Lecture Notes on Recursive Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 8
Thu Sep 18, 2025

1 Introduction

Using type structure to capture common constructions available in programming languages, we have built a rich set of primitives in our programming language (see 07-isomorphisms-rules.pdf for a summary of the rules). Booleans turned out be representable using generic constructions, since $\mathit{bool} = 1 + 1$. However, natural numbers would be

$$\mathit{nat} = 1 + (1 + (1 + \cdots))$$

which cannot be expressed already. However, we can observe that the tail of the sum is equal to the whole sum. That is,

$$\mathit{nat} = 1 + \mathit{nat}$$

We won't be able to achieve such an equality, but we *can* achieve an *isomorphism*

$$\mathit{nat} \cong 1 + \mathit{nat}$$

with two functions to witness the isomorphism.

$$\mathit{nat} \quad \underset{\text{fold}}{\overset{\text{unfold}}{\cong}} \quad 1 + \mathit{nat}$$

Actually, unfold and fold will not be functions but language primitives because we want them to apply to a large class of recursively defined types.

2 Recursive Types

The more general type constructor that solves recursive type equations is written as $\mu\alpha.\,\tau$. Mu (μ) here stands for "recursive", α is a type variable with scope τ.[1] The general picture to keep in mind is that a recursive type $\mu\alpha.\,\tau$ should be isomorphic to its *unfolding* $[\mu\alpha.\,\tau/\alpha]\tau$.

$$\mu\alpha.\,\tau \quad \underset{\text{fold}}{\overset{\text{unfold}}{\cong}} \quad [\mu\alpha.\,\tau/\alpha]\tau$$

[1] Sometimes $\mu\alpha.\,\tau$ is reserved for so-called *inductive types* which requires some restrictions on the occurrences of α in τ. Since our type is indeed inductive (rather than, for example, coinductive) whenever these restrictions are satisfied, we will use the notation $\mu\alpha.\,\tau$. It is also common in the literature.

Once we have defined the fold and unfold expressions with their statics and dynamics, we will have to check that these two types are indeed isomorphic.

As an example, consider

$$\mathit{nat} = \mu\alpha.\, 1 + \alpha$$

Does this give us the desired isomorphism? Let's check:

$$\begin{array}{rcl} \mathit{nat} & = & \mu\alpha.\, 1 + \alpha \\ & \cong & [\mu\alpha.\, 1 + \alpha/\alpha](1 + \alpha) \\ & = & 1 + (\mu\alpha.\, 1 + \alpha) \\ & = & 1 + \mathit{nat} \end{array}$$

So, yes, we get the desired isomorphism. Here are some other examples of types with recursive definitions we'd like to represent in a similar manner.

Lists	$\mathit{list}\ \tau$	$\cong$	$1 + (\tau \times \mathit{list}\ \tau)$
Binary Trees	tree	$\cong$	$1 + (\mathit{tree} \times \mathit{nat} \times \mathit{tree})$
Binary Numbers	bin	$\cong$	$\mathit{list}\ (1 + 1)$

For example, binary trees of natural numbers would then be explicitly defined as

$$\begin{array}{rcl} \mathit{tree} & = & \mu\alpha.\, 1 + (\alpha \times \mathit{nat} \times \alpha) \\ & \cong & 1 + (\mathit{tree} \times \mathit{nat} \times \mathit{tree}) \end{array}$$

and satisfy the desired isomorphism.

3 Fold and Unfold

Let's recall the principal isomorphism we would like to have:

$$\mu\alpha.\, \tau \quad \underset{\mathsf{fold}}{\overset{\mathsf{unfold}}{\cong}} \quad [\mu\alpha.\, \tau/\alpha]\tau$$

Each new type we have comes with some constructors for values of the new type and some destructors. Computation arises when a destructor meets a constructor. According to the display above, fold should be the *constructor* (because it results in something of type $\mu\alpha.\, \tau$), while unfold is a destructor. Reading the types off the above desired isomorphism:

$$\frac{\Gamma \vdash e : [\mu\alpha.\, \tau/\alpha]\tau}{\Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\, \tau}\ \mathsf{tp/fold} \qquad \frac{\Gamma \vdash e : \mu\alpha.\, \tau}{\Gamma \vdash \mathsf{unfold}\ e : [\mu\alpha.\, \tau/\alpha]\tau}\ \mathsf{tp/unfold}$$

