Preliminary Examination II
Computer Science 312, Cornell University 15 April 2003

Solutions

1. True/False [10 pts]

   (parts a–e; **4** points off for each wrong answer, 2 points off for each blank answer, minimum problem score 0.)

   (a) Imperative data abstractions have mutators.

   true

   (b) The red-black tree invariant ensures that every path from the root to the leaf has the same height.

   false

   (c) The worst-case performance of hash-table lookup is $O(n)$ where n is the number of elements, even if the number of buckets is n.

   true

   (d) The update operation (`:=`) is $O(1)$ in a language implementation that manages memory through reference counting.

   false

   (e) Breadth-first search of a graph requires time proportional to $n \lg n$ where n is the number of nodes and edges in the graph.

   false

2. Zardoz Refs [20 pts]

   For each of the following expressions, give a *value* that causes the expression to evaluate to 42 if the box $\square$ is replaced by that value.

   (a) [10 pts]

```
let
   val zardoz: 'a * 'a ref * 'a ref * 'a ref ref -> unit =
      □
   val x: int ref ref = ref(ref(1))
   val y: int ref ref = ref(ref(1))
   val z: int ref = ref(8)
in
   zardoz(6,z,!x,y);
   (!(!x))  * (!(!y) - 1)
end
```

   Answer:

```
fn(a,b,c,d) => (
  d := b;
  c := a
 )
```

(b) [10 pts]

```
let val zardoz: unit->unit->int = [ ]
    val f = zardoz()
in
  f() + f() + 1
end
```

Answer:

```
fn() => let val x = ref 19
in fn() => (x := !x + 1; !x)
end
```

3. Correctness [25 pts]

Consider the following data abstraction for Booleans and its implementation:

```
signature BOOL = sig
  (* A "boolean" is a Boolean with the usual operations. *)
  type boolean
  val true: boolean
  val false: boolean
  val and_: boolean * boolean -> boolean
  val or: boolean * boolean -> boolean
end
structure Boolean :> BOOL = struct
  type boolean = int
  val true = 1
  val false = 0
  fun and_(x,y) = x*y handle overflow => true
  fun or(x,y) = x+y handle overflow => true
end
```

(a) [3 pts] What is the abstraction function for this implementation?

Answer:

AF(0) = false, AF(any positive integer) = true

(b) [3 pts] What is the representation invariant maintained by this implementation?

Answer:

The representation is nonnegative.

(c) [4 pts] Now let's prove that the function `or` is implemented correctly. Start by stating a proposition that, if true, means `or` is implemented correctly. This proposition should be expressed using the abstraction function *AF* and the representation invariant *RI*.

Answer:

For all `boolean`*'s* x *and* y *such that* $RI(x)$ *and* $RI(y)$ *hold,*

$$((AF(x) \vee AF(y)) = AF(\texttt{or}(x, y))) \;\&\; RI(\texttt{or}(x, y))$$

(d) [15 pts] Prove the proposition you stated in part 3(c). *Hint*: consider possible cases on x and y.

Answer:

By assumption, $RI(x)$ *and* $RI(y)$ *(they are both nonnegative), so we can consider three cases: (1) both zero, (2) one zero and the other positive, and (3) both positive.*
Case 1. $x = y =$ `0`*. The function clearly evaluates to zero, and 3(c) holds:*

$$AF(x) \vee AF(y) = false = AF(0) = AF(\texttt{or}(x, y))$$

and $RI(0)$*.*
Case 2. Without loss of generality assume $y =$ `0`*. Then* $\texttt{or}(x, y)$ *evaluates to* x*, which is positive and hence represents true. This satisfies 3(c):*

$$AF(x) \vee AF(y) = true = AF(x) = AF(\texttt{or}(x, y))$$

and $RI(x)$*.*
Case 3. $\texttt{or}(x, y)$ *steps to* x `+` y `handle overflow => true`*. If there is no overflow, then* $x + y$ *must evaluate to a positive number because it is the sum of two positive numbers. In that case the result represents true. If there is overflow, the expression also evaluates to a representation of true. Therefore the result always represents true and is always positive:*

$$AF(x) \vee AF(y) = true = AF(\texttt{or}(x, y))$$

and $RI(\texttt{or}(x, y))$ *holds because* $\texttt{or}(x, y) > 0$*.*

4. Complexity [20 pts]

Let $T(n)$ be the time to perform a merge sort of n elements. The recurrence is:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n + 1$$

Let's `prove` that $T(n) = O(n)$ for all n by induction on n.

