Assignment 8
Representation Independence

15-814: Types and Programming Languages
Frank Pfenning

Due Sat Nov 15, 2025
95 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 three separate files:

- `hw08.pdf` with the written answers to the questions.
- `hw08.cbv` with the two implementations of queues from Problem 1
- `hw08.prf` with the the proof terms from Problem 2.

1 Representation Independence [55 pts]

Task 1 (5 pts) Prove that all well-typed implementations of the type $\exists\alpha.(\mathbf{zero} : \alpha)\,\&\,(\mathbf{succ} : \alpha \to \alpha)$ are logically equivalent.

We now consider two implementation of queues in a functional language. Our choice of abstract type for queues is

$$\mathsf{QUEUE}\ \alpha = \exists q.\,(\mathbf{empty} : q)\ \&\ (\mathbf{enq} : \alpha \to q \to q)\ \&\ (\mathbf{deq} : q \to (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times q))$$

or, in the concrete syntax of LAMBDA:

```
type QUEUE = \a. ?q. ('empty : q)
                   & ('enq : a -> q -> q)
                   & ('deq : q -> ('nil : 1) + ('cons : a * q))
```

This type is almost the abstract version of polymorphic lists, except we have renamed the constructors to `'empty` and `'enq`. The reason is that the implementation should behave as a queue. For example, if we start with an empty queue and then enqueue 0 followed by enqueuing 1, then dequeueing will first give us 0 and then 1. In the starter code hw08.cbv there is a similar example. In your implementation you may use the `nil`, `cons`, `append`, and `reverse` operations as you wish and also implement additional auxiliary functions.

We start with a straightforward, but inefficient implementation.

Task 2 (10 pts) Implement `QList1 : !a. QUEUE a` where the queue is implemented as a simple list. New elements should be inserted at the back of the list, and dequeuing should take from the front of the list.

Next is a more efficient implementation with amortized constant cost for a sequence of enqueue and dequeue operations on a queue. For this bound to hold, the queue must be used in a *singlethreaded* manner (which we will eventually recognize as *affine*).

Task 3 (10 pts) Implement `QList2 : !a. QUEUE a` where the queue is implemented as two lists, `inp` and `out`. New elements are added to the front of the input list `inp`. For dequeuing, elements are taken from the front of the output list `out`. When the output list is empty, the input list is reversed to become the new output list before an element is dequeued. If both input and output lists are empty, then dequeueing returns `'nil ()` to indicate that the queue is empty.

The goal is to set up the proof that the two implementations are logically equivalent.

Task 4 (10 pts) Define a relation R between values of type list S and type list $S \times$ list S, given any relation S between the list elements. This relation should be suitable to show the logical equivalence of type QList1 S and QList2 S.

Task 5 (10 pts) Show the part of the proof of logical equivalence regarding the enqueue operation. Feel free to use high level notations for lists. You should state any lemmas regarding the `nil`, `cons`, `append`, and `reverse` operations you need, but you don't need to prove them.

Task 6 (5 pts) Assume we have given a well-typed polymorphic definition

```
decl Queue = !a. QUEUE a
defn Queue = ...
```

State an interesting property that parametricity alone guarantees regarding

```
Q.'deq (Q.'enq v (Q.'empty))
```

assuming the expression is well-typed (that is, you are in a context where we have opened up value of type QUEUE τ, naming it Q, as in `test` of hw08.cbv). In particular, what you deduce should not rely on the correctness of the implementation, just its type.

Task 7 (5 pts) Sketch the proof of the property from the previous task.

2 Proof Terms [40 pts]

Proof terms can be checked in the LAMBDA implementation by using files with the `.prf` extension. This rules out recursive types and also recursive programs, with otherwise adhering to the LAMBDA syntax. You can find examples from Lecture 17 in the file lec17.prf.

Task 8 (20 pts) One proposition is *more general* than another if we can instantiate the propositional variables in the first to obtain the second. For example, $A \supset (B \supset A)$ is more general than $A \supset (\bot \supset A)$ (with $[\bot/B]$) and $(C \wedge D) \supset (B \supset (C \wedge D))$ (with $[C \wedge D/A]$), but not more general than $C \supset (D \supset E)$.

For each of the following proof terms, give the most general proposition proved by it. (We are justified in saying "*the most general*" because the most general proposition is unique up to the names of the propositional variables.) Please include your answer in the `hw08.prf` file. For example, for

0. $\lambda u.\, u$

you would write

```
decl proof0 : !A. A -> A
defn proof0 = /\A. \u. u
```

1. $\lambda u.\, \lambda w.\, \lambda k.\, w\,(u\,k)$
2. $\lambda w.\, \langle(\lambda u.\, w\,(\mathbf{l} \cdot u)), (\lambda k.\, w\,(\mathbf{r} \cdot k))\rangle$
3. $\lambda x.\, (x.\mathbf{l})\,(x.\mathbf{r})\,(x.\mathbf{r})$
4. $\lambda x.\, \lambda y.\, \lambda z.\, (x\,z)\,(y\,z)$

Task 9 (20 pts) Write out a proof term for each of the following propositions. As you know from this lecture, this is the same as writing a program of the translated type in our program language without the use of fixed points.

5. $(A \wedge (A \supset \bot)) \supset B$
6. $(A \vee (A \supset \bot)) \supset (((A \supset \bot) \supset \bot) \supset A)$
7. $(A \vee (B \wedge C)) \supset (A \vee B) \wedge (A \vee C)$
8. $((A \vee B) \wedge (A \vee C)) \supset (A \vee (B \wedge C))$