CS 410/510 Compiling Advanced Languages Take-home Final – due 12 noon, Friday, June 9, 2000

Pencil and Paper

1. Consider the CPS version of the list length function, more or less as shown in class:

```
let len (xs : 'a list) : int =
  let rec len' : 'a list * (int -> int) -> int =
     function
       (h::t,k) -> len'(t, fun i -> k (i+1))
     | ([],k) -> k 0
  in len' (xs, fun i -> i)
```

(a) Using the technique described in class to represent closures as elements of a algebraic (variant) type, transform this function so that it manipulates algebraic type values instead of higher-order continuation functions.

(b) Consider the function

```
let g (xs : 'a list) : int =
  let rec g' : 'a list * int -> int =
     function
       (h::t,k) -> g'(t,k+1)
     | ([],k) -> k
  in g' (xs,0)
```

Prove that `g` also computes the length of a list by systematically transforming your solution from part (a) into `g`. You may use the following two facts (in addition to standard facts about arithmetic). Assume an infinite range of integers, i.e., don't worry about overflow.

(Fact i) Any datatype of the form

`type nat = S of nat | Z`

is isomorphic to the natural numbers $\{0, 1, 2, \ldots\}$, where `Z` corresponds to 0, `S(Z)` corresponds to 1, `S(S(Z))` corresponds to 2, and so on. The natural numbers are, of course, a subset of the integers.

(Fact ii) Given functions

```
let rec f1 (a:int,b:int) : int =
  if a = 0 then b else f (a-1,b+1)

let rec f2 (a:int,b:int) : int = a+b
```

Then, $\forall a, b \geq 0,$ `f1(`a`,`b`) = f2(`a`,`b`)`.

Programming

Consider again the PNF language and its interpreters, as provided on the web page for this assignment. The files for environments (`env.ml`,`env.mli`), the core language (`tcore.ml`) and its type-checker (`check.ml`) are unchanged. The files for PNF (`pnf.ml`) and the core-to-PNF translator (`tcore2pnf.ml`) are almost as before The only change is that every *use* of a variable now has the variable's type attached; this makes it possible to compute the type of an arbitrary PNF expression without needing an environment to type the free variables. There is a PNF type-checker (`pnfcheck.ml`) that does this. In addition, there are three interpreters: in "direct style" (`pnfinterp.ml`) much as before, in CPS (`pnfinterpc.ml`), and using an explicit frame stack (`pnfinterps.ml`); the latter two are as described in lecture.

Most of the interesting activity in evaluting a PNF program takes place during evaluation of `Let` expressions. But some `Let` expressions are simpler than others. If the defining expression is a trivial expression, or is an expression kind that itself can only mention trivial expressions (i.e., a `Record`, `Select`, or `Variant`), then we can "unroll" the interpreter code to avoid making a recursive call (or building a new continuation or frame) when evaluating the `Let`. E.g., we could modify the orignal "direct style" version of the interpreter to look something like this:

```
let rec eval env (e:exp) =
  match e with
    Let(x,Triv trexp,b) ->
      eval (extend env (x,trivial env trexp)) b
  | Let(x,Variant(l,tre),b) ->
      eval (extend env
                (x,Vvariant(l,trivial tre))) b
  | ...
```

(If we inline the definition of `apply`, we can also unroll bindings to `App`s of primitive functions in this way.) The unrolled form makes it clear that evaluating these kinds of `Let`s requires only a small fixed amount of work, and in particular, does not require a non-tail call to `eval` (or, in the continuation- or frame-based interpreter, the construction of a new continuation or frame).

If the defining expression is an `App` of a user-defined function, or a `Switch`, `Fix`, or nested `Let`, we cannot do this unrolling. In the case of `App`, this is scarcely surprising; we would expect to need to save the "calling" context before evaluating a call to an arbtrary user-defined function. But what about the other three cases?

It turns out that we can make a further restriction on PNF programs to simply *rule out* `Let` bindings to `Switch`, `Fix`, or `Let` expressions, and then define a translation from (full) PNF to our restricted subset, which I'll call "extra-normalized" PNF. For simplicity, we also require that expressions which *may* be `Let`-bound (i.e., `Triv`, `Record`, `Select`, `Variant`, and `App`) *must* be `Let`-bound. An interpreter for extra-normlized PNF can be particularly simple, because it makes only one non-tail recursive call.

File `pnfinterpn.ml` contains such an interpreter; file `pnfnorm.ml` converts an arbitrary

PNF program to an "extra-normalized" one. (There is one extra restriction on the input to `pnfnorm`: all case arms must be explicit `Abs` expressions. This is not an important restriction in practice, and could be overcome by eta-expansion, at the expense of some complication in the code.)

A typical execution command would be:

```
Pnfinterpn.evaluate (Pnfnorm.norm(Tcore2pnf.translate d))
```

where `d` is a core expression defined in file `examples.ml`. As a sanity check on the translation and normalization processes, one could type:

```
Pnfcheck.check (Pnfnorm.norm(Tcore2pnf.translate d))
```

So much for preliminaries; here are the problems.

2. [Easy] By loading and executing the various pieces, show what `Pnfnorm.norm` does to the (translations of the) example expressions `ex1`,`ex2` and `ex3` defined in `example.ml`. Generalize from these to explain the effect of extra-normalization on arbitrary programs and argue (informally) why it doesn't change the meaning of a program.

3. [Harder] `Pnfnorm.ml` has one major drawback: it can cause an arbitrarily large piece of code to be duplicated because the continuation argument `k` is used in *each* arm of a `Let`-bound `Switch` expression. This is illustrated by `ex3`, for example, where the code to construct the record {`a=`,`b=`} is repeated for each arm of the `if`.

Modify `Pnfnorm.ml` to stop this code duplication from occurring, by introducing a new *function* to which the result of the `Switch` arm will be passed. For example (using informal syntax) `ex3` should be extra-normalized to

```
\ z -> let q = \ x -> {a=x,b=3}
       in if z then q 1 else q 2
```

4. [Not *too* hard] As noted, the `eval` function in `pnfinterpn.ml` makes only one non-tail recursive call. By using explicit frame stacks (in the style of `pnfinterps.ml`), even this one call can be made tail-recursive. Once all calls are tail-recursive, `eval` can easily be rewritten as an imperative loop, with no recursive calls at all. Write such a version of `pnfinterpn.ml`. You'll want to maintain the current environment, current stack, and current expression as CAML `ref` variables. The stack changes only when evaluating a `Let(_,App,_)` binding or a `Triv` expression.

Note: you may find it easier to develop this imperative version by first systematically converting `pnfinterpn.ml` to CPS (in the style of `pnfinterpc.ml`) and then to using explicit frame stacks (in the style of `pnfinterps.ml`). But just hand in the imperative version.