We consider one of the most important properties of genetic programs over other forms of supervised machine learning, to be that the learned model is *human readable*. We note here other related examples of functional genetic programs that we exclude from consideration due to the impracticality of their object language, meta language, or both. (Briggs and O'Neill, 2008) shows how we may use SKI combinators for strongly typed genetic programming to sidestep some of the issues involved when representing variables. (Binard and Felty, 2007) give us a specification for a genetic program that learns types as well as values in the system F. However Binard and Felty represent types and terms in church-encoding and give an implementation in the C language, so there is a question of whether their abstraction-based genetic programming system is a functional genetic program. (Křen and Neruda, 2014) are concerned with parameterising the genetic program over different search strategies and show improvements with an object language of simply typed lambda calculus.

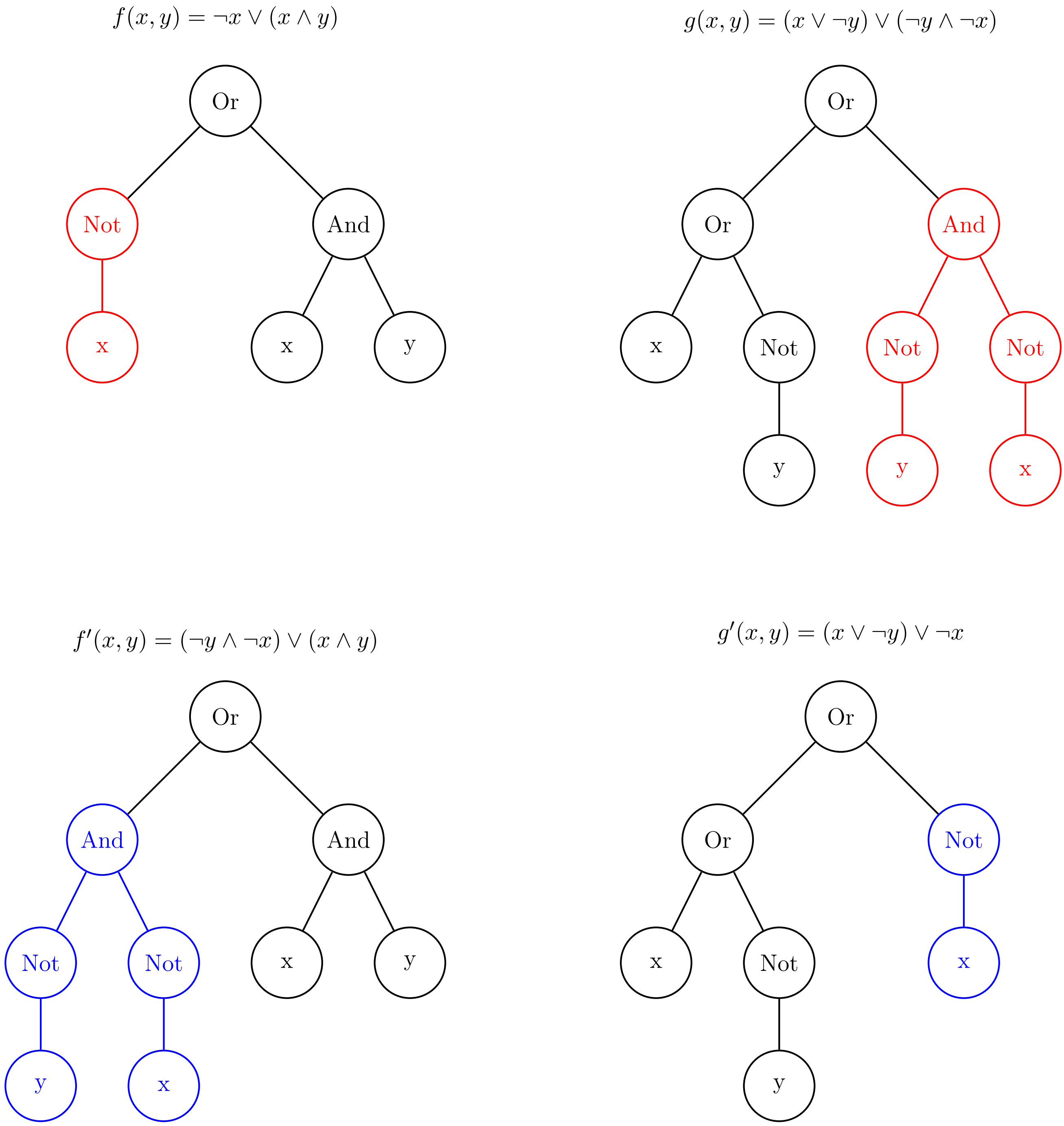


Figure 2: Crossover on expression trees.

3 TRAVERSAL OVER GADT-EMBEDDED HOAS TREES

We may ultimately view a crossover operation as a substitution. We would like to substitute some particular expression in our syntax tree, with some other particular expression. In order to perform a crossover operation on two GADT embedded higher order abstract syntax trees we first need to examine ways in which we may traverse the tree structure. We also must preserve the properties of HOAS that meet our use case requirements, namely offloading variable binding and type-checking to our meta language. This chapter examines the traditional techniques functional programmers have to traverse ordinary inductive algebraic data types and why these techniques are not suited to a crossover operation on generalised algebraic data types. We then examine techniques for tree manipulation of ordinary ADTs and present a solution for traversal and expression substitution for GADT embedded trees.

3.1 Folds and Maps - Traversal Over Adts

Before we explore ways to traverse well-typed GADT-embedded structures, it will be beneficial to have an understanding of common methods of traversing inductive algebraic data types. Fold and map are well known abstractions that traverse and transform these types, and are frequently introduced in beginning texts on functional programming (Bird, 2014; Allen and Moronuki, 2017; Lipovaca, 2011, for example). We cover them again here as a refresher and to show that we cannot implement these directly over generalised algebraic data type embedded higher order abstract syntax trees.

Definition 13 (Typing rules for $foldr$ and $foldl$)**.**

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{foldr} : (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B} \to \mathsf{T}\ \mathsf{A} \to \mathsf{B}}\ \Gamma \vdash \mathsf{Foldable}\ \mathsf{T}$$

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{foldl} : (\mathsf{B} \to \mathsf{A} \to \mathsf{B}) \to \mathsf{B} \to \mathsf{T}\ \mathsf{A} \to \mathsf{B}}\ \Gamma \vdash \mathsf{Foldable}\ \mathsf{T}$$

The fold abstraction (often called reduce in imperative languages) traverses the inductive data type and yields up elements of underlying structure, applying them to the given function and accumulating value. This usually causes a reduction in the structure (hence the common name in imperative programming). The fold abstraction is expressed in modern pure functional languages as the pair of the foldr and foldl functions functions (see definition 13). These two functions differ in the order in which they

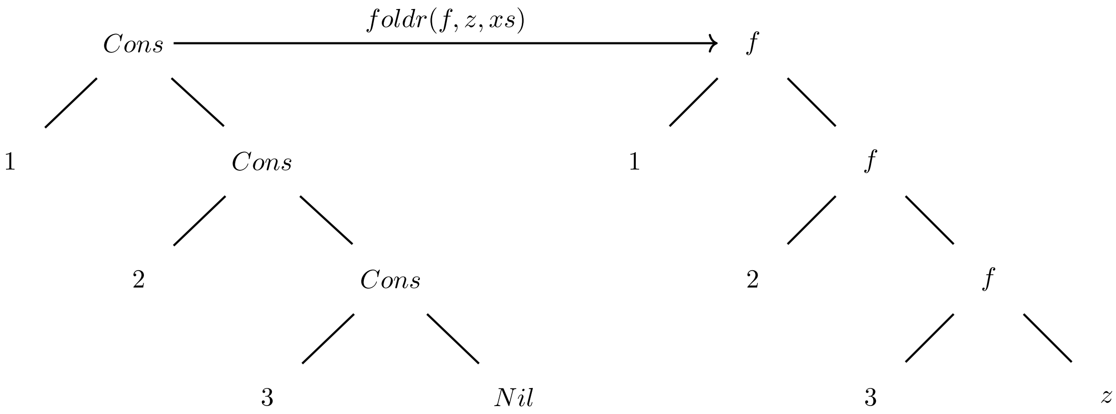


Figure 1: A graphical representation of $foldr$ on lists.

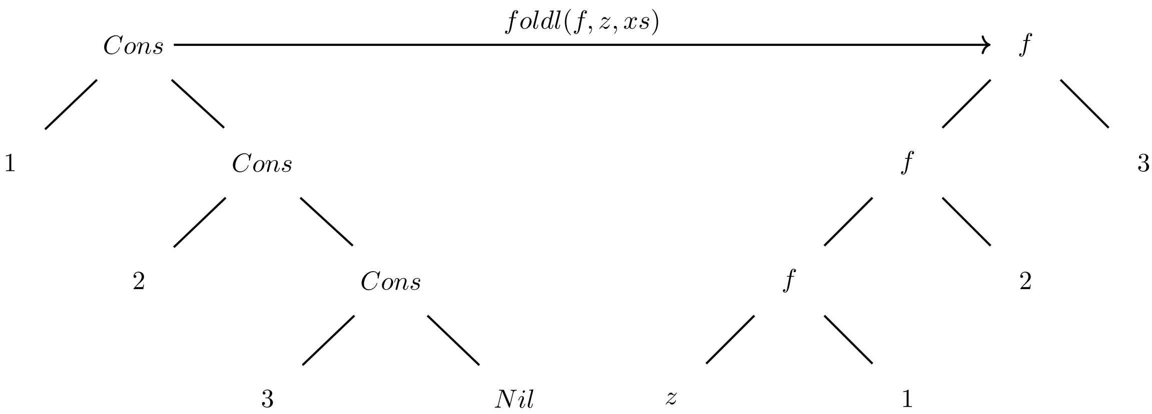


Figure 2: A graphical representation of $foldl$ on lists.

accumulate the intermediate values of the function, as we illustrate in figures 1 and 2. In addition to the well-known cons-list structure, we may also fold over abstract syntax trees and accumulate the underlying values (Meijer and Jeuring, 1995). If a structure is fold-able it is customary to declare it a member of a Foldable interface in languages that support ad-hoc polymorphism (recall Chapter 2 (2.1.3)). Usually a definition of foldr is enough to declare membership of this interface as we may define foldl solely in terms of foldr.

Definition 14 (The typing rule for *map*)**.**

$$\frac{\Gamma \vdash \mathsf{F} : \mathsf{Type} \rightarrow \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \mathsf{map} : (\mathsf{A} \rightarrow \mathsf{B}) \rightarrow \mathsf{F}\ \mathsf{A} \rightarrow \mathsf{F}\ \mathsf{B}}\ \Gamma \vdash \mathsf{Functor}\ \mathsf{F}$$

A simpler abstraction than fold, map (see definition 14) transforms every underlying element in a structure, by applying a given function to that element. Figure 3 shows this graphically. Folds and maps are used to traverse and transform a wide variety of data types and are common abstractions that functional programmers reach for. Johann and Ghani (2008) show us that semantically GADTs are not members of the functor interface or type class. Type indexes are not necessarily polymorphic in the same way that type parameters are, this is precisely the restriction that GADTs relax. Instead GADTs rely on pattern matching each value to refine the appropriate type index. Consider example 10, a GADT provided by Johann and Ghani to represent terms:

Example 10 (A GADT to represent terms from (Johann and Ghani, 2008))**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Term}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A}}{\Gamma \vdash \mathsf{Const}(\mathsf{x}) : \mathsf{Term}\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{B}}{\Gamma \vdash \mathsf{Pair}(\mathsf{x}, \mathsf{y}) : \mathsf{Term}\ (\mathsf{A} \times \mathsf{B})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{A} : \mathsf{Type} \\ \Gamma \vdash \mathsf{B} : \mathsf{Type}\end{array} \qquad \Gamma \vdash \mathsf{A} \to \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{f} : \mathsf{Term}\ \mathsf{A} \to \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Term}\ \mathsf{A}}{\Gamma \vdash \mathsf{App}(\mathsf{f}, \mathsf{x}) : \mathsf{Term}\ \mathsf{B}}$$

Johann and Ghani consider the example of trying to map over the values of $\mathsf{Pair}(\mathsf{x}, \mathsf{y})$. We would like to express something of the form $\mathsf{map}(\mathsf{f}, \mathsf{Pair}(\mathsf{x}, \mathsf{y})) = \mathsf{Pair}(\mathsf{u}, \mathsf{v})$ for some u and v. There is some semantic ambiguity here, we promised in the type of map that $\mathsf{f} : \mathsf{A} \to \mathsf{B}$, so in the case of pairs we have $\mathsf{f} : \mathsf{A} \times \mathsf{B} \to \mathsf{U} \times \mathsf{V}$ but as map would iterate through the structure of the Term, the types of the sub expressions change. A function that maps pairs will fail to type check when given the term $\mathsf{Const}(2)$ as an argument. Yet a pair of such terms, say $\mathsf{Pair}(\mathsf{Const}(2), \mathsf{Const}(2))$, is a completely valid value of type Term. As the type indices of the sub expressions change we cannot iterate through them with only one particular function without further restrictions on the type of the sub-expressions (for example, forcing them all to be ad-hoc polymorphic in the same manner).

Johann and Ghani continue, showing how we may derive a higher order functors and folds to structurally traverse basic GADTs. Yakushev et al. (2009) also show how we may use recursion schemes and fixed points to traverse and structurally evaluate [1] generalised algebraic data types. We leave the details of this approach to the interested reader however, as there is still semantic ambiguity around substitution of sub-expressions *by*

1 http://www.timphilipwilliams.com/posts/2013-01-16-fixing-gadts.html Accessed on 07/10/19.

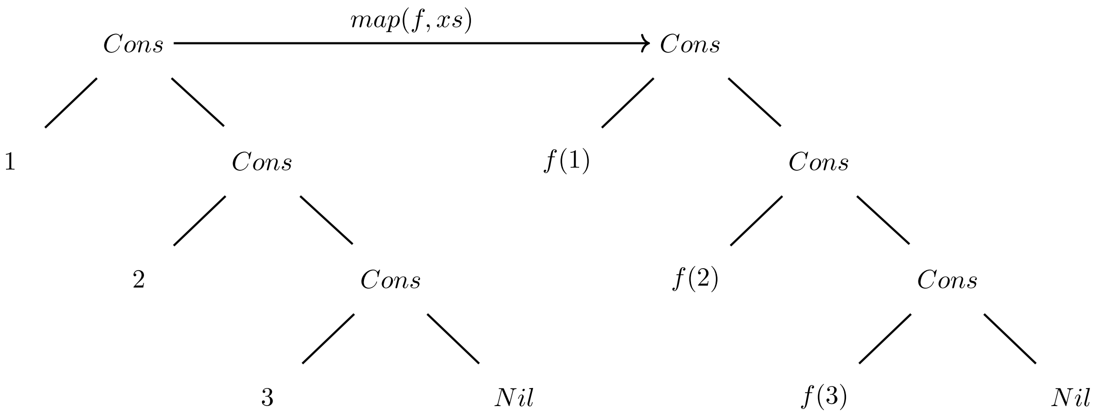


Figure 3: A graphical representation of *map* on lists.

position. In order for a fold or map, whether first-order or higher-order, to transform an inductive type, it may only examine the structure of its argument one node at a time. This suggests we require a structure that handles positions.

3.2 The Zipper

Huet (1997) gives us the Zipper data structure as a means to walk down trees in a non-destructive manner. The zipper is usually represented as the pairing of a focused sub-expression, and a type capturing one-hole contexts such that the original tree may be recovered by substituting the focus into the hole. We present a similar example to Lipovaca (2011, chapter 14) by way of introduction. We first start be defining a parameterised algebraic data type to represent binary trees in definition 15.

Definition 15 (The binary tree type)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Tree}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Empty} : \mathsf{Tree}\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Tree}\ \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{l} : \mathsf{Tree}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{r} : \mathsf{Tree}\ \mathsf{A}}{\Gamma \vdash \mathsf{Node}(\mathsf{x}, \mathsf{l}, \mathsf{r}) : \mathsf{Tree}\ \mathsf{A}}$$

Here we see that a tree is either empty, or contains a value and two sub-trees (although these sub-trees may in turn be empty). We define the type of one-hole contexts as a product of the parent value p and a sibling tree s as we see in definition 16.

Definition 16 (The type of one-hole contexts for binary trees)**.** Types:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Context\ A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Tree\ A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{p} : \mathsf{A} \qquad \Gamma \vdash \mathsf{s} : \mathsf{Tree\ A}}{\Gamma \vdash \mathsf{LeftContext}(\mathsf{p}, \mathsf{s}) : \mathsf{Context\ A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Tree\ A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{p} : \mathsf{A} \qquad \Gamma \vdash \mathsf{s} : \mathsf{Tree\ A}}{\Gamma \vdash \mathsf{RightContext}(\mathsf{p}, \mathsf{s}) : \mathsf{Context\ A}}$$

Once we have the context type we can represent a zipper as the product of some focus value, and a list of one-hole contexts that represent the path taken through the tree so far. Each step (definition 17) down the tree causes the list to grow, tracking the path taken from the root. Each step up the tree shrinks the list and rebuilds the new focus to include the value of the previous focus as a sub tree. This ensures that any modifications made to the focus value will propagate back up the tree.

Definition 17 (Functions to walk along a binary tree)**.**

$$\begin{aligned}
&\mathsf{left} : \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \rightarrow \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \\
&\mathsf{left}(\mathsf{Empty}, \mathsf{ctx}) = (\mathsf{Empty}, \mathsf{ctx}) \\
&\mathsf{left}(\mathsf{Node}(\mathsf{x}, \mathsf{l}, \mathsf{r}), \mathsf{ctx}) = (\mathsf{l}, \mathsf{Cons}(\mathsf{LeftContext}(\mathsf{x}, \mathsf{r}), \mathsf{ctx}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{right} : \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \rightarrow \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \\
&\mathsf{right}(\mathsf{Empty}, \mathsf{ctx}) = (\mathsf{Empty}, \mathsf{ctx}) \\
&\mathsf{right}(\mathsf{Node}(\mathsf{x}, \mathsf{l}, \mathsf{r}), \mathsf{ctx}) = (\mathsf{r}, \mathsf{Cons}(\mathsf{RightContext}(\mathsf{x}, \mathsf{l}), \mathsf{ctx}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{up} : \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \rightarrow \mathsf{Tree\ A} \times \mathsf{List\ (Context\ A)} \\
&\mathsf{up}(\mathsf{x}, \mathsf{Cons}(\mathsf{LeftContext}(\mathsf{p}, \mathsf{r}), \mathsf{ctx})) = (\mathsf{Node}(\mathsf{p}, \mathsf{x}, \mathsf{r}), \mathsf{ctx}) \\
&\mathsf{up}(\mathsf{x}, \mathsf{Cons}(\mathsf{RightContext}(\mathsf{p}, \mathsf{l}), \mathsf{ctx})) = (\mathsf{Node}(\mathsf{p}, \mathsf{x}, \mathsf{l}), \mathsf{ctx})
\end{aligned}$$

Given that the up function will substitute the focus back into the tree, we can perform a crossover operation by taking a pair of trees and exchanging their foci as per definition 18. Subsequent applications of the function up will then rebuild each tree from which we may then project out the underlying focus from the zipper.

