

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

Problem Set 8 Solutions

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`.

Solution:

- `recordrec`

$$\frac{\forall i \in \{1, \dots, n\}. TE[I_1 : T_1, \dots, I_n : T_n] \vdash E_i : T_i}{TE \vdash (\text{recordrec } (I_1 \ E_1) \ \dots \ (I_n \ E_n)) \quad [\text{recordrec}]} : (\text{recordof } (I_1 \ T_1) \ \dots \ (I_n \ T_n))$$

The use of $TE[I_1 : T_1, \dots, I_n : T_n]$ in the antecedent means that the definitions in a `recordrec` are mutually recursive. If we had used just TE then the definitions would not be mutually recursive. A similar difference can be found in the rules for `let` and `letrec`.

- with-fields

$$\frac{TE \vdash E_r : (\text{recordof } (I_1 T_1) \dots (I_n T_n))}{TE[I_1 : (\text{genPure}T_1, TE, E_r, [E_b]), \dots, I_n : (\text{genPure}T_1, TE, E_r, [E_b])] \vdash E_b : T_b} \quad [\text{with-fields}]$$

We use

$$TE[I_1 : (\text{genPure}T_1, TE, E_r, [E_b]), \dots, I_n : (\text{genPure}T_1, TE, E_r, [E_b])]$$

for typing E_b , which allows the identifiers I_i to be used polymorphically in the body E_b . In order to generalize the type of a module binding, we require that E_r is pure and that E_b does not mutate any of the I_i 's. This is similar to the typing rule for `let` in Fig. 14.21.

- b. 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.

Solution: Consider the following expressions:

$$\begin{aligned} &(\text{abs } (r \ x) \dots (\text{with-fields } r \ x)) \\ &(\text{abs } (r1 \ r2) (\text{with-fields } r1 (\text{with } r2 \ x))) \end{aligned}$$

In the first expression it is ambiguous whether the x in the body of the `with-fields` refers to the procedure argument x or a definition of x in record r . In the second it is ambiguous whether the definition for x is supplied by record $r1$ or record $r2$. These ambiguities cannot be resolved using a unification-based reconstruction algorithm. The programmer must specify the source of the definition for each identifier.

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

Solution:

- recordrec

$$\frac{\forall_{i=1}^n . R[E_i] \quad TE[I_j : \tau_j]_{j=1}^n [I\tau]_{=} = \langle T_i, CS_i \rangle}{R[(\text{recordrec } ((I_i E_i)_{i=1}^n))] \quad TE = \langle (\text{recordof } ((I_i : \tau_i)_{j=1}^n) \quad), \uplus_{i=1}^n CS_i \uplus (\uplus_{i=1}^n \{\tau_i \doteq T_i\}) \rangle}$$

- with-fields

$$\frac{R[E_r] \quad TE = \langle (\text{recordof } ((I_i : T_i)_{i=1}^n) \quad), CS_r \rangle}{R[(\text{with-fields } ((I_i E_i)_{i=1}^n) E_r E_b)] \quad TE = \langle T_b, CS_b \uplus CS_r \uplus T_r \doteq (\text{recordof } ((I_i : T_i)_{i=1}^n) \quad) \rangle}$$

The use of $RgenPure$ makes the identifiers I_i polymorphic in the body E_b .

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 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 scor expression. Write down the *Meta-CPS* rule for $(\text{scor } E_1 E_2)$.

Solution:

$$\begin{aligned} \mathcal{MCP}S_{exp}[\text{scor } E_1 E_2] \\ = (\lambda m. (\mathcal{MCP}S_{exp}[E_1] \\ \quad (\lambda V_1. (\text{let } ((I_k (mc \rightarrow exp m))) \\ \quad \quad (\text{if } V_1 \\ \quad \quad \quad ((id \rightarrow mc I_k) \#t) \\ \quad \quad \quad (\mathcal{MCP}S_{exp}[E_2] (id \rightarrow mc I_k))))))) \end{aligned}$$

- 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))))))
```

Solution:

```
(flare ()
  (let ((z 2))
    (let ((y (abs (x) (* x z))))
      (y 42))))
```

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. $live < busy$

Solution: The program is guaranteed to have a storage leak — i.e., there is at least one block that is unreachable and has not been freed. This could explain the memory exhaustion problems. There may also be dangling references (reachable blocks that have been freed) as long as the number of words in dangling references is less than the number of words in the storage leak.

- ii. $live > busy$

Solution: The program is guaranteed to have a dangling reference — i.e., there is at least one block that is reachable but has already been freed. Accessing a dangling pointer could be responsible for the crashes. There may also be a storage leak as long as the number of words in the storage leak is less than the number of words in the dangling references.

iii. *live = busy*

Solution: The program may be in a safe state or it may not. There may be a combination of unfreed dead storage (storage leak) and freed live storage (dangling reference) such that the amount of mismanaged storage exactly balances.

- 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?

Solution: He would change `free` to change the tag in the size header to be different from the block header tag and the broken heart tag. Thus when the garbage collector begins scanning a pointer that points to a block with this unique freed tag, it can signal a dangling reference error.

- 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.

Solution: Many students answered this question by saying that the garbage collector splits memory into two semi-spaces and thus only has half the total amount of memory available at any one time. However, the garbage collected version could still run out of memory even if given twice as much memory as the non-garbage collected version.

This can happen if there is a dangling reference — a pointer to a freed block of memory. The garbage collector will copy the dangling block, while the non-garbage collected version is free to reuse it.

How can there be a dangling reference if the program verification is correct? It could be a bug in the compiler that was used to compile the verified program. The program has freed some block of memory, but the compiler has not arranged to “zero out” all pointers to the block.

This is one of the reasons one might choose to use flat environment structures for closures, so that they only hold onto values that will be needed. Linked closures typically hold onto more than just the free variables of a function. To follow this one step further, the compiler might actively remove the value from the environment after the last reference to that variable.