Fachbereich Mathematik und Informatik
Prof. Dr. K. Ostermann

Sebastian Erdweg, seba@informatik
Tillmann Rendel, rendel@informatik

Philipps Universität Marburg

October 29, 2009

Programming Languages and Types
Homework Assignment 3

Please hand in your homework by email to `mailto:pllecture@informatik.uni-marburg.de` until November 5.

H3.1 Strictness points

In our lazy interpreter, there are two points where we need to force evaluation of expression closures (by invoking `strict`): the function position of an application and arithmetic primitives. Now consider a version of the `FAE/L` interpreter where all instances of `strict` are removed and this line

```
[id (v) (lookup v env)]
```

is replaced with:

```
[id (v) (strict (lookup v env))]
```

The idea is that the only time the interpreter returns an expression closure is on looking up an identifier in the environment. If we force its evaluation right away, we can be sure no other part of the interpreter will get an expression closure, so removing those other invocations of strict will do no harm.
Is it possible to write a program that will produce a different results under the original interpreter and modified interpreters? Let the interpreted language feature arithmetic, first-class functions, with, if0, and rec (even though not all of these are in our in-class lazy interpreter – add them to your interpreter if you need them in your example and want to test).
If so, hand in an example program and the result under each interpreter, and clearly identify which interpreter will produce each result. If it's not possible, defend why one cannot exist.

please turn the page!

H3.2 An alternative representation of programs

Consider this alternative representation of programs.

```
(define-type FAE-HOAS
  [num (n number?)]
  [add (lhs FAE-HOAS?) (rhs FAE-HOAS?)]
  [fun (f procedure?)]
  [app (fun-expr FAE-HOAS?) (arg-expr FAE-HOAS?)])
```

In this representation, instead of writing this for the program `((lambda (x) (+ 3 x) 7)`

```
(define test1
  (app
   (fun 'x (add (num 3) (id 'x)))
   (num 7)))
```

you write this:

```
(define test1
  (app
   (fun (lambda (x) (add (num 3) x)))
   (num 7)))
```

Instead of dealing with closures, identifiers, environments, substitution and this stuff all the time, we can define the following interpreter for `FAE-HOAS`:

```
(define (interp expr)
   (type-case FAE-HOAS expr
     [num (n) expr]
     [add (l r) (num (+ (num-n (interp l)) (num-n (interp r))))]
     [fun (f) expr]
     [app (the-fun the-arg)
          (interp ((fun-f (interp the-fun)) (interp the-arg)))]))
```

This interpreter is deceptively simple. Try it! Discuss how the interpreter works and how it relates to the `FAE` interpreter. Is it equivalent to the `FAE` interpreter? What happened to closures, identifiers and environments? Discuss the merits of this interpreter with regard to the taxonomy presented in Sec. 11.4 of the textbook.