Lecture Notes on Representation Independence

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
Thu Oct 30, 2025

In Lecture 11 we introduced existential types and explored how they seem to enforce data abstraction. But we didn't have the tools to prove it—now we do! We determine the clauses for existential types in the definition of the binary logical relations and then apply the definition to show logical equality of different implementations of the same abstract type.

1 Existential Types Revisited

Recall the typing rules for existential types.

$$\frac{\Gamma \vdash \sigma \ \mathit{type} \quad \Gamma \vdash e : [\sigma/\alpha]\tau}{\Gamma \vdash \langle [\sigma], e\rangle : \exists\alpha.\,\tau}\ \mathsf{tp/paire}$$

$$\frac{\Gamma \vdash e : \exists\alpha.\,\tau \quad \Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{case}\ e\ (\langle [\alpha], x\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}^*$$

(*) α not in Γ, e, or τ'

For the same reason as for universal types, we cannot simply define $[\exists\alpha.\,\tau]$ with reference to $[\sigma/\alpha]\tau$: it would break the inductive nature of the definition. Instead, the type variable α has to stand for a relation R on values.

(∃) $v \sim v' \in [\exists\alpha.\,\tau]$ iff $v = \langle [\sigma], v_1\rangle$ and $v' = \langle [\sigma'], v_1'\rangle$ for some closed types σ, σ' and values v_1, v_1', and there is a relation R such that $v_1 \sim v_1' \in [[R/\alpha]\tau]$.

We mostly use this when R is a relation between values $v_1 : \sigma$ and $v_1' : \sigma'$, but that is not required. The fundamental theorem extends to this case, with reasoning analogous to the universal case.

2 Abstraction Representations of Numbers

Our previous example for existential types considered integers to be represented as a pair of natural numbers (so that $\langle n, k\rangle$ represents $n - k$) or as a pair of a sign and the absolute value. In this lecture we consider just natural numbers, where one (unary) representation is by zero and

the successor function while the more efficient representation in base 2. How can we arrange an implementation that makes it look like numbers are defined in unary (so, for example, we can use simple induction proofs), but this implementation can be swapped out for bit strings without affecting any client.

For reference, we start with the following definitions

```
type nat = $nat. ('zero : 1) + ('succ : nat)
decl zero : nat
defn zero = fold 'zero ()
decl succ : nat -> nat
defn succ = \n. fold 'succ n

type bin = $bin. ('b0 : bin) + ('b1 : bin) + ('e : 1)
decl bzero : bin
defn bzero = fold 'e ()

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

We assume that our binary representation allows leading zeros so that, for example, the number $1 = (1e)_2 = (10e)_2 = (100e)_2 = \cdots$. As in this remark, we use a shorthand for bit strings, which is related to their rendering in LAMBDA in the obvious way.

We now make the representation of the concrete type `nat` abstract by using constructor functions rather than labels of a sum. We start with

$$\mathsf{NAT} = \exists\alpha.\, (\mathbf{zero} : \alpha)\ \&\ (\mathbf{succ} : \alpha \to \alpha)\ \&\ \ldots$$

where we left room for more. To use an implementation $\mathsf{N} : \mathsf{NAT}$ we would "open" it and then construct elements. For example:

```
type NAT = ?a. ('zero : a) & ('succ : a -> a) & ...

decl Nat : NAT
defn Nat =

decl example : ...
defn example = case Nat of (([a], N) => N.'succ (N.'succ (N.'zero)))
```

This illustrates how to construct some unknown representation of the number 2, but it does not type-check. That's because `N.'succ (N.'succ (N.'zero)) : a` but `a` is not allowed to escape the scope of the binding `[a]`. So we need some way to observe the numbers. Our bias here is towards the unary representation, so we declare a function out : $\alpha \to (\mathbf{zero} : 1) + (\mathbf{succ} : \alpha)$ that exposes one layer of the natural number so we can pattern-match against it.

Here is an example, which will return false because the number 2 is not 0.

```
type NAT = ?a. ('zero : a) & ('succ : a -> a)
               & ('out : a -> ('zero : 1) + ('succ : a))

decl Nat : NAT
defn Nat = ...

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

```
decl example : bool
defn example = case Nat of (([a], N) =>
     case N.'out (N.'succ (N.'succ (N.'zero))) of
     ( 'zero () => 'true()
     | 'succ y => 'false() ))
```

It should be clear that this is an example of very general construction of turning an concrete type into an abstract one. In fact, a similar transformation is applied to `datatype` declarations in the *Harper-Stone* [2000] interpretation of Standard ML. Because of the existential quantifier in the translation, each datatype declaration generates a fresh type. So, for example, with

```
datatype nat = zero | succ of nat
datatype pos =        succ of nat
```

there is no subtyping relationship between `pos` and `nat`, and we cannot mix them. We say that datatype declarations in ML are *generative*.