Definition 18 (A crossover operation (arbitrarily called f) for zippers on binary trees)**.**

Let $\mathsf{Zipper\ A}$ denote $\mathsf{Tree\ A} \times \mathsf{List}\ (\mathsf{Context\ A})$.

$$\mathsf{f} : \mathsf{Zipper\ A} \times \mathsf{Zipper\ A} \to \mathsf{Zipper\ A} \times \mathsf{Zipper\ A}$$
$$\mathsf{f}((\mathsf{focus}_1, \mathsf{ctx}_1), (\mathsf{focus}_2, \mathsf{ctx}_2)) = ((\mathsf{focus}_2, \mathsf{ctx}_1), (\mathsf{focus}_1, \mathsf{ctx}_2))$$

The zipper then is a structure to handle operations on nodes of tree-shaped inductive data types when we are concerned with a nodes position. The zipper may extract and substitute the node regardless of its particular structure, and allows manipulation of a particular focus in constant time.

$$
\begin{array}{l}
\texttt{data D : } I \to \texttt{Set where} \\
\quad \texttt{K : } (j : J) \to (e : E) \to \texttt{Q}[j,e] \to \texttt{D}(d_1[j,e]) \to \ldots \to \texttt{D}(d_k[j,e]) \to \texttt{D}(c[j])
\end{array}
$$

$$
\begin{array}{l}
\texttt{data Ctx : } I \to \texttt{Set where} \\
\quad \texttt{[] : } \{m : I\} \to \texttt{Ctx}(m) \\
\left\{ \begin{array}{l}
\texttt{K}_i \texttt{ : } \{j : J\} \to \{e : E\} \to \texttt{Q}[j,e] \to \\
\quad \underbrace{\texttt{D}(d_1[j,e]) \to \ldots \to \texttt{D}(d_k[j,e])}_{\text{with only } \texttt{D}(d_i[j,e]) \text{ missing}} \to \texttt{Ctx}(c[j]) \to \texttt{Ctx}(d_i[j,e])
\end{array} \right\}_{i \in |k|}
\end{array}
$$

$$
\begin{array}{l}
\texttt{data Zipper : } I \to \texttt{Set where} \\
\quad \texttt{\_}\triangleright\texttt{\_ : } \{m : I\} \to \texttt{D}(m) \to \texttt{Ctx}(m) \to \texttt{Zipper}(m) \\
\quad \texttt{failure : } \{m : I\} \to \texttt{Zipper}(m)
\end{array}
$$

$$
\begin{array}{l}
\texttt{down}_i \texttt{ : } \{j : J\} \to \{e : E\} \to \texttt{Zipper}(c[j]) \to \texttt{Zipper}(d_i[j,e]) \\
\texttt{down}_i(\texttt{K q a}_1 \ldots \texttt{a}_{i-1}\ \texttt{a}_i\ \texttt{a}_{i+1} \ldots \texttt{a}_k \triangleright C) = (\texttt{a}_i \triangleright \texttt{K}_i\ \texttt{q a}_1 \ldots \texttt{a}_{i-1}\ \texttt{a}_{i+1} \ldots \texttt{a}_k\ C)
\end{array}
$$

$$
\begin{array}{l}
\texttt{upfrom}_i \texttt{ : } \{j : J\} \to \{e : E\} \to \texttt{Zipper}(d_i[j,e]) \to \texttt{Zipper}(c[j]) \\
\texttt{upfrom}_i(\texttt{a}_i \triangleright \texttt{K}_i\ \texttt{q a}_1 \ldots \texttt{a}_{i-1}\ \texttt{a}_{i+1} \ldots \texttt{a}_k\ C) = (\texttt{K q a}_1 \ldots \texttt{a}_{i-1}\ \texttt{a}_i\ \texttt{a}_{i+1} \ldots \texttt{a}_k \triangleright C)
\end{array}
$$

Figure 4: A zipper over a simple inductive family D (Hamana and Fiore, 2011).

3.3 A Type Dependent Zipper

(Hamana and Fiore, 2011) give us a general specification for a zipper over simple inductive families in their category-theoretic treatment of generalised algebraic data types and inductive families. They show this in an Agda-like syntax (Norell, 2008) as in figure 4, subject to the following definitions:

- D denotes a simple inductive family.
- Q denotes a constant (with respect to the type under definition) type, optionally indexed by values from J and E.
- K denotes some value constructor tag of D.
- $d_k[j,e]$ represents up-to k values of D optionally indexed by values of types J and E.
- $c[j]$ represents some number of values from J.
- i denotes the ith hole for the context. If K is a binary value constructor then $i = 2$, where $i = 1$ may denote the left inductive D value, and $i = 2$ may denote the right inductive D value.

The specification given in figure 4 does not hold when I = Set (the type of types in Agda). Consider this zipper applied to our Term GADT (from our earlier example 10), as we show in example 11.

Example 11 (A Hamana and Fiore Context type on the Term GADT shown by Johann and Ghani)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Ctx_{hf}}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash [\ ]_{\mathsf{hf}} : \mathsf{Ctx_{hf}}\ \mathsf{A}}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash \mathsf{A}, \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{A} \times \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{Term}\ \mathsf{B} : \mathsf{Type} \\ \Gamma \vdash \mathsf{Ctx_{hf}}\ (\mathsf{A} \times \mathsf{B}) : \mathsf{Type} & \Gamma \vdash \mathsf{r} : \mathsf{Term}\ \mathsf{B} & \Gamma \vdash \mathsf{c} : \mathsf{Ctx_{hf}}\ (\mathsf{A} \times \mathsf{B}) \end{array}}{\Gamma \vdash \mathsf{PairLeft_{hf}}(\mathsf{r}, \mathsf{c}) : \mathsf{Ctx_{hf}}\ \mathsf{A}}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash \mathsf{A}, \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{A} \times \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{Term}\ \mathsf{A} : \mathsf{Type} \\ \Gamma \vdash \mathsf{Ctx_{hf}}\ (\mathsf{A} \times \mathsf{B}) : \mathsf{Type} & \Gamma \vdash \mathsf{l} : \mathsf{Term}\ \mathsf{A} & \Gamma \vdash \mathsf{c} : \mathsf{Ctx_{hf}}\ (\mathsf{A} \times \mathsf{B}) \end{array}}{\Gamma \vdash \mathsf{PairRight_{hf}}(\mathsf{l}, \mathsf{c}) : \mathsf{Ctx_{hf}}\ \mathsf{B}}$$

$$\frac{\begin{array}{cccc} \Gamma \vdash \mathsf{A}, \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{A} \to \mathsf{B} : \mathsf{Type} & & \\ \Gamma \vdash \mathsf{Term}\ \mathsf{A} : \mathsf{Type} & \Gamma \vdash \mathsf{Ctx_{hf}}\ \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{r} : \mathsf{Term}\ \mathsf{A} & \Gamma \vdash \mathsf{c} : \mathsf{Ctx_{hf}}\ \mathsf{B} \end{array}}{\Gamma \vdash \mathsf{AppLeft_{hf}}(\mathsf{r}, \mathsf{c}) : \mathsf{Ctx_{hf}}\ (\mathsf{A} \to \mathsf{B})}$$

$$\frac{\begin{array}{cccc} \Gamma \vdash \mathsf{A}, \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{A} \to \mathsf{B} : \mathsf{Type} & & \\ \Gamma \vdash \mathsf{Term}\ \mathsf{A} : \mathsf{Type} & \Gamma \vdash \mathsf{Ctx_{hf}}\ \mathsf{B} : \mathsf{Type} & \Gamma \vdash \mathsf{r} : \mathsf{Term}\ \mathsf{A} \to \mathsf{B} & \Gamma \vdash \mathsf{c} : \mathsf{Ctx_{hf}}\ \mathsf{B} \end{array}}{\Gamma \vdash \mathsf{AppRight_{hf}}(\mathsf{l}, \mathsf{c}) : \mathsf{Ctx_{hf}}\ \mathsf{A}}$$

Definition 19 (A Hamana and Fiore zipper on $Term$)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Zipper_{hf}}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\begin{array}{cccc} \Gamma \vdash \mathsf{A} : \mathsf{Type} & & & \\ \Gamma \vdash \mathsf{Term}\ \mathsf{A} : \mathsf{Type} & \Gamma \vdash \mathsf{Ctx_{hf}}\ \mathsf{A} : \mathsf{Type} & \Gamma \vdash \mathsf{e} : \mathsf{Term}\ \mathsf{A} & \Gamma \vdash \mathsf{c} : \mathsf{Ctx}_{\eta}\ \mathsf{A} \end{array}}{\Gamma \vdash \mathsf{Zip_{hf}}(\mathsf{e}, \mathsf{c}) : \mathsf{Zipper_{hf}}\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Fail_{hf}} : \mathsf{Zipper_{hf}}\ \mathsf{A}}$$

We must now define a $\mathtt{down}_i$ for each non-root value in our $\mathsf{Ctx_{hf}}$ type. To illustrate why the specification given in figure 4 does not hold, consider counterexample

12. Common dependent type systems (e.g those used in Agda, Idris and Coq) cannot unify the A type used in the index of the return type $\mathsf{Zipper_{hf}}\ (\mathsf{A} \to \mathsf{B})$ with the A used to construct the input parameter in $\mathsf{App}(\mathsf{f}, \mathsf{x})$. In order to correctly represent this we would need to return a Σ type to capture the fact that we do not know precisely what the index of the return zipper may be. The other drawback of Hamana and Fiore's specification for the zipper over inductive families is that we must change what function we call to traverse the zipper depending on the structure of the current node. We avoided pursuing higher order functors and folds for similar reasons, we would like to express "walking left" or "walking right" down a gadt-embedded syntax tree regardless of the current focus structure.

Example 12 (Traversal function that will not type check for a $Zipper_{hf}$ on the $Term$ type)**.**

$$\begin{aligned}
&\mathsf{down_{hf-AppLeft}} : \mathsf{Zipper_{hf}}\ \mathsf{B} \to \mathsf{Zipper_{hf}}\ (\mathsf{A} \to \mathsf{B})\\
&\mathsf{down_{hf-AppLeft}}(\mathsf{Zip_{hf}}(\mathsf{App}(\mathsf{f}, \mathsf{x}), \mathsf{c})) = \mathsf{Zip_{hf}}(\mathsf{f}, \mathsf{AppLeft_{hf}}(\mathsf{x}, \mathsf{c}))\\
&\mathsf{down_{hf-AppLeft}}(\mathsf{Zip_{hf}}(\mathsf{e}, \mathsf{c})) = \mathsf{Fail_{hf}}\\
&\mathsf{down_{hf-AppLeft}}(\mathsf{Fail_{hf}}) = \mathsf{Fail_{hf}}
\end{aligned}$$

We will then adapt Hanama and Fiore's description of a dependently typed zipper to allow us to traverse GADT-embedded higher order abstract syntax trees. We first start by defining expressions in a small language ζ. This language is just complex enough to force us to handle ad-hoc polymorphism, along with sub-expressions of different type to a parent expression. As we are using a deep embedding for ζ, the meta language (the dependently typed language within which we chose to implement the zipper) will handle any type checking and variable binding (see Chapter 2 (2.1.3), and Chapter 2 (2.2)).

As we see in definition 20 we index an expression in ζ by any type that we may derive from a global typing context Γ. We also see that there are only three expressions in this language:

- Lit_ζ allows us to embed a value from our meta language into ζ and return an Expr_ζ indexed by the type of this value.
- Add_ζ allows us to embed two values of some ad-hoc polymorphic type $\Gamma \vdash \mathsf{Num}\ \mathsf{A}$ which requires an implementation of at least addition for type A. We then return an Expr_ζ indexed by the same ad-hoc polymorphic type.
- Const_ζ allows us to embed two values, the first of a parametrically polymorphic type A, the second of a (possibly identical) parametrically polymorphic type B. We then return an Expr_ζ indexed by the first type A.

Definition 20 (A small embedded language with the potential for differing type indices in sub expressions along with values of ad-hoc polymorphic types)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Expr}_\zeta\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{a} : \mathsf{A}}{\Gamma \vdash \mathsf{Lit}_\zeta(\mathsf{a}) : \mathsf{Expr}_\zeta\ \mathsf{A}}\ (\Gamma \vdash \mathsf{Show}\ \mathsf{A})$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{A}}{\Gamma \vdash \mathsf{Add}_\zeta(\mathsf{x}, \mathsf{y}) : \mathsf{Expr}_\zeta\ \mathsf{A}}\ (\Gamma \vdash \mathsf{Num}\ \mathsf{A})$$

$$\frac{\Gamma \vdash \mathsf{A}, \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{B}}{\Gamma \vdash \mathsf{Const}_\zeta(\mathsf{x}, \mathsf{y}) : \mathsf{Expr}_\zeta\ \mathsf{A}}$$

One advantage of a generalised algebraic data type embedding is the ease with which we may define an evaluation function for our language. We simply map each term in our embedded language to a function in our meta language. The function argument and return types must match the indices of the sub-expressions and the parent expression respectively, as we see in definition 21. Here we give the types of the ad-hoc polymorphic function $+$ as $+ : Num\ a \Rightarrow a \to a \to a$ and the parametrically polymorphic function $const$ as $const : a \to b \to a$ respectively.

Definition 21 (An evaluation function for $Expr_\zeta$). .

$$\begin{aligned}
&\mathsf{eval}\ : \mathsf{Expr}_\zeta\ \mathsf{A} \to \mathsf{A} \\
&\mathsf{eval}(\mathsf{Lit}_\zeta(\mathsf{x})) && = \mathsf{x} \\
&\mathsf{eval}(\mathsf{Add}_\zeta(\mathsf{x}, \mathsf{y})) && = \mathsf{eval}(\mathsf{x}) + \mathsf{eval}(\mathsf{y}) \\
&\mathsf{eval}(\mathsf{Const}_\zeta(\mathsf{x}, \mathsf{y})) && = const(\mathsf{eval}(\mathsf{x}), \mathsf{eval}(\mathsf{y}))\ \text{where}\ const(x, y) = x
\end{aligned}$$

Following from definition 20 we saw that we may index any expression in ζ by any type that we may derive in our meta language. This includes the uninhabited type $\perp$. In order to allow us to express effectively total functions going forward we must discharge a proof obligation to say that while $\mathsf{Expr}_\zeta\ \perp$ is a valid type, it cannot be inhabited by any values (and therefore is effectively the $\perp$ type). We use our definition of eval to prove this in theorem 1.

Theorem 1 ($Expr_\zeta \to \perp$). $\mathsf{Expr}_\zeta\ \perp$ *is uninhabited.*

Proof.

$$\begin{aligned}
&\mathsf{e} : \mathsf{Expr}_\zeta\ \perp && \text{Assume } \mathsf{Expr}_\zeta\ \perp \text{ is inhabited.} && (1) \\
&\mathsf{eval}(\mathsf{e}) : \perp. && \text{By definition 21.} && (2) \\
&\textit{ex falso quodlibet} && \text{By line 2.} && (3)
\end{aligned}$$

□

Given our well-typed embedding Expr_ζ we show from definition 22 that we may *calculate* the appropriate type when "walking" down the abstract syntax tree. We define an indexed family of types where the index is a value of $\mathsf{Expr}_\zeta\ \mathsf{A}$ and the elements are values of our universe of discourse type: Type . One indexed family is defined for each direction that we would like to walk down. For invalid directions, (going right or left on the unary value Lit_ζ for example) we return the $\perp$ type. For the valid directions we return the appropriate type of the sub-expression, given the direction and parent expression.

In ζ the maximum arity of an expression is two and the only unary expression Lit_ζ is a terminating node (the base case in our inductive definition of Expr_ζ). Only two directions are sufficient for walking down an expression tree in ζ: walking down the left sub-expression, or walking down the right sub-expression. In particular we see in the Const_ζ case that we return the type of the right sub-expression B, from definition 20.

Definition 22 (indexed families representing left and right movement down an $Expr_\zeta$ tree)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Expr}_\zeta\ \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Left}_\zeta\ : \mathsf{Expr}_\zeta\ \mathsf{A} \to \mathsf{Type}} \qquad \frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Expr}_\zeta\ \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Right}_\zeta\ : \mathsf{Expr}_\zeta\ \mathsf{A} \to \mathsf{Type}}$$

$$\begin{aligned}
&\mathsf{Left}_\zeta : \mathsf{Expr}_\zeta\ \mathsf{A} \to \mathsf{Type} \\
&\mathsf{Left}_\zeta\ (\mathsf{Lit}_\zeta(\mathsf{x})) && = \perp \\
&\mathsf{Left}_\zeta\ (\mathsf{Add}_\zeta(\mathsf{x},\mathsf{y})) && = \mathsf{A} \\
&\mathsf{Left}_\zeta\ (\mathsf{Const}_\zeta(\mathsf{x},\mathsf{y})) && = \mathsf{A}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Right}_\zeta : \mathsf{Expr}_\zeta\ \mathsf{A} \to \mathsf{Type} \\
&\mathsf{Right}_\zeta\ (\mathsf{Lit}_\zeta(\mathsf{x})) && = \perp \\
&\mathsf{Right}_\zeta\ (\mathsf{Add}_\zeta(\mathsf{x},\mathsf{y})) && = \mathsf{A} \\
&\mathsf{Right}_\zeta\ (\mathsf{Const}_\zeta(\mathsf{x},\mathsf{y})) && = \mathsf{B}
\end{aligned}$$

Recall the zipper as a pairing of a focus and a context from section 3.2. The context must contain enough information to rebuild the parent node given a focus. In particular the context must specify which direction was taken, from which parent, and the context under which that parent was focused. We define a new type-indexed inductive family Ctx_ζ to capture this information.

The typing rule for our Ctx_ζ is straightforward. We then define three possible values:

- Root_ζ represents the root, or top, of an expression tree.
- $\mathsf{L}_\zeta(\mathsf{e},\mathsf{ctx})$ represents a context state from moving left on a parent expression e where that parent had a context of ctx. The type index of this context is calculated by using $\mathsf{Left}_\zeta(\mathsf{e})$.
- $\mathsf{R}_\zeta(\mathsf{e},\mathsf{ctx})$ represents a context state from moving right. Otherwise this value is identical to the left context except for the type index, which is now $\mathsf{Right}_\zeta(\mathsf{e})$.

