

This is a reference sheet together with some notes for the last assignment (HW8). Please also refer to the notes posted to the lectures section of the course website. An abbreviated form of these notes will be included with the final exam (notes sheet will be provided to you before the final exam review session).

1 λ -calculus

Syntax:

$$\text{Expr} ::= \text{Var} \mid \lambda \text{Var} . \text{Expr} \mid \text{Expr Expr}$$

Free variables:

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(\lambda x . e) &= \text{fv}(e) \setminus \{x\} \\ \text{fv}(e_1 e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \end{aligned}$$

Substitution:

$$\begin{aligned} x[x \mapsto e'] &= e' \\ y[x \mapsto e'] &= y \quad \text{where } x \neq y \\ (\lambda x . e)[x \mapsto e'] &= (\lambda x . e) \\ (\lambda y . e)[x \mapsto e'] &= (\lambda y . e) \quad \text{where } x \neq y \\ (e_1 e_2)[x \mapsto e'] &= (e_1[x \mapsto e'])(e_2[x \mapsto e']) \end{aligned}$$

Equivalences:

$$\begin{aligned} \lambda x . e &\equiv_{\alpha} \lambda y . e[x \mapsto y] \quad \text{where } y \notin \text{fv}(e) \\ (\lambda x . e) e' &\equiv_{\beta} e[x \mapsto e'] \quad \text{where } x \notin \text{fv}(e) \\ e &\equiv_{\eta} \lambda x . ex \quad \text{where } x \notin \text{fv}(e) \end{aligned}$$

2 Combinatory Logic

All computable functions can be expressed in terms of the S and K combinators. These are simply two functions, defined as follows. Note that the identity function, for example, can be defined in terms of S and K .

$$\begin{aligned} S x y z &= x z (y z) \\ K x y &= x \\ I &= S K K \end{aligned}$$

Bracket abstraction—transformation from λ -calculus to CL:

$$\begin{aligned} \lambda x . \rho &\Rightarrow [x]\rho \\ [x]x &\Rightarrow S K K \\ [x]y &\Rightarrow K y \quad \text{where } x \neq y \\ [x]K &\Rightarrow K K \\ [x]S &\Rightarrow K S \\ [x](\rho \rho') &\Rightarrow S ([x]\rho) ([x]\rho') \end{aligned}$$

3 Natural Semantics

The semantics of the λ -calculus can also be specified in another form known as *natural semantics*:

$$\begin{aligned} (E, v) &\Downarrow v && \text{(values)} \\ (E, x) &\Downarrow x E && \text{(variables)} \\ (E, \lambda x . e') &\Downarrow \text{Cl}(x, e', E) && \text{(closures)} \\ \frac{(E, e_1) \Downarrow \text{Cl}(x, e', E') \quad (E, e_2) \Downarrow v \quad ([x \mapsto v]E', e') \Downarrow v'}{(E, e_1 e_2) \Downarrow v'} &&& \text{(application or } \beta\text{-reduction)} \end{aligned}$$

Please refer to MP6 for an example of natural semantics for a richer language.

4 Church Booleans

Alonzo Church devised an encoding similar to the following to encode Boolean values (`true` and `false`). The basic idea is that a Boolean acts like an if-then-else statement, which either executes the then-clause (`true`) or the else-clause (`false`).

```
true  =  $\lambda then . \lambda else . then$ 
false =  $\lambda then . \lambda else . else$ 
and   =  $\lambda x . \lambda y . x y \text{ false}$ 
not   =  $\lambda x . \lambda then . \lambda else . x \text{ else } then$ 
...   = ...
```

5 Church Numerals

Alonzo Church devised an encoding similar to the following to encode natural numbers ($0, 1, 2, \dots$). The basic idea is that a number n acts like a for-loop that runs n times.

```
zero  =  $\lambda s . \lambda z . z$ 
succ  =  $\lambda n . \lambda s . \lambda z . s (n s z)$ 
one   =  $\text{succ zero} = \lambda s . \lambda z . s z$ 
two   =  $\text{succ one} = \lambda s . \lambda z . s (s z)$ 
...   = ...
add   =  $\lambda n . \lambda m . \lambda s . \lambda z . n s (m s z)$ 
```