We decide that $\mathsf{fold}\ e$ is a value only if e is a value. This is so that, for example, when we write $v : \mathit{nat}$, the value v will actually directly represent a natural number instead of some expression that might result in a natural number (see Exercise 1).

$$\frac{e\ \mathit{value}}{\mathsf{fold}\ e\ \mathit{value}}\ \mathsf{val/fold}$$

The interesting rule for stepping (usually the first one to write) is the one where a destructor meets a constructor.

$$\frac{v\ \mathit{value}}{\mathsf{unfold}\ (\mathsf{fold}\ v) \mapsto v}\ \mathsf{step/unfold/fold}$$

Does this rule preserve types? Let's say we have

$$\cdot \vdash \mathsf{unfold}\ (\mathsf{fold}\ v) : \sigma$$

By inversion (only the unfold rule could have this conclusion), we obtain

$$\cdot \vdash \mathsf{fold}\ v : \mu\alpha.\,\tau$$

where $\sigma = [\mu\alpha.\,\tau/\alpha]\tau$. Applying inversion again, we get

$$\cdot \vdash v : [\mu\alpha.\,\tau/\alpha]\tau$$

which is also the type of unfold (fold v). Therefore, the rule step/unfold satisfies type preservation.

We now only need to add rules to reach values and redices, so-called *congruence rules*.

$$\frac{e \mapsto e'}{\mathsf{fold}\ e \mapsto \mathsf{fold}\ e'}\ \mathsf{step/fold} \qquad \frac{e \mapsto e'}{\mathsf{unfold}\ e \mapsto \mathsf{unfold}\ e'}\ \mathsf{step/unfold}_0$$

It is a matter of checking the progress theorem and also verifying the desired isomorphism to ensure that we now have enough rules. A student suggested

$$\frac{}{\mathsf{fold}\ (\mathsf{unfold}\ e) \mapsto e}\ ?$$

which is eminently reasonable, but turned out to be unnecessary. Instead, we find that fold (unfold e) is extensionally equivalent to e at type $\mu\alpha.\,\tau$.

4 Examples

Before we check our desired properties, let's write some examples on natural numbers (in our unary representation).

$$\begin{array}{lcl}
\mathit{nat} & = & \mu\alpha.\,1 + \alpha \\
 & \cong & 1 + \mathit{nat} \\
 & & \\
\mathit{zero} & : & \mathit{nat} \\
\mathit{zero} & = & \mathsf{fold}\ (\mathbf{l} \cdot \langle\rangle) \\
 & & \\
\mathit{one} & : & \mathit{nat} \\
\mathit{one} & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathit{zero}) \\
 & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathsf{fold}\ (\mathbf{l} \cdot \langle\rangle)) \\
 & & \\
\mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{succ} & = & \lambda n.\,\mathsf{fold}\ (\mathbf{r} \cdot n) \\
 & & \\
\mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{pred} & = & \lambda n.\,\mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{l} \cdot x_1 \Rightarrow \mathit{zero} \mid \mathbf{r} \cdot x_2 \Rightarrow x_2)
\end{array}$$

At this point we realize that we cannot write any function that recurses over a natural number. Unlike the λ-calculus, the representation here as a sum and a recursive types only allows us to implement a case construct. This is not a significant obstacle, since we will shortly add general recursion to our language and then functions like addition, multiplication, exponentiation, and greatest common divisor can be implemented simply and uniformly. On the positive side, we have a constant-time predecessor, which we did not have for Church numerals.

5 Preservation and Progress

We have already seen the key idea in the preservation theorem; all other cases are simple and follow familiar patterns.

For progress, we first need a canonical form theorem. We get the new case

(vi) If $\cdot \vdash v : \mu\alpha.\,\tau$ and v *value* then $v = \mathsf{fold}\ v'$ for a value v'.

This follows, as before, by analyzing the cases for typing and values.

The critical case in the proof of progress (by rule induction on the given typing derivation) is

$$\frac{\cdot \vdash e_1 : \mu\alpha.\,\tau}{\cdot \vdash \mathsf{unfold}\ e_1 : [\mu\alpha.\,\tau/\alpha]\tau}\ \mathsf{tp/unfold}$$

If $e_1 \mapsto e_1'$ then, by rule, $\mathsf{unfold}\ e_1 \mapsto \mathsf{unfold}\ e_1'$. If e_1 is a value, then the canonical forms theorem tells us that $e_1 = \mathsf{fold}\ v_2$ for some value v_2. Therefore, the $\mathsf{step/unfold}$ applies and $\mathsf{unfold}\ (\mathsf{fold}\ v_2) \mapsto v_2$.

