

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
Department of Electrical Engineering and Computer Science

## Problem Set 8

### Due December 4

#### Problem 1 Type Reconstruction

In this problem, you will extend the type system of FLARE to support the `recordrec` and `with-fields` forms. (FLARE is presented in section 13.5.1 of the course text, and `recordrec` and `with-fields` are discussed in section 7.2.3.)

The expression and type grammars of FLARE are extended as follows:

$$\begin{aligned} E & ::= (\text{recordrec } (I \ E)^*) \mid (\text{with-fields } (I^*) \ E_r \ E_b) \mid \dots \\ T & ::= (\text{recordof } (I \ T)^*) \mid \dots \end{aligned}$$

The  $(I^*)$  that appears in the `with-fields` expression is the list of all of the identifiers which are defined by the record value of  $E_r$ .

There are many examples of the use of `recordof` and `with-fields` throughout the course notes. We will just give one example here to illustrate how we would like you to type them. The expression

```
(let ((r (recordrec (id (lambda (x) x))))
      (with-fields (id) r (if (id #f) (id 3) (id 4))))
```

should be well-typed, and have type `int`. Note in particular the polymorphic use of `id` in the body of the `with` expression.

- Extend the typing rules of FLARE (given in Figure 13.21 of the course text) to handle `recordrec` and `with-fields`.
- Suppose the syntax of `with-fields` were changed to be

$$(\text{with-fields } E_r \ E_b).$$

Briefly explain how this would prevent you from providing a working type reconstruction algorithm. You should provide a relevant example input expression.

- Extend the type reconstruction algorithm of FLARE (presented in Figures 13.23 of the course notes) to handle `recordrec` and `with-fields`.

#### Problem 2 Compilation

Do exercise 17.25 on page 981 of the course text.

### Problem 3 Pragmatics

- a. Assume we have an expression  $(\text{scor } E_1 E_2)$  that computes the short-circuit OR: first, it evaluates  $E_1$ ; if the result is true, the entire  $\text{scor}$  expression is true ( $E_2$  should NOT be evaluated in this case); otherwise,  $E_2$  is evaluated too and its value is the value of the entire  $\text{scor}$  expression. Write down the *Meta-CPS* rule for  $(\text{scor } E_1 E_2)$ .
- b. A FLARE program was run through the TORTOISE compiler to the point after the lambda-lifting stage. Below is the output. What was the original FLARE program?

```
(fil (ktop.3)
  (let* ((z.0 2)
        (abs.4 (@mprod subr.14 z.0))
        (code.9 (@mget 1 abs.4)))
    (app code.9 abs.4 42 ktop.3))
  (def subr.14
    (abs (clo.13 x.2 k.7)
      (let* ((z.0 (@mget 2 clo.13))
            (t.8 (@* x.2 z.0))
            (code.11 (@mget 1 k.7)))
        (app code.11 k.7 t.8))))))
```

### Problem 4 Memory Management

Ben Bitdiddle has been called in by the Analog Equipment Corporation to consult on a difficult memory management problem. Analog uses Balsa, a programming language that stack allocates all continuations and environments. Since Balsa does not support a garbage collector, heap storage must be explicitly managed by programmers via calls to the procedures `malloc` and `free`:

$$E ::= (\text{malloc } E) \mid (\text{free } E) \mid \dots$$

Here is the informal description of `malloc` and `free` from the Balsa ANSI Standard:

- `(malloc E)`: If the value of  $E$  is a positive integer  $n$ , returns a location for a block of storage that is  $n + 1$  words long. The first word of the returned block is a size header; the other  $n$  words are uninitialized.
- `(free E)`: If the value of  $E$  is a location, frees the storage at that location and returns an unspecified value.

Analog is having problems with a very large Balsa application (called “The Titanic” by the development staff) that eventually always either mysteriously crashes or runs out of heap space. Ben suspects that the programmers who wrote the application are not properly deallocating storage.

In order to debug Analog’s problem Ben decides to write a standard stop-and-copy garbage collector for Balsa. He modifies `malloc` and `free` to keep track of the total amount of “busy” storage — `malloc` increments a `*busy*` counter with the number of words in the object it creates and `free` decrements the `*busy*` counter by the number of words in the object it frees. In Ben’s

system `free` does not actually free any storage. Instead, when storage is exhausted, the garbage collector runs and copies live storage (storage that is reachable from a root set) from old space into new space.

- a. Let *live* be the number of words copied during a garbage collection and *busy* be the value of the `*busy*` counter at the time of the garbage collection. In each of the following situations encountered while executing the program in Ben's system *with* garbage collection, describe the implications for executing the original Balsa program *without* garbage collection:
  - i.  $live < busy$
  - ii.  $live > busy$
  - iii.  $live = busy$
- b. A *dangling reference* is a pointer to a freed block of memory that is still reachable from the root set. How could Ben modify his garbage collector to detect dangling references?
- c. Ben tries his garbage collector out on another program from Analog Equipment Corporation. The development staff at AEC have used program verification techniques to prove that this program does not have a storage leak and in fact it runs just fine without garbage collection. However, when Ben runs the program with garbage collection, the garbage collector runs out of memory. What is going on? You should assume that the program verification techniques were valid and used correctly.