Definition 23 (A context type to rebuild $Expr_\varsigma$ trees)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Ctx}_\varsigma\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Root}_\varsigma : \mathsf{Ctx}_\varsigma\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\varsigma\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\varsigma\ \mathsf{A} \qquad \Gamma \vdash \mathsf{Right}_\varsigma : \mathsf{Expr}_\varsigma\ \mathsf{A} \to \mathsf{Type}}{\Gamma \vdash \mathsf{R}_\varsigma(\mathsf{x}, \mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \mathsf{Right}_\varsigma(\mathsf{x})}$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\varsigma\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\varsigma\ \mathsf{A} \qquad \Gamma \vdash \mathsf{Left}_\varsigma : \mathsf{Expr}_\varsigma\ \mathsf{A} \to \mathsf{Type}}{\Gamma \vdash \mathsf{L}_\varsigma(\mathsf{x}, \mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \mathsf{Left}_\varsigma(\mathsf{x})}$$

From the definition of Ctx_ς alone, it is not immediately apparent why indexing the left and right values by the respective indexed families is necessary. This becomes clear once we define a type to represent our pairing of a focus and a context. We define a type indexed $\mathsf{Zipper}_\varsigma$ as effectively a type-indexed product. Any given $\mathsf{Zipper}_\varsigma\ \mathsf{A}$ must have been defined using an $\mathsf{Expr}_\varsigma\ \mathsf{A}$ and a $\mathsf{Ctx}_\varsigma\ \mathsf{A}$ as per definition 24 below. From definition 23 we saw that either the type index will unify with any type (in the case of Root_ς), or that the index must arise as a result of Left_ς or Right_ς. Note that all type parameters for each argument value in the zipper must be the same. This ensures that any attempt to describe an incorrect zipper state through the pairing of a context and an invalid focus type is automatically ill-typed and will be rejected as such by the compiler of our dependently typed meta language. This gives us a measure of type safety when implementing this Zipper type.

Definition 24 (A type indexed Zipper over $Expr_\varsigma$ and Ctx_ς)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Zipper}_\varsigma\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\varsigma\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\varsigma\ \mathsf{A}}{\Gamma \vdash \mathsf{Zip}_\varsigma(\mathsf{x}, \mathsf{ctx}) : \mathsf{Zipper}_\varsigma\ \mathsf{A}}$$

We now have enough information to “walk” down our zipped expression trees. We define a function for each direction from a Σ type to a Σ type. The second element of the dependent pair is the type indexed $\mathsf{Zipper}_\varsigma\ \mathsf{a}$ and the first element is the specific type index of the second element (a). Recall from Chapter 2 (2.1.6) that a Σ may also be read as a first order logic proposition as per the Curry-Howard isomorphsim. In this instance we claim that a given direction function below has the following interpretation: $\exists a, Zipper(a) \rightarrow \exists b, Zipper(b)$ where $a, b \in \mathsf{Type}$. Informally we may think of this as a function that maps zippers to zippers, but where we may not be absolutely certain of their type index. There are two general cases to consider in each direction function:

- If the input zipper pf has a focus of $\mathsf{Lit}_\varsigma(\mathsf{x})$ then we return the input zipper unchanged. We could also add an error-case in our zipper, or return a value of type $\mathsf{Maybe}\ \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\varsigma\ \mathsf{a}$ if we wanted to explicitly fail on this case.

- If the input zipper pf has a focus of binary expression e with some left and right sub expressions of x and y respectively. Here we return a new dependent pair. The second element is a zipper with the appropriate sub expression as the focus (x if walking right, y if walking left). The context is the appropriate direction value (L_ς or R_ς) of the previous focus e and the context under which that previous focus was derived ctx. The first element then contains the type index of this zipper.

Definition 25 (Left and right direction functions over a $Zipper_\varsigma$)**.**

$$\mathsf{left}_\varsigma : \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\varsigma\ \mathsf{a} \rightarrow \Sigma\, \mathsf{b} : \mathsf{Type} \,.\, \mathsf{Zipper}_\varsigma\ \mathsf{b}$$

$$\mathsf{left}_\varsigma(\mathsf{t}, \mathsf{pf}) = \begin{cases} (\mathsf{t}, \mathsf{pf}), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Lit}_\varsigma(\mathsf{x}). \\ (\mathsf{Left}_\varsigma(\mathsf{e}), \mathsf{Zip}_\varsigma(\mathsf{x}, \mathsf{L}_\varsigma(\mathsf{e}, \mathsf{ctx}))), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Add}_\varsigma(\mathsf{x}, \mathsf{y}). \\ (\mathsf{Left}_\varsigma(\mathsf{e}), \mathsf{Zip}_\varsigma(\mathsf{x}, \mathsf{L}_\varsigma(\mathsf{e}, \mathsf{ctx}))), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Const}_\varsigma(\mathsf{x}, \mathsf{y}) \end{cases}$$

$$\mathsf{right}_\varsigma : \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\varsigma\ \mathsf{a} \rightarrow \Sigma\, \mathsf{b} : \mathsf{Type} \,.\, \mathsf{Zipper}_\varsigma\ \mathsf{b}$$

$$\mathsf{right}_\varsigma(\mathsf{t}, \mathsf{pf}) = \begin{cases} (\mathsf{t}, \mathsf{pf}), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Lit}_\varsigma(\mathsf{x}). \\ (\mathsf{Right}_\varsigma(\mathsf{e}), \mathsf{Zip}_\varsigma(\mathsf{y}, \mathsf{R}_\varsigma(\mathsf{e}, \mathsf{ctx}))), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Add}_\varsigma(\mathsf{x}, \mathsf{y}). \\ (\mathsf{Right}_\varsigma(\mathsf{e}), \mathsf{Zip}_\varsigma(\mathsf{y}, \mathsf{R}_\varsigma(\mathsf{e}, \mathsf{ctx}))), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{ctx}) \text{ where } \mathsf{e} = \mathsf{Const}_\varsigma(\mathsf{x}, \mathsf{y}). \end{cases}$$

We also require a way to move back up the expression tree. Recall that our goal is to use this structure to facilitate a type preserving crossover operator. We may think of this as having to substitute in an expression tree at a given node. Given our zipper structure, we would ideally like to ensure that any such substitution operation propagates through the parent contexts when moving towards the root. We define the function up_ς in definition 26. We see from the definition that there are three general contexts: the input focus is at the root, the focus arose from going left on a binary expression, or the focus arose from going right on a binary expression.

If the context represents the root of the tree (Root_ς), we return the given input and up_ς effectively acts as the identity function. If the context represents a focus obtained by going left or right on binary expression then we return a new zipper, with the parent expression as the focus, and the parent context as the context. However, we substitute the appropriate sub-expression of the parent, with the focus from the input zipper.

This will do nothing in the case of an ordinary movement, but allows us to propagate any substitutions through the context.

Recall from Chapter 2 (2.1.7) that functions using dependent types must be total. Definition 26 is clearly partial. We claim that up_ς is *effectively* total however, if we are able to prove that all the missing cases are uninhabited and therefore cannot arise. The missing definitions are for zippers with contexts that attempt to walk left or right on the unary function Lit_ς. We show that these values are impossible by proving that they are uninhabited below in theorem 2 and theorem 3.

Definition 26 (The up_ς function rebuilds a $Zipper_\varsigma$.)**.**

$$\mathsf{up}_\varsigma : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{Zipper}_\varsigma\ \mathsf{a} \to \Sigma\, \mathsf{b} : \mathsf{Type}\,.\, \mathsf{Zipper}_\varsigma\ \mathsf{b}$$

$$\mathsf{up}_\varsigma(\mathsf{t}, \mathsf{pf}) = \begin{cases} (\mathsf{t}, \mathsf{pf}), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{x}, \mathsf{Root}_\varsigma). \\ (\mathsf{Left}_\varsigma(\mathsf{c}), \mathsf{Zip}_\varsigma(\mathsf{Add}_\varsigma(\mathsf{e}, \mathsf{y}), \mathsf{ctx})), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{L}_\varsigma(\mathsf{c}, \mathsf{ctx})) \text{ where} \\ & \quad \mathsf{c} = \mathsf{Add}_\varsigma(\mathsf{x}, \mathsf{y}). \\ (\mathsf{Left}_\varsigma(\mathsf{c}), \mathsf{Zip}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{e}, \mathsf{y}), \mathsf{ctx})), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{L}_\varsigma(\mathsf{c}, \mathsf{ctx})) \text{ where} \\ & \quad \mathsf{c} = \mathsf{Const}_\varsigma(\mathsf{x}, \mathsf{y}). \\ (\mathsf{Right}_\varsigma(\mathsf{c}), \mathsf{Zip}_\varsigma(\mathsf{Add}_\varsigma(\mathsf{x}, \mathsf{e}), \mathsf{ctx})), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{R}_\varsigma(\mathsf{c}, \mathsf{ctx})) \text{ where} \\ & \quad \mathsf{c} = \mathsf{Add}_\varsigma(\mathsf{x}, \mathsf{y}). \\ (\mathsf{Right}_\varsigma(\mathsf{c}), \mathsf{Zip}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{x}, \mathsf{e}), \mathsf{ctx})), & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{R}_\varsigma(\mathsf{c}, \mathsf{ctx})) \text{ where} \\ & \quad \mathsf{c} = \mathsf{Const}_\varsigma(\mathsf{x}, \mathsf{y}). \end{cases}$$

Theorem 2 ($L_\varsigma(Lit_\varsigma(x), ctx) \to \bot$)**.** *Context* $\mathsf{L}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}), \mathsf{ctx})$ *is uninhabited.*

Proof.

$$\begin{aligned} &\mathsf{L}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}), \mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \mathsf{Left}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x})) && \text{By definition 23.} && (4)\\ &\mathsf{Left}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x})) : \bot && \text{By definition 22.} && (5)\\ &\mathsf{L}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}), \mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \bot && \text{By lines 4 and 5.} && (6)\\ &\mathsf{Zip}_\varsigma(\mathsf{e}, \mathsf{L}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}), \mathsf{ctx})) : \mathsf{Zipper}_\varsigma\ \bot && \text{By definition 24 and line 6.} && (7)\\ &\mathsf{e} : \mathsf{Expr}_\varsigma\ \bot && \text{By definition 24 and line 7.} && (8)\\ &\textit{ex falso quodlibet} && \text{By theorem 1 and line 8.} && (9) \end{aligned}$$

□

Theorem 3 ($R_\varsigma(Lit_\varsigma(x), ctx) \to \bot$)**.** *Context* $\mathsf{R}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}), \mathsf{ctx})$ *is uninhabited.*

Proof.

$$
\begin{aligned}
&\mathsf{R}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}),\mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \mathsf{Right}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x})) && \text{By definition 23.} && (10)\\
&\mathsf{Right}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x})) : \bot && \text{By definition 22.} && (11)\\
&\mathsf{R}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}),\mathsf{ctx}) : \mathsf{Ctx}_\varsigma\ \bot && \text{By lines 10 and 11.} && (12)\\
&\mathsf{Zip}_\varsigma(\mathsf{e},\mathsf{R}_\varsigma(\mathsf{Lit}_\varsigma(\mathsf{x}),\mathsf{ctx})) : \mathsf{Zipper}_\varsigma\ \bot && \text{By definition 24 and line 12.} && (13)\\
&\mathsf{e} : \mathsf{Expr}_\varsigma\ \bot && \text{By definition 24 and line 13.} && (14)\\
&\textit{ex falso quodlibet} && \text{By theorem 1 and line 14.} && (15)
\end{aligned}
$$

□

We will also use up_ς to define a convenience function for moving back to the root node and propagating any focus substitutions. It is important to stress that the definition given below is not strictly total due to the recursive call of top_ς . We should provide a proof of well-founded recursion to show that top_ς operates on structurally smaller objects. However, we will leave such a proof as an exercise to the reader as the definition of up_ς straightforwardly returns a context one level higher in the expression tree. Interested readers will find well-documented resources on well-founded recursion proofs for Coq [2], Agda [3] , and Idris [4] . Readers may also refer to the work of Tomé Cortiñas and Swierstra (2018) for a well-founded recursion proof over well-typed expression trees wrapped in similar object to our zipper.

Definition 27 (top_ς rebuilds a $Zipper_\varsigma$ to the root)**.**

$$\mathsf{top}_\varsigma : \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a} \to \Sigma\,\mathsf{b} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{b}$$

$$\mathsf{top}_\varsigma(\mathsf{t},\mathsf{pf}) = \begin{cases} (\mathsf{t},\mathsf{pf}) & \text{if } \mathsf{pf} = \mathsf{Zip}_\varsigma(\mathsf{e},\mathsf{Root}_\varsigma). \\ \mathsf{top}_\varsigma(\mathsf{up}_\varsigma(\mathsf{t},\mathsf{pf})) & \text{otherwise.} \end{cases}$$

We define a substitution function, subst_ς, to substitute a focus from a zipper of a given type index, with a new focus expression of the same type index. This is achieved by using the dependent pair projection function fst on the supplied Σ type xs to calculate the type index of the supplied Expr_ς (recall Chapter 2 (2.1.6)).

Definition 28 ($subst_\varsigma$ substitutes a new $Expr_\varsigma$ in the focus on a $Zipper_\varsigma$)**.**

$$\mathsf{subst}_\varsigma : \mathsf{x} : (\Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}) \to \mathsf{Expr}_\varsigma\ \mathsf{fst}(\mathsf{x}) \to \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}$$

$$\mathsf{subst}_\varsigma\ ((\mathsf{t},\mathsf{Zip}_\varsigma(\mathsf{e},\mathsf{c})),\mathsf{x}) = (\mathsf{t},\mathsf{Zip}_\varsigma(\mathsf{x},\mathsf{c}))$$

2 https://coq.inria.fr/library/Coq.Init.Wf.html Accessed on 05/10/19.
3 https://github.com/agda/agda-stdlib/blob/master/src/Induction/WellFounded.agda Accessed on 05/10/19.
4 https://www.idris-lang.org/docs/current/base_doc/docs/Prelude.WellFounded.html Accessed on 05/10/19 .

Lastly, definition 29 shows how we may recover the expression tree under the zipper focus for future evaluation or substitution.

Definition 29 ($extract_\zeta$ extracts an $Expr_\zeta$ under the focus of a $Zipper_\zeta$)**.**

$$\mathsf{extract}_\zeta : \mathsf{x} : (\Sigma\, \mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\zeta\ \mathsf{a}) \rightarrow \mathsf{Expr}_\zeta\ \mathsf{fst}(\mathsf{x})$$
$$\mathsf{extract}_\zeta(\mathsf{t}, \mathsf{Zip}_\zeta(\mathsf{e}, \mathsf{ctx})) = \mathsf{e}$$

3.4 Dependent Zipper Examples

In this section we give a series of example functions and figures that show how we perform type preserving substitution and movement along well-typed expression trees. We start with example 13 defining an expression that adds the natural number 2, to the result of the function $const(2, Test)$ where we recall that $const$ returns the value of the first argument given. So effectively example 13 is adding 2 to 3. What makes this expression interesting is that while the second argument to $const$ never is evaluated, it never-the-less forms part of the expression, and we should be able to substitute in a new value for the string $Test$ in a manner that guarantees that the expression is still well-typed.

Example 13 (An $Expr_\zeta$ Nat containing a sub-expression of $Expr_\zeta$ $String$)**.**

$$\mathsf{ex}_1 : \mathsf{Expr}_\zeta\ \mathsf{Nat}$$
$$\mathsf{ex}_1 = \mathsf{Add}_\zeta(\mathsf{Lit}_\zeta(2), \mathsf{Const}_\zeta(\mathsf{Lit}_\zeta(3), \mathsf{Lit}_\zeta(Test)))$$

Once we have defined an expression, we define a zipper with the root (or entire expression) as the focus as per example 14.

Example 14 (A $Zipper_\zeta$ Nat from $ex_1 : Expr_\zeta\ Nat$)**.**

$$\mathsf{ex}_2 : \mathsf{Zipper}_\zeta\ \mathsf{Nat}$$
$$\mathsf{ex}_2 = \mathsf{Zip}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta)$$

Recall that our direction functions for zippers operates on a Σ type as its parameter, pairing a type with a zipper indexed by that type. We wrap the zipper from example 14 with a Σ type by giving the type index of the zipper as the first element of the dependent pair, and value for ex_1 as the second element. We see this in example 15.

Example 15 (A Σ type of $ex_2 : Zipper_\zeta\ Nat$)**.**

$$\mathsf{ex}_3 : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\zeta\ \mathsf{a}$$
$$\mathsf{ex}_3 = (\mathsf{Nat}, \mathsf{ex}_2)$$

We then observe from example 16 that moving left gives us a dependent pair with the type of the left sub expression as the first element, and a zipper for the second element. This zipper contains the *value* of the left sub expression as the focus, and a context that describes how we arrived at the focus (by going "$\mathsf{L}_\zeta$eft" on the parent expression and parent context).

Example 16 (Moving the focus $left_\zeta$ on ex_3)**.**

$$\begin{aligned} \mathsf{left}_\zeta(\mathsf{ex}_3) &: \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a} \\ \mathsf{left}_\zeta(\mathsf{ex}_3) &= (\mathsf{Nat}, \mathsf{Zip}_\zeta(\mathsf{Lit}_\zeta(2), \mathsf{L}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta))) \end{aligned}$$

Similarly, moving right from the root node gives us the appropriate dependent pair value with the return type of *const* (a natural number in this case), and a zipper with the right sub expression as the focus. We see this in example 17.

Example 17 (Moving the focus $right_\zeta$ on ex_3)**.**

$$\begin{aligned} \mathsf{right}_\zeta(\mathsf{ex}_3) &: \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a} \\ \mathsf{right}_\zeta(\mathsf{ex}_3) &= (\mathsf{Nat}, \mathsf{Zip}_\zeta(\mathsf{Const}_\zeta(\mathsf{Lit}_\zeta(3), \mathsf{Lit}_\zeta(Test)), \mathsf{R}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta))) \end{aligned}$$

Example 18 shows us concretely why it is necessary to give the type of the direction functions as $\Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a} \rightarrow \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$. Firstly, we compose direction functions with other direction functions to form a *path*, regardless of the particular type index of a given node. Secondly we may project out the type of a focus to perform type calculations, an example of which we shall see momentarily. It is important to note that the first element of the dependent pair has changed to the correct type given the new type index of the corresponding second element.

Example 18 (Moving the focus $right_zeta$ on $right_\zeta(ex_3)$)**.**

$$\begin{aligned} \mathsf{right}_\zeta(\mathsf{right}_\zeta(\mathsf{ex}_3)) &: \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a} \\ \mathsf{right}_\zeta(\mathsf{right}_\zeta(\mathsf{ex}_3)) &= (\mathsf{String}, \mathsf{Zip}_\zeta(\mathsf{Lit}_\zeta(Test), \mathsf{R}_\zeta(\mathsf{Const}_\zeta(\mathsf{Lit}_\zeta(3), \mathsf{Lit}_\zeta(Test)), \mathsf{R}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta)))) \end{aligned}$$

Recall that subst_ζ has the type $\mathsf{x} : (\Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}) \rightarrow \mathsf{Expr}_\zeta\ \mathsf{fst}(\mathsf{x}) \rightarrow \Sigma\, \mathsf{a} : \mathsf{Type} \,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$. The expression given as the second argument to the function must have the same type index as the first *element* of the dependent pair that we give subst_ζ as the first argument. Effectively fst *calculates* the type index of the second parameter given the first parameter. From our previous definition of zippers (definition 24) the type index of a given zipper must be the same as the type index of its focus, we now see type preserving substitution in example 19.