Returning to our example, we can give a *unary* implementation of type `NAT` directly using the concrete type `nat`.

```
decl UNat : NAT
defn UNat = ([nat], (| 'zero => fold 'zero ()
                     | 'succ => \n. fold 'succ n
                     | 'out => \n. unfold n |) )
```

For the binary implementation, we need to define the out function. Because leading zeros are allowed, this has to be recursive.

```
decl bout : bin -> ('zero : 1) + ('succ : bin)
defn bout = $bout. \x. case unfold x of
            ( 'e () => 'zero ()
            | 'b1 y => 'succ (fold 'b0 y)
            | 'b0 y => case bout y of
                       ( 'zero () => 'zero ()
                       | 'succ z => 'succ (fold 'b1 z) ) )
                          (* x = 2y = 2(z+1) = 2z+2, so x-1 = 2z+1 *)

decl BNat : NAT
defn BNat = ([bin], (| 'zero => fold 'e ()
                     | 'succ => \x. inc x
                     | 'out => \x. bout x |) )
```

If would like an implementation that is independent of the representation, we can define a function from the existential type. In SML terminology this would be called a *functor* because existential types are considered part of the module layer which is separate from the core language.

Here, we could write:

```
decl is_zero : !a. (('zero : 1) + ('succ : a)) -> bool
defn is_zero = /\a. \s. case s of ( 'zero() => 'true() | 'succ y => 'false () )

decl functor : NAT -> bool
defn functor = \Nat. case Nat of (([a], N) =>
     is_zero [a] (N.'out (N.'succ (N.'succ (N.'zero)))))

eval ex_unary = functor UNat
eval ex_binary = functor BNat
```

Note that the `is_zero` function must be polymorphic so that it can be used in the scope of `[a]`, where `a` is not known at the time of typechecking.

For reference, the complete implementation can be found in lec16.cbv.

If the unary and binary implementation are indeed equivalent, then the outcome of the two evaluations at the end must be the same. In this example, they are both false. This is what we are about to verify now.

3 Logical Equivalence on Abstract Types

In our example, we ask if

$$\mathsf{UNat} \sim \mathsf{BNat} \in [\mathsf{NAT}]$$

which unfolds into demonstrating that there is a relation R

$$\begin{array}{l}\mathsf{zero} \sim \mathsf{bzero} \in [R] \\ \mathsf{succ} \sim \mathsf{inc} \in [R \to R] \\ \lambda n.\, \mathbf{unfold}\ n \sim \lambda x.\, \mathsf{bout}\ x \in [R \to (\mathbf{zero} : 1) + (\mathbf{succ} : R)]\end{array}$$

It is generally most straightforward if the relation R directly relates elements of the implementation types. Here, we are looking for a relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$.

The relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ we seek needs to relate natural numbers in two different representations. It is convenient and general to define such relations by using inference rules. In particular, this will allow us to prove properties by *rule induction*. An alternative approach would be to define such relations as functions, but because representations are often not unique this is not quite as general.

Once we have made this decision, the relation could be based on the structure of $x : \mathit{bin}$ or on the structure of $n : \mathit{nat}$. The latter may run into difficulties because each number actually corresponds to infinitely many numbers in binary form: just add leading zeros that do not contribute to its value. Therefore, we define it based on the binary representation. In order to define it concisely, we use a representation function for (mathematical) natural numbers k into our language of values defined by

$$\begin{array}{lcl}\overline{0} & = & \mathsf{fold}\ \mathbf{zero} \cdot \langle\,\rangle \\ \overline{n+1} & = & \mathsf{fold}\ \mathbf{succ} \cdot \overline{n}\end{array}$$

We also write binary number representations in compressed form with the least significant bit first:

$$\begin{array}{lcl}\mathsf{0}x & = & \mathsf{fold}\ \mathbf{b0} \cdot x \\ \mathsf{1}x & = & \mathsf{fold}\ \mathbf{b1} \cdot x \\ \mathsf{e} & = & \mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle\end{array}$$

Recall the ambiguity that e, 0e, 00e etc. all represent the natural number 0.

