15-150 Spring 2012
Lab 6

February 22, 2012

1 Introduction

The goal for this lab is to make you more familiar with continuations, and proofs about continuations.Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.

2 Proofs about Continuations

Consider the following two functions:

```
fun sum (t : int tree) : int =
    case t of
        Empty => 0
      | Leaf x => x
      | Node(t1,t2) => sum t1 + sum t2

fun sumc (t : int tree) (k : int -> int) : int =
    case t of
        Empty => k 0
      | Leaf x => k x
      | Node (l,r) => sumc l (fn a => sumc r (fn b => k (a + b)))
```

We would like to prove that the continuation-based `sumc` behaves the same as `sum`, when you apply `sumc` to the identity continuation:

Theorem 1. *For all* `t : int tree`*,* `sumc t (fn x => x)` $\cong$ `sum t`*.*

We can try to prove this theorem by induction on `t`.

Task 2.1 Start the case for `Node(l,r)`. Explain why the proof breaks down.

To fix this, we can generalize the theorem to consider an arbitrary continuation `k`:

Theorem 2. *For all values `t : int tree`, `k : int -> int`, `sumc t k` $\cong$ `k(sum t)`*

This says that `sumc` computes the same sum as `sum`, and then passes this sum to `k`.

To combat the problem you encountered above, it is necessary to quantify over `k` *in the predicate that is proved by induction*, so that the inductive hypotheses are general enough.

Task 2.2 Prove "for all `t`, P(`t`)" by induction on `t`, where P is defined by

$$P(\texttt{x}) = \text{for all } \texttt{k}, \texttt{sumc x k} \cong \texttt{k(sum x)}$$

You may assume that `sum` is total.

Case for `Empty`:

To show:

Proof: Assume a continuation `k`.

Case for `Leaf x`:

To show:

Proof:

Case for `Node(l,r)`:
IH 1:

IH 2:

To show:

Proof:

When you use an IH, carefully note which continuation the “for all” is instantiated with.

Have the TAs check your proof before continuing!

$\boxed{\textbf{Solution 2.2}}$

Theorem 3. *For all values* `t : int tree`, `k : int -> int`, `sumc t k` $\cong$ `k(sum t)`

Case for `Empty`:
To show: for all `k`, `sumc Empty k` $\cong$ `k (sum Empty)`

Proof: Assume a continuation `k`

```
 sumc Empty k
 ≅ case Empty of Empty ⇒ k 0 | ...       (Step)
 ≅ k 0                                   (Step)
 ≅ k (case Empty of Empty ⇒ 0 | ...)     (Reverse Step into sum)
 ≅ k (sum Empty)                         (Reverse Step)
```

Case for `Leaf x`:
To show: For all `k`, `sumc (Leaf x) k` $\cong$ `k (sum Leaf x)`

Proof: Assume a continuation `k`

```
 sumc (Leaf x) k
 ≅ case (Leaf x) of ... | Leaf x ⇒ k x) | ...        (Step)
 ≅ k x                                               (Step)
 ≅ k (case (Leaf x) of ...  | Leaf x ⇒ x | ...)      (Reverse Step into sum)
 ≅ k (sum Leaf x)                                    (Reverse Step)
```

Case for `Node(t1,t2)`:

IH 1: For all `k1`, `sumc l k1` $\cong$ `k1 (sum l)`

IH 2: For all `k2`, `sumc r k2` $\cong$ `k2 (sum r)`

To show: For all `k`, `sumc Node(l,r) k` $\cong$ `k (sum Node(l,r))`

Proof: Assume a continuation `k`

```
sumc Node(l,r) k
≅ case Node(l,r) of ...                              (Step)
      | Node(l,r) ⇒ sumc l (fn a ⇒ ...)
≅ sumc l (fn a ⇒ sumc r (fn b ⇒ k (a + b)))         (Step)
≅ (fn b ⇒ k (a + b))) (sum l)                        (IH1 taking k1 as
                                                     fn a ⇒ sumc r (fn b ⇒ k (a + b))
≅ sumc r (fn b ⇒ k (sum l + b))                      (Step. sum total so sum l valuable)
≅ (fn b ⇒ k (sum l + b)) (sum r)                     (IH2 taking k2 as
                                                     fn b ⇒ k (sum l + b)
≅ k (sum l + sum r)                                  (Step. sum total so sum r valuable)
≅ k(case Node(l,r) of ...                            (Reverse Step into sum)
       | Node(l,r) ⇒ sum l + sum r)
≅ k(sum Node(l,r))                                   (Reverse Step)
```

Q.E.D.

3 Programming with Continuations

3.1 Answer types

`sumc` can in fact be given a more general type than above. Starting from

```
fun sumc (t : int tree) (k : ?) : ? = <as above>
```

Task 3.1 Infer the most general type for `sumc`. Explain why this makes sense.