Base case: $T(1) = 1 = O(1)$
Assume that for any m, $1 \leq m < n$, $T(m) = O(m)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n + 1 \\ &= 2O(n/2) + n + 1 \qquad \text{by IH} \\ &= O(2(n/2) + n + 1) \\ &= O(2n + 1) \\ &= O(n). \end{aligned}$$

"QED".

(a) [1 pt] What is the correct asymptotic complexity of merge sort?

Answer:

$O(n \lg n)$

(b) [10 pts] What's wrong with this proof? Explain briefly how and where the reasoning is incorrect.

Answer:

The notation $f(n)$ is $O(g(n))$ means that the ratio $|f(n)/g(n)|$ is bounded by some constant for all *natural numbers n. It is essential that both f and g are functions depending on some argument n.*
The induction hypothesis ($T(n)$ is $O(n)$) is stated for fixed *n; therefore, it is meaningless - there is no dependency on any argument.*
The induction hypothesis for a fixed n can be rewritten as $const_1$ is $O(const_2)$. We could imagine that both constants depend on some argument n'. Then the statement $const_1(n')$ is $O(const_2(n'))$ is trivially true, but it doesn't say anything about the property that we are trying to prove – $T(n)$ is $O(n)$ – because that property is really a statement about functions, not a statement about the values of those functions at any given n.

Consider the following recurrence:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n^2$$

Use the substitution method to determine whether the following statements are true or false. Show your work.

(c) [3 pts] $T(n)$ is $O(n)$

Answer:

Substitute kn for $T(n)$. LHS: kn. RHS: $kn+n^2$. Since RHS is asymptotically smaller than LHS for any k, this statement is false.

(d) [3 pts] $T(n)$ is $O(n \lg n)$

Answer:

Substitute $kn \lg n$ for $T(n)$. LHS: $kn \lg n$. RHS: $2k(n/2)\lg(n/2)+n^2 = kn \lg n - kn + n^2$. Again, the $O(n^2)$ term on the RHS dominates and the statement is false.

(e) [3 pts] $T(n)$ is $O(n^2)$

Answer:

Substitute kn^2 for $T(n)$. LHS: kn^2. RHS: $2k(n^2/4)+n^2 = (k/2+1)n^2$. If $k > 2$ then the LHS is asymptotically larger than the RHS, hence $T(n)$ is $O(n^2)$.

5. Type Checking [25 pts]

The following two functions transform a two-argument function between its curried and uncurried forms:

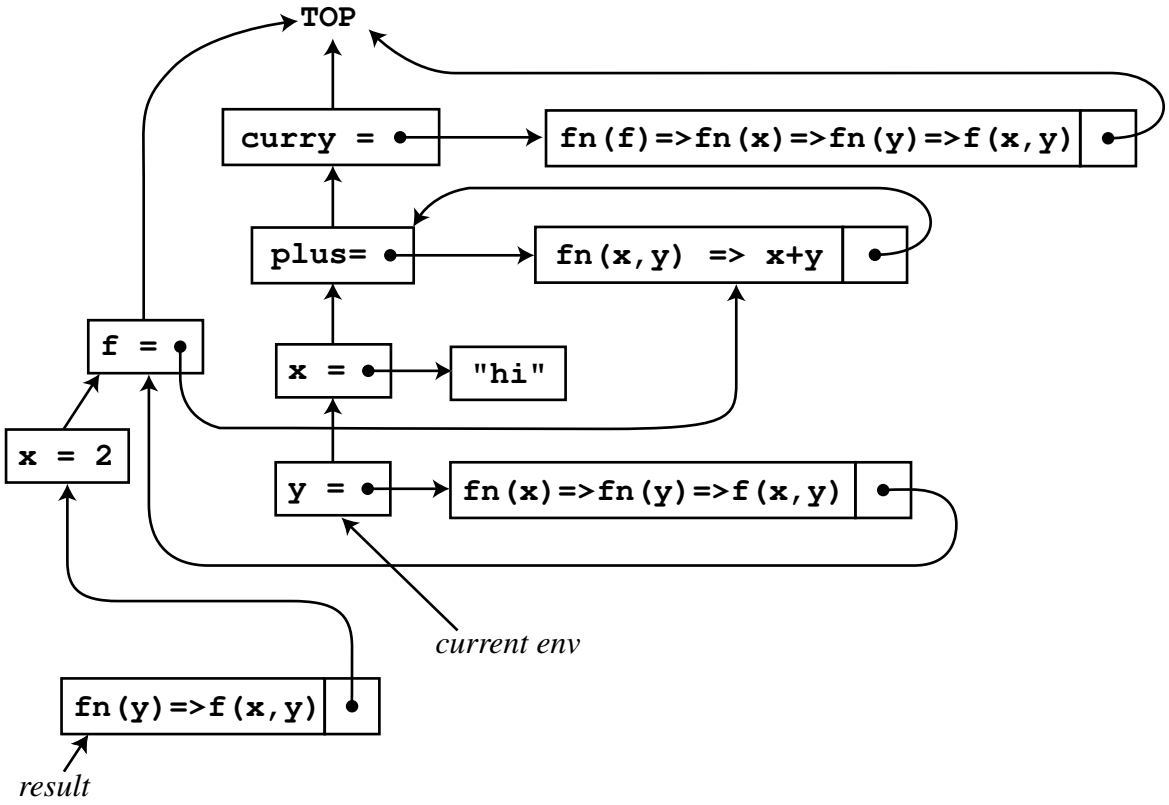
```
val curry: ('a*'b->'c) -> ('a->'b->'c) =
    fn (f: 'a*'b->'c) => fn(x:'a) => fn(y:'b) => f(x,y)
val uncurry: ('a->'b->'c) -> ('a*'b->'c) =
    fn (f: 'a->'b->'c) => fn(x:'a, y':b) => f x y
```