We then define:

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e \qquad \frac{\overline{k}\ R\ x}{\overline{2k}\ R\ \mathsf{0}x}\ R_0 \qquad \frac{\overline{k}\ R\ x}{\overline{2k+1}\ R\ \mathsf{1}x}\ R_1$$

As usual, we consider $n\ R\ x$ to hold if and only if we can derive it using these rules.

4 Verifying the Relation

Because our signature exposes three fields, we now have to check three properties:

$$\begin{array}{l}\mathsf{zero} \sim \mathsf{bzero} \in [R] \\ \mathsf{succ} \sim \mathsf{inc} \in [R \to R] \\ \mathsf{out} \sim \mathsf{bout} \in [R \to (\mathbf{zero} : 1) + (\mathbf{succ} : R)]\end{array}$$

We already have by definition that $v \sim v' \in [R]$ iff $v\ R\ v'$. For convenience, we also define the notation $e\ \mathbb{R}\ e'$ to stand for $e \approx e' \in [\![R]\!]$, which means that $e \mapsto^* v$ and $e' \mapsto^* v'$ with $v\ R\ v'$

Lemma 1 $\mathsf{zero} \sim \mathsf{bzero} \in [R]$.

Proof: Since $\overline{0} = \mathsf{zero}$ and $\mathsf{e} = \mathsf{bzero}$ this is just the contents of rule R_e. □

Lemma 2 $\mathsf{succ} \sim \mathsf{inc} \in [R \to R]$.

Proof: By definition of logical equality, this is equivalent to showing

For all values $n : \mathsf{nat}$ *and* $x : \mathsf{bin}$ *with* $n\ R\ x$ *we have* $(\mathsf{succ}\ n)\ \mathbb{R}\ (\mathsf{inc}\ x)$.

Since R is defined inductively by a collection of inference rules, the natural attempt is to prove this by rule induction on the given relation, namely $n\ R\ x$.

Case: Rule

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

with $n = \overline{0}$ and $x = \mathsf{e}$. We have to show that $(\mathsf{succ}\ \overline{0})\ \mathbb{R}\ (\mathsf{inc}\ \mathsf{e})$

$\mathsf{succ}\ \overline{0} \mapsto^* \overline{1}$	By defn. of succ
$\mathsf{inc}\ \mathsf{e} \mapsto^* 1\mathsf{e}$	By defn. of inc
$\overline{1}\ R\ 1\mathsf{e}$	By rules R_1 and R_e

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k}\ R\ 0y}\ R_0$$

where $x = 0y$ and $n = \overline{2k}$. To prove is $(\mathsf{succ}\ \overline{2k})\ \mathbb{R}\ (\mathsf{inc}\ 0y)$.

$\mathsf{succ}\ \overline{2k} \mapsto^* \overline{2k+1}$	By defn of succ
$\mathsf{inc}\ 0y \mapsto^* 1y$	By defn. of inc
$\overline{k}\ R\ y$	Premise in this case
$\overline{2k+1}\ R\ 1y$	By rule R_1

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ 1y}\ R_1$$

where $n = \overline{2k+1}$ and $x = 1y$. To show: $(\mathsf{succ}\ \overline{2k+1})\ \mathbb{R}\ (\mathsf{inc}\ 1y)$.

$\mathsf{succ}\ \overline{2k+1} \mapsto^* \overline{2k+2}$	By defn. of succ
$\mathsf{inc}\ 1y \mapsto^* \mathsf{b0}\ (\mathsf{inc}\ y) \mapsto^* 0z$ where $\mathsf{inc}\ y \mapsto^* z$	By defn. of inc
Remains to show: $\overline{2k+2}\ R\ 0z$	
$\overline{k}\ R\ y$	Premise in this case
$(\mathsf{succ}\ \overline{k})\ \mathbb{R}\ (\mathsf{inc}\ y)$	By ind. hyp.
$\overline{k+1}\ R\ z$	By defn. of $\mathbb{R}$ and succ
$\overline{2(k+1)}\ R\ 0z$	By rule R_0
$\overline{2k+2}\ R\ 0z$	By arithmetic

□

In order to prove the relation between the implementation of the predecessor function we write out the interpretation of the type $(\mathbf{zero} : 1) + (\mathbf{succ} : R)$.

$v \sim v' \in [(\mathbf{zero} : 1) + (\mathbf{succ} : R)]$ iff ($v = \mathbf{zero} \cdot \langle\,\rangle$ and $v' = \mathbf{zero} \cdot \langle\,\rangle$)
or ($v = \mathbf{succ} \cdot v_1$ and $v' = \mathbf{succ} \cdot v_1'$ and $v_1\ R\ v_1'$.

