CS611 Lecture 14 Imperative features, continuation-passing style October 1, 2001

Scribe: Christopher Capobianco and Yan Zhang Lecturer: Andrew Myers

## 1 $uF$ extension to C

### 1.1 Review of $uF$

First let us take a look back at the $uF$ language:

$$\begin{array}{rcl} e & ::= & b \mid x \mid e_1 \oplus e_2 \mid \textbf{if}\, e_0\, \textbf{then}\, e_1\, \textbf{else}\, e_2 \mid \langle e_1, e_2 \rangle \\ & \mid & \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{let}\ \ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{rec}\ \ y\ (\lambda x\ e) \\ b & ::= & n \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \mid s \\ v & ::= & b \mid \lambda x\ e \mid \langle v_1, v_2 \rangle \end{array}$$

### 1.2 Definition of C

Now we will extend $uF$ to allow access to both the values stored in variables, and the location at which they are stored

$$e \ ::= \ \ldots \mid e_1 = e_2 \mid *\,e \mid e_1; e_2 \mid \&e$$

We call this extension of the $uF$ language $C$ for obvious reasons, as it models the behavior of the C programming language with respect to pointers and variables.

### 1.3 $lvalues$ and $rvalues$

In actual C, an $lvalue$ is an expression that can appear on the left hand side of an assignment operation, and likewise, an $rvalue$ is an expression that can (only) appear on the right hand side. An $lvalue$ is an expression that describes the location of an variable used in the program. Furthermore, an variable's $rvalue$ is the value stored at the location described by the $lvalue$.

These concepts will be important when we proceed to the translation of $C$ into $uF_!$

## 2 Translation of C into $uF_!$

### 2.1 Review of $uF_!$

The $uF_!$ language, like C, was an extension of $uF$. Here is a quick refresher of $uF_!$ syntax:

$$e \ ::= \ \ldots \mid \mathsf{ref}\ e \mid !e \mid e_1; e_2 \mid e_1 := e_2$$

### 2.2 Translation of $C$ into of $uF_!$

We now will translate $C$ into $uF$!. To do this we define two translation functions $\mathcal{R}$ and $\mathcal{L}$. The translation $\mathcal{R}[\![e]\!]$ takes the $rvalue$ of an expression in our $C$ language and translates into an equivalent expression in $uF_!$. Similarly, $\mathcal{L}[\![e]\!]$ takes the $lvalue$ of an expression $e$ in $C$ and gives an expression in $uF_!$. It is important to notice that not all expressions have $lvalues$- e.g. it does not make sense to find the location of the expression $2+2$; i.e. $\&(2+2)$ is an illegal expression. Likewise, we will not define $\mathcal{L}[\![e_1 \oplus e_2]\!]$ assuming that well-formed programs do not contain expressions like $\&(2+2)$ or $(2+2) = 5$.

$$\begin{aligned}
\mathcal{R}[\![x]\!] &= \,!x \\
\mathcal{L}[\![x]\!] &= x \\
\mathcal{R}[\![*e]\!] &= \,!\mathcal{R}[\![e]\!] \\
\mathcal{L}[\![*e]\!] &= \mathcal{R}[\![e]\!] \\
\mathcal{R}[\![\&e]\!] &= \mathcal{L}[\![e]\!] \\
\mathcal{R}[\![\lambda x.\, e]\!] &= \lambda x' \;\mathsf{let}\; x = \mathsf{ref}\; x' \;\mathsf{in}\; \mathcal{R}[\![e]\!] \\
\mathcal{R}[\![e_0\ e_1]\!] &= \mathcal{R}[\![e_0]\!]\mathcal{R}[\![e_1]\!]
\end{aligned}$$

Some program languages allow you to pass in a reference to a function; this allows that function to change the value stored in the variable passed in. This technique is called call-by-refernce. Here is a program in Pascal that uses the call-by-reference method to write an increment function (similar to the C programming language operator ++). The value of $y$ at the end of this fragment of a program is 3, since the value y refers to is incremented by the call to inc.

```
function inc(var x::int):int
  begin
     inc := x, x := x + 1
  end

. . .

y:int = 2;
inc(y);
```

