Lecture Notes on Elaboration

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
Tuesday, October 6, 2020

1 Introduction

We have spent a lot of time analyzing and designing the essence of a programming language, starting from first principles. The focus has been on the *statics* (the type system), the *dynamics* (the rules for how to evaluate programs), and understanding the relationship between them in a mathematically rigorous way.

There is, of course, a lot more to a real programming language. At the "front end" there is the *concrete syntax* according to which the program text is parsed. The result of parsing is either some *abstract syntax* or an error message if the program is not well-formed according to the grammar defining its syntax. At the "back end" there are concerns about how a language might be executed efficiently, or *compiled* to machine language so it can run even faster. In this course we will say little about issues of grammar, concrete syntax, parsers or parser generators, because we want to focus on the deeper semantic issues where we have accumulated a lot of knowledge about language design.

In today's lecture we will look at *elaboration*, which is a translation mediating between specific forms of concrete syntax and internal representation in abstract syntax. Elaborating the program allows us to provide some conveniences that make it easy to write and read concise programs without giving up the sound underlying principles we have learned about in this course so far.

2 An Example: Binary Numbers

Before binary numbers, we introduce the concrete syntax of LAMBDA when applied to call-by-value functional programs (recognized by the `.cbv` extension). We see a few items of concrete syntax. We use `$` for recursion, both at the level of types (to stand for ρ) and at the level of terms (to stand for fix). Instead of writing $\mathbf{l}.e$ and $\mathbf{r}.e$ (with different fonts being unavailable in the ASCII source) we write $'\mathbf{l}\, e$ and $'\mathbf{r}\, e$ (pronounced "tick l" and "tick r"). Finally, we interpose the keyword `of` between the subject of the case expressions and the branches in order to avoid an ambiguous grammar.

We also see the new kind of declaration **eval** $x = e$ which evaluates $e \mapsto^* v$ and defines x to stand for the resulting value v. Remember that this is quite different from the normal form of e, as we have discussed multiple times.

```
type nat = $a. 1 + a    % == 1 + nat

decl zero : nat
decl succ : nat -> nat
defn zero = fold ('l ())     % ~ fold (l.<>)
defn succ = \n. fold ('r n)

eval two = succ (succ zero)

decl pred : nat -> nat
defn pred = \n. case (unfold n) of ('l _ => zero | 'r m => m)

eval one = pred two

decl plus : nat -> nat -> nat
defn plus = $plus. \n. \k.
  case (unfold n)
    of ('l _ => k | 'r m => succ (plus m k))

eval three = plus two one

decl times : nat -> nat -> nat
defn times = $times. \n. \k.
   case (unfold n)
     of ('l _ => zero | 'r m => plus (times m k) k)

eval six = times three two
```

Listing 1: Unary natural numbers in call-by-value LAMBDA

The unary representation of numbers is perfect from the foundational point of view, but impractical. In particular, representation of numbers become very large, and operations on them very slow. But we already know a better representation: binary numbers, which are (finite) sequences of bits 0 and 1.

Binary numbers (type *bin*) are generated by three constructors:

1. e : *bin* where e represents 0,
2. *b0* : *bin* $\to$ *bin* where *b0* $\overline{x}$ represents $2x$, and
3. *b1* : *bin* $\to$ *bin* where *b1* $\overline{x}$ represents $2x + 1$.

This representation means that we see the least significant bit first. For example, $6 = (110)_2$ wwould be represented by *b0* $($*b1* $($*b1* $e))$. This "little-endian" representation is well-suited for operations on binary numbers; the representation where we write the bits in the order we are used to not so much (consider, for example, the increment function defined below).