Lemma 3 $\mathsf{out} \sim \mathsf{bout} \in [R \to (\mathbf{zero} : 1) + (\mathbf{succ} : R)]$

Proof: By[1] definition of logical equality, this is equivalent to showing

For all values $n : \mathsf{nat}$ *and* $x : \mathsf{bin}$ *with* $n\ R\ x$ *we have* $\mathsf{out}\ n \approx \mathsf{bout}\ x \in [\![1 + R]\!]$.

We break this down into two properties, based on n.

(i) For all $\overline{0}\ R\ x$ we have $\mathsf{out}\ \overline{0} \approx \mathsf{bout}\ x \in [\![(\mathbf{zero} : 1)]\!]$.

(ii) For all $\overline{k+1}\ R\ x$ we have $\mathsf{out}\ \overline{k+1} \approx \mathsf{bout}\ x \in [\![(\mathbf{succ} : R)]\!]$.

For part (i), we note that $\mathsf{out}\ \overline{0} \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$, so all that remains to show is that $\mathsf{bout}\ x \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$ for all $\overline{0}\ R\ x$. We prove this by rule induction on the derivation of $\overline{0}\ R\ x$.

Case(i):

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

where $x = \mathsf{e}$. Then $\mathsf{bout}\ x = \mathsf{bout}\ \mathsf{e} \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$.

Case(i):

$$\frac{\overline{k}\ R\ y}{\overline{2k}\ R\ 0y}\ R_0$$

where $x = 0y$ and $2k = 0$ and therefore also $k = 0$. Then

$\mathsf{bout}\ y \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$	By ind. hyp.
$\mathsf{bout}\ 0y \mapsto^* \mathbf{zero} \cdot \langle\,\rangle$	By defn. of bout and computation

[1] We skipped this part of the proof in lecture.

Case(i):

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ 1y}\ R_1$$

This case is impossible since $2k + 1 \neq 0$.

Now we come to Part (ii). We note that out $\overline{k+1} \mapsto^* \mathbf{succ} \cdot \overline{k}$ so what we have to show is that

(ii)′ For all $\overline{k+1}\ R\ x$ we have bout $x \mapsto^* \mathbf{succ} \cdot y$ with $\overline{k}\ R\ y$.

We prove this by rule induction on the derivation of $\overline{k+1}\ R\ x$.

Case(ii):

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

is impossible since $\overline{0} \neq \overline{k+1}$.

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j}\ R\ 0y}\ R_0$$

where $k + 1 = 2j$ and $x = 0y$.

$j = j' + 1$ for some j'	Since $j > 0$ by arithmetic
bout $y \mapsto^* \mathbf{succ} \cdot z$ with $\overline{j'}\ R\ z$	By ind. hyp.
bout $0y \mapsto^* \mathbf{succ} \cdot 1z$	By defn. of bout
$\overline{2j'+1}\ R\ 1z$	By rule R_1
$\overline{k}\ R\ 1z$	By equality

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j+1}\ R\ 1y}\ R_1$$

for $k + 1 = 2j + 1$ and $x = 1y$. Then

bout $1y \mapsto^* \mathbf{succ} \cdot 0y$	By defn. of bout
$\overline{j}\ R\ y$	Premise in this case
$\overline{2j}\ R\ 0y$	By rule R_0
$\overline{k}\ R\ 0y$	By equality

□

5 Concrete Types vs. Abstract Types[2]

An interesting observation about the logical equivalence of the two implementation of counters is that, had we omitted the decrement operation from the interface, then universal relation ($n\ U\ x$ for all values n : nat and x : bin) also allows us to prove equivalence. This is because without the decrement we can create a counter and increment it, but can never observe any of its properties.

This raises the question how we should more generally observe properties of elements of abstract type. There is no universal answer: different applications or libraries require different choices. A particularly frequent and useful technique is to endow abstract types with a *view*, realized by a function called *expose* or *out*.