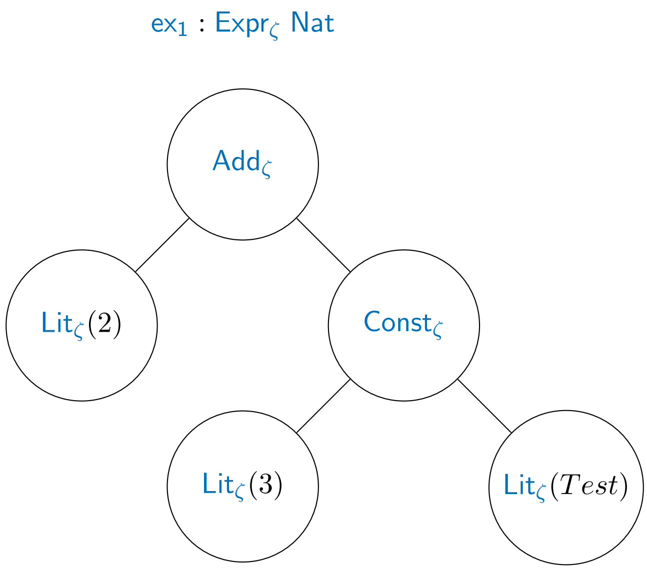


Figure 5: A graphical representation of example 13.

Example 19 (Substituting a new focus on $right_\varsigma(right_\varsigma(ex_3))$)**.**

$$\begin{aligned}
&\mathsf{z} : \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}\\
&\mathsf{z} = \mathsf{right}_\varsigma(\mathsf{right}_\varsigma(\mathsf{ex}_3))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello)) \ : \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}\\
&\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello)) \ = (\mathsf{String}, \mathsf{Zip}_\varsigma(\mathsf{Lit}_\varsigma(Hello), \mathsf{R}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Test)), \mathsf{R}_\varsigma(\mathsf{ex}_1, \mathsf{Root}_\varsigma))))
\end{aligned}$$

Lastly, we may then recover the original (modulo the substituted node) zipped expression through successive calls to up_ς as we see in example 20. Our expression is still well-typed and guaranteed to be so through the type checker in our dependently typed meta language.

Example 20 (Rebuilding $subst_\varsigma(z, Lit_\varsigma(Hello))$)**.**

$$\begin{aligned}
&\mathsf{up}_\varsigma(\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello))) \ : \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}\\
&\mathsf{up}_\varsigma(\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello))) \ = (\mathsf{Nat}, \mathsf{Zip}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Hello)), \mathsf{R}_\varsigma(\mathsf{ex}_1, \mathsf{Root}_\varsigma)))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{up}_\varsigma(\mathsf{up}_\varsigma(\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello)))) \ : \Sigma\,\mathsf{a} : \mathsf{Type}\,.\,\mathsf{Zipper}_\varsigma\ \mathsf{a}\\
&\mathsf{up}_\varsigma(\mathsf{up}_\varsigma(\mathsf{subst}_\varsigma(\mathsf{z}, \mathsf{Lit}_\varsigma(Hello)))) \ = (\mathsf{Nat}, \mathsf{Zip}_\varsigma(\mathsf{Add}_\varsigma(\mathsf{Lit}_\varsigma(2), \mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Hello))), \mathsf{Root}_\varsigma))
\end{aligned}$$

$\mathsf{ex}_2 : \mathsf{Zipper}_\varsigma\ \mathsf{Nat}$

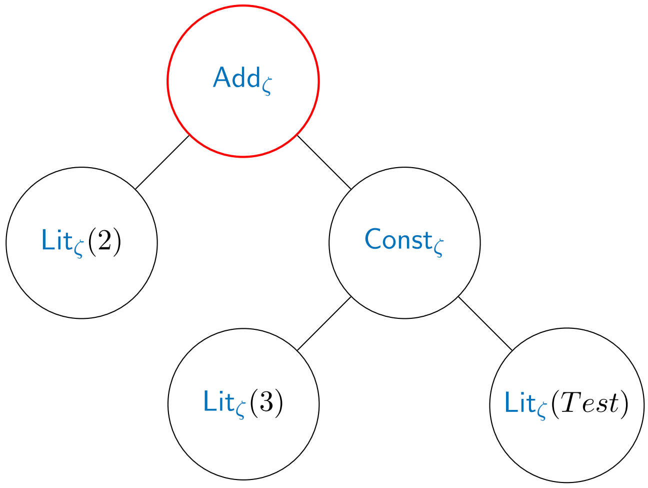


Figure 6: A graphical representation of a zipper on the expression seen in example 14.

3.5 A Zipper-Like Structure

It is important to mention that the $\mathsf{Zipper}_\varsigma$ structure we have defined here is not equivalent to a zipper from the algebra of types perspective. McBride (2001) shows that the derivative of a type is the type of one-hole context. The Ctx_ς (definition 23) we have defined has no "hole", or missing type parameter that matches the focus. This concept of a hole is clearly visible in Hamana and Fiore's specification of $\mathsf{Ctx}_{\mathsf{hf}}$ (example 11). The return type of each non-empty (or non-root) context represents the missing sub-expression, which the $\mathsf{Zipper}_{\mathsf{hf}}$ then pairs up with the context as the focus. In contrast our Ctx_ς carries around the entire parent expression, so the sub-expression is not missing, and is duplicated in the zipper twice. Once in the parent expression inside the context, and one again as the focus. This is the price we pay for the ability to call a generic right_ς or left_ς function irrespective of the current focus. The context type is indexed by a function family, and that family requires an expression. Our $\mathsf{Zipper}_\varsigma$ type is still *semantically* a zipper however, and will satisfy our use case of enabling a type preserving crossover operation.

$(\mathsf{Nat}, \mathsf{Zip}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta)) : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$

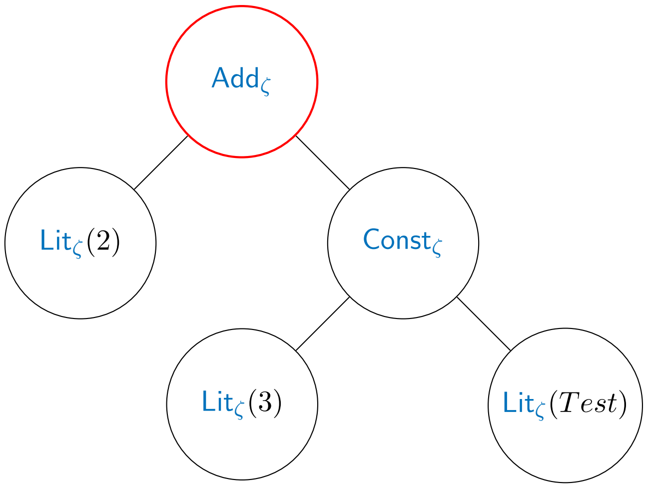


Figure 7: A Σ type that wraps the zipper in example 14.

$(\mathsf{Nat}, \mathsf{Zip}_\zeta(\mathsf{Lit}_\zeta(2), \mathsf{L}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta))) : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$

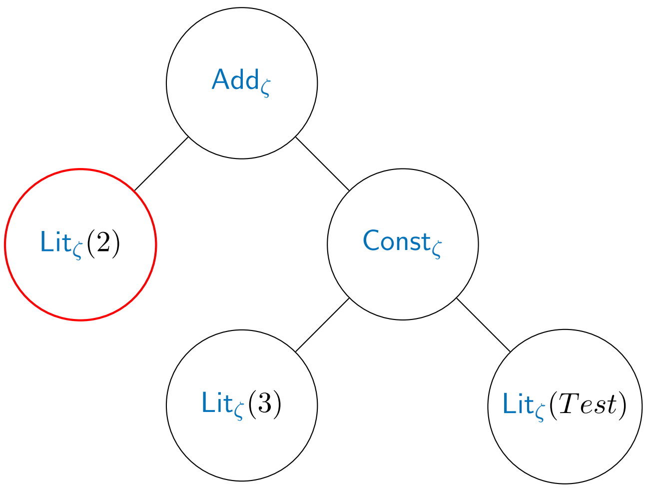


Figure 8: Moving the focus left from the root node.

$$(\mathsf{Nat}, \mathsf{Zip}_\zeta(\mathsf{Const}_\zeta(\mathsf{Lit}_\zeta(3), \mathsf{Lit}_\zeta(Test)), \mathsf{R}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta))) : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$$

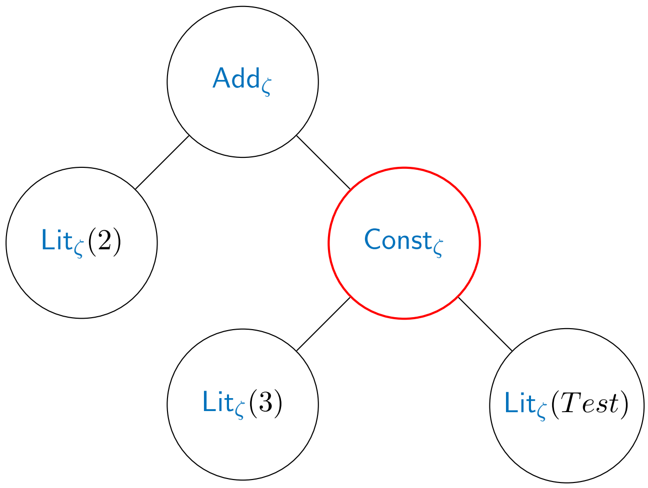


Figure 9: Moving the focus right from the root node.

$$(\mathsf{String}, \mathsf{Zip}_\zeta(\mathsf{Lit}_\zeta(Test), \mathsf{R}_\zeta(\mathsf{Const}_\zeta(\mathsf{Lit}_\zeta(3), \mathsf{Lit}_\zeta(Test)), \mathsf{R}_\zeta(\mathsf{ex}_1, \mathsf{Root}_\zeta)))) : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{Zipper}_\zeta\ \mathsf{a}$$

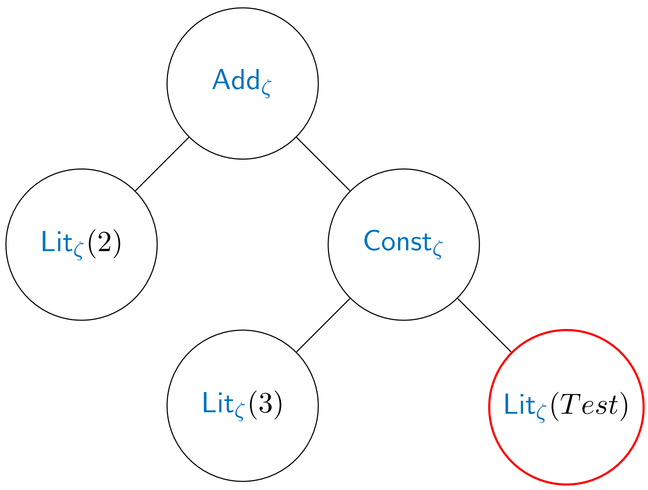
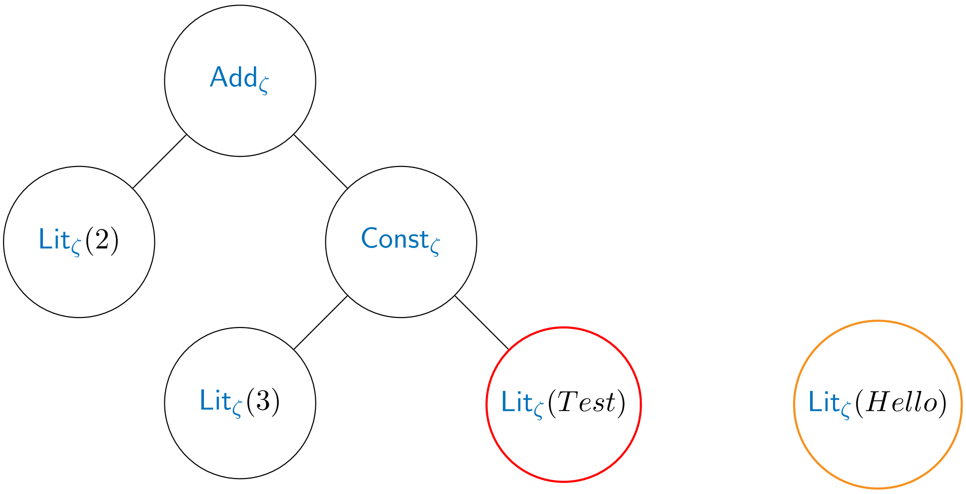


Figure 10: Moving the focus right twice from the root node.

$$\text{Let } \mathsf{z} = \mathsf{Zip}_\varsigma(\mathsf{Lit}_\varsigma(Test), \mathsf{R}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Test)), \mathsf{R}_\varsigma(\mathsf{ex}_1, \mathsf{Root}_\varsigma))) : \mathsf{Zipper}_\varsigma\ \mathsf{String}$$

$(\mathsf{String}, \mathsf{z}) : \Sigma\, \mathsf{a} : \mathsf{Type}\, .\, \mathsf{Zipper}_\varsigma\ \mathsf{a}$ $\qquad$ $\mathsf{Lit}_\varsigma(Hello) : \mathsf{Expr}_\varsigma\ \mathsf{fst}(\mathsf{String}, \mathsf{z})$



$$\text{Let } \mathsf{x} = \mathsf{Zip}_\varsigma(\mathsf{Lit}_\varsigma(Hello), \mathsf{R}_\varsigma(\mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Test)), \mathsf{R}_\varsigma(\mathsf{ex}_1, \mathsf{Root}_\varsigma))) : \mathsf{Zipper}_\varsigma\ \mathsf{String}$$

$$\text{Let } \mathsf{y} = \mathsf{Zip}_\varsigma(\mathsf{Add}_\varsigma(\mathsf{Lit}_\varsigma(2), \mathsf{Const}_\varsigma(\mathsf{Lit}_\varsigma(3), \mathsf{Lit}_\varsigma(Hello))), \mathsf{Root}_\varsigma) : \mathsf{Zipper}_\varsigma\ \mathsf{Nat}$$

$(\mathsf{String}, \mathsf{x}) : \Sigma\, \mathsf{a} : \mathsf{Type}\, .\, \mathsf{Zipper}_\varsigma\ \mathsf{a}$ $\qquad$ $\mathsf{top}_\varsigma\ (\mathsf{Nat}, \mathsf{y}) : \Sigma\, \mathsf{a} : \mathsf{Type}\, .\, \mathsf{Zipper}_\varsigma\ \mathsf{a}$

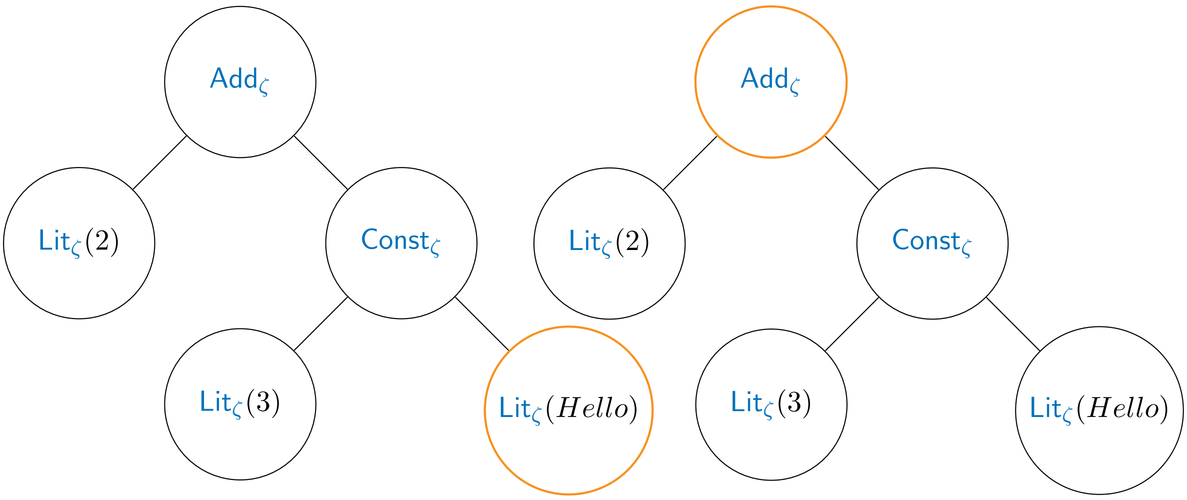


Figure 11: Substituting a new expression for the focus.
Then rebuilding the zipper.

4 VERIFIED CROSSOVER ON PHOAS TREES

In this chapter we discuss the necessary mechanics for adapting our language to the "real world". This involves a variety of awkward solutions and work-arounds so that we may actually do something useful. We discuss the notion of parametricity and why types are usually only "almost" first class citizens of dependently typed functional languages. We then define a method in which we can bypass this restriction for a specific closed universe. We go on to discuss issues of Monadic IO and the necessity of being able to print information. This has a large flow-on effect leading to a complete redefinition of our types so far. We then discuss the necessary mechanics for a random well-typed crossover, and give the definitions and types to achieve this, discussing any proofs or language specific issues along the way. Finally, we present a proof of concept program in the Idris programming language.

4.1 Pattern Matching on Types and Parametricity

A type preserving crossover operation must (by definition) have some manner of determining whether or not two types are equal. However, in common dependently typed functional programming languages (e.g Idris, Agda) pattern matching on type is often prohibited as it would break parametricity [1]. Wadler (1989) explains parametricity as a result allowing theorems to be derived from universally quantified parametrically polymorphic (recall Chapter 2 (2.1.2)) types (many functional programming languages implicitly quantify polymorphic type variables, e.g Idris, Haskell, etc). To give an example, consider the function $f : \forall a.a \to a$. Wadler shows that there are two theorems about the possible definitions for f under these conditions: $f(x) = x$ or $f(x) = f(x)$. In dependently typed languages we gain access to the universe of discourse (the type of types: type, set, prop, * - depending on the language). The above theorems now will not hold if we can match on the type of a instead. We could say $f(Bool) = Int$ or $f(\mathbb{N}) = \mathbb{N} \to \mathbb{N}$. In order to avoid this, pattern matching on types us often disallowed in dependently typed functional languages.

In order to match on types then it is usually necessary to define a universe of types on which we would like to match at run time. (Peyton Jones et al., 2016) shows how we may use an open (easily extensible and maintainable) approach to defining universes, however for the purposes of simplicity here, we define a closed universe of types for

1 Brady indicates the restriction on parametricity will be relaxed in Idris 2.
See https://twitter.com/edwinbrady/status/1088811837376876545 Accessed on 16/10/19.

matching in definition 30. We express this through a type indexed GADT where each constructor tag returns the GADT, with the index specified to the type in question.