The types of the constructors lead us to the recursive equation

$$\begin{array}{rcl} \mathit{bin} & = & \rho\alpha.\, \alpha + (\alpha + 1) \\ & \cong & \mathit{bin} + (\mathit{bin} + 1) \end{array}$$

and the definitions

$$\begin{array}{rcl} \mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{l} \cdot x) \\ \mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{r} \cdot \mathbf{l} \cdot x) \\ e & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathbf{r} \cdot \langle\,\rangle) \end{array}$$

On this representation we can now define the binary increment function *inc*. We would like it to satisfy the specification

$$\begin{array}{l} \mathit{inc}\ (\mathit{b0}\ x) = \mathit{b1}x \\ \mathit{inc}\ (\mathit{b1}\ x) = \mathit{b0}\ (\mathit{inc}\ x) \\ \mathit{inc}\ (e) = \mathit{b1}\ e \end{array}$$

From this we can derive a closed form definition, where we have to be respect that fact that *inc* is defined *recursively*.

```
type bin = $a. a + (a + 1)  % == bin + (bin + 1)

decl b0 : bin -> bin
decl b1 : bin -> bin
decl e : bin

```

```
defn b0 = \x. fold ('l x)
defn b1 = \x. fold ('r ('l x))
defn e = fold ('r ('r ()))

decl inc : bin -> bin
defn inc = $inc. \x. case (unfold x)
                       of ( 'l y => b1 y
                          | 'r y => case y
                                     of ( 'l z => b0 (inc z)
                                        | 'r z => b1 e
                                        )
                          )

eval _6 = b0 (b1 (b1 e))
eval _7 = inc _6
eval _8 = inc _7
```

Listing 2: Binary numbers in call-by-value LAMBDA

We see the output of the last three evaluations

```
defn _6 = fold 'l fold 'r 'l fold 'r 'l fold 'r 'r ()
defn _7 = fold 'r 'l fold 'r 'l fold 'r 'l fold 'r 'r ()
defn _8 = fold 'l fold 'l fold 'l fold 'r 'l fold 'r 'r ()
```

which we can recognize as the binary representations of 6, 7, and 8 where `'l` represents a bit 0, `'r 'l` represents a bit 1, and `'r 'r` represents the terminator for the bit sequence.

One step towards a more natural (and readable) representation is to generalize the binary sum to an variadic sums, which we discuss in Section 4

3 Isomorphism Revisited

The representation of binary numbers has a feature which is common in the representation of complex data, but haven't seen so far: there are multiple different representations of the same data. For example, *e* and *b0 e* both represent the number 0, because $2 \times 0 = 0$. In fact, each number has infinitely many representations: we can just add leading zeros to every representation. In Exercise 1 we explore how to remove this ambiguity from the representation of binary numbers, but this is certainly not possible in other examples.

We could try to show that the translation between unary and binary numbers are an isomorphism (which will fail). For that purpose, we define the following translations:

$$\begin{array}{lcl} \mathit{nat2bin} & : & \mathit{nat} \to \mathit{bin} \\ \mathit{bin2nat} & : & \mathit{bin} \to \mathit{nat} \end{array}$$

These serve as the "strawman" proposal for a pair of functions witnessing an isomorphism. We write them here in the concrete syntax of LAMBDA.

```
decl nat2bin : nat -> bin
decl bin2nat : bin -> nat

(*
 * nat2bin zero = e
 * nat2bin (succ n) = inc (nat2bin n)
 *)
defn nat2bin = $nat2bin. \n.
   case (unfold n) of ('l _ => e | 'r m => inc (nat2bin m))

(*
 * bin2nat (b0 x) = times two (bin2nat x)
 * bin2nat (b1 x) = succ (times two (bin2nat x))
 * bin2nat (e) = zero
 *)
defn bin2nat = $bin2nat. \x.
   case (unfold x)
     of ( 'l y => times two (bin2nat y)
        | 'r y => case y of ( 'l z => succ (times two (bin2nat z))
                            | 'r z => zero
                            )
        )
```

However, these two functions do not form an isomorphism because any alternative form of a binary number with leading zeros, when mapped to a unary number and back will be standardized in the sense that the leading zeros will be erased.

$$\begin{array}{lcl} \mathit{standardize} & : & \mathit{bin} \to \mathit{bin} \\ \mathit{standardize} & = & \mathit{nat2bin} \circ \mathit{bin2nat} \end{array}$$

On the domain of binary numbers the function *standardize* represents a *retract*, mapping any number to its standard form without leading zeros.