6 The Y Combinator

Church numerals give a limited form of iteration, but not general recursion. Since our definition of λ -calculus does not include a `let` or `let rec` construct, some other equivalent notion must be constructed. The fixpoint combinator known as the Y combinator serves this purpose. Y is defined to respect the following equation, for any expression F :

$$Y F = F (Y F)$$

In other words, Y provides access to arbitrarily many copies of F . The following λ -calculus expression is a solution to the above equation:

$$Y = \lambda F . (\lambda x . F (x x))(\lambda x . F (x x))$$

For example, the following two programs, on the left, using `let rec`, and on the right, using the Y combinator, are equivalent. The advantage of the Y combinator is that it's not an additional language construct, it's simply a function.

$$\text{let rec } f = \dots f \dots \text{ in } f \quad \equiv \quad Y (\lambda f . \dots f \dots)$$

7 Simply-typed λ -calculus

The *simply-typed λ -calculus* is a restriction of the λ -calculus which introduces *simple types* for all variables and terms. An important property of the simply-typed λ -calculus is that simplification of all well-typed expressions terminates (you can simplify typed expressions using the semantics of the untyped λ -calculus, simply ignoring the types).

Syntax:

$$\begin{aligned} \text{Type} &::= \bullet \mid \text{Type} \rightarrow \text{Type} \\ \text{Expr} &::= \text{Var} \mid \lambda \text{Var} : \text{Type} . \text{Expr} \mid \text{Expr Expr} \end{aligned}$$

8 Continuations

A *continuation* is an abstraction of *the rest of the program*. This was the subject of MP7, so you should still have a good idea of how to work with continuations.

CPS transform: takes a program and rewrites it to an equivalent program in continuation-passing style. Given here for the λ -calculus, but can be defined analogously for other (functional) programming languages.

$$\begin{aligned} \text{CPS}(x) &= \lambda k . k x \\ \text{CPS}(\lambda x . e) &= \lambda k . k (\lambda k' . \lambda x . \text{CPS}(e) k') \\ \text{CPS}(e_1 e_2) &= \lambda k . \text{CPS}(e_2) (\text{CPS}(e_1) (\lambda f . f k)) \end{aligned}$$

Note that the result of the CPS transform is closer to machine code in the sense that function calls have essentially been weakened to *jump* or *go to* statements. However, in general the generated code will be suboptimal: there may be sections of code which can be simplified, or *optimized*, even without knowledge the inputs that will be used with the program.

9 Pseudo-assembly Language

Let's use the following simplified syntax for assembly language. While the semantics is not specified formally, it should be intuitive. Our constants are numbers and labels, and our variables are the registers, R_0, R_1, R_2, \dots (assume infinitely many registers unless otherwise specified; you can give the registers arbitrary names, as long as they start with R).

$R_i = K$	load constant K into register R_i
$R_i = R_j$	copy the contents of register R_j into register R_i
$R_i = *R_j$	load the value at the memory address stored in R_j into register R_i
$*R_i = R_j$	store the contents of register R_j into memory at the address stored in R_i
$R_i = R_j \text{ op } R_k$	where op is one of $+$, $-$, $*$, $/$, and , or , xor
push R_i	push the contents of register R_i onto the stack
pop R_i	pop the next value off the stack, into register R_i
jmp <i>label</i>	unconditional jump to <i>label</i>
jmp $*R_i$	unconditional indirect jump to address stored in R_i
jg R_i, R_j, label	jump to <i>label</i> if $R_i > R_j$
jeq R_i, R_j, label	jump to <i>label</i> if $R_i = R_j$
$R_i = \phi(R_j, R_k)$	SSA ϕ operator: R_i gets the value of R_j or R_k depending on the control flow

Example: note that the code accepts parameters and returns its result on the stack.

```
f : pop Rret
    pop Rcount
    R0 = 0
    R1 = 1
    Rsum = 0
loop : jg R0, Rcount, exit
      Rsum = Rsum + Rcount
      Rcount = Rcount - R1
      jmp loop
exit : push Rsum
      jmp *Rret
```

SSA form, or *static single assignment* form simply means that in the code, each (logical) register is assigned a value exactly *once*. The ϕ operator is used to collect values assigned along different control flow paths into a single register—for example, the following implements the code “if $x > y$ then $x = x + 1$ else $x = y$.”

```
    jg Ry, Rx0, else
    R1 = 1
    Rx1 = Rx0 + R1
    jmp done
else : Rx2 = Ry
done : Rx3 =  $\phi(R_{x1}, R_{x2})$ 
      // from here on, use Rx3 for x
```