Assignment 5
Mutual Recursion

15-814: Types and Programming Languages
Frank Pfenning

Due Tuesday, October 13, 2020

You should hand in two files

- `hw05.pdf` with your written solutions to the questions
- `hw05.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 Programming with Lists

For the purpose of this task, use the language from Lecture 10, as defined in the rule sheet 10-rectypes-rules.pdf which should be consistent with the LAMBDA implementation. Specifically, sums are binary type constructors with injections **l** and **r**.

Task 1 (L10.3, 15 pts) Consider the type of lists of natural numbers

$$\mathit{list} = \rho\alpha.\,(\mathit{nat} \times \alpha) + 1 \cong (\mathit{nat} \times \mathit{list}) + 1$$

Define the following functions

(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.

(iii) $\mathit{append} : \mathit{list} \to \mathit{list} \to \mathit{list}$, appending two lists.

(iv) $\mathit{reverse} : \mathit{list} \to \mathit{list}$, reversing a list.

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

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

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.

(vi) Design a type and implementation for *primitive recursion* over lists, defining a function *plist*.

You should include all functions from this task (including *plist*) in your `hw05.cbv` file.

2 Mutually Recursive Types

For the purpose of the tasks in this section, use the language from Lecture 11 with variadic sums, as given by the defined in the rule sheet at 10-rectypes-rules.pdf, updated as described in Section L11.7. You do not need to provide the functions from these problems in your `hw05.cbv` file, but you are welcome to do so. If so, please indicate this in your written solution.

Task 2 (L11.1, 20 points) 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*).

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

(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 L11.4 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).

Task 3 (L11.2, 25 pts) 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*.

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

(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

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

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).