Definition 30 (*TypeRep*: A data type that allows pattern matching on a type at run time)**.**

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{TypeRep}_\eta : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}}{\Gamma \vdash \mathsf{TBool} : \mathsf{TypeRep}_\eta\ \mathsf{Bool}} \quad \frac{\Gamma \vdash \mathsf{Nat} : \mathsf{Type}}{\Gamma \vdash \mathsf{TNat} : \mathsf{TypeRep}_\eta\ \mathsf{Nat}} \quad \frac{\Gamma \vdash \mathsf{String} : \mathsf{Type}}{\Gamma \vdash \mathsf{TString} : \mathsf{TypeRep}_\eta\ \mathsf{String}}$$

$$\frac{\Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{TypeRep}_\eta\ \mathsf{B}}{\Gamma \vdash \mathsf{TFunc}_\eta(\mathsf{x},\mathsf{y}) : \mathsf{TypeRep}_\eta\ \mathsf{A} \to \mathsf{B}}$$

$$\frac{\Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{TypeRep}_\eta\ \mathsf{B}}{\Gamma \vdash \mathsf{TPair}_\eta(\mathsf{x},\mathsf{y}) : \mathsf{TypeRep}_\eta\ \mathsf{A} \times \mathsf{B}}$$

$$\frac{\Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{TypeRep}_\eta\ \mathsf{B}}{\Gamma \vdash \mathsf{TSum}_\eta(\mathsf{x},\mathsf{y}) : \mathsf{TypeRep}_\eta\ \mathsf{A} + \mathsf{B}}$$

4.2 Random Numbers, Io Monads and Pretty Printing

The crossover operation selects nodes for crossover uniformly at random (Koza, 1992, p. 101). This usually implies an external source of randomness, which in turn suggests that our programs will likely need to be run inside the IO Monad (by reading from /dev/urandom, or the system clock time, etc) (Peyton Jones, 2010). We then require a way to display these randomly generated values back to the user. This presents a problem if our expression language contains a term for lambda abstraction. Consider the following extension to our language ζ from Chapter 3 (Definition 20):

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} \to \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{f} : \mathsf{A} \to \mathsf{Expr}_\zeta\ \mathsf{B}}{\Gamma \vdash \mathsf{Lam}_\zeta(\mathsf{f}) : \mathsf{Expr}_\zeta\ (\mathsf{A} \to \mathsf{B})}$$

We can begin by defining a pretty printing function over our newly extended Expr_ζ GADT.

$$\mathsf{pretty}_\zeta : \mathsf{Expr}_\zeta\ \mathsf{A} \rightarrow \mathsf{String}$$
$$\mathsf{pretty}_\zeta(\mathsf{Lit}_\zeta(\mathsf{x})) = show(\mathsf{x})$$
$$\mathsf{pretty}_\zeta(\mathsf{Add}_\zeta(\mathsf{x},\mathsf{y})) = \text{“Add(”} +\!\!+\ \mathsf{pretty}_\zeta(\mathsf{x}) +\!\!+ \text{“,”} +\!\!+\ \mathsf{pretty}_\zeta(\mathsf{y}) +\!\!+ \text{“)”}$$
$$\mathsf{pretty}_\zeta(\mathsf{Const}_\zeta(\mathsf{x},\mathsf{y})) = \text{“Const(”} +\!\!+\ \mathsf{pretty}_\zeta(\mathsf{x}) +\!\!+ \text{“,''} +\!\!+\ \mathsf{pretty}_\zeta(\mathsf{y}) +\!\!+ \text{“)”}$$
$$\mathsf{pretty}_\zeta(\mathsf{Lam}_\zeta(\mathsf{f})) = \lambda x.\mathsf{pretty}_\zeta(\mathsf{f}(x))\ \text{this will not type check.}$$

In the final case here we would like to come up with some a value so that we may apply it to the f parameter in our Lam_ζ constructor. However as we are working in a type system with parametricity there is no value that can satisfy the polymorphic type $\forall a.a$. We cannot “reach under” the lambda binder to print the lambda abstraction expression case. In languages like Haskell that do not enforce a positivity restriction, the lambda abstraction case is usually defined as `Lam : (Expr a -> Expr b) -> Expr (a -> b)`, which we can then combine with an additional dummy constructor in our expression language: `Print : String -> Expr a`. We need a stronger solution when working with dependent types.

4.3 From Hoas to Phoas

(Chlipala, 2008) gives us a method to solve the problem of reaching under lambda abstraction constructors in a GADT-embedded language through parametric higher order abstract syntax (PHOAS) (Chlipala, 2013, chapter 17). We add a polymorphic function family (V) to our expression indices. We then represent the types in our associated constructors as a the result of applying the appropriate type to the family V. We define a more expressive language than ζ, providing terms for lambda abstraction, function application, product types and sum types. We refer to this richer language as η as in definition 31.

Definition 31 (A GADT-embedded parameteric higher order abstract syntax language $Expr_\eta$)**.**

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \rightarrow \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A}}{\Gamma \vdash \mathsf{Lit}_\eta(\mathsf{t}_1, \mathsf{x}) : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}}\ (\Gamma \vdash \mathsf{Show}\ \mathsf{A})$$

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{x} : \mathsf{V}(\mathsf{A})}{\Gamma \vdash \mathsf{Var}_\eta(\mathsf{t}_1, \mathsf{x}) : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{f} : \mathsf{V}(\mathsf{A}) \to \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}\end{array}}{\Gamma \vdash \mathsf{Lam}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{f}) : \mathsf{Expr}_\eta\ \mathsf{V}\ (\mathsf{A} \to \mathsf{B})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{B} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{f} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{B}\end{array}}{\Gamma \vdash \mathsf{App}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{f}, \mathsf{x}) : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}\end{array}}{\Gamma \vdash \mathsf{Add}_\eta(\mathsf{t}_1, \mathsf{x}, \mathsf{y}) : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}}\ (\Gamma \vdash \mathsf{Num}\ \mathsf{A})$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}\end{array}}{\Gamma \vdash \mathsf{Const}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}, \mathsf{y}) : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{y} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}\end{array}}{\Gamma \vdash \mathsf{Pair}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}, \mathsf{y}) : \mathsf{Expr}_\eta\ \mathsf{V}\ (\mathsf{A} \times \mathsf{B})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \\ \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A}\end{array}}{\Gamma \vdash \mathsf{ELeft}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}) : \mathsf{Expr}_\eta\ \mathsf{V}\ (\mathsf{A} + \mathsf{B})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{A} : \mathsf{Type} \\ \Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{t}_1 : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{TypeRep}_\eta\ \mathsf{B} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}\end{array}}{\Gamma \vdash \mathsf{ERight}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}) : \mathsf{Expr}_\eta\ \mathsf{V}\ (\mathsf{A} + \mathsf{B})}$$

With the addition of the typed lambda calculi terms Lam_η and App_η we require a method to partially apply constructor functions in Expr_η if we would like to preserve partial function application. Definition 32 shows how we may transform a function from expressions to a weak-HOAS variant that satisfies the positivity requirement of dependently typed languages. We also define a smart constructor lam_η to more conveniently weaken any partial applications we might find. consider the expression $\mathsf{Add}_\eta(\mathsf{t}_1, \mathsf{Lit}_\eta(\mathsf{t}_1, 2)) : \mathsf{Num}\ \mathsf{A} \Rightarrow \mathsf{Expr}_\eta\ \mathsf{A} \to \mathsf{Expr}_\eta\ \mathsf{A}$ We may wish to transform this into a type $\mathsf{Expr}_\eta\ \mathsf{A} \to \mathsf{A}$ for evaluation or use in higher order functions such as map. We can then extend it using lam_η from definition 33 for the result $\mathsf{lam}_\eta(t_1, t_1, \mathsf{Add}_\eta(t_1, \mathsf{Lit}_\eta(2,))) : \mathsf{Num\ A} \Rightarrow \mathsf{Expr}_\eta\ \mathsf{A} \to \mathsf{A}$.

Definition 32 (*weaken*$_\eta$ transforms a HOAS style lambda abstraction to a weak HOAS style abstraction)**.**

$$\begin{aligned}
&\mathsf{weaken}_\eta : \mathsf{TypeRep}_\eta\ \mathsf{A} \to (\mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}) \to (\mathsf{V}(\mathsf{A}) \to \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B})\\
&\mathsf{weaken}_\eta(t_1, f) = \lambda x : \mathsf{V}(\mathsf{A}).\, f(\mathsf{Var}_\eta(t_1, x))
\end{aligned}$$

Definition 33 (*lam*$_\eta$ is a convenience function for weaking HOAS-style functions to avoid direct calls to *weaken*$_\eta$)**.**

$$\begin{aligned}
&\mathsf{lam}_\eta : \mathsf{TypeRep}_\eta\ \mathsf{A} \to \mathsf{TypeRep}_\eta\ \mathsf{B} \to (\mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{B}) \to \mathsf{Expr}_\eta\ \mathsf{V}\ (\mathsf{A} \to \mathsf{B})\\
&\mathsf{lam}_\eta(t_1, t_2, f) = \mathsf{Lam}_\eta(t_1, t_2, \mathsf{weaken}_\eta(t_1, f))
\end{aligned}$$

The most notable change in our evaluation function for expressions is in the new $\mathsf{V} : \mathsf{Type} \to \mathsf{Type}$ parameter. To recover the fully polymorphic evaluation we used in Chapter 3 (21) we must give V as the identity function on types in definition 34.

Definition 34 (An evaluation function for *Expr*$_\eta$)**.**

$$\begin{aligned}
&\mathsf{eval}_\eta : \mathsf{Expr}_\eta\ (\lambda x : \mathsf{Type}\,.\,x)\ \mathsf{A} \to \mathsf{A}\\
&\mathsf{eval}_\eta(\mathsf{Lit}_\eta(t_1, x)) && = x\\
&\mathsf{eval}_\eta(\mathsf{Var}_\eta(t_1, x)) && = x\\
&\mathsf{eval}_\eta(\mathsf{Lam}_\eta(t_1, t_2, f)) && = \lambda x : \mathsf{A}.\, \mathsf{eval}_\eta(f(x))\\
&\mathsf{eval}_\eta(\mathsf{App}_\eta(t_1, t_2, f, x)) && = (\mathsf{eval}_\eta(f))(\mathsf{eval}_\eta(x))\\
&\mathsf{eval}_\eta(\mathsf{Add}_\eta(t_1, x, y)) && = \mathsf{eval}_\eta(x) + \mathsf{eval}_\eta(y)\\
&\mathsf{eval}_\eta(\mathsf{Const}_\eta(t_1, t_2, x, y)) && = \mathit{const}(\mathsf{eval}_\eta(x), \mathsf{eval}_\eta(y))\\
&\mathsf{eval}_\eta(\mathsf{Pair}_\eta(t_1, t_2, x, y)) && = (\mathsf{eval}_\eta(x), \mathsf{eval}_\eta(y))\\
&\mathsf{eval}_\eta(\mathsf{ELeft}_\eta(t_1, t_2, x)) && = \mathsf{Left}(\mathsf{eval}_\eta(x))\\
&\mathsf{eval}_\eta(\mathsf{ERight}_\eta(t_1, t_2, y)) && = \mathsf{Right}(\mathsf{eval}_\eta(y))
\end{aligned}$$

The additional flexibility we gain from our extension to PHOAS is readily apparent in our pretty printing function for Expr_η (definition 35). Here we see that by forcing the V parameter to always return a string, we fool the type checker into believing that the variables occurring are strings or functions of strings. This is particularly interesting in our Lam_η(,c,a)se. We can define a de-Bruijn style index through the natural number parameter to the function, and then print nested lambda abstractions. We can now "reach under" the argument f because our PHOAS parameter causes f to have the type String for its first argument. Once we pass a string to f we can then manipulate the return expression as needed.

Definition 35 (A pretty printer for $Expr_\eta$).

$$
\begin{aligned}
&\mathsf{pretty}_\eta : \mathsf{Nat} \to \mathsf{Expr}_\eta\ (\lambda\, \mathsf{x} : \mathsf{Type}\,.\,\mathsf{String})\ \mathsf{A} \to \mathsf{String}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Lit}_\eta(\mathsf{t}_1, \mathsf{x})) &&= \mathit{show}(\mathsf{x})\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Var}_\eta(\mathsf{t}_1, \mathsf{x})) &&= \text{“(Var ”} +\!\!+\ \mathsf{x} +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Lam}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{f})) &&=\\
&\qquad \text{Let } x = \text{“x”} +\!\!+ \mathit{show}(\mathsf{k})\\
&\qquad \text{in “(Lam ”} +\!\!+\ x +\!\!+ \text{“}\Rightarrow\text{”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{S}(\mathsf{k}), \mathsf{f}(x)) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{App}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{f}, \mathsf{x})) &&= \text{“(App ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{f}) +\!\!+ \text{“ ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{x}) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Add}_\eta(\mathsf{t}_1, \mathsf{x}, \mathsf{y})) &&= \text{“(”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{f}) +\!\!+ \text{“ + ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{x}) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Const}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}, \mathsf{y})) &&= \text{“(Const ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{x}) +\!\!+ \text{“ ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{y}) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{Pair}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x}, \mathsf{y})) &&= \text{“(”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{x}) +\!\!+ \text{“,”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{y}) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{ELeft}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{x})) &&= \text{“(Left ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{x}) +\!\!+ \text{“)”}\\
&\mathsf{pretty}_\eta(\mathsf{k}, \mathsf{ERight}_\eta(\mathsf{t}_1, \mathsf{t}_2, \mathsf{y})) &&= \text{“(Right ”} +\!\!+ \mathsf{pretty}_\eta(\mathsf{k}, \mathsf{y}) +\!\!+ \text{“)”}
\end{aligned}
$$

While we can reach under the lambda binders for a specific type with this PHOAS implementation, it carries with it a cost. We must now “carry around” the V parameter anywhere throughout our program that we suspect we may wish to eventually print or evaluate an expression. This includes inside zippers and lists. We will redefine these types slightly differently to ease our proof obligations while carrying around the PHOAS parameter where necessary.

4.4 Zipper Correctness By Construction

We begin our extended definitions of zippers to accommodate the PHOAS parameter by defining improved indexed families for traversal along the tree. We say these are improved as the index has changed from Type to $\mathsf{Maybe\ Type}$. This changes the burden of proof from one of proving that expression values are uninhabited, to a simpler one of proving that constructor tag values are not equal. We can see these changes in definition 36

Definition 36 (Traversal families for movement along a $Zipper_\eta$).

$$
\begin{aligned}
&\mathsf{Right}_\eta\ : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{Maybe\ Type}\\
&\mathsf{Right}_\eta(\mathsf{e}) = \begin{cases}
\mathsf{Just}(\mathsf{B}), & \text{if } \mathsf{e} = \mathsf{Pair}_\eta(\mathsf{a}, \mathsf{b}, \mathsf{x}, \mathsf{y}).\\
\mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{App}_\eta(\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{x}).\\
\mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{Add}_\eta(\mathsf{a}, \mathsf{x}, \mathsf{y}).\\
\mathsf{Just}(\mathsf{B}), & \text{if } \mathsf{e} = \mathsf{Const}_\eta(\mathsf{a}, \mathsf{b}, \mathsf{x}, \mathsf{y}).\\
\mathsf{Nothing}, & \text{otherwise.}
\end{cases}
\end{aligned}
$$

$$\mathsf{Left}_\eta \ : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{Maybe\ Type}$$

$$\mathsf{Left}_\eta(\mathsf{e}) = \begin{cases} \mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{Pair}_\eta(\mathsf{a},\mathsf{b},\mathsf{x},\mathsf{y}). \\ \mathsf{Just}(\mathsf{A} \to \mathsf{B}), & \text{if } \mathsf{e} = \mathsf{App}_\eta(\mathsf{a},\mathsf{b},\mathsf{f},\mathsf{x}). \\ \mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{Add}_\eta(\mathsf{a},\mathsf{x},\mathsf{y}). \\ \mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{Const}_\eta(\mathsf{a},\mathsf{b},\mathsf{x},\mathsf{y}). \\ \mathsf{Nothing}, & \text{otherwise.} \end{cases}$$

$$\mathsf{Down}_\eta \ : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{Maybe\ Type}$$

$$\mathsf{Down}_\eta(\mathsf{e}) = \begin{cases} \mathsf{Just}(\mathsf{A}), & \text{if } \mathsf{e} = \mathsf{ELeft}_\eta(\mathsf{a},\mathsf{b},\mathsf{x}). \\ \mathsf{Just}(\mathsf{B}), & \text{if } \mathsf{e} = \mathsf{ERight}_\eta(\mathsf{a},\mathsf{b},\mathsf{x}). \\ \mathsf{Nothing}, & \text{otherwise.} \end{cases}$$

We alter the Ctx_η to index by values of $\mathsf{Just}(\mathsf{Type})$, along with indexing by the PHOAS parameter. This further reduces the chance of programmer error from writing an invalid context. The cases where the appropriate function family returns $\mathsf{Nothing}$ cannot arise.

Definition 37 (A context type, Ctx_η that is correct by construction)**.** Types:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Maybe\ A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Ctx}_\eta\ \mathsf{V}\ (\mathsf{Maybe\ A}) : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{Maybe\ A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Root}_\eta : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Just}(\mathsf{A})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Just}(\mathsf{A})\end{array}}{\Gamma \vdash \mathsf{L}_\eta(\mathsf{x},\mathsf{ctx}) : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Left}_\eta(\mathsf{A})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Just}(\mathsf{A})\end{array}}{\Gamma \vdash \mathsf{R}_\eta(\mathsf{x},\mathsf{ctx}) : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Right}_\eta(\mathsf{A})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{V}\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Just}(\mathsf{A})\end{array}}{\Gamma \vdash \mathsf{D}_\eta(\mathsf{x},\mathsf{ctx}) : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Down}_\eta(\mathsf{A})}$$

Our definition of the zipper is straightforward, only extending the type with the additional index for the PHOAS parameter. We also allow our Zip_η function to take a value of $\mathsf{TypeRep}_\eta$ to simplify some operations going forward.

Definition 38 ($Zipper_\eta$ over $Expr_\eta$ and Ctx_η types)**.** Types:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{t} : \mathsf{TypeRep}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{x} : \mathsf{Expr}_\eta\ \mathsf{A} \qquad \Gamma \vdash \mathsf{ctx} : \mathsf{Ctx}_\eta\ \mathsf{V}\ \mathsf{Just}(\mathsf{A})\end{array}}{\Gamma \vdash \mathsf{Zip}_\eta(\mathsf{t}, \mathsf{x}, \mathsf{ctx}) : \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A}}$$

4.5 Random Well-Typed Crossover Positions

Now that we have a zipper type carrying a PHOAS index we may turn our attention towards the mechanics of implementing a type safe crossover. We saw in Chapter 2 (2.3.1) that crossover selects randomly the node in each tree. It follows then that a type preserving crossover will select randomly from those nodes that happen to be of a particular type. We may represent all possible nodes of a given type as a list of zippers that all contain a focus of that particular type. Once again we must index this otherwise ordinary list data type by the PHOAS index as we see in definition 39.

Definition 39 (A list data type with an additional PHOAS index)**.**

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}}{\Gamma \vdash \mathsf{PNil}_\eta : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}$$

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}{\Gamma \vdash \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs}) : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}$$

We must also describe then common list processing functions and proof data types in order to work with a PHOAS indexed list. We first define a type that gives us a proof by construction. We saw from Chapter 2 (2.1.7) that types may be represented as propositions, which are proven (or witnessed) by corresponding programs (or values). We now give a concrete example of this in the $\mathsf{InPList}_\eta$ type given in definition 40, which represents a proof that some natural number is a valid index into the list. $\mathsf{InPList}_\eta$ is indexed by a Nat and our PList_η. It is easy enough to envisage a proof by cases approach where we consider a natural number n as a valid index into the list xs.

- n is 0 and xs is empty: This is false as zero is not a valid index into the empty list.
- n is the successor of k and xs is empty: Also false.
- n is 0 and the list is non-empty: True, zero is always a valid index in a non-empty list.
- n is the successor k and xs is non-empty: This may only be true by induction on the length of xs.