While common, ambiguous representations such as *bin* have their dangers. In particular, we *could* define functions that make no sense at all from the numerical standpoint because they behave differently on different representations of the same number! For example, the function *bad* below returns *true* for the standard representation and *false* for a nonstandard one, even though these two representation are supposed to be indistinguishable.

```
decl bad : bin -> 1 + 1
defn bad = \x. case (unfold x) of ('l x => 'r () | 'r y => 'l () )

eval tt = bad (b0 e)
eval ff = bad e
```

It is therefore important, when working on ambiguous representation, to keep in mind and reason about whether functions are correct with respect to different representations of the same elements. In more general type theories this kind of construction with the guarantee that accompanies it is called a *quotient type*.

4 Variadic Sums

Once we know that the sum is associative and commutative with unit 0 we can introduce a more general notation that is useful for practical purposes: rather than just using labels **l** and **r** for a binary sum, we can allow a *finite set I* of *tags* or *label* (think of them as strings) and write

$$(i_1 : \tau_1) + \cdots + (i_n : \tau_n)$$

where each summand is marked with a distinct label i. We also write this in abstract syntax as

$$\sum_{i \in I} (i : \tau_i)$$

The empty type 0 arises from $I = \{\,\}$ and we might define

$$\begin{array}{lcl}
\textit{bool} & = & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\
\textit{option}\ \tau & = & (\mathbf{none} : 1) + (\mathbf{some} : \tau) \\
\textit{order} & = & (\mathbf{less} : 1) + (\mathbf{equal} : 1) + (\mathbf{greater} : 1) \\
\textit{nat} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \textit{nat}) \\
 & = & \rho\alpha.\, (\mathbf{zero} : 1) + (\mathbf{succ} : \alpha) \\
\textit{list}\ \tau & \cong & (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \textit{list}\ \tau) \\
 & = & \rho\alpha.\, (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha) \\
\textit{bin} & \cong & (\mathbf{b0} : \textit{bin}) + (\mathbf{b1} : \textit{bin}) + (\mathbf{e} : 1) \\
 & = & \rho\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)
\end{array}$$

This generalized form of sum also comes with a generalized constructor (allowing any label of a sum) and case expression (requiring a branch for each label of a sum). For example, we might have the following definitions.

$$\begin{array}{lcl}
\mathit{bin} & = & \rho\alpha.\,(\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1) \\[4pt]
\mathit{b0} & : & \mathit{bin} \to \mathit{bin} \\
\mathit{b1} & : & \mathit{bin} \to \mathit{bin} \\
\mathit{e} & : & \mathit{bin} \\[4pt]
\mathit{b0} & = & \lambda x.\,\mathsf{fold}\ (\mathbf{b0} \cdot x) \\
\mathit{b1} & = & \lambda x.\,\mathsf{fold}\ (\mathbf{b1} \cdot \cdot x) \\
\mathit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \\[4pt]
\mathit{inc} & : & \mathit{bin} \to \mathit{bin} \\
\mathit{inc} & = & \mathsf{fix}\ \mathit{inc}.\,\lambda x.\,\mathsf{case}\ (\mathsf{unfold}\ x) \\
 & & \quad\quad (\ \mathbf{b0} \cdot y \Rightarrow \mathit{b1}\ y \\
 & & \quad\quad \mid \mathbf{b1} \cdot y \Rightarrow \mathit{b0}\ (\mathit{inc}\ y) \\
 & & \quad\quad \mid \mathbf{e} \cdot \_ \Rightarrow \mathit{b1}\ \mathit{e})
\end{array}$$

5 "Syntactic Sugar"

A simple form of elaboration is to eliminate some simple forms of "syntactic sugar" and translate them into an internal form to simplify downstream processing. A good example are the following definitions:

$$\begin{array}{lcl}
\mathsf{bool} & \triangleq & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\
\mathsf{true} & \triangleq & \mathbf{true} \cdot \langle\,\rangle \\
\mathsf{false} & \triangleq & \mathbf{false} \cdot \langle\,\rangle \\
\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \triangleq & \mathsf{case}\ e_1\ (\mathbf{true} \cdot \_ \Rightarrow e_2 \mid \mathbf{false} \cdot \_ \Rightarrow e_3)
\end{array}$$

Here, we used another common convention, name we use an underscore (_) in place of a variable name if that variable does not occur in its scope (here, this scope would be e_2 for the first underscore and e_3 for the second. Such a syntactic transformation could take place before or after type checking.

