Spare Time Teaching's 2013/2014 Note Collection

Kristoffer Andersen
Christian Clausen
Mikkel Kringelbach
Richard Möhn
Mathias Pedersen

Edited by Christian Clausen

Foreword

This is the first official collection of notes from Spare Time Teaching. The people who have contributed to this collection were either bachelors or masters students, and every subject presented has been studied and prepared during the author's spare time.

The topics in this edition fall mainly within the field of programming languages, more specifically: Lambda Calculus, type theory (dependent types, type inference, and type classes), functional and logic programming, semantics, abstract interpretation, and low level programming (assembly and bit manipulation).

This edition contains notes from all the talks that were presented with corresponding pdf, therefore there may be more notes, source code, and other material avalible on Spare Time Teaching's website (`daimi.au.dk/~christia`). The notes are ordered with respect to the order they were presented in.

The content of this collection is not to be considered as research, but rather as experience reports, tutorials, or supplementing material for presentations.

The contributing authors to this edition are:

- Kristoffer Andersen
- Christian Clausen
- Mikkel Kringelbach
- Richard Möhn
- Mathias Pedersen

With special thanks to Kent Grigo for his corrections, comments, and suggestions for this note collection.

Contents

Intuition to Lambda Calculus

Christian Clausen

December 30, 2013

1 Grammar

```
t ::= v
    | λ v . t
    | t t
```

2 Reduction rules

2.1 Metarules

```
"⇝" means "syntax for"
x y z ⇝ (x y) z
λ x. t1 t2 ⇝ λx. (t1 t2)
λ x y . t ⇝ λ x . λ y . t
t[t'/x] is substitution
```

2.2 α Renaming

```
λ x . t ⟶α λ y . t[y/x]
```

2.3 η Expansion

```
f ⟶η λ x . f x
```

2.4 β Reduction

$(\lambda\ x\ .\ t_1)\ t_2\ \longrightarrow^{\beta}\ t_1[t_2/x]$

3 The Identity Function

3.1 Function

$id \equiv \lambda\ x\ .\ x$

3.2 Evaluation

$id\ id = (\lambda\ x\ .\ x)\ (\lambda\ x\ .\ x) \longrightarrow^{\alpha} (\lambda\ x\ .\ x)\ (\lambda\ y\ .\ y) \longrightarrow^{\beta} x[(\lambda\ y\ .\ y)/x] = \lambda\ y\ .\ y = id$

$id\ id = (\lambda\ x\ .\ x)\ id \longrightarrow^{\beta} x[id/x] = id$

4 Some Terminology

Abstraction, normalization, free variable, name capture, combinator, weak-head normal form, closed terms, ...

5 Combinators

$I \equiv \lambda\ x\ .\ x$
$K \equiv \lambda\ x\ y\ .\ x$
$B \equiv \lambda\ x\ y\ z\ .\ x\ (y\ z)$
$S \equiv \lambda\ f\ g\ x\ .\ f\ x\ (g\ x)$
$K^* \equiv \lambda\ x\ y\ .\ y$

```
K I   ↠  K*
K* I   ↠  I
S K I   ↠  I
S K K   ↠  I
S (K S) K   ↠  B

X = λ x . x S K
X X   ↠  S K (K K)
X (X X)   ↠  S K
X (X (X X))   ↠  K
X (X (X (X X)))   ↠  S
```

6 Booleans

6.1 Definition

```
T ≡ λ t f . t
F ≡ λ t f . f
```

6.2 Not

```
not ≡ λ b . λ t f . b f t
not ≡ λ b . b (λ t f . f) (λ t f . t)
not ≡ λ b . b F T
¬a ⇝ not a
```

6.3 And

```
and ≡ λ b₁ b₂ . b₁ b₂ F
a & b ⇝ and a b
```

6.4 Exercise: Or

```
or ≡ λ b1 b2 . b1 T b2
a | b ⇝ and a b
```

7 Numbers

7.1 Definition

```
nat ::= 0
      | Succ nat
```

```
0 ≡ λ fs vz . vz
mk_succ ≡ λ n . λ fs vz . fs (n fs vz)
succ ≡ mk_succ
```

Church numerals

```
0 ≡ λ s z . z
1 ≡ λ s z . s z
2 ≡ λ s z . s (s z)
...
```

7.2 Is zero?

```
is_zero? ≡ λ n . n (λ α . F) T
```

7.3 Plus

```
plus ≡ λ m n . m succ n
a + b ⇝ plus a b
```

7.4 Times

```
times ≡ λ m n . m (plus n) 0
a * b ⇝ times a b
```

8 Pairs

8.1 Definition

```
pair ::= P * *
```

```
mk_pair ≡ λ a b . λ f_p . f_p a b
<a, b> ⇝ mk_pair a b
π_1 ≡ λ p . p (λ a b . a)
π_2 ≡ λ p . p (λ a b . b)
```

8.2 Swap

```
swap ≡ λ p . <π_2 p, π_1 p>
swap ≡ λ p . mk_pair (π_2 p) (π_1 p)
```

8.3 Sliding Window

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a + b>}$$

```
fib ≡ λ n . π_1 (n f <0, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<succ a, a * b>}$$

```
fac ≡ λ n . π_2 (n f <1, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a>}$$

```
is_even? ≡ λ n . π_1 (n f <T, F>)
```