6 Isorecursive Types

The new type constructor $\mu\alpha.\,\tau$ we have defined is called an *isorecursive type*, because we have and isomorphism

$$\mu\alpha.\,\tau \quad \underset{\mathsf{fold}}{\overset{\mathsf{unfold}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\cong}}}} \quad [\mu\alpha.\,\tau/\alpha]\tau$$

rather than an equality between the two types (which would be *equirecursive*). But is it really an isomorphism? Let's check the two directions.

First, we need to check that $\mathsf{unfold}\ (\mathsf{fold}\ v) = v$ for any value $v : [\mu\alpha.\,\tau/\alpha]\tau$. But immediately (by rule $\mathsf{step/unfold}$) we have

$$\mathsf{unfold}\ (\mathsf{fold}\ v) \mapsto v$$

so the two are certainly equal.

In the other direction, we need to verify that

$$\mathsf{fold}\ (\mathsf{unfold}\ v) \stackrel{?}{=} v \qquad \text{for any value } v : \mu\alpha.\,\tau$$

By the canonical forms theorem, $v = \mathsf{fold}\ v'$ for some value v'. Then we reason

$$\begin{array}{lll} & \mathsf{fold}\ (\mathsf{unfold}\ v) \\ = & \mathsf{fold}\ (\mathsf{unfold}\ (\mathsf{fold}\ v')) \\ \mapsto & \mathsf{fold}\ v' \\ = & v \end{array}$$

So, an isorecursive type is indeed isomorphic to its unfolding.

7 Excursion: Embedding the Untyped λ-Calculus

Now that we have recursive types, perhaps we can type $\lambda x.\, x\, x$, which we previously proved to have no type. And if that works, why stop there? Why not type the Y combinator itself? In an earlier lecture we convinced ourselves that $\lambda x.\, x\, x : \tau \to \sigma$ for any types τ and τ satisfying $\tau = \tau \to \sigma$. That's because x needs to take itself as an argument.

This does not seem promising, since we still cannot solve this equation! But we may be able to approximate it by an *isomorphism*. Can we find a type U such that $U \cong U \to \tau_2$. The unspecified type τ_2 gets in the way, so let's try it with $\tau_2 = U$. So, we have to solve

$$U \quad \underset{\text{fold}}{\overset{\text{unfold}}{\cong}} \quad U \to U$$

In our language, any recursive type equation has a solution (perhaps degenerate), so we just set

$$U = \mu\alpha.\, \alpha \to \alpha \cong U \to U$$

Let's try to type self-application at type $U \to U$.

$$\frac{\overset{?}{x : U \vdash x\, x : U}}{\cdot \vdash \lambda x.\, x\, x : U \to U}\ \mathsf{tp/lam}$$

This still does not work, but we can *unfold* the type of the first occurrence of x so it matches the type of its argument!

$$\frac{\dfrac{\dfrac{\dfrac{}{x : U \vdash x : U}\ \mathsf{tp/var}}{x : U \vdash \mathsf{unfold}\ x : U \to U}\ \mathsf{tp/unfold} \quad \dfrac{}{x : U \vdash x : U}\ \mathsf{tp/var}}{x : U \vdash (\mathsf{unfold}\ x)\ x : U}\ \mathsf{tp/app}}{\cdot \vdash \lambda x.\, (\mathsf{unfold}\ x)\ x : U \to U}\ \mathsf{tp/lam}$$

So, lo and behold, if we are willing to insert an unfold we can now type-check self-application.

Curious: can we do the same with the Y combinator? The answer is yes, but let's be even more ambitious: let's translate the whole untyped λ-calculus into our language! We write M for untyped expressions to distinguish them from the target language expressions e.

$$\text{Untyped Exps} \quad M \quad ::= \quad x \mid \lambda x.\, M \mid M_1\, M_2$$