If we want to be able to use the call-by-reference feature we must extend the $C$ defintion:

$$e \;::=\; \ldots \;|\; \mathsf{rfn}\; x\; e \;|\; \mathsf{rapply}\; e_0\; e_1$$

rfn takes the place of the $\lambda$ for functions that take in a reference rather a value. rapply is used to evaluate call-by-reference functions. There translations are as below:

$$\begin{aligned}
\mathcal{R}[\![\mathsf{rfn}\; x\; e]\!] &= \lambda x\; \mathcal{R}[\![e]\!] \\
\mathcal{R}[\![\mathsf{rapply}\; e_0\; e_1]\!] &= \mathcal{R}[\![e_0]\!]\mathcal{L}[\![e_1]\!]
\end{aligned}$$

## 3 Continuations

### 3.1 Translations as they were

So far we have done a number of translations: $C$ to $uF_!$, $uF_!$ to $uF$, $uF$ to $CBV$ $\lambda$ calculus, etc. All of these translations were done using a direct semantic translation. This style of translation translates the control structure in the source language into a corresponding form in the destination language:

$$\begin{aligned}
\mathcal{T}[\![\lambda x.\, e]\!] &\mapsto \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_0\ e_1]\!] &\mapsto \mathcal{T}[\![e_0]\!]\mathcal{T}[\![e_1]\!]
\end{aligned}$$

This type of translation is straightforward and works well when the source language looks a lot like the target: a $\lambda$-term in the source language maps to some $\lambda$-term of the destination language, etc. However, for translating a variety of language features, it is actually rather limiting.

### 3.2 Thinking About Continuations

Consider the following code:

$$\textbf{if}\, x \;\leq\; 0\, \textbf{then}\, x = 1\, \textbf{else}\, x = -1$$

We have described the evaluation of this by using an *evaluation context*. We have a hole in the program where we could put a *redex* and evaluate in that context:

$$\textbf{if}\, [\;]\, \textbf{then}\, x = 1\, \textbf{else}\, x = -1$$

We can instead think of the context as a function that takes in the *redex* to evaluate as an argument:

$$(\lambda y\; \textbf{if}\, y\, \textbf{then}\, x = 1\, \textbf{else}\, x = -1)\;(x \;\leq\; 0)$$

Using this idea we can make explicit the control stuctures, and extend it to the whole program. We model all transfers of control as functions where the function accpets values but does not return. The control of the program does not return to the caller of the function, but instead control proceeds in a continuing fashion.

## 4 CPS $\lambda$-calculus

Continuation-passing style is a particular way of writing lambda calculus terms. We can define CPS terms subset as lambda calculus terms with to:

$$\begin{array}{rcl} e & ::= & x \;\mid\; \lambda\; x\; s \;\mid\; \mathsf{halt} \\ s & ::= & e_1\; e_2 \end{array}$$

Here the *lambda* expression is a continuation; it waits for the value of $x$ to continue the rest of the programs. And we also add a special continuation halt, which halts the program and return the value of argument it takes.

For example, we have the following programs:

$$\begin{array}{c} s \\ \mathsf{halt}\; e \\ \mathsf{halt}\; (\lambda\; x\; s) \end{array}$$

### 4.1 Operational Semantics

For the small step semantics, we have:

$$(\lambda\; x\; s)e \mapsto\; s\{e/x\}$$

where the evaluation does one step every time. There is no rule for evaluation in a context because there are no evaluation contexts, unlike in ordinary $\lambda$ calculus.

For the large step semantics, we have: $\dfrac{s\{e/x\} \Downarrow (\mathsf{halt}\; v)}{(\lambda\; x\; s)\; e \Downarrow (\mathsf{halt}\; v)}$

Notice the proof tree of a CPS program is simply stack. This implies that if we implement an interpreter, there is no need to walk back down the proof tree– precisely because there are no evaluation contexts to resume executing.