6 Data Constructors and Pattern Matching

As another example, consider the definition of the natural numbers in unary form:

$$\mathit{nat} = \rho\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \alpha)$$

This is unnecessarily difficult to read because we have to remember that α really is supposed to stands for *nat* on the right hand. Easier to read is

$$\mathit{nat} \cong (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{nat})$$

Moreover, the labels may sometimes be a bit awkward to use, so perhaps we could "automatically" define

$$\begin{array}{lcl} \mathit{zero} & : & 1 \to \mathit{nat} \\ \mathit{zero} & = & \lambda u.\, \mathbf{zero} \cdot u \\ \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\, \mathbf{succ} \cdot n \end{array}$$

Notice there the difference between the *function succ* (in italics) and the *label* **succ** (in bold). Maybe we could even go further and eliminate the $1 \to \mathit{nat}$ because we already know that $1 \to \tau \cong \tau$, in which case we would obtain

$$\begin{array}{lcl} \mathit{zero} & : & \mathit{nat} \\ \mathit{zero} & = & \mathbf{zero} \cdot \langle\, \rangle \end{array}$$

Finally, it would be nice if we could simplify pattern matching as well. Instead of, for example,

$$\begin{array}{lcl} \mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{pred} & = & \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{zero} \cdot \_ \Rightarrow \mathit{zero} \mid \mathbf{succ} \cdot n' \Rightarrow n') \end{array}$$

it would be easier to read and understand if we could write

$$\begin{array}{lcl} \mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\[1ex] \mathit{pred}\ \mathit{zero} & = & \mathit{zero} \\ \mathit{pred}\ (\mathit{succ}\ n') & = & n' \end{array}$$

This would somehow only make sense if "*zero*" was understood not only as a constant of type *nat*, but also that it corresponded to a label **zero** with the same name so we can elaborate it into the case of the internal definition of predecessor shown just before. And similarly for *succ* and **succ**.

In fact, modern functional languages such as Haskell, OCaml, or Standard ML provide syntax for data type definitions that provide essentially the above functionality, and more. In ML we would write:

```
datatype nat = Zero | Succ of nat
fun pred Zero = Zero
  | pred (Succ n') = n'
```

In OCaml it might be

```
type nat = Zero | Succ of nat;;
let pred n = match n with
  | Zero -> Zero
  | Succ n' -> n';;
```

And Haskell:

```
data Nat = Zero | Succ Nat

pred :: Nat -> Nat
pred Zero = Zero
pred (Succ n') = n'
```

The type we gave here for `pred` is optional, but it is often helpful to explicitly state the type of a function. We should also keep in mind that the dynamics of `Zero` and `Succ` is different in Haskell because it is a call-by-need ("lazy") language.

We refer to `Zero` and `Succ` as *data constructors*, which means they are simultaneously functions (or constants in the case of `Zero`) to constructs values of a sum, and labels so we can pattern-match against them.

7 Generalizing Sums

Let's recall our language so far:

$$
\begin{array}{llcll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \mid \rho\alpha.\, \tau & \\
\text{Expressions} & e & ::= & x & \text{(variables)} \\
 & & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
 & & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
 & & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
 & & \mid & \mathbf{l} \cdot e \mid \mathbf{r} \cdot e \mid \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) & (+) \\
 & & \mid & \mathsf{case}\ e\ (\,) & (0) \\
 & & \mid & \mathsf{fold}\ e \mid \mathsf{unfold}\ e & (\rho) \\
 & & \mid & f \mid \mathsf{fix}\ f.\, e & \text{(recursion)}
\end{array}
$$

Except for functions and recursive types, the destructors are of the form $\mathsf{case}\ e\ (\ldots)$. We will now unify these constructs even more, replacing the primitive unfold e by a new one, case e (fold $x \Rightarrow e'$). We can then define *Unfold* as a function

$$\begin{array}{lcl} \mathit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & \triangleq & \lambda x.\,\mathsf{case}\ x\ (\mathsf{fold}\ x \Rightarrow x) \end{array}$$

See Exercise 3 for more on this restructuring of the language.