Consider the following code that uses `curry`:

```
let fun plus(x:int,y:int):int = x+y
    val x:string = "hi"
    val y: int->int->int = curry plus
in
    y 2
end
```

(a) [10 pts] Show the contents of the environment (including both `x`'s) and the heap at the end evaluating the `let` block (but while `x` and `y` are still in scope.) Part of the diagram is drawn below; complete it. (Note that `curry` is assumed to be already present in the environment as shown.) You may use the back side as scratch paper or redraw the whole figure there if you need more room.

Answer:



A type can be interpreted as a logical proposition, where the product type operator `*` corresponds to Boolean "and" ($\wedge$), the datatype separator `|` corresponds to Boolean "or" ($\vee$), and the function type operator `->` corresponds to Boolean implication ($\Rightarrow$). For example, the type `'a->('b->'c)` corresponds to a proposition $A \Rightarrow (B \Rightarrow C)$. A type like `int` corresponds to the proposition "some integer exists".

Remarkably, a proposition holds[1] only when we can find a term whose type corresponds to that of the proposition. For example, the proposition $A \Rightarrow A$ holds for all propositions A; the term `fn(x:'a)=>x` is a proof of this claim! Conversely, if a proposition is false, then we can find no term that has the corresponding type (if we aren't allowed to use certain SML features: for example, refs or any recursion leading to nontermination). Thus, there is no term of the type `'a->'b` because the proposition $A \Rightarrow B$ is not true for all A and B.

In logic, we can show that two propositions X and Y are equivalent (written $X \equiv Y$) by showing both $X \Rightarrow Y$ and $Y \Rightarrow X$. Because propositions are types, the equivalence of two propositions has a computational significance: it means that there must exist a pair of functions that map back and forth between the types that correspond to the two propositions. For example, the `curry` and `uncurry` functions prove the logical equivalence $A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C$.

(b) [8 pts] In logic, $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$, because "and" distributes over "or". To represent "or", we use this datatype:

```
datatype ('x, 'y) sum = Left of 'x | Right of 'y
```

Prove this logical equivalence by implementing these declarations with functions that always terminate:

```
val forward: 'a*('b,'c) sum -> ('a*'b, 'a*'c) sum
val backward: ('a*'b, 'a*'c) sum -> 'a*('b,'c) sum
```

Write down all type declarations explicitly.

Answer:

```
fun forward(a:'a, s:('b,'c) sum): ('a*'b, 'a*'c) sum =
  case s of
    Left(b:'b) => Left(a,b)
  | Right(c:'c) => Right(a,c)
fun backward(s: ('a*'b, 'a*'c) sum): 'a*('b,'c) sum =
  case s of
      Left(a:'a,b:'b) => (a, Left(b))
    | Right(a:'a,c:'c) => (a, Right(c))
```

(c) [7 pts] Similarly, show $A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C)$ by defining two terminating functions that map between the types `'a->'b*'c` and `('a->'b)*('a->'c)`.

Answer:

```
fun forward(f: 'a->'b*'c): ('a->'b)*('a->'c) =
  (fn(a:'a) => #1(f a), fn(a:'a) => #2(f a))
fun backward(p: 'a->'b * 'a->'c): 'a->'b*'c =
  fn(a:'a) => ((#1 p) a, (#2 p) a)
```

[1] in "constructive logic", which isn't as powerful as the classical logic covered in CS 280.