The "trick" then is to disallow the false cases, and only have values in our $\mathsf{InPList}_\eta$ type that represent the true cases. In both of these cases the list is non-empty. Here InNow_η returns an $\mathsf{InPList}_\eta$ indexed by zero and a non-empty list, and $\mathsf{InAfter}_\eta$ expects another proof type where the indexing natural number has increased by one.

Definition 40 ($InPList_\eta$ represents a proof by construction of valid $PList_\eta$ indices)**.**
xs.
Types:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{k} : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}{\Gamma \vdash \mathsf{InPList}_\eta\ \mathsf{k}\ \mathsf{xs} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}{\Gamma \vdash \mathsf{InNow}_\eta : \mathsf{InPList}_\eta\ \mathsf{Z}\ \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs})}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \\ \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{k} : \mathsf{Nat} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}\end{array}}{\Gamma \vdash \mathsf{InAfter}_\eta(\mathsf{xs}) : \mathsf{InPList}_\eta\ \mathsf{S}(\mathsf{k})\ \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs})}$$

Here in definition 41 we see a similar proof structure that captures the proposition that a list is non-empty.

Definition 41 (A constructive proof that a $PList_\eta$ is not empty)**.**

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}{\Gamma \vdash \mathsf{NonEmpty}_\eta\ \mathsf{xs} : \mathsf{Type}}$$

Values:

$$\frac{\Gamma \vdash \mathsf{V} : \mathsf{Type} \to \mathsf{Type} \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma \vdash \mathsf{x} : \mathsf{A} \qquad \Gamma \vdash \mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}}{\Gamma \vdash \mathsf{IsNonEmpty}_\eta : \mathsf{NonEmpty}_\eta\ \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs})}$$

We can define the append function on lists as normal (definition 42).

Definition 42 (The append function for $PList_\eta$).

$$\begin{aligned}
&\mathsf{++}_\eta : \mathsf{PList}_\eta\ \mathsf{A}\ \to \mathsf{PList}_\eta\ \mathsf{A}\ \to \mathsf{PList}_\eta\ \mathsf{A} \\
&\mathsf{PNil}_\eta \mathrel{\mathsf{++}_\eta} \mathsf{ys} = \mathsf{ys} \\
&\mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs}) \mathrel{\mathsf{++}_\eta} \mathsf{ys} = \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs} \mathrel{\mathsf{++}_\eta} \mathsf{ys})
\end{aligned}$$

From our earlier definition 40, we now define an indexing function into a PList_η. We say that this function is *verified* with respect to functional correctness as it demands that a proof be satisfied that the index is valid.

Definition 43 (A verified index function for $PList_\eta$).

$$\begin{aligned}
&\mathsf{index}_\eta : \mathsf{Nat} \to (\mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ \mathsf{A}) \to \mathsf{NonEmpty}_\eta\ \mathsf{xs} \to \mathsf{A} \\
&\mathsf{index}_\eta(\mathsf{Z}, \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs}), \mathsf{pf}) = \mathsf{x} \\
&\mathsf{index}_\eta(\mathsf{S}(\mathsf{k}), \mathsf{PCons}_\eta(\mathsf{x}, \mathsf{xs}), \mathsf{InAfter}_\eta(\mathsf{p})) = \mathsf{index}_\eta(\mathsf{k}, \mathsf{xs}, \mathsf{p})
\end{aligned}$$

Now that we have defined common list operations over our PList_η type we define a function to flatten our zipped Expr_η trees into a PList_η of zippers, with one element for each focus. $\mathsf{flatten}_\eta$ (definition 44) considers Lit_η, Var_η, and Lam_η as the leaves or tips of the tree and capital types (A,B,U, $\mathsf{A} \times \mathsf{b}$ etc) are the types of the corresponding type index of the zipper. We flatten a zipper into a list of Σ types where the first element of the dependent pair is the specific type index of the zipper, and the second element is a pair of the $\mathsf{TypeRep}_\eta$ and the associated zipper. This allows us to easily filter such a list based on its run-time type representation values. Once again we omit the proofs for well-founded recursion for flatten. Each recursive call to the left and right sub-trees on binary expressions are clearly structurally smaller.

Definition 44 (Flatten a $Zipper_\eta$ into a $PList_\eta$ of Σ types).

Let S represent the set of all binary terms in language η.

$S = \{\mathsf{App}_\eta, \mathsf{Add}_\eta, \mathsf{Const}_\eta, \mathsf{Pair}_\eta\}$

$\forall s \in S,$ Let s_l denote the left sub-expression in s and let s_r denote the right sub-expression in s.

Let t denote some $\mathsf{TypeRep}_\eta\ \mathsf{A}$.

Let zr represent a zipper of s_r by focusing right on s in some context ctx.

Let zl represent a zipper of s_l by focusing left on s in some context ctx.

$\mathsf{zr} = \mathsf{Zip}_\eta(\mathsf{t}, s_r, \mathsf{R}_\eta(s, \mathsf{ctx}))$

$\mathsf{zl} = \mathsf{Zip}_\eta(\mathsf{t}, s_l, \mathsf{L}_\eta(s, \mathsf{ctx}))$

$$\mathsf{flatten}_\eta : \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A} \to \mathsf{PList}_\eta\ \mathsf{V}\ (\Sigma\, \mathsf{a} : \mathsf{Type}\,.\, \mathsf{TypeRep}_\eta\ \mathsf{a} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{a})$$

$$\mathsf{flatten}_\eta(\mathsf{Zip}_\eta(\mathsf{t},\mathsf{e},\mathsf{c})) = \begin{cases} \mathsf{PCons}_\eta((\mathsf{A},(\mathsf{t},\mathsf{Zip}_\eta(\mathsf{t},\mathsf{e},\mathsf{c}))),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} = \mathsf{Lit}_\eta(\mathsf{u},\mathsf{x}) \\ \mathsf{PCons}_\eta((\mathsf{A},(\mathsf{t},\mathsf{Zip}_\eta(\mathsf{t},\mathsf{e},\mathsf{c}))),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} = \mathsf{Var}_\eta(\mathsf{u},\mathsf{x}) \\ \mathsf{PCons}_\eta((\mathsf{A}\to\mathsf{B},(\mathsf{t},\mathsf{Zip}_\eta(\mathsf{t},\mathsf{e},\mathsf{c}))),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} = \mathsf{Lam}_\eta(\mathsf{u},\mathsf{v},\mathsf{f}) \\ \mathsf{zd} \mathbin{++_\eta} \mathsf{PCons}_\eta((\mathsf{A}+\mathsf{B},(\mathsf{t},\mathsf{Zip}_\eta(\mathsf{u},\mathsf{e},\mathsf{c}))),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} = \mathsf{ELeft}_\eta(\mathsf{t},\mathsf{u},\mathsf{x}) \\ \quad \text{where } \mathsf{zd} = \mathsf{Zip}_\eta(\mathsf{t},\mathsf{x},\mathsf{D}_\eta(\mathsf{e},\mathsf{c})) & \\ \mathsf{zd} \mathbin{++_\eta} \mathsf{PCons}_\eta((\mathsf{A}+\mathsf{B},(\mathsf{t},\mathsf{Zip}_\eta(\mathsf{u},\mathsf{e},\mathsf{c}))),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} = \mathsf{ERight}_\eta(\mathsf{t},\mathsf{u},\mathsf{x}) \\ \quad \text{where } \mathsf{zd} = \mathsf{Zip}_\eta(\mathsf{u},\mathsf{x},\mathsf{D}_\eta(\mathsf{e},\mathsf{c})) & \\ \mathsf{zl} \mathbin{++_\eta} \mathsf{zr} \mathbin{++_\eta} \mathsf{PCons}_\eta((\mathsf{U},(\mathsf{u},\mathsf{Zip}_\eta(\mathsf{t},\mathsf{e},\mathsf{c})),\mathsf{PNil}_\eta) & \text{if } \mathsf{e} \in S. \\ \quad \text{where } \mathsf{U} \text{ is the type index of } \mathsf{u}. & \end{cases}$$

4.6 Verified Crossover

We may now define the functions necessary for achieving a type-preserving crossover. We first define $\mathsf{selectNode}_\eta$ that guarantees a valid index into any non-empty PList_η given any natural number, and returns that index. We may achieve a valid index by taking the modulo of natural number n by the length of the list. Depending on which particular language this function may be implemented in, the burden of proof to show that the modulo operation always results in a valid index (pf_2) may be quite complex. We refer interested readers to appendix A.4 and appendix A.6 for the re-implementation of the modulo function using finite sets (Bove and Dybjer, 2008), and accompanying proofs which were necessary to implement this function in the Idris programming language.

Definition 45 ($selectNode_\eta$ selects a pair of zippers from a $PList_\eta$ given any natural number n)**.**

$$\mathsf{selectNode}_\eta : \mathsf{Nat} \to (\mathsf{xs} : \mathsf{PList}_\eta\ \mathsf{V}\ ) \to \mathsf{NonEmpty}_\eta\ \mathsf{xs} \to \Sigma\, \mathsf{a} : \mathsf{Type}\,.\,(\mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{a} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{a})$$
$$\mathsf{selectNode}_\eta(\mathsf{n},\mathsf{xs},\mathsf{pf}) = \mathsf{index}_\eta(mod(\mathsf{n}, length(\mathsf{xs})),\mathsf{xs},\mathsf{pf}_2)$$

Our $\mathsf{flatten}_\eta$ function gives us the $Sigma$ type dependent pair where the second element is a product of a zipper and it's type representation. To compare $\mathsf{TypeRep}_\eta$ values for equality we can take the Cartesian product of two flattened zippers to give us a $\mathsf{PList}_\eta\ \mathsf{V}\ (\Sigma\, \mathsf{y} : \mathsf{Type}\,.\,(\mathsf{TypeRep}_\eta\ \mathsf{y} \times \mathsf{Zipper}_\eta\ \mathsf{y}\ ) \times \Sigma\, \mathsf{z} : \mathsf{Type}\,.\,(\mathsf{TypeRep}_\eta\ \mathsf{z} \times \mathsf{Zipper}_\eta\ \mathsf{z}\ ))$. This will ensure that every possible point of comparison between the two trees exists in the list. We may then define a function that returns only those elements in the list where the type representation matches. It is important to note that the return type ensures that the two zippers now have the same type index. Without this crossover would not be possible. We call our function $\mathsf{typeRepEq}_\eta$ in definition 46. Here f is a function that includes the pair in the list if the type representations are equal. We refer interested readers to appendix A.7 for our implementation of f in Idris. This implementation is complex and involves dependent pattern matching and a number of proofs. Other dependently typed languages may be able to express f more succinctly.

Definition 46 (*typeRepEq$_\eta$* returns a list of valid crossover points)**.**

$$\begin{aligned}\mathsf{typeRepEq}_\eta : \mathsf{PList}_\eta\ \mathsf{V}\ (\Sigma\, \mathsf{y} : \mathsf{Type}\,.\,(\mathsf{TypeRep}_\eta\ \mathsf{y} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{y}) \times \Sigma\, \mathsf{z} : \mathsf{Type}\,.\,(\mathsf{TypeRep}_\eta\ \mathsf{z} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{z})) \to \\ \mathsf{PList}_\eta\ \mathsf{V}\ \Sigma\, \mathsf{a} : \mathsf{Type}\,.\,(\mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{a} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A}) \\ \mathsf{typeRepEq}_\eta(\mathsf{xs}) = \mathsf{foldr}(\mathsf{f}, \mathsf{PNil}_\eta, \mathsf{xs})\end{aligned}$$

The definition of a type preserving crossover is now trivial. Given a Σ type of a pair of zippers indexed by the first element, we may then return a pair of zippers indexed by that element. We then only ensure that we exchange the foci between the two zippers as in definition 47.

Definition 47 (*xover$_\eta$*, A type preserving crossover)**.**

$$\begin{aligned}\mathsf{xover}_\eta : (\mathsf{z} : \Sigma\, \mathsf{a} : \mathsf{Type}\,.\,(\mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A} \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{A})) \to (\mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{fst}\,(\mathsf{z}) \times \mathsf{Zipper}_\eta\ \mathsf{V}\ \mathsf{fst}\,(\mathsf{z})) \\ \mathsf{xover}_\eta(\mathsf{t}_1, (\mathsf{Zip}_\eta(\mathsf{t}_1, \mathsf{e}_1, \mathsf{ctx}_1), \mathsf{Zip}_\eta(\mathsf{t}_2, \mathsf{e}_2, \mathsf{ctx}_2)) = (\mathsf{Zip}_\eta(\mathsf{t}_1, \mathsf{e}_2, \mathsf{ctx}_1), \mathsf{Zip}_\eta(\mathsf{t}_2, \mathsf{e}_1, \mathsf{ctx}_2))\end{aligned}$$

4.7 Proof of Concept

We provide a proof of concept of a type-preserving crossover operation implemented in the Idris programming language (version 1.3.2) (Brady, 2017). We chose Idris due to the syntactic similarity it shares with Haskell, the support it offers for programming with dependent types, and to put a fairly recently developed language through its paces. The source code for our implementation can be found in appendix A. Figure 12 demonstrates the possible crossover positions for one of the examples given in section A.8, appendix A. There are 17 possible total crossover positions, 4×4 possible exchanges for natural numbers, and 1 possible exchange for strings.

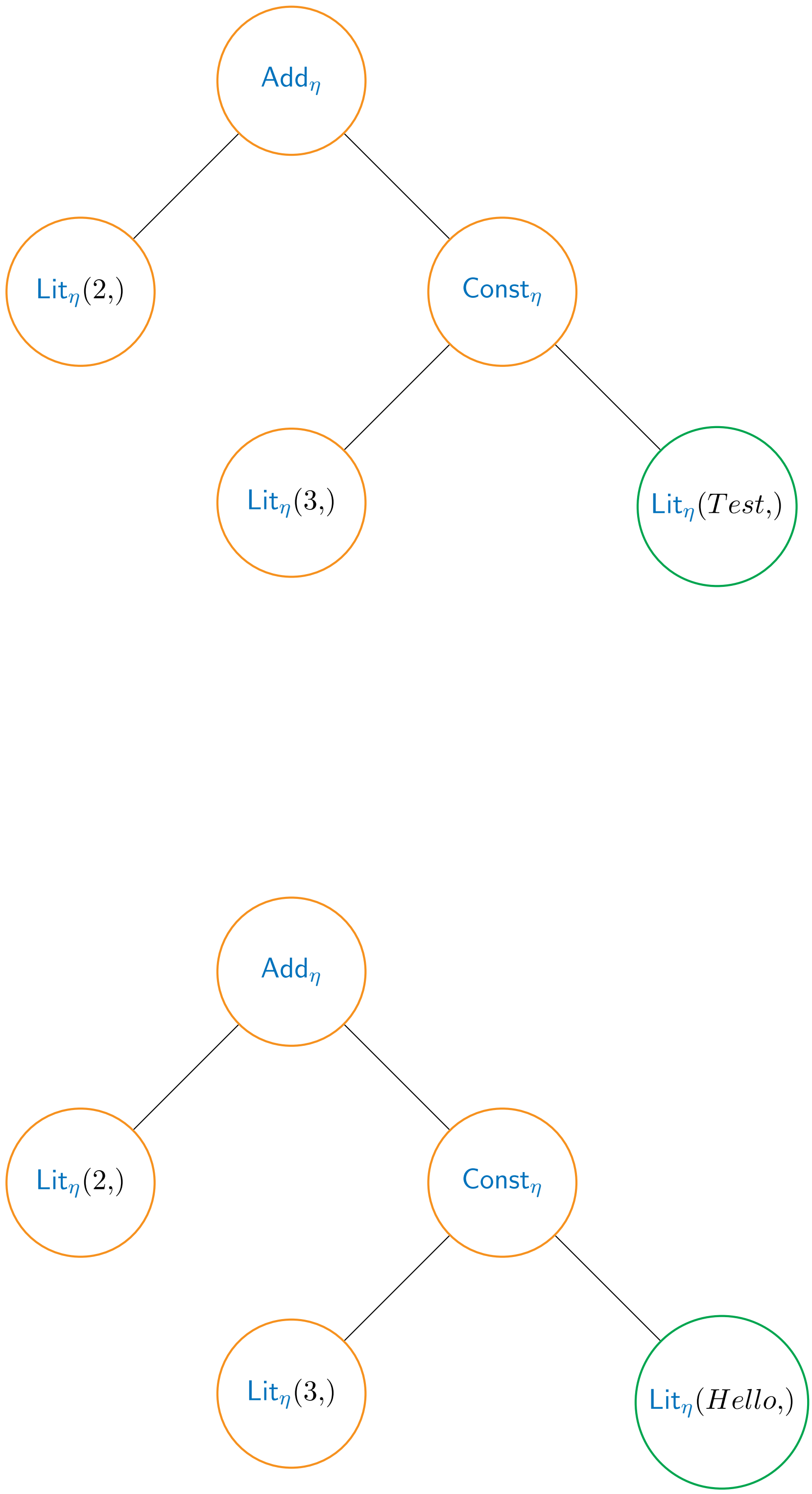


Figure 12: The well-typed crossover points between two expressions.
The orange nodes for natural numbers.
The green nodes for strings.

5 CONCLUSION

We have shown that Hamana and Fiore's specification for a zipper over simple inductive families, such as the GADT of well-typed terms does not hold by way of a counter example that will not type check. We provide an alternative structure to Hamana and Fiore's zipper that allows us to traverse well-typed GADT-embedded expression trees and perform type correct substitution. We also have given a well-typed expressive object language η and defined a verified type-preserving crossover operation using our zipper-like structure. We have the expressivity of parametrically polymorphic and higher-kinded terms that Yu and Clack show in PolyGP, along with the verification and type-safety guarantees from Diehl's stack-based genetic program. Finally we have provided a working proof of concept of our type-preserving crossover operation in the Idris programming language, along with any necessary proofs required for the implementation.

5.1 Further Work

For further interesting research there are a plethora of approaches remaining. Readers interested in embedded syntax and dependent types may examine an implementation of our zipper structure over other forms of well-typed object languages representations such as the well-typed interpreter (Augustsson and Carlsson, 1999). Readers who are interested in genetic programming may define and implement a mutation operation over our higher order or parametrically higher order abstract syntax trees. Readers interested in general functional programming may investigate an open world approach to our run-time type representations (Peyton Jones et al., 2016). Interested readers may also investigate the type-theoretic and category-theoretic properties of our zipper-like structure, particularly with regard to the derivatives of one-hole contexts. Finally readers with an interest in more experimental computer science may wish to apply our well-typed crossover operation to supervised learning problems and give experimental results.

REFERENCES

C. Allen and J. Moronuki. Haskell programming from first principles. *Gumroad (ebook)*, 2017.

R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2009.

L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*, 1999.

F. Binard and A. Felty. An abstraction-based genetic programming system. In *Proceedings of the 9th annual conference companion on Genetic and evolutionary computation*, pages 2415–2422. ACM, 2007.

R. Bird. *Thinking functionally with Haskell.* Cambridge University Press, 2014.

A. Bove and P. Dybjer. Dependent types at work. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 57–99. Springer, 2008.

E. Brady. *Type-driven development with Idris.* Manning Publications Company, 2017.

F. Briggs and M. O'Neill. Functional genetic programming and exhaustive program search with combinator expressions. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 12(1):47–68, 2008.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM, 2008.

A. Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2013.

L. Diehl. Verified stack-based genetic programming via dependent types. In *Proceedings of AAIP 2011 4th International Workshop on Approaches and Applications of Inductive Programming*, page 17, 2011.

A. P. Engelbrecht. *Computational Intelligence: An Introduction.* Wiley Publishing, 2007.

M. Hamana and M. Fiore. A foundation for gadts and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 59–70. ACM, 2011.

J. H. Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

G. Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

P. Johann and N. Ghani. Foundations for structured programming with gadts. In *ACM SIGPLAN Notices*, volume 43, pages 297–308. ACM, 2008.

J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

T. Křen and R. Neruda. Generating lambda term individuals in typed genetic programming using forgetful A*. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1847–1854. IEEE, 2014.

M. Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

C. McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *International School on Advanced Functional Programming*, pages 228–266. Springer, 1995.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

D. J. Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2): 199–230, 1995.

U. Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

T. Perkis. Stack-based genetic programming. In *Proceedings of the first ieee conference on evolutionary computation. ieee world congress on computational intelligence*, pages 148–153. IEEE, 1994.

S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. 2010.

S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical report, Technical Report MS-CIS-05-26, Univ. of Pennsylvania, 2004.

S. Peyton Jones, S. Weirich, R. A. Eisenberg, and D. Vytiniotis. A reflection on types. In *A List of Successes That Can Change the World*, pages 292–317. Springer, 2016.

F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM sigplan notices*, volume 23, pages 199–208. ACM, 1988.

B. C. Pierce and C. Benjamin. *Types and programming languages*. MIT press, 2002.

B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

C. Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1-2):11–49, 2000.

C. Tomé Cortiñas and W. Swierstra. From algebra to abstract machine: a verified generic construction. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, pages 78–90. ACM, 2018.

P. Wadler. Theorems for free! In *FPCA*, volume 89, pages 347–359. Citeseer, 1989.

P. Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ACM SIGPLAN Notices*, volume 38, pages 249–262. ACM, 2003.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices*, volume 38, pages 224–235. ACM, 2003.

A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices*, volume 44, pages 233–244. ACM, 2009.

T. Yu and C. Clack. Polygp: A polymorphic genetic programming system in haskell. *Genetic Programming*, 98, 1998.

A SOURCE CODE

A.1 Run-Time Type Representations: Typeable.idr

```
%default total

||| A run-time representation for a type.
public export
data TypeRep : Type -> Type where
  TBool : TypeRep Bool
  TNat : TypeRep Nat
  TInteger : TypeRep Integer
  TString : TypeRep String
  TFunc : TypeRep a -> TypeRep b -> TypeRep (a -> b)
  TPair : TypeRep a -> TypeRep b -> TypeRep (a, b)
  TSum : TypeRep a -> TypeRep b -> TypeRep (Either a b)

||| A Pi Type to extract the underlying type of the run-time representation.
||| TODO: Is this used anywhere?
public export
ExtractTy : (TypeRep a) -> Type
ExtractTy {a} t = a

||| Boolean equality between type representations
||| Unfortunately Idris is not smart enough to correctly match
||| all the false cases by default.
public export
beqType : TypeRep a -> TypeRep b -> Bool
beqType TBool TBool = True
beqType TBool TNat = False
beqType TBool TInteger = False
beqType TBool TString = False
beqType TBool (TFunc x y) = False
beqType TBool (TPair x y) = False
beqType TBool (TSum x y) = False
beqType TNat TBool = False
beqType TNat TNat = True
beqType TNat TInteger = False
```

```
beqType TNat TString = False
beqType TNat (TFunc x y) = False
beqType TNat (TPair x y) = False
beqType TNat (TSum x y) = False
beqType TInteger TBool = False
beqType TInteger TNat = False
beqType TInteger TInteger = True
beqType TInteger TString = False
beqType TInteger (TFunc x y) = False
beqType TInteger (TPair x y) = False
beqType TInteger (TSum x y) = False
beqType TString TBool = False
beqType TString TNat = False
beqType TString TInteger = False
beqType TString TString = True
beqType TString (TFunc x y) = False
beqType TString (TPair x y) = False
beqType TString (TSum x y) = False
beqType (TFunc x y) TNat = False
beqType (TFunc x y) TBool = False
beqType (TFunc x y) TInteger = False
beqType (TFunc x y) TString = False
beqType (TFunc x y) (TFunc a b) = beqType x a && beqType y b
beqType (TFunc x y) (TPair a b) = False
beqType (TFunc x y) (TSum a b) = False
beqType (TPair x y) TNat = False
beqType (TPair x y) TBool = False
beqType (TPair x y) TInteger = False
beqType (TPair x y) TString = False
beqType (TPair x y) (TFunc a b) = False
beqType (TPair x y) (TPair a b) = beqType x a && beqType y b
beqType (TPair x y) (TSum a b) = False
beqType (TSum x y) TNat = False
beqType (TSum x y) TBool = False
beqType (TSum x y) TInteger = False
beqType (TSum x y) TString = False
beqType (TSum x y) (TFunc a b) = False
beqType (TSum x y) (TPair a b) = False
beqType (TSum x y) (TSum a b) = beqType x a && beqType y b

||| Pretty printing for runtime representations of Types.
public export
prettyTy : TypeRep a -> String
```

```
prettyTy TBool = "Bool"
prettyTy TNat = "Nat"
prettyTy TInteger = "Integer"
prettyTy TString = "String"
prettyTy (TFunc x y) = "(" ++ prettyTy x ++ " -> " ++ prettyTy y ++ ")"
prettyTy (TPair x y) = "(" ++ prettyTy x ++ ", " ++ prettyTy y ++ ")"
prettyTy (TSum x y) = "(Either " ++ prettyTy x ++ " " ++ prettyTy y ++ ")"
```

A.2 Well-Typed Expression Trees: Expr.idr

```
import Typeable.Typeable

%default total

||| PHOAS Expression Langauge
||| see 'Typeable' for TypeRep explaination.
public export
data Expr : (v : Type -> Type) -> (a : Type) -> Type where
  Lit : Show a => {t1 : TypeRep a} -> a -> Expr v a
  Var : {t1 : TypeRep a} -> v a -> Expr v a
  Lam : {t1 : TypeRep a} -> {t2 : TypeRep b} ->
        (v a -> Expr v b) -> Expr v (a -> b)
  App : {t1 : TypeRep a} -> {t2 : TypeRep b} -> Expr v (a -> b) ->
        Expr v a -> Expr v b
  Add : Num a => Expr v a -> {t1 : TypeRep a} -> Expr v a -> Expr v a
  Cnst : {t1 : TypeRep a} -> {t2 : TypeRep b} -> Expr v a -> Expr v b ->
         Expr v a
  EPair : {t1 : TypeRep a} -> {t2 : TypeRep b} -> Expr v a -> Expr v b ->
          Expr v (a, b)
  ELeft : {t1 : TypeRep a} -> {t2 : TypeRep b} -> Expr v a ->
          Expr v (Either a b)
  ERight : {t1 : TypeRep a} -> {t2 : TypeRep b} -> Expr v b ->
           Expr v (Either a b)

||| Weaken a non-positive function into a strictly positive one.
public export
weaken : {t1 : TypeRep a} -> (Expr v a -> Expr v b) ->
         (v a -> Expr v b)
weaken {t1} f = \x => f (Var {t1} x)

||| Smart constructor for the 'Lam' case so weaken doesn't need to be
||| explicitly called.
```

```
public export
lam : {t1 : TypeRep a} -> {t2 : TypeRep b} -> (Expr v a -> Expr v b) ->
      Expr v (a -> b)
lam {t1}{t2} f = Lam {t1} {t2} (weaken {t1} f)

||| Polymorphic evaluator.
public export
eval : Expr (\x => x) a -> a
eval (Lit x) = x
eval (Var x) = x
eval (Lam f) = \x => eval (f x)
eval (App f x) = (eval f) (eval x)
eval (Add x y) = (eval x) + (eval y)
eval (Cnst x y) = const (eval x) (eval y)
eval (EPair x y) = (eval x, eval y)
eval (ELeft {b} x) = Left (eval x)
eval (ERight {a} x) = Right (eval x)

||| A HOAS encoding of `Expr a -> String` is too polymorphic
||| to reach under the lambda binding to access the body.
||| So here we pretty print with PHOAS.
||| (Needed to identify expressions/typereps in the IO monad)
public export
prettyE : Nat -> Expr (\_ => String) a -> String
prettyE k (Lit x) = show x
prettyE k (Var x) = "(Var " ++ x ++ ")"
prettyE k (Lam f) =
  let x = "x" ++ (show k)
  in "(Lam " ++ x ++ " => " ++ prettyE (succ k) (f x) ++ ")"
prettyE k (App f x) = "(App " ++ prettyE k f ++ " " ++ prettyE k x ++ ")"
prettyE k (Add x y) = "(" ++ prettyE k x ++ " + " ++ prettyE k y ++ ")"
prettyE k (Cnst x y) = "(Const " ++ prettyE k x ++ " " ++ prettyE k y ++ ")"
prettyE k (EPair x y) = "(Pair " ++ prettyE k x ++ " " ++ prettyE k y ++ ")"
prettyE k (ELeft x) = "(Left " ++ prettyE k x ++ ")"
prettyE k (ERight x) = "(Right " ++ prettyE k x ++ ")"
```

A.3 Zipper-Like Traversal of Expressions: Zipper.idr

```
import Expr.Expr
import Typeable.Typeable

%default total
```

```
||| The indexed family calculating the type when walking left on an Expr.
public export
GoLeft : Expr v a -> Maybe Type
GoLeft (Lit x) = Nothing
GoLeft (Var x) = Nothing
GoLeft (Lam f) = Nothing
GoLeft (EPair {a} x y) = Just a
GoLeft (ELeft x) = Nothing
GoLeft (ERight x) = Nothing
GoLeft (App {a}{b} f x) = Just (a -> b)
GoLeft (Add {a} x y) = Just a
GoLeft (Cnst {a} x y) = Just a

||| The indexed family calculating the type when walking right on an Expr.
public export
GoRight : Expr v a -> Maybe Type
GoRight (Lit x) = Nothing
GoRight (Var x) = Nothing
GoRight (Lam f) = Nothing
GoRight (EPair {b} x y) = Just b
GoRight (ELeft x) = Nothing
GoRight (ERight x) = Nothing
GoRight (App f {a} x) = Just a
GoRight (Add x {a} y) = Just a
GoRight (Cnst x {b} y) = Just b

||| The indexed family calculating the type when walking down on an Expr.
public export
GoDown : Expr v a -> Maybe Type
GoDown (Lit x) = Nothing
GoDown (Var x) = Nothing
GoDown (Lam {a}{b} f) = Nothing
GoDown (App f x) = Nothing
GoDown (Add x y) = Nothing
GoDown (Cnst x y) = Nothing
GoDown (EPair x y) = Nothing
GoDown (ELeft {a}{b} x) = Just a
GoDown (ERight {b}{a} x) = Just b

||| An ordinary Cons list but indexed by the PHOAS parameter so I don't
||| need to carry it around inside Sigma types everwhere.
public export
```

```
data PHOASList : (v : Type -> Type) -> (a : Type) -> Type where
  Nil  : PHOASList v a
  (::) : a -> PHOASList v a -> PHOASList v a

||| Append for PHOASList.
public export
(++) : PHOASList v a -> PHOASList v a -> PHOASList v a
(++) Nil xs = xs
(++) (x :: xs) ys = x :: xs ++ ys

||| InBounds for PHOASLists
public export
data InPList : (k : Nat) -> (xs : PHOASList v a) -> Type where
  InNow : InPList Z (x :: xs)
  InAfter : InPList k xs -> InPList (S k) (x :: xs)

||| Index for PHOASLists
public export
index : (n : Nat) -> (xs : PHOASList v a) -> {auto prf: InPList n xs} -> a
index Z (x :: xs) = x
index (S k) (x :: xs) {prf = InAfter p} = index k xs

||| The context contains all information necessary to rebuild the expression
||| tree.
public export
data Context : (Type -> Type) -> Maybe Type -> Type where
  Root : Context v (Just a)
  L : (x : Expr v a) -> Context v (Just a) -> Context v (GoLeft x)
  R : (x : Expr v a) -> Context v (Just a) -> Context v (GoRight x)
  D : (x : Expr v a) -> Context v (Just a) -> Context v (GoDown x)

||| A modified Hamana-Fiore style dependent-zipper.
public export
data Zipper : (Type -> Type) -> Type -> Type where
  Zip : {z1 : TypeRep a} -> Expr v a -> Context v (Just a) -> Zipper v a

||| extract the focus from a Zipper.
public export
extract : Zipper v a -> Expr v a
extract (Zip e c) = e

||| A convenience function to wrap a zipper up in a Sigma type.
public export
```

```
wrap : Zipper v a -> (v : (Type -> Type) ** a : Type ** Zipper v a)
wrap {v} {a} z = (v ** a ** z)

||| Rebuild a tree.
public export
up : (a : Type ** Zipper v a) -> (b : Type ** Zipper v b)
up (t ** pf) =
  case pf of
    (Zip {z1} e {a} Root) => (a ** Zip e {z1} Root)
    (Zip e (L (Lit x) c)) impossible
    (Zip e (L (Lam f) c)) impossible
    (Zip e (L (App {t1}{t2} f x) c)) => (_ ** (Zip {z1=t2} (App {t1}{t2} e x) c))
    (Zip e (L (Add {t1} x y) c)) => (_ ** (Zip {z1=t1} (Add {t1} e y) c))
    (Zip e (L (Cnst {t1}{t2}  x y) c)) => (_ ** (Zip {z1=t1} (Cnst {t1}{t2} e y) c))
    (Zip e (L (EPair {t1}{t2} {a}{b} x y) c)) =>
      ((a,b) ** (Zip {z1=TPair t1 t2} (EPair {t1}{t2} e y) c))
    (Zip e (L (ELeft x) c)) impossible
    (Zip e (L (ERight x) c)) impossible
    (Zip e (R (Lit x) c)) impossible
    (Zip e (R (Lam f) c)) impossible
    (Zip e (R (App {t1}{t2}{b} f x) c)) => (b ** (Zip {z1=t2} (App {t1}{t2} f e) c))
    (Zip e (R (Add {t1} {a} x y) c)) => (a ** (Zip {z1=t1} (Add {t1} x e) c))
    (Zip e (R (Cnst {t1}{t2}{a} x y) c)) =>
      (a ** (Zip {z1=t1} (Cnst {t1}{t2} x e) c))
    (Zip e (R (EPair {t1}{t2} {a}{b} x y) c)) =>
      ((a,b) ** (Zip {z1=TPair t1 t2} (EPair {t1}{t2} x e) c))
    (Zip e (R (ELeft x) c)) impossible
    (Zip e (R (ERight x) c)) impossible
    (Zip e (D (Lit x) c)) impossible
    (Zip e (D (Lam f) c)) impossible
    (Zip e (D (App f x) c)) impossible
    (Zip e (D (Add x y) c)) impossible
    (Zip e (D (Cnst x y) c)) impossible
    (Zip e (D (EPair x y) c)) impossible
    (Zip e (D (ELeft {t1}{t2} {a} {b} x) c)) =>
      (Either a b ** (Zip {z1=TSum t1 t2} (ELeft {t1}{t2} e) c))
    (Zip e (D (ERight {t1}{t2} {a} {b} x) c)) =>
      (Either a b ** (Zip {z1=TSum t1 t2} (ERight {t1} {t2} e) c))

||| A convenience function to move the focus all the way to the root
public export
top : (a ** Zipper v a) -> (b ** Zipper v b)
top (a ** (Zip {z1} e Root)) = (a ** (Zip {z1} e Root))
```

```
top (a ** (Zip {z1} e c)) =
  let s = assert_smaller (a ** (Zip e {z1} c)) (up (a ** (Zip {z1} e c)))
  in top s

||| Flatten a zipper into a list of all possible foci
public export
flatten : Zipper v a ->
          PHOASList v (x : Type ** (TypeRep x, Zipper v x))
flatten (Zip {z1} e@(Lit {t1} {a} x) c) = [(a ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(Var {t1} {a} x) c) = [(a ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(Lam {t1}{t2}{a}{b} f) c) = [(a -> b ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(App {t1}{t2} {a}{b} f x) c) =
  let zl = assert_total $ flatten (Zip {z1 = TFunc t1 t2} f (L e c))
      zr = assert_total $ flatten (Zip {z1=t1} x (R e c))
  in zl ++ zr ++ [(b ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(Add {t1}{a} x y) c) =
  let zl = assert_total $ flatten (Zip {z1=t1} x (L e c))
      zr = assert_total $ flatten (Zip {z1=t1} y (R e c))
  in zl ++ zr ++ [(a ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(Cnst {t1}{t2} x y) c) =
  let zl = assert_total $ flatten (Zip {z1=t1} x (L e c))
      zr = assert_total $ flatten (Zip {z1=t2} y (R e c))
  in zl ++ zr ++ [(a ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(EPair {t1}{t2}{a}{b} x y) c) =
  let zl = assert_total $ flatten (Zip {z1=t1} x (L e c))
      zr = assert_total $ flatten (Zip {z1=t2} y (R e c))
  in zl ++ zr ++ [((a, b) ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(ELeft {t1}{t2} {a}{b} x) c) =
  let zd = assert_total $ flatten (Zip {z1=t1} x (D e c))
  in zd ++ [(Either a b ** (z1, (Zip {z1} e c)))]
flatten (Zip {z1} e@(ERight {t1}{t2}{a}{b} x) c) =
  let zd = assert_total $ flatten (Zip {z1=t2} x (D e c))
  in zd ++ [(Either a b ** (z1, (Zip {z1} e c)))]

||| substitute in a new focus expression into a zipper
public export
subst : (x : (v : (Type -> Type) ** a : Type ** Zipper v a)) ->
        Expr (fst x) (fst (snd x)) -> Zipper (fst x) (fst (snd x))
subst (v ** x ** (Zip {z1} e' c)) e = Zip {z1} e c

||| Pretty printer for contexts
public export
prettyC : Context (\_ => String) a -> String
```

```
prettyC Root = " Root"
prettyC (L e c) = "(L " ++ prettyE 0 e ++ prettyC c ++ ")"
prettyC (R e c) = "(R " ++ prettyE 0 e ++ prettyC c ++ ")"
prettyC (D e c) = "(D " ++ prettyE 0 e ++ prettyC c ++ ")"

||| pretty printer for zippers
public export
prettyZ : Zipper (\_ => String) a -> String
prettyZ (Zip e c) = "Zip " ++ prettyE 0 e ++ " [C: " ++ prettyC c ++ "]"
```

A.4 A Reimplementation of Mod for Simpler Proofs: Mod.idr

```
import Data.Fin
%default total

||| helper function on finite sets rather than natural numbers
public export
modf : (x, y : Nat) -> {auto prf : GT y Z} -> Fin y
modf Z (S k) = FZ
modf (S k) (S j) with (strengthen (modf k (S j)))
  modf (S k) (S j) | Left bounds = FZ
  modf (S k) (S j) | Right rem = FS rem

||| mod on natural numbers using finite sets
public export
modfin : (x, y : Nat) -> {auto prf : GT y Z} -> Nat
modfin x y = finToNat (modf x y)
```