We try to devise a translation $\ulcorner - \urcorner$ such that

$$\ulcorner M \urcorner : U$$

for any untyped expression M. To be more precise, assume the untyped expression has free variables $x_1, \ldots, x_n$, then we aim for

$$x_1 : U, \ldots, x_n : U \vdash \ulcorner M \urcorner : U$$

The reason all variables have type U because in the source they stand for an arbitrary untyped expression. We define

$$\begin{array}{lcl} \ulcorner x \urcorner & = & x \\ \ulcorner \lambda x.\, M \urcorner & = & \mathsf{fold}\ (\lambda x. \ulcorner M \urcorner) \\ \ulcorner M_1\, M_2 \urcorner & = & (\mathsf{unfold}\ \ulcorner M_1 \urcorner)\ \ulcorner M_2 \urcorner \end{array}$$

We suggest you go through these definitions and type-check them, keeping in mind the all-important

$$U \quad \underset{\mathsf{fold}}{\overset{\mathsf{unfold}}{\cong}} \quad U \to U$$

The type-correctness of this translation means we have a very direct representation of the whole *untyped λ-calculus* in our language, using only a single type U (but exploiting recursive types). Therefore, the untyped λ-calculus is sometimes referred to as the *unityped λ-calculus* because it can be represented with a single universal type U.

Since the Y combinator is only a particular untyped λ-expression, we can also translate it into the target.

However, there is still a fly in the ointment: even though we know the target is well-typed, we don't know if it behaves correctly, operationally. Under some definitions it does not. For example, $\lambda x.\, \Omega$ has no normal form, but $\ulcorner \lambda x.\, \Omega \urcorner = \mathsf{fold}\ (\lambda x. \ulcorner \Omega \urcorner)$ is a value and does not take a step. We will discuss at a later point how to bridge this gap, which is not straightforward.

Here is the code we wrote in LAMBDA in lecture. Here `$` stands for μ, so that `$U. U -> U` stands for $\mu U.\, U \to U$. Note that `$` binds a type variable, so we could equally well have written `$a. a -> a`, but it is easier to ready if we reuse the intended name on the left-hand side of the definition as the name of the bound variable.

```
1  type U = $U. U -> U
2  decl lam : (U -> U) -> U
3  decl app : U -> (U -> U)
4
5  defn lam = \f. fold f
6  defn app = \e1. \e2. (unfold e1) e2
7
8  defn omega = lam (\x. app x x)
9  defn Omega = app omega omega
10
11 eval omega_val = omega
12
13 fail eval 100000 Omega_val = Omega
```

Listing 1: Untyped λ-calculus in LAMBDA

8 Fixed Point Expressions

We have added recursive types that solve recursive type equations. But in order to write all the programs we want (for example, on natural numbers all the recursive functions) we also need recursively defined expressions. The Y combinator is not directly available to us in the needed generality, even though it can be defined at type U. Instead we add a primitive, $\mathsf{fix}\ f.\, e$, where f is

a variable. It is not a value, and it steps by *unrolling* the fixed point:

$$\frac{}{\mathsf{fix}\ f.\, e \mapsto [\mathsf{fix}\ f.\, e/f]e}\ \mathsf{step/fix}$$

This "unrolling" is quite similar to unfolding a recursive type, but at the level of expressions. However, it is independent of recursive types and can be applied in full generality. One particular example is $\mathsf{fix}\ f.\, f \mapsto \mathsf{fix}\ f.\, f$ so in this language we can define $\bot = \mathsf{fix}\ f.\, f$ (see Exercise L6.3). Emboldened by this property, we imagine we might have in general

$$\frac{\Gamma, f : \boxed{} \vdash e : \boxed{}}{\Gamma \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{tp/fix}$$

but there are still some holes in this typing rule.

We want preservation to hold (progress is trivial to extend, because a fixed point always steps) so we need that

$$\cdot \vdash \mathsf{fix}\ f.\, e : \tau \text{ implies } \cdot \vdash [\mathsf{fix}\ f.\, e/f]e : \tau$$

From this we can deduce two things: first, $e : \tau$ because that is the result of substitution. And, second, for the substitution property to hold we need that $f : \tau$ so we can substitute $[\mathsf{fix}\ f.\, e/f]e$. Filling in this information:

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{tp/fix}$$

Now we have settled both statics and dynamic and have fixed point expressions available to us. For example

$$\begin{array}{lcl} \mathit{plus} & : & \mathit{nat} \to (\mathit{nat} \to \mathit{nat}) \\ \mathit{plus} & = & \mathsf{fix}\ p.\, \lambda n.\, \lambda k.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{l} \cdot \_ \Rightarrow k \mid \mathbf{r} \cdot m \Rightarrow \mathit{succ}\ (p\ m\ k)) \end{array}$$

There are a few unpleasant things about fixed point expressions. One is that it is neither a constructor nor a destructor of any particular type, but is applicable at any type τ. It thus violates one of the design principles of our language that we have followed so far. We may interpret this as an indication that recursion is a fundamental computational principle separate from any particular typing construct, but this is not a universally held view.