Streamlining our language a little bit further, we now officially generalize the sum from binary to n-ary, allowing labels i to be drawn from a finite index set I. The case construct for the sums then has a branch for each $i \in I$. Our previous constructs are a special case, with $\tau_1 + \tau_2 \triangleq \sum_{i \in \{\mathbf{l},\mathbf{r}\}}(i : \tau_i) = (\mathbf{l} : \tau_1) + (\mathbf{r} : \tau_2)$ and $0 \triangleq \sum_{i \in \emptyset}(i : \tau_i)$.

$$\begin{array}{llcll} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \sum_{i \in I}(i : \tau_i) \mid \rho\alpha.\,\tau & \\ \text{Expressions} & e & ::= & x & \text{(variables)} \\ & & \mid & \lambda x.\,e \mid e_1\,e_2 & (\to) \\ & & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\ & & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\ & & \mid & i \cdot e \mid \mathsf{case}\ e\ (i \cdot x \Rightarrow e')_{i \in I} & (\textstyle\sum) \\ & & \mid & \mathsf{fold}\ e \mid \mathsf{case}\ e\ (\mathsf{fold}\ x \Rightarrow e') & (\rho) \\ & & \mid & f \mid \mathsf{fix}\ f.\,e & \text{(recursion)} \end{array}$$

Except for functions, all destructors are now case-expressions. Functions are different because values are of the form $\lambda x.\,e$ that we cannot match against because we assumed that they are not observable outcomes of computation.

For sums, we have the following generalized statics and dynamics. Key is that we have to check all branches of a case expressions, and all of them have the same type τ'.

$$\frac{k \in I \quad \Gamma \vdash e : \tau_k}{\Gamma \vdash k \cdot e : \sum_{i \in I}(i : \tau_i)}\ \mathsf{tp/sum}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e'_i : \tau' \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} : \tau'}\ \mathsf{tp/cases}$$

$$\frac{e\ \mathit{value}}{i \cdot e\ \mathit{value}}\ \mathsf{val/sum}$$

$$\frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\ \mathsf{step/inject}$$

$$\frac{e_0 \mapsto e'_0}{\mathsf{case}\ e_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto \mathsf{case}\ e'_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I}}\ \mathsf{step/cases}_0$$

$$\frac{k \in I \quad v\ \mathit{value}}{\mathsf{case}\ (k \cdot v)\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto [v/x_k]e'_k}\ \mathsf{step/cases/inject}$$

8 Nesting Case Expressions

As another example, let's consider a function *half* on natural numbers that is supposed to round down. We write it down in a pattern-matching style.

$$\begin{array}{lcl}
\mathit{half} & : & \mathit{nat} \to \mathit{nat} \\[1ex]
\mathit{half}\ \mathit{zero} & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ (\mathit{succ}\ n'')) & = & \mathit{succ}\ (\mathit{half}\ n'')
\end{array}$$

This could be elaborated into two nested case expressions and a use of recursion. To avoid an even deeper nesting of cases, we use *Unfold* as defined in the previous section.

$$\begin{array}{l}
\mathit{half} = \mathsf{fix}\ h.\ \lambda n.\ \mathsf{case}\ (\mathit{Unfold}\ n)\ (\,\mathbf{zero} \cdot \_ \Rightarrow \mathit{zero} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{succ} \cdot n' \Rightarrow \mathsf{case}\ (\mathit{Unfold}\ n')\ (\,\mathbf{zero} \cdot \_ \Rightarrow \mathit{zero} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{succ} \cdot n'' \Rightarrow \mathit{succ}\ (h\ n'')))
\end{array}$$

Such nested case expressions naturally lead to the question on how to define arbitrarily nested patterns and how they should be typed and evaluated, which we will discuss in the next lecture.