A.5 A List Data Type with a PHOAS Parameter: PHOASList.idr

```
%default total

||| The ordinary list data type, carrying around an extra PHOAS parameter
public export
data PHOASList : (v : Type -> Type) -> (a : Type) -> Type where
  Nil  : PHOASList v a
  (::) : a -> PHOASList v a -> PHOASList v a

||| Append on PHOASLists
```

```
public export
(++) : PHOASList v a -> PHOASList v a -> PHOASList v a
(++) Nil xs = xs
(++) (x :: xs) ys = x :: xs ++ ys

||| A proof that a given index is a valid index into a PHOASList
public export
data InPList : (k : Nat) -> (xs : PHOASList v a) -> Type where
  InNow : InPList Z (x :: xs)
  InAfter : InPList k xs -> InPList (S k) (x :: xs)

||| An index function into a PHOASList
public export
index : (n : Nat) -> (xs : PHOASList v a) -> {auto prf: InPList n xs} -> a
index Z (x :: xs) = x
index (S k) (x :: xs) {prf = InAfter p} = index k xs

||| Length defined for PHOASList
public export
length : PHOASList v a -> Nat
length [] = Z
length (x :: xs) = S (length xs)

||| A non-empty proof for PHOASLists
public export
data NonEmptyPL : (xs : PHOASList v a) -> Type where
  IsNonEmptyPL : NonEmptyPL (x :: xs)

||| A proof that an empty non-empty proof is uninhabited for PHOASLists
public export
implementation Uninhabited (NonEmptyPL []) where
  uninhabited IsNonEmptyPL impossible

||| A Show implementation for PHOASLists
public export
implementation Show a => Show (PHOASList v a) where
  show Nil = "[]"
  show (x :: xs) = "[" ++ show x ++ showl xs
    where
    showl : Show a => PHOASList v a -> String
    showl Nil = "]"
    showl (y :: ys) = "," ++ show y ++ showl ys
```

```
||| Semigroup implementation for PHOASLists
public export
implementation Semigroup (PHOASList v a) where
  (<+>) = (++)

||| Monoid implementation for PHOASLists
public export
implementation Monoid (PHOASList v a) where
  neutral = []

||| Foldable implementation for PHOASLists
public export
implementation Foldable (PHOASList v) where
  foldr f acc [] = acc
  foldr f acc (x :: xs) = f x (foldr f acc xs)

  foldl f acc [] = acc
  foldl f acc (x :: xs) = f (foldl f acc xs) x

||| Functor implementation for PHOASLists
public export
implementation Functor (PHOASList v) where
  map f [] = []
  map f (x :: xs) = f x :: map f xs

||| Applicative implementation for PHOASLists
public export
implementation Applicative (PHOASList v) where
 pure x = [x]
 fs <*> xs = concatMap (\f => map f xs) fs

||| Monad implementation for PHOASLists
public export
implementation Monad (PHOASList v) where
  xs >>= f = concatMap f xs
```

A.6 Necessary Proofs for Crossover: Proofs.idr

```
import Data.Fin

import Mod.Mod
import Typeable.Typeable
```

```
import PHOASList.PHOASList

%default total

||| A proof that if the run-time representation of two types are equal
||| Then those types are constructively equal.
public export
typeRepInj : (prf : TypeRep a = TypeRep b) -> (a = b)
typeRepInj Refl = Refl

||| Congruence on two equality proofs.
public export
cong2 : {f : Type -> Type -> Type} -> (a = c) -> (b = d) -> (f a b = f c d)
cong2 Refl Refl = Refl

||| Specific two equality congruence proof for product types.
public export
cong2p : (a = c) -> (b = d) -> (a, b) = (c, d)
cong2p Refl Refl = Refl

||| An equality of products implies a product of equalities.
public export
p2cong : (prf : (a, b) = (c, d)) -> (a = c, b = d)
p2cong Refl = (Refl, Refl)

||| Specific two equality congruence proof for function types.
public export
cong2f : (a = c) -> (b = d) -> (a -> b) = (c -> d)
cong2f Refl Refl = Refl

||| Proof of the injectivity of a boolean AND when True on pair.
public export
bandIsInjective : (x, y : Bool) -> (prf: (x && y) = True) -> (x = True, y = True)
bandIsInjective False False prf = absurd prf
bandIsInjective False True prf = absurd prf
bandIsInjective True False prf = absurd prf
bandIsInjective True True prf = (Refl, Refl)

||| Proof that boolean equality between run-time type representations
||| implies constructive equality between types.
public export
beqTypeReflectsEq : {a, b : Type} -> (x: TypeRep a) -> (y: TypeRep b) ->
                    (prf : beqType x y = True) -> (a = b)
```

```
beqTypeReflectsEq TBool TBool prf = Refl
beqTypeReflectsEq TBool TNat prf = absurd prf
beqTypeReflectsEq TBool TInteger prf = absurd prf
beqTypeReflectsEq TBool TString prf = absurd prf
beqTypeReflectsEq TBool (TFunc x y) prf = absurd prf
beqTypeReflectsEq TBool (TPair x y) prf = absurd prf
beqTypeReflectsEq TBool (TSum x y) prf = absurd prf
beqTypeReflectsEq TNat TBool prf = absurd prf
beqTypeReflectsEq TNat TNat prf = Refl
beqTypeReflectsEq TNat TInteger prf = absurd prf
beqTypeReflectsEq TNat TString prf = absurd prf
beqTypeReflectsEq TNat (TFunc x y) prf = absurd prf
beqTypeReflectsEq TNat (TPair x y) prf = absurd prf
beqTypeReflectsEq TNat (TSum x y) prf = absurd prf
beqTypeReflectsEq TInteger TBool prf = absurd prf
beqTypeReflectsEq TInteger TNat prf = absurd prf
beqTypeReflectsEq TInteger TInteger prf = Refl
beqTypeReflectsEq TInteger TString prf = absurd prf
beqTypeReflectsEq TInteger (TFunc x y) prf = absurd prf
beqTypeReflectsEq TInteger (TPair x y) prf = absurd prf
beqTypeReflectsEq TInteger (TSum x y) prf = absurd prf
beqTypeReflectsEq TString TBool prf = absurd prf
beqTypeReflectsEq TString TNat prf = absurd prf
beqTypeReflectsEq TString TInteger prf = absurd prf
beqTypeReflectsEq TString TString prf = Refl
beqTypeReflectsEq TString (TFunc x y) prf = absurd prf
beqTypeReflectsEq TString (TPair x y) prf = absurd prf
beqTypeReflectsEq TString (TSum x y) prf = absurd prf
beqTypeReflectsEq (TFunc x y) TBool prf = absurd prf
beqTypeReflectsEq (TFunc x y) TNat prf = absurd prf
beqTypeReflectsEq (TFunc x y) TInteger prf = absurd prf
beqTypeReflectsEq (TFunc x y) TString prf = absurd prf
beqTypeReflectsEq (TFunc x y) (TFunc z w) prf =
  let p1 = bandIsInjective (beqType x z) (beqType y w) prf
      rec1 = beqTypeReflectsEq x z (fst p1)
      rec2 = beqTypeReflectsEq y w (snd p1)
  in cong2f rec1 rec2
beqTypeReflectsEq (TFunc x y) (TPair z w) prf = absurd prf
beqTypeReflectsEq (TFunc x y) (TSum z w) prf = absurd prf
beqTypeReflectsEq (TPair x y) TBool prf = absurd prf
beqTypeReflectsEq (TPair x y) TNat prf = absurd prf
beqTypeReflectsEq (TPair x y) TInteger prf = absurd prf
beqTypeReflectsEq (TPair x y) TString prf = absurd prf
```

```
beqTypeReflectsEq (TPair x y) (TFunc z w) prf = absurd prf
beqTypeReflectsEq (TPair x y) (TPair z w) prf =
  let p1 = bandIsInjective (beqType x z) (beqType y w) prf
      rec1 = beqTypeReflectsEq x z (fst p1)
      rec2 = beqTypeReflectsEq y w (snd p1)
  in cong2 rec1 rec2
beqTypeReflectsEq (TPair x y) (TSum z w) prf = absurd prf
beqTypeReflectsEq (TSum x y) TBool prf = absurd prf
beqTypeReflectsEq (TSum x y) TNat prf = absurd prf
beqTypeReflectsEq (TSum x y) TInteger prf = absurd prf
beqTypeReflectsEq (TSum x y) TString prf = absurd prf
beqTypeReflectsEq (TSum x y) (TFunc z w) prf = absurd prf
beqTypeReflectsEq (TSum x y) (TPair z w) prf = absurd prf
beqTypeReflectsEq (TSum x y) (TSum z w) prf =
  let p1 = bandIsInjective (beqType x z) (beqType y w) prf
      rec1 = beqTypeReflectsEq x z (fst p1)
      rec2 = beqTypeReflectsEq y w (snd p1)
in cong2 rec1 rec2

||| Proof that x != 0 implies that x >= 0 forall natural numbers.
public export
notZimpliesGTZ : (x : Nat) -> (prf : Not (x = 0)) -> GT x Z
notZimpliesGTZ Z prf = void (prf Refl)
notZimpliesGTZ (S k) prf = LTESucc LTEZero

||| Proof that x > 0 implies that x != 0 forall natural numbers.
public export
gTZimpliesZ : (x : Nat) -> (prf : GT x Z) -> Not (x = 0)
gTZimpliesZ Z prf = absurd prf
gTZimpliesZ (S k) prf = SIsNotZ

||| Proof that for a given non-empty PHOASList, that the length
||| of such a list must be greater than zero.
public export
nonEmptyImpliesGTZ : (xs : PHOASList v a) -> (prf : NonEmptyPL xs) ->
                      GT (length xs) Z
nonEmptyImpliesGTZ [] prf = absurd prf
nonEmptyImpliesGTZ (x :: xs) prf = LTESucc LTEZero


||| Proof that for a given element and given PHOASList, the length of the list
||| built from prepending the elemnt to the original list must be
||| greater than zero.
```

```
consGTZ : (x : a) -> (xs : PHOASList v a) -> GT (length (x :: xs)) Z
consGTZ x [] = lteRefl
consGTZ x (y :: ys) = LTESucc LTEZero

||| Proof that forall Natural numbers: 'y', given a list of length y: 'xs'
||| and a proof that y is not zero. Then xs must not be empty.
lenGTZimpliesNonEmpty : (y : Nat) -> (xs : PHOASList v a) -> (prf : y = length xs) ->
                        (prf2 : GT y Z) -> NonEmptyPL xs
lenGTZimpliesNonEmpty Z xs prf prf2 = absurd prf2
lenGTZimpliesNonEmpty (S k) [] prf prf2 = absurd prf
lenGTZimpliesNonEmpty (S k) (x :: xs) prf prf2 = IsNonEmptyPL

||| Proof that for all elements of a finite set 'f' with a supremum 'n'
||| Any (successful) increments to the index of 'f' must be <= n.
public export
finLTBound : (f : Fin n) -> LTE (S (finToNat f)) n
finLTBound FZ = LTESucc LTEZero
finLTBound (FS x) =
  let rec = finLTBound x
  in LTESucc rec

||| Proof that for all natural numbers: n given a PHOASList: xs
||| and a proof that n < length xs then element at index n is
||| guaranteed to be in the PHOASList.
public export
ltLenAlwaysBound : (n : Nat) -> (xs : PHOASList v a) ->
                   (prf : LTE (S n) (length xs)) -> InPList n xs
ltLenAlwaysBound Z [] prf = absurd prf
ltLenAlwaysBound Z (x :: xs) prf = InNow
ltLenAlwaysBound (S k) (x :: xs) prf =
  let rec = ltLenAlwaysBound k xs
  in InAfter (rec (fromLteSucc prf))

||| proof that natural numbers x and y, given a proof that y > 0
||| then mod x y + 1 <= y
public export
modfNlteN : (x, y : Nat) -> {auto prf : GT y Z} -> LTE (S (modfin x y)) y
modfNlteN x y = finLTBound (modf x y)
```

A.7 Verified Type-Preserving Crossover: Crossover.idr

```
import Data.Fin

import Mod.Mod
import Typeable.Typeable
import Expr.Expr
import PHOASList.PHOASList
import Zipper.Zipper
import Proofs.Proofs

%default total

||| Given a list of Sigma types of a pair of Zippers,
|||   and some (presumably random) natural number:
|||   select that index.
public export
selectNode : (n : Nat) -> (xs :
             PHOASList v (a : Type ** (Zipper v a, Zipper v a))) ->
             {auto prf : NonEmptyPL xs} ->
             (a : Type ** (Zipper v a, Zipper v a))
selectNode n xs {prf} =
  let p1 = nonEmptyImpliesGTZ xs prf
      p2 = modfNlteN n (length xs) {prf = p1}
      p3 = ltLenAlwaysBound (modfin n (length xs)) xs p2
  in index (modfin n (length xs)) xs {prf=p3}

||| typeRepEq takes a PHOASList of the cartesian product of all possible
||| crossover pairs, and builds a PHOASList of only those pairs where crossover
||| is possible.
public export
typeRepEq : (xs : PHOASList v
            ((x : Type ** (TypeRep x, Zipper v x)),
            (y : Type ** (TypeRep y, Zipper v y)))) ->
            PHOASList v (a : Type ** (Zipper v a, Zipper v a))
typeRepEq xs = foldr f [] xs
  where
    f : {v : Type -> Type} ->
        (xy : ((x : Type ** (TypeRep x, Zipper v x)),
              (y : Type ** (TypeRep y, Zipper v y)))) ->
        PHOASList v (a : Type ** (Zipper v a, Zipper v a)) ->
        PHOASList v (a : Type ** (Zipper v a, Zipper v a))
    f ((t1 ** (r1, z1)),(t2 ** (r2, z2))) xs with (beqType r1 r2) proof p
```

```
      f ((t1 ** (r1, z1)),(t2 ** (r2, z2))) xs | True =
        let p1 = beqTypeReflectsEq r1 r2 (sym p)
            z3 = replace (sym p1) z2
        in (_ ** (z1, z3)) :: xs
      f ((t1 ** (r1, z1)),(t2 ** (r2, z2))) xs | False = xs

||| A type preserving crossover function that is correct by construction
public export
xover : (zs : (a : Type ** (Zipper v a, Zipper v a))) ->
        (Zipper v (fst zs), Zipper v (fst zs))
xover (t ** (Zip {z1} e1 c1, Zip e2 c2)) = (Zip {z1} e2 c1, Zip {z1} e1 c2)
```

A.8 Crossover Random Positions of a Given Type: Main.idr

```
import Typeable.Typeable
import Mod.Mod
import Expr.Expr
import PHOASList.PHOASList
import Proofs.Proofs
import Zipper.Zipper
import Crossover.Crossover

import System

%default total

ex1 : Expr v Nat
ex1 = Add {t1=TNat} (Lit {t1=TNat} 2)
       (Cnst {t1=TNat} {t2=TString}
          (Lit {t1 = TNat} 3) (Lit {t1 = TString} "Test"))

ex2 : Expr v Nat
ex2 = Add {t1=TNat} (Lit {t1=TNat} 2)
       (Cnst {t1=TNat} {t2=TString}
          (Lit {t1 = TNat} 3) (Lit {t1 = TString} "Hello"))

zipExp1 : Zipper v Nat
zipExp1 = Zip {z1=TNat} ex1 Root

zipExp2 : Zipper v Nat
zipExp2 = Zip {z1=TNat} ex2 Root

```

```
||| The cartesian product of possible focus pairs between zipExp1 and zipExp2
cartProdZipExp : PHOASList v ((x : Type ** (TypeRep x, Zipper v x)),
                              (y : Type ** (TypeRep y, Zipper v y)))
cartProdZipExp = (\a, b => (a, b)) <$> (flatten zipExp1) <*> (flatten zipExp2)


||| A control function that selects a crossover point and does not
||| perform crossover
noOver : Nat -> ((a : Type ** Zipper v a), (b : Type ** Zipper v b))
noOver n = (top x, top y)
  where
    xys : (a : Type ** (Zipper v a, Zipper v a))
    xys = selectNode n (typeRepEq cartProdZipExp)

    x : (a : Type ** Zipper v a)
    x = (DPair.fst xys ** (Basics.fst (DPair.snd xys)))

    y : (a : Type ** Zipper v a)
    y = (DPair.fst xys ** (Basics.snd (DPair.snd xys)))

||| A function that selects the same point as 'noover' and performs the
||| crossover operation
testOver : Nat -> ((a : Type ** Zipper v a), (b : Type ** Zipper v b))
testOver n =
  let xys = xover $ selectNode n (typeRepEq cartProdZipExp)
      x = (_ ** Basics.fst xys)
      y = (_ ** Basics.snd xys)
in (top x, top y)

||| A convenience function to read bytes from /dev/urandom.
getChars : File -> List (IO (Either FileError Char)) -> Nat ->
           List (IO (Either FileError Char))
getChars f xs Z = fgetc f :: xs
getChars f xs (S k) = fgetc f :: getChars f xs k

main : IO ()
main = do
  let path = "/dev/urandom"

  Right (FHandle ptr) <- openFile path Read | Left err => do
    putStrLn (show err)
  cs <- sequence $ getChars (FHandle ptr) [] 4

```

```
Right str <- pure (sequence cs) | Left err => do
  putStrLn (show err)

closeFile (FHandle ptr)
let ints = (\x => cast x {to=Int}) <$> str
(x :: xs) <- pure ints | [] => putStrLn "Error somehow we have an empty List"

let i = x
let n = cast i {to=Nat}
let xs' = (\x => cast x {to=Nat}) <$> ints
let s = show i
let control = noOver n {v=\x => x}
let xo = testOver n {v = \x => x}
let list = typeRepEq cartProdZipExp {v= (\x => String)}
let xolen = length list
let xomod = modfin n xolen
let printctrl = noOver n {v = \_ => String}
let printxo = testOver n {v = \_ => String}
let c1 = extract $ DPair.snd $ Basics.fst control
let p1 = extract $ DPair.snd $ Basics.fst printctrl
let c2 = extract $ DPair.snd $ Basics.snd control
let p2 = extract $ DPair.snd $ Basics.snd printctrl
let x1 = extract $ DPair.snd $ Basics.fst printxo
let x2 = extract $ DPair.snd $ Basics.snd printxo

putStrLn "-------------------GENERATING RANDOM NUMBER----------------"
putStrLn ("Random number is: " ++ s)
putStrLn ("List of numbers: " ++ show (x :: xs))
putStrLn "----------PRINTING CONTROL EXPRESSIONS (No Crossover!)------"
putStrLn (prettyE 0 p1)
putStrLn (prettyE 0 p2)
putStrLn "-----------NUMBER OF VALID POSITIONS FOR CROSSOVER----------"
putStrLn (show xolen)
putStrLn "------------POSITION CHOSEN FROM VALID POSITIONS------------"
putStrLn (show xomod)
putStrLn "-------------PRINTING CROSSED OVER EXPRESSIION VALUES---------"
putStrLn (prettyE 0 x1)
putStrLn (prettyE 0 x2)
putStrLn "---------------------------END PROGRAM-----------------------"
```