The second one is that in $\mathsf{fix}\ f.\, e$ the variable f does not stand for a value (like all other variables x we have used so far) but a expression (we substitute $\mathsf{fix}\ f.\, e$ for f, and that's not a value). To avoid this latter issue, in call-by-value languages sometimes the fixed point expression is limited to functions, as in $\mathsf{fun}\ f(x) = e$ where e can depend on both x and f.

In LAMBDA we reuse `$` to stand for fix for expressions, since it serves the same purpose and obeys corresponding laws to the type constructors. Below is our sample code using recursive types and fixed points from lecture.

This code also illustrates general *labeled sums* which have the form $(\ell_1 : \tau_1) + \cdots + (\ell_n : \tau_n)$. We recover the usual binary sum with $\ell_1 = \mathbf{l}$ and $\ell_2 = \mathbf{r}$. The *tags* or *labels* ℓ_i occupy a different name space and should not be confused with variables. We therefore write them bold in the mathematical presentation and prefix them with a tick mark (`'`) in LAMBDA code. The use of general tagged sums make actual code significantly more readable.

```
type bool = ('true : 1) + ('false : 1)

decl true : bool
decl false : bool

defn true = 'true ()
defn false = 'false ()

decl and : bool -> bool -> bool
defn and = \b. \c. case b of ('true u => c | 'false u => false)
conv and true true = true
conv and true false = false
conv and false true = false
conv and false false = false

decl and' : bool -> bool -> bool
defn and' = \b. \c. case b of ('true _ => case c of ('true _ => true | 'false _ => false)
                             |'false _ => false)

fail conv and = and'

fail type nat = ('zero : 1) + ('succ : nat)

type nat = $nat. ('zero : 1) + ('succ : nat)

decl zero : nat
decl succ : nat -> nat

defn zero = fold 'zero ()
defn succ = \n. fold 'succ n

eval one = succ zero

decl pred : nat -> nat
defn pred = \n. case unfold n
                  of ( 'zero _ => zero
                     | 'succ m => m )
eval two = pred (succ (succ one))

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

eval five = plus (plus two one) two

type list = $list. ('nil : 1) + ('cons : nat * list)
decl sum : list -> nat
defn sum = $sum. \l. case (unfold l) of ( 'nil _ => zero
                                        | 'cons (n, t) => plus n (sum t) )

type tree = $tree. ('leaf : 1) + ('node : tree * nat * tree)
```

Listing 2: Sample recursive types in LAMBDA

Exercises

Exercise 1 Prove adequacy of natural number encodings in type *nat*.

1. Define a (mathematical) function $\ulcorner n \urcorner$ on natural numbers n such that $\cdot \vdash \ulcorner n \urcorner : \mathit{nat}$ and $\ulcorner n \urcorner$ *value*.
2. Define a (mathematical) function $\llcorner v \lrcorner$ on values v with $\cdot \vdash v : \mathit{nat}$ returning the number represented by v.
3. Prove that the pair of functions $\ulcorner - \urcorner$ and $\llcorner - \lrcorner$ witness an isomorphism between the usual (mathematical) natural numbers and closed values of type *nat*.

Exercise 2 Consider the combinators Y and Z. Here Z, the call-by-value fixed point combinator, is defined as

$$Z = \lambda f.\, (\lambda x.\, f\, (\lambda v.\, x\, x\, v))\, (\lambda x.\, f\, (\lambda v.\, x\, x\, v))$$

1. Exhibit a difference between Y and Z under that assumption that the pure untyped λ-calculus follows a call-by-value evaluation strategy.
2. Give the translation $\ulcorner Z \urcorner : U$ into the universal type.

Exercise 3 Consider the type of lists of natural numbers

$$\mathit{list} = \mu\alpha.\, (\mathbf{nil} : 1) + (\mathbf{cons} : \mathit{nat} \times \alpha) \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \mathit{nat} \times \mathit{list})$$

Define the following functions (including *plist*) file. Feel free to use any definition of *nat* consistent with the natural numbers.

(i) $\mathit{nil} : \mathit{list}$, the empty list.