Task 3.2 Your answer should say that `sumc` is polymorphic, with one type variable. Give two example `k`s, which require instantiating the type variable to two different types.

$\boxed{\textbf{Solution 3.2}}$ SML performs typechecking statically by just looking at the code, so we should be able to as well. Let's start with a copy of `sumc` where we replace the annotations at the top of the function with letters standing for any type. We also have added line numbers for reference.

```
(1) fun sumc (t : α) (k : β) : γ =
(2)    case t of
(3)      Empty => k 0
(4)    | Leaf x => k x
(5)    | Node (t1, t2) => sumc t1 (fn a => sumc t2 (fn b => k (a + b)))
```

Going line-by-line:

Line (1) `sumc` is a function with type α `->` β `->` γ, for some types α, β, γ.

Line (2) Since we're casing on `t` and the case statement has a branch for `Empty` on (3), then α must be δ `tree` for some type δ. We don't know anything about δ because we haven't yet used anything from the tree.

Line (3) The case for `Empty` is consistent with the assumption that `t` has type δ `tree` but still tells us nothing about δ. We see that `k` is applied to some arguments, so we know that β must be ϵ `->` ζ for some types ϵ and ζ. More specifically, we see that `k` is applied to the value `0` of type `int` so ϵ must be `int`.

Taking stock, we now know that

> `sumc` has type δ `tree -> (int ->` ζ`) ->` γ

for some types δ, ζ and γ.

Line (4) The case for `Leaf x` is consistent with the assumption that `t` has type δ `tree` but still tells us nothing about δ. Inside the branch for `Leaf`, `x` has type δ and we see that `k` is applied to `x`. We already figured out that the argument type of `k` must be `int` so that means that δ is `int`.

Taking stock, we now know that

> `sumc` has type `int tree -> (int ->` ζ`) ->` γ

for some types ζ and γ.

Line (5) The case for `Node(t1,t2)` is consistent with the assumption that `t` has type `int tree`. That means that we get the assumptions that `t1` and `t2` both have type `int tree` to the right of the `=>`.

The expression we're now considering is

```
sumc t1 (fn a => sumc t2 (fn b => k (a + b)))
```

Let's work from the inside out.

We know that `a` and `b` both have type `int` because they're bound by `fn` in the continuation arguments to `sumc`—and we know that the continuation has type `int ->` ζ. Therefore, `k (a+b)` has type ζ and `fn b => k(a+b)` has type `int ->` ζ.

Since `t2` has type `int tree`, then the whole expression

```
sumc t2 (fn b => k (a + b)))
```

has type γ and the expression

```
fn a => sumc t2 (fn b => k (a + b)))
```

has type `int ->` γ. Since we're using this as the third argument to a call to `sumc`, we know that it must have type `int ->` ζ, so therefore $\gamma = \zeta$. Since `t1` also has type `int tree` the whole expression type checks.

That gives us that

> `sumc` has type `int tree -> (int ->` ζ`) ->` ζ

Since we've run out of code, we have no more information about the types to add, and we're done.

If you delete the type annotations on the code in the file, SML will run roughly the same sort of analysis as above, and come back with

```
sumc :  int tree -> (int -> 'a) -> 'a
```

It tries to pick sane names for type variables in polymorphic functions, so even though we ran across 6, the final result will be with α.

The intuitive reason that this function is polymorphic is that we never actually use the result of the continuation as we recurr down the tree. We take old continuations and bury them inside new ones, but we don't do anything like

```
case k (a+b)
  of ...
```

that would constrain the return type. This corresponds to what we saw during an evaluation of `sumc` on a particular initial continuation: since it gets evaluated last, all we know is that it is applied to the sum of the tree. If we want to know the sum of the tree, then `fn x => x` is the right choice, but `Int.toString` which has type `int -> string` or

```
fn x => List.tabulate(x, (fn _ => "hello!"))
```

which has type `int -> string list` would also type check. (The latter would raise an exception if the sum of the tree happens to be negative, though.)

3.2 Find

Task 3.3 Write a function

```
fun find (p : 'a -> bool) (t : 'a tree) : 'a option = ...
```

such that

- if there is some `x` in `t` for which `p x == true` then `find p t == SOME x`. If there is more than one `x` that satisfies `p`, return the left-most one in the tree.
- `find p t == NONE` if there is no such `x`.

Task 3.4 Write a function

```
fun find_cont (p : 'a -> bool) (t : 'a tree) (k : 'a option -> 'b) : 'b = ...
```

such that (1) `find_cont p t k` $\cong$ `k (find p t)` and (2) `find_cont` uses constant stack space.

Have the TAs check your code before leaving!