Exercises

Exercise 1 It is often intuitive to define types in a mutually recursive way. As a simple example, consider how to define binary numbers in *standard form*, that is, not allowing leading zeros. We define binary numbers in standard form (*std*) mutually recursively with strictly positive binary numbers (*pos*).

$$\begin{array}{lcl} \mathit{std} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \\ \mathit{pos} & \cong & \qquad\qquad (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \end{array}$$

(i) Using only *std*, *pos*, and function types formed from them, give all types of *e*, *b0*, and *b1* defined as follows:

$$\begin{array}{lcl} \mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b0} \cdot x) \\ \mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b1} \cdot x) \\ \mathit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \end{array}$$

(ii) Define the types *std* and *pos* explicitly in our language using the ρ type former so that the isomorphisms stated above hold.

(iii) Does the function *inc* from Section 4 have type $\mathit{std} \to \mathit{pos}$? You may use all the types for *b0*, *b1* and *e* you derived in part (i). Then either explain where the typing fails or indicate that it has that type. You do not need to write out a typing derivation.

(iv) Write a function $\mathit{pred} : \mathit{pos} \to \mathit{std}$ that returns the predecessor of a strictly positive binary number. You must make sure your function is correctly typed, where again you may use all the types from part (i).

Exercise 2 It is often convenient to define functions by mutual recursion. As a simple example, consider the following two functions on bit strings determining if it has *even or odd parity*.

$$\begin{array}{lcl} \mathit{bin} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) \\[1ex] \mathit{even} & : & \mathit{bin} \to \mathit{bool} \\ \mathit{odd} & : & \mathit{bin} \to \mathit{bool} \\[1ex] \mathit{even}\ \mathit{e} & = & \mathit{true} \\ \mathit{even}\ (\mathit{b0}\ x) & = & \mathit{even}\ x \\ \mathit{even}\ (\mathit{b1}\ x) & = & \mathit{odd}\ x \\[1ex] \mathit{odd}\ \mathit{e} & = & \mathit{false} \\ \mathit{odd}\ (\mathit{b0}\ x) & = & \mathit{odd}\ x \\ \mathit{odd}\ (\mathit{b1}\ x) & = & \mathit{even}\ x \end{array}$$

(i) Write a function *parity* with a single fixed point constructor and use it to define *even* and *odd*. Also, state the type of your *parity* function explicitly.

(ii) More generally, our simple recipe for implementing a recursively specified function using the fixed point constructor in our call-by-value language goes from the specification

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ f\,x & = & h\,f\,x \end{array}$$

to the implementation

$$f \;=\; \mathsf{fix}\, g.\, \lambda x.\, h\,g\,x$$

It is easy to misread these, so remember that by our syntactic convention, $h\,f\,x$ stands for $(h\,f)\,x$ and similarly for $h\,g\,x$. Give the type of h and show by calculation that f satisfies the given specification by considering $f\,v$ for an arbitrary value v of type τ_1.

(iii) A more general, *mutually recursive* specification would be

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ g & : & \sigma_1 \to \sigma_2 \\ f\,x & = & h_1\,f\,g\,x \\ g\,y & = & h_2\,f\,g\,y \end{array}$$

Give the types of h_1 and h_2.

(iv) Show how to explicitly define f and g in our language from h_1 and h_2 using the fixed point constructor and verify its correctness by calculation as in part (ii). You may use any other types in the language introduced so far (pairs, unit, sums, polymorphic, and recursive types).

Exercise 3 In the language where the primitive unfold has been replaced by pattern matching, we can define the following two functions:

$$\begin{array}{lcl} \mathit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & = & \lambda x.\,\mathsf{case}\;x\;(\mathsf{fold}\;x \Rightarrow x) \\[1ex] \mathit{Fold} & : & [\rho\alpha.\,\tau/\alpha]\tau \to \rho\alpha.\,\tau \\ \mathit{Fold} & = & \lambda x.\,\mathsf{fold}\;x \end{array}$$

Prove that *Fold* and *Unfold* are witnessing a type isomorphism.