As an example, let's reconsider the (concrete) type of binary numbers:

$$\mathsf{bin} = \mu\mathsf{bin}.\ (\mathbf{b0} : \mathsf{bin}) + (\mathbf{b1} : \mathsf{bin}) + (\mathbf{e} : 1)$$

This concrete type allows clients to construct numbers with leading zeros, which may be undesirable because it complicates certain algorithms (e.g., equality of binary numbers). In this case, one solution would be to split the type bin into positive numbers pos and numbers in standard form std (with no leading zeros), which we did in an earlier exercise. However, now all client code has to be aware of these two types and use them appropriately. Alternatively, we can create an abstract type providing the constructors in the interface. to start with, we would have

$$\begin{array}{lll}\mathsf{BIN} = \exists\alpha.\ (\mathbf{b0} : \alpha \to \alpha) & \% \textit{ b0} \\ \quad \&(\mathbf{b1} : \alpha \to \alpha) & \% \textit{ b1} \\ \quad \&(\mathbf{e} : \alpha) & \% \textit{ e} \\ \quad \&\ldots\end{array}$$

The implementation of these constructors can make sure that only numbers with no leading zeros are ever created. But how do we *observe* a value of the abstract type? The technique is to provide a function out : $\alpha \to \tau$ where τ is usually a sum that the client can pattern match against. Here we would have

$$\begin{array}{lll}\mathsf{BIN} = \exists\alpha.\ (\mathbf{b0} : \alpha \to \alpha) & \% \textit{ b0} \\ \quad \&(\mathbf{b1} : \alpha \to \alpha) & \% \textit{ b1} \\ \quad \&(\mathbf{e} : \alpha) & \% \textit{ e} \\ \quad \&(\alpha \to (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1))\end{array}$$

The result out v where v is a value of the abstract type allows one level of pattern matching. The value tagged by **b0** or **b1** is again of abstract type and we must apply out again. If we want to allow multiple levels of pattern matching we would need some special syntax to designate out as a view with a corresponding pattern constructor, say, out^{-1}. Then matching the value $v : \alpha$ against the pattern out^{-1} p will evaluate out $v \mapsto^* w$ and match w against p.

We show the implementation of this abstract type in LAMBDA. In this example, the out function just has to unfold the recursive type to expose the sum underneath.

```
1 type BIN = ?a. (a -> a)    % b0 = \n. 2n
2              * (a -> a)    % b1 = \n. 2n+1
3              * a           % e = 0
4              * (a -> (('b0 : a) + ('b1 : a) + ('e : 1))) % out
5
```

[2]Not covered in lecture this year

```
decl Bin : BIN
defn Bin = ([bin], \x. case x of ( fold 'e () => e
                                 | _  => fold 'b0 x ),
                    \x. fole 'b1 x, e, \x. unfold x)
```

The only other interesting part of this is the constructor corresponding to the tag **b0** ensures that it never constructs 0e but returns e instead, thereby making the representation unique.

6 Polymorphic Lists

In functional languages lists are usually represented by a so-called *type constructor* list : type→type. That is, for any type τ, we would have

list $\tau = \mu\beta.\,(\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \beta)$

We have not introduced type constructors into our language, so we cannot express this directly. But we can formulate it as an abstract type. Essentially, the implementation is a *function* which takes an element type τ as an argument and returns in instance of an existential type for this particular τ.

```
type LIST = !a. ?b. b             % nil
                  * (a * b -> b)  % cons x l
                  * (b -> ('nil : 1) + ('cons : a * b)) % out l
```

There is, however, a quirk with the implementation that often comes up with abstract types. If we have an implementation of lists, for example

```
decl List : LIST
defn List = /\a. ([$list. ('nil : 1) + ('cons : a * list)],
                  fold 'nil (),
                  \p. fold 'cons p,
                  \l. unfold l)
```

then two different uses of this, e.g., List [nat] and List [nat] are incompatible because there is no way the type checker can know that the different abstract types are actually equal. We summarize this sometimes by saying that abstract types are *generative* as explained earlier in this lecture because every time an implementation of an abstract type is opened, a fresh type variable is generated to stand for the implementation type.

This implementation of lists, by the way, is called a *functor* in languages in the ML family, because it is a module-level function. We think of it this way because it is a function that returns an abstract type when given a type.

7 The Upshot

Because the two implementations are logically equal we can replace one implementation by the other without changing any client's behavior. This is because all clients are parametric, so their behavior does not depend on the library's implementation.

It may seem strange that this is possible because we have picked a particular relation to make this proof work. Let us reexamine the tp/casee rule:

$$\frac{\Gamma \vdash e : \exists\alpha.\,\tau \quad \Gamma, \alpha\ \mathsf{type}, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{case}\ e\ (\langle[\alpha], x\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}$$

In the second premise we see that the client e' is checked with a fresh type α and $x : \tau$ which may mention α. If we reify this into a function, we find

$$\Lambda\alpha.\, \lambda x.\, e' : \forall\alpha.\, \tau \to \tau'$$

where τ' does not depend on α.