### 4.2 Two arguments funtion of CPS

In lambda calculus, we have

$$\lambda\ (x\ y)\ e \Rightarrow \lambda\ x\ (\lambda\ y\ e)$$

But this does not hold in CPS anymore. This problem arises because in CPS, the body of the function of a lambda term is a statement, not an expression; it doesn't return a value and neither does the function itself.

So:

$$\lambda\ (x\ y)\ s \neq \lambda\ x\ (\lambda\ y\ s)$$

In order to sove this problem, we add 2-argument functions to CPS. Change the CPS grammar as:

$$\begin{array}{rcl} e & ::= & x \ \mid\ \lambda\ x\ s \ \mid\ \lambda(x\ y)\ s \\ s & ::= & e_0\ e_1 \ \mid\ e_0\ e_1\ e_2 \end{array}$$

Now we can take 2 arguments directly, with an associated evaluation rule.

$$(\lambda(x\ y)\ s)\ e_0\ e_1 \longmapsto s\{(e_0/x)\ (e_1/y)\}$$

Here we can consider $y$ as a continuation which takes the result of what the function did with $x$ and passes to the rest of the program.

## 5 CPS translation

We can make a translation from a lambda calculus expression to CPS statement. Let $\mathcal{T}$ be such a translation function:

$$\mathcal{T}[\![e]\!] = s$$

We wonder if $e$ is converted to $s$, and $e$ is evaluated to $v$, will $s$ be evaluated to $\mathsf{halt}\ \mathcal{T}[\![v]\!]$? This turns out not to be quite true, but it does evaluate to some equivalent term.

To define CPS translation, first we need to make sure the converted program ends with $\mathsf{halt}$:

$$\mathcal{T}[\![e]\!] = \mathcal{D}[\![e]\!]\ \mathsf{halt}$$

The contract satisfied by the semantic function $\mathcal{D}$ is:

$$\mathcal{D}[\![e]\!]k = s$$

Where $s$ is a statement that sends the value of $e$ to $k$. $k$ is continuation that takes the result of $e$ and continues the program. We can then define the translation from CBV $\lambda$ calculus to CPS:

$$\begin{array}{rcl} \mathcal{D}[\![x]\!]k & = & x \\ \mathcal{D}[\![\lambda\ x\ e]\!]k & = & k(\lambda(x\ k')\mathcal{D}[\![e]\!]k') \\ \mathcal{D}[\![e_0\ e_1]\!]k & = & \mathcal{D}[\![e_0]\!](\lambda f\ \mathcal{D}[\![e_1]\!](\lambda v\ (f\ v\ k))) \end{array}$$

Here is an example for CPS translation:

$$\begin{array}{lcl}
\mathcal{D}[\![(\lambda\ x(\lambda\ y\ y))1]\!]\ \mathsf{halt} & = & \mathcal{D}[\![\lambda\ x(\lambda\ y\ y)]\!](\lambda\ f\ \mathcal{D}[\![1]\!](\lambda\ v\ f\ v\ \mathsf{halt})) \\
 & = & (\lambda\ f\ \mathcal{D}[\![1]\!](\lambda\ v\ f\ v\ \mathsf{halt}))(\lambda\ (\ x\ k')\ k'\ \mathcal{D}[\![\lambda\ y\ y]\!]k') \\
 & = & (\lambda\ f\ (\lambda\ v\ f\ v\ \mathsf{halt})1)(\lambda\ (x\ k')\ k'(\lambda(y\ k'')(k''y))) \\
 & = & (\lambda\ (x\ k')\ k'(\lambda(y\ k'')(k''y)))\ 1\ \mathsf{halt} \\
 & = & \mathsf{halt}\ (\lambda(y\ k'')(k''y)) \\
 & = & \mathsf{halt}\ \mathcal{D}[\![\lambda y\ y]\!]
\end{array}$$