(ii) $\mathit{cons} : \mathit{nat} \times \mathit{list} \to \mathit{list}$, adding an element to a list. Include at least 1 test.

(iii) $\mathit{append} : \mathit{list} \to \mathit{list} \to \mathit{list}$, appending two lists. Include at least 1 test.

(iv) $\mathit{reverse} : \mathit{list} \to \mathit{list}$, reversing a list. Include at least 1 test.

(v) $\mathit{itlist} : \mathit{list} \to \forall\beta.\, (\mathit{nat} \times \beta \to \beta) \to \beta \to \beta$ satisfying

$$\begin{array}{lcl} \mathit{itlist}\ \mathit{nil}\ [\tau]\ f\ c & = & c \\ \mathit{itlist}\ (\mathit{cons}\ \langle n, l \rangle)\ [\tau]\ f\ c & = & f\ \langle n, \mathit{itlist}\ l\ [\tau]\ f\ c \rangle \end{array}$$

where you may take equality to be extensional. This captures *iteration* over lists, for the special case where the elements are all natural numbers. You do not need to prove the correctness of your representation, nor provide any testing.

(vi) Design a type and implementation for *primitive recursion* over lists, defining a function *plist*. Note that we do *not* ask for primitive recursion over the naturals contained in the list, only over the list itself. You do not need to prove the correctness of *plist*, nor provide any testing.

Exercise 4 It is often intuitive and useful to define types in a mutually recursive way. For example, we might specify the even and odd natural numbers in unary representation with the following desired isomorphisms:

$$\begin{array}{lcl} \mathit{even} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{odd}) \\ \mathit{odd} & \cong & () + (\mathbf{succ} : \mathit{even}) \end{array}$$

Here the empty parenthesis $()$ are used to indicate that $(\mathbf{succ} : \mathit{even})$ is a disjoint sum with just a single alternative. The only value v of type *odd* would be fold $(\mathbf{succ} \cdot v')$ with $v' : \mathit{even}$. Part of this task will be to find a representation of such types using the explicit recursive type constructor $\mu\alpha.\,\tau$.

Let the type of bit strings (which, during lecture, we used to represent numbers in binary form) be defined as

$$\begin{array}{lcl} \mathit{bits} & \cong & (\mathbf{b0} : \mathit{bits}) + (\mathbf{b1} : \mathit{bits}) + (\mathbf{e} : 1) \\ \mathit{bits} & = & \mu\alpha.\,(\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1) \end{array}$$

We say a bit string has parity 0 if it has an even number of 0s and 1 if it has an odd number of 1s. The answer to the questions below should be included in your solution.

(i) Define isomorphisms to be satisfied by two types *bits0* and *bits1*, where the values of type *bits0* are exactly the bit strings with parity 0, and the values of type *bits1* are exactly the bit strings with parity 1.

(ii) Give explicit definitions *bits0* $= \ldots$ and *bits1* $= \ldots$ using the recursive type constructor $\mu\alpha.\,\tau$ satisfying this specification.

(iii) We now define a type

$$\mathit{parity} \quad = \quad (\mathbf{p0} : 1) + (\mathbf{p1} : 1)$$

Define a function $\mathit{parity} : \mathit{bits} \to \mathit{parity}$ that computes the parity of the given bit string.

(iv) Next we define

$$\begin{array}{lcl} \mathit{par0} & = & (\mathbf{p0} : 1) + () \\ \mathit{par1} & = & () + (\mathbf{p1} : 1) \end{array}$$

It should be the case that

$$\begin{array}{lcll} \mathit{parity}\ v_0 & \mapsto^* & w_0 & \text{where } w_0 : \mathit{par0} \text{ if } v_0 : \mathit{bits0} \\ \mathit{parity}\ v_1 & \mapsto^* & w_1 & \text{where } w_1 : \mathit{par1} \text{ if } v_1 : \mathit{bits1} \end{array}$$

Does your implementation of *parity* have either following types?

$$\begin{array}{lcl} \mathit{parity} & : & \mathit{bits0} \to \mathit{par0} \\ \mathit{parity} & : & \mathit{bits1} \to \mathit{par1} \end{array}$$

If not, explain why not. We are not looking for a paraphrase of the error message, but a brief analysis why the two types above may be difficult to verify for a type-checker.

If yes, explain briefly which feature of your implementation made it possible for the type-checker to verify both of these properties.

The explanations should be included your solution. You may use the delimited comments `(* <comment> *)` for this purpose.