By Reynolds's parametricity theorem we know that this function is parametric. This can now be applied for any relation R to conclude that if $v_0 \sim v_0' \in [[R/\alpha]\tau]$ then $(\Lambda\alpha\lambda x.\, e')[R]\, v_0 \approx (\Lambda\alpha.\, \lambda x.\, e')[R]\, v_0' \in [\![[R/\alpha]\tau']\!]$. But α does not occur in τ', so this is just saying that $[R/\alpha, v_0/x]e' \approx [R/\alpha, v_0'/x]e' \in [\![\tau']\!]$. So the result of substituting the two different implementations is equivalent.

Exercises

Exercise 1 We can represent integers a as pairs $\langle x, y\rangle$ of natural numbers where $a = x - y$. We call this the *difference representation* and call the representation type diff.

$$\begin{array}{lcl} \mathsf{nat} & = & \mu\alpha.\, (\mathbf{zero} : 1) + (\mathbf{succ} : \mathsf{alpha}) \\ \mathsf{diff} & = & \mathsf{nat} \times \mathsf{nat} \end{array}$$

In your answers below you may use *constructors* zero : nat and succ : nat → nat to construct terms of type nat. If you need auxiliary functions on natural numbers, you should define them.

1. Define a function nat2diff : nat→diff that, when given a representation of the natural number n returns an integer representing n.

2. Define a constant d_zero : diff representing the integer 0 as well as functions dplus : diff → diff → diff and dminus : diff → diff → diff representing addition and subtraction on integers, respectively.

3. Consider the type
   $$\mathsf{ord} = (\mathbf{lt} : 1) + (\mathbf{eq} : 1) + (\mathbf{gt} : 1)$$
   that represents the outcome of a comparison (**lt** = "less than", **eq** = "equal", **gt** = "greater than"). Define a function dcompare : diff → diff → ord to compare the two integer arguments. Again, you may use lt, eq and gt as constructors.

Exercise 2 We consider an alternative *signed representation* of integers where

$$\mathsf{sign} = (\mathbf{pos} : \mathsf{nat}) + (\mathbf{neg} : \mathsf{nat})$$

where $\mathbf{pos} \cdot x$ represents the integer x and $\mathbf{neg} \cdot x$ represents the integer $-x$. In your answers below you may use pos and neg as data constructors, to construct elements of type sign. Define the following functions in analogy with Exercise 1:

1. nat2sign : nat → sign

2. s_zero : sign

3. s_plus : sign → sign → sign

4. s_minus : sign → sign → sign

5. s_compare : sign $\to$ sign $\to$ ord

Exercise 3 In this exercise we pursue two different implementations of an integer counter, which can become negative (unlike the natural number counter in this lecture). The functions are simpler than the ones in Exercise 1 and Exercise 2 so that the logical equality argument is more manageable. We specify a signature

$$\mathsf{INTCTR} = \exists\alpha.\,(\mathbf{new} : \alpha)\ \&\ (\mathbf{inc} : \alpha \to \alpha)\ \&\ (\mathbf{dec} : \alpha \to \alpha)\ \&\ (\mathbf{is0} : \alpha \to \mathsf{bool})$$

where new, inc, dec and is0 have their obvious specification with respect to integers, generalizing the CTR type defined in the last lecture and used in this one.

1. Define the constants and functions d_zero, d_inc, d_dec and d_is0 for the implementation where type ictr = diff from Exercise 1.
2. Define the constants and functions szero, s_inc, s_dec and s_is0 for the implementation where type ictr = sign from Exercise 2.

Now consider the two definitions

$$\mathsf{DiffCtr} : \mathsf{INTCTR} = \langle [\mathsf{diff}], \langle\!| \mathsf{d\_zero}, \mathsf{d\_inc}, \mathsf{d\_dec}, \mathsf{d\_is0} |\!\rangle\rangle$$
$$\mathsf{SignCtr} : \mathsf{INTCTR} = \langle [\mathsf{sign}], \langle\!| \mathsf{s\_zero}, \mathsf{s\_inc}, \mathsf{s\_dec}, \mathsf{s\_is0} |\!\rangle\rangle$$

4. Prove that DiffCtr $\sim$ SignCtr $\in$ [INTCTR] by defining a suitable relation R : diff $\leftrightarrow$ sign and proving that the corresponding projections of the lazy record are related.

References

Robert Harper and Christopher A. Stone. A type-theoretic interpretation of Standard ML. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 341–388. MIT Press, 2000.