Assignment 4
Type Isomorphisms
Sample Solution

15-814: Types and Programming Languages
Frank Pfenning

Due Tue Sep 30, 2025
70 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please use the attached template to typeset your assignment and make sure to include your full name and Andrew ID. For the written problems, you may also submit handwritten answers that have been scanned and are **easily legible**.

Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~fp/courses/15814-f25/assignments.html`.

You should hand in two files separately:

- `hw04.pdf` with the written answers to the questions.
- `hw04.cbv` with the code, where the solutions to the problems are clearly marked and auxiliary code (either from lecture or your own) is included so it passes the LAMBDA checker.

1 Isomorphism Witnesses

Task 1 (20 pts) Exhibit the functions *Forth* and *Back* witnessing the following isomorphisms. You do not need to prove that they constitute an ismorphism, just show the functions. We remain here in the pure language of where every function is terminating.

(i) $\tau \times (\sigma + \rho) \cong (\tau \times \sigma) + (\tau \times \rho)$

(ii) $2 \to \tau \cong \tau \times \tau$

(iii) $1 \to \tau \cong \tau$

(iv) $0 \to \tau \cong 1$

(v) $(\sigma + \rho) \to \tau \cong (\sigma \to \tau) \times (\rho \to \tau)$

Include the appropriately polymorphic functions in your file `hw04.cbv`. For example, in the implementation of part (iii) your functions should have type *forth_iii* : $\forall\alpha.\ (1 \to \alpha) \to \alpha$ and *back_ii* : $\forall\alpha.\ \alpha \to (1 \to \alpha)$.

See file hw04soln.cbv

Task 2 (20 pts) Verify that the composition $\mathit{Forth} \circ \mathit{Back} \approx \lambda g.\, g$ where *Forth* and *Back* coerce from a curried function to its tupled counterpart.

$$\begin{array}{lcl} \mathit{Forth} & : & ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho)) \\ \mathit{Forth} & = & \lambda f.\, \lambda x.\, \lambda y.\, f\, \langle x, y \rangle \end{array}$$

$$\begin{array}{lcl} \mathit{Back} & : & (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho) \\ \mathit{Back} & = & \lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow g\, x\, y) \end{array}$$

```
Forth o Back
=?=
\g. g : (t -> s -> r) -> (t -> s -> r)
(* by extensionality, it is sufficient to check the result
 * of applying both sides to arbitrary value f : t -> s -> r
 *)
(Forth o Back)(f)
= Forth(Back(f))
= Forth(\p. case p of ((x, y) => f x y))
= \x. \y. (\p. case p of ((x, y) => f x y)) (x, y)
=?= f : t -> s -> r
(* by extensionality, it is sufficient to check the
 * result of apply both sides to arbitrary v : t and w : s
 *)
(\x. \y. (\p. case p of ((x, y) => f x y)) (x, y)) v w
= (\p. case p of ((x, y) => f x y)) (v, w)
= case (v, w) of ((x, y) => f x y)
= f v w
=?=
f v w : r
(* and extensional equality has been verified *)
```

2 Recursive Types

Balanced ternary numbers are a representation of integers with some remarkable properties. This representation has three digits with values -1, 0, and 1. It represents any integer uniquely (assuming no leading 0s) and has some nice symmetry properties. For example, a number is negated just be negating every digit. An early computer built in Moscow in 1958 actually used balanced ternary numbers and ternary logic, instead of the binary system we are now used to. For example, the Wikipedia article on Balanced Ternary provides an introduction and more details.

In this problem, we ask you to implement ternary numbers and some simple operations on that representation.

Task 3 (30 pts) Implement the following types and functions in your `hw04.cbv` file.

```
type tern = ...

decl zero : tern                  % 0
decl plus1 : tern                 % 1
decl minus1 : tern                % -1
decl inc : tern -> tern           % increment
decl dec : tern -> tern           % decrement
decl neg : tern -> tern           % negate
decl plus : tern -> tern -> tern % addition

type sign = ('neg : 1) + ('zero : 1) + ('pos : 1)
decl sign_of : tern -> sign
```

You may define other types, functions, and predicates as you see fit. Efficiency should not be a significant consideration. Because we would like to represent integers of arbitrary size, we recommend a representation as a sequence of digits, with the least significant bit coming first (known as "little endian"). Your representation may allow leading 0 digits, which would be "trailing" if you choose the recommended representation.

Also provide a few test cases for each function.

See hw04soln.cbv