CS 5860
Thur.
Nov 3, 2011
A Note on Recursion and Introduction to Event Logic

PLAN

1. The fix constructor (also called Y-combinator) and recursive function definition
   - the 3x+1 function as an example
   - recursion as $F = \lambda(x. F(x,F))$
   - recursion as $\text{fix}(\lambda(f. \lambda(x. F(x,f))))$

2. In the final week we will mention type theory and discuss partial types (bar types) as an approach to partial correctness, a key topic as we saw last time.
   (A good research topic is to use partial types to reexpress Manna's theory.)

3. Review Peterson's Mutual Exclusion Algorithm

4. Message passing version of Peterson, by Wu and Rahli.

5. A Logical Model of Asynchronous Distributed Computing Events, causal-order, event orderings, etc.
Computability in All Types

Here is how Computational Type Theory (CTT) defines recursive functions. Consider the \( 3x+1 \) function with natural number inputs.

\[
f(x) = \begin{cases} 
1 & \text{if } x=0 \\
\text{if even}(x) \text{ then } f(x/2) & \text{else if even}(x) \text{ then } f(x/2) \\
& \text{else } f(3x+1) \\
& \text{fi} \\
& \text{fi}
\end{cases}
\]

Alternative Syntax

\[
f = \text{function}(x. \text{ if } x=0 \text{ then } 1 \\
& \text{else if even}(x) \text{ then } f(x/2) \\
& \text{else } f(3x+1))
\]

Using Lambda Notation

\[
f = \lambda(x. \begin{cases} 
1 & \text{if } x=0 \\
\text{if even}(x) \text{ then } f(x/2) & \text{else if even}(x) \text{ then } f(x/2) \\
& \text{else } f(3x+1))
\end{cases}
\]

Here is a related term with function input \( f \)

\[
\lambda(f. \lambda(x. \begin{cases} 
1 & \text{if } x=0 \\
\text{if even}(x) \text{ then } f(x/2) & \text{else if even}(x) \text{ then } f(x/2) \\
& \text{else } f(3x+1))
\end{cases})
\]

The recursive function is computed using this term.

Defining Recursive Functions in CTT

\[
\text{fix}(\lambda(f. \lambda(x. \begin{cases} 
1 & \text{if } x=0 \\
\text{if even}(x) \text{ then } f(x/2) & \text{else if even}(x) \text{ then } f(x/2) \\
& \text{else } f(3x+1))
\end{cases}))
\]

Recursion in General

\( f(x) = F(f,x) \) is a recursive definition, also \( f = \lambda(x.F(f,x)) \) is another expression of it, and the CTT definition is:

\[
\text{fix}(\lambda(f. \lambda(x. F(f,x))))
\]

which reduces in one step to:

\[
\lambda(x.F(\text{fix}(\lambda(f. \lambda(x. F(f,x)))),x))
\]

by substituting the fix term for \( f \) in \( \lambda(x.F(f,x)) \).

Non-terminating Computations

CTT defines all general recursive functions, hence non-terminating ones such as this

\[
\text{fix}(\lambda(x.x))
\]

which in one reduction step reduces to itself.

This system of computation is a simple functional programming language. In CTT it is essentially the programming language also used in the metatheory, ML. Later we add non-functional features as well.
Recall the shared memory computing model

A critical section (cs) is an execution of a process \( P_i \) during which it has exclusive access to (a portion of) the state, e.g. \( P_i \) needs to write to memory without interference from other processes.

We saw that two processes that share the variables \( Q_1, Q_2 \) and \( \text{Turn} \) can provide mutually exclusive access to the critical portion of the state. Here is the code.

\[
P_1
\]
\[
Q_1, Q_2 : \text{Bool}
\]
\[
\text{Turn} : \{0, 1, 3\}
\]
\[
\text{to enter CS 3}
\]
\[
Q_1 := \text{true}; \text{Turn} := 1
\]
\[
\text{Wait until } (\neg Q_2 \lor \text{Turn} = 2)
\]
\[
\text{enter CS}
\]
\[
\text{exit CS}
\]
\[
Q_1 := \text{false}
\]
\[
\text{end}
\]

\[
P_2
\]
\[
Q_1, Q_2 : \text{Bool}
\]
\[
\text{Turn} : \{0, 1, 3\}
\]
\[
\text{to enter CS 3}
\]
\[
Q_2 := \text{true}; \text{Turn} := 2
\]
\[
\text{Wait until } (\neg Q_1 \lor \text{Turn} = 1)
\]
\[
\text{enter CS}
\]
\[
\text{exit CS}
\]
\[
Q_2 := \text{false}
\]
\[
\text{end}
\]
We can fairly easily see these properties of the algorithm.

Correctness: only one process is in the critical section "at a time," if \( P_i \) is in its CS then \( P_j \) for \( j \neq i \) is not.

Liveness: if a process requests entry to CS, it will eventually get access.

Now we want to consider a message passing distributed version of the algorithm where the processes \( P_1, P_2 \) want to have mutually exclusive access to a resource at a separate location from those of \( P_1, P_2 \). The picture is

\[
\begin{array}{c}
\text{P1} \\
\downarrow \\
\text{R} \\
\uparrow \\
\text{P2}
\end{array}
\]

The arrows are communication channels, and communication is asynchronous. There is no global clock and no fixed time for a communication to complete. The processes can run at different speeds. We do assume that the communication channels are reliable, sometimes that they are FIFO, i.e. messages arrive on a channel in the order sent. (We can relax these assumptions by adding a process on the channel that drops, duplicates, and reorders messages if we want those assumptions.)
Jason Wu and Vincent Rahli developed a distributed version of Peterson's algorithm based on token passing. Their algorithm is essentially the pseudo-code below. They have implemented it in Event ML.

![Diagram](image)

One process starts with a token, and we can think of the value of a Boolean variable token? indicating whether the process has it. Each process executes this code to enter CS and to respond to requests for the token.

```plaintext
enterCS
    if token? then enterCS (busy := true; CS; busy := false)
    else (request_token; await token? then enterCS)

token request received
    if busy then (await !busy then send_token)
    else send_token
```

Consider

**Correctness**

**Liveness**

See notes **Peterson's Algorithm in a Distributed Environment** by Wu and Rahli.