Class Announcements

- Fifth homework is due Wednesday, April 5
- Second project has been posted and is due Monday, April 17
- Midterms
  - Second midterm will be on April 11.
    Final (third midterm) exam on Thursday, May 4, noon - 3:00pm timeslot. Location: most likely this room.
    Any CONFLICTS with other classes?

Review - Function application

Computation in the lambda calculus is based on the concept or reduction (rewrite rules). The goal is to "simplify" an expression until it can no longer be further simplified.

$((\lambda x.M)N) \quad \Rightarrow_{\beta} \quad [N/x]M \qquad (\beta\text{–reduction})$

Replace all free occurences of **x** (formal parameter) in **M** (body of function) by **N** (actual parameter), where the replacement is caputure free (capture-free substitution)

$(\lambda x.M) \quad \Rightarrow_{\alpha} \quad \lambda y.[y/x]M \qquad (\alpha\text{–reduction})$

if $y \notin free(M)$

Renaming used to enable capture-free substitution

Note:

- α–reduction does not reduce the complexity.
- β–reduction: corresponds to application, models computation.

Review - Reduction

- A subterm of the form (λx.M)N is called a ***redex*** (reduction expression).
- A reduction is any sequence of β–reductions and α–reductions.
- A term that cannot be β–reduced is said to be in β–normal form (normal form).
- A subterm that is an abstraction or a variable is said to be in head normal form.

Does a normal form always exist?

Examples:
($(\lambda$x.(xx))(λx.(xx)))

Programming in lambda calculus

The lambda calculus has very few constructs and it is therefore easy to reason *about it*.

Question: Is the lambda calculus too simple, i.e., can we express all computable functions in the lambda calculus?

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly β–reductions).

Logical constants and operations (incomplete list):

true $\equiv$ λa.λb.a *select–first*

false $\equiv$ λa.λb.b *select–second*

cond $\equiv$ λm.λn.λp.((p m)n)

not $\equiv$ λx.((x false) true)

equiv $\equiv$ λx.λy.((x y) false)

or $\equiv$ λx.λy. ((x true) y)

Programming in lambda calculus

What about data structures?

data structures:

pairs can be represented as

$$[M\ .\ N]\ \equiv\ \lambda \text{z.((z M) N)}$$

first $\equiv$ λx.(x true)	(*car*)
second $\equiv$ λx.(x false)	(*cdr*)
build $\equiv$ λx.λy.λz.((z x) y)	(*cons*)

Programming in lambda calculus

What about arithmetic constants and operations?

There are many options here. Let's look at the system proposed by Church:

$0 \equiv \lambda$fx.x
$1 \equiv \lambda$fx.(f x)
$2 \equiv \lambda$fx.(f (f x))
...
n $\equiv \lambda f x.(\underbrace{f(f(\ldots(f}_{n\ times} x)\ldots)) \equiv \lambda f x.(f^n x)$

The natural number **n** is represented as a function that applies a function f n–times to its argument x.

succ $\equiv \lambda$m.(λfx.(f (m f x)))
add $\equiv \lambda$mn.(λfx.((m f) (n f x)))
mult $\equiv \lambda$mn.(λfx.((m (n f)) x))
isZero? $\equiv \lambda$m.((m (true false)) true)

Programming in lambda calculus

Examples:

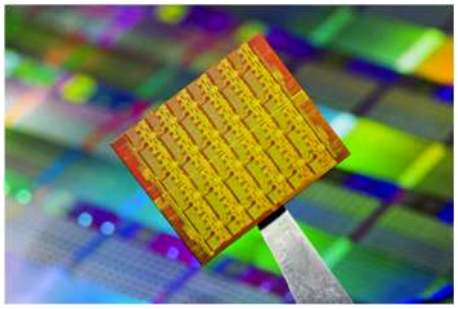
(mult 2 3) =
$((\lambda$mn.$(\lambda$fx.((m (n f)) x))) 2 3) =
$\lambda f_0 x_0.((2\ \boxed{(3\ f_0)})\ x_0)$ =
$\lambda f_0 x_0.((2\ ((\lambda$fx.(f (f (f x))))$\ f_0))\ x_0)$ =
$\lambda f_0 x_0.((2\ (\lambda$x.$(f_0\ (f_0\ (f_0$ x)))))$\ x_0)$ =
$\lambda f_0 x_0.(\boxed{(2\ (\lambda x_1.(f_0^3\ x_1)))}\ x_0)$ =
$\lambda f_0 x_0.((\lambda$x.$((\lambda x_1.(f_0^3\ x_1))\ \boxed{(\ (\lambda x_1.(f_0^3\ x_1))$ x)}$\ ))\ x_0)$ =
$\lambda f_0 x_0.((\lambda$x.$\boxed{(\ (\lambda x_1.(f_0^3\ x_1))\ (f_0^3$ x))}$)\ x_0)$ =
$\lambda f_0 x_0.\boxed{((\lambda$x.$(f_0^3\ (f_0^3$ x)))$\ x_0)}$ =
$\lambda f_0 x_0.(f_0^3\ (f_0^3\ x_0))$ =
λfx.$(f^6$ x) = 6

Programming with Concurrency

Why do we care about concurrency?

- Today, concurrency is nearly everywhere (peta-flops supercomputers to high-end smart phones).
- Necessary to keep "Moore's Law" alive due to power/heat dissipation limits.
- Some form of parallel programming will be required, i.e., automatic tools have not been able to hide all aspects of concurrency.

⇒ Need to understand the basics of parallel programming



Programming with Concurrency

Two ways of thinking about concurrency?

data-centric view: partition the data that can be worked on in parallel (data-level parallelism);

⇒ your work is determined by the data that you are assigned to work on.

task-centric view: partition the work that can be done concurrently (task-level parallelism);

⇒ your data is determined by the work that you have to do

What tasks have "to travel" to what data (data-centric) or what data has "to travel" to what tasks (task-centric) are symmetric problems.

Programming with Concurrency

Task-level parallelism can be performed at different levels:

1. **Instruction-level** parallelism (ILP) – typically exploited by hardware or compiler
2. **Loop-level parallelism** – single loop iterations are considered individual tasks
3. **Procedure-level** parallelsim – different procedures may be executed concurrently
4. **Process-level** parallelism – different programs may be executed concurrently

Will concentrate on loop-level parallelism

Next Lecture

Things to do:

- Loop level parallelism
- Dependence testing
- Vectorization vs. Parallelization
- OpenMP tutorial on our website