9 Lists

9.1 Definition

```
list ::= Nil
       | Cons * list
```

```
mk_cons ≡ λ hd tl . λ f_c v_n . f_c hd (tl f_c v_n)
hd :: tl ⇝ mk_cons hd tl
nil ≡ λ f_c v_n . v_n
```

9.2 Sum

```
sum_list ≡ λ xs . xs plus 0
```

10 Trees

10.1 Definition

```
tree ::= Leaf *
       | Node tree tree
```

```
mk_node ≡ λ t_1 t_2 . λ f_n f_l . f_n (t_1 f_n f_l) (t_2 f_n f_l)
mk_leaf ≡ λ n . λ f_n f_l . f_l n
```

10.2 Sum

```
sum_tree ≡ λ t . t plus id
```

11 Unbounded Iteration

```
ω ≡ λ x . x x
Ω ≡ ω ω
```

11.1 But Y

11.1.1 Curry

W_f = λx . f (x x)
Y_C ≡λ f . W_f W_f

11.1.2 Turing

A ≡ λ x y . y (x x y)
Y_T ≡ A A

11.2 Example: Streams

$\widehat{\mathbb{N}}$ ≡ λ self n . <n, self (succ n)>
$\mathbb{N}$ ≡ Y $\widehat{\mathbb{N}}$ 0

$\widehat{\text{zip}}$ ≡λ self xst yst . <π_1 xst, self yst (π_2 xst)>
zip ≡ Y $\widehat{\text{zip}}$

Day 1
Untyped Lambda Calculus

Christian Clausen

January 31, 2014

1 Syntax

```
t ::= x              VAR(new)
    | lam x. t       ABS(new)
    | t t            APP(new)

x y z ⇝ (x y) z
lam x. t1 t2 ⇝ lam x. (t1 t2)
lam x y. t ⇝ lam x. lam y. t
let x = t1 in t2 ⇝ (lam x. t2) t1
```

2 Evaluation

2.1 Version 1

$$\frac{}{x \Downarrow x}\textsc{Var} \qquad \frac{t \Downarrow t'}{\lambda x.t \Downarrow \lambda x.t'}\textsc{Abs}$$

$$\frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[t_2'/x] \Downarrow t'}{t_1\ t_2 \Downarrow t'}\textsc{App}_1 \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}\textsc{App}_2$$

2.2 Version 2

$$\frac{\sigma(x) = t \quad (\sigma, t) \Downarrow t'}{(\sigma, x) \Downarrow t'}\textsc{Var} \qquad \frac{(\sigma[x \mapsto x], t) \Downarrow t'}{(\sigma, \lambda x.t) \Downarrow \lambda x.t'}\textsc{Abs}$$

$$\frac{(\sigma, t_1) \Downarrow \lambda x.t \quad (\sigma, t_2) \Downarrow t_2' \quad (\sigma[x \mapsto t_2'], t) \Downarrow t'}{(\sigma, t_1\ t_2) \Downarrow t'}\textsc{App}_1 \qquad \frac{(\sigma, t_1) \Downarrow t_1' \quad (\sigma, t_2) \Downarrow t_2'}{(\sigma, t_1\ t_2) \Downarrow t_1'\ t_2'}\textsc{App}_2$$

$\sigma(x)$ has to be unique.

3 Typing rules

There are no types in the untyped lambda calculus.

4 Examples

```
T ≡ lam t f. t
F ≡ lam t f. f
and ≡ lam b₁ b₂. b₁ b₂ F
and T T ⇓ T
and F T ⇓ F
T T ⇓ lam f. T
```

5 Proofs

As there are no types, there are no proofs.

6 Variations

- Thunks.
- Call by name.
- Allowing unbound variables.

- De Bruijn indices. Representing variables as the number of lambdas since it was bound; $\lambda x.x$ would be $\lambda 0$, $\lambda xy.x$ would be $\lambda\lambda 1$.

  Now we have a 3-step model for substitution without an environment. For $(\lambda x.t_1)\ t_2$, we have:

  1. Find the (free) occurences of x in t_1.
  2. Remove the λ and subtract 1 from all the free variables in t_1.
  3. Put t_2 in the places found in (1), and add 1 to the free variables of t_2 each time you encounter a λ.

Curry-Howard Correspondance

Christian Clausen

February 6, 2014

1 Propositional Logic

1.1 Hypothesis

We assume the axiom:

$$\frac{}{\Gamma_1, A, \Gamma_2 \vdash A}\textsc{Hyp}$$

Which in words means that if A is already one of our assumptions, we may use it.

1.2 Top

$$\frac{}{\Gamma \vdash \top}\top i$$

1.3 Bottom

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\bot e$$

1.4 Conjunction

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge i$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge e_1$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge e_2$$

1.5 Disjunction

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee i_1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee i_2$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee e$$

1.6 Implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to i$$

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to e$$

1.7 Syntactic Sugar

Not:

$$\neg A \equiv A \to \bot$$

Bi-implication:

$$A \leftrightarrow B \equiv (A \to B) \wedge (B \to A)$$

2 Propositional Logic as Programs

2.1 Hypothesis

Here HYP would be a variable access.

2.2 Top

$$\frac{}{\Gamma \vdash \texttt{()} : \top} \top i$$

2.3 Bottom

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \texttt{raise (Contradiction M)} : A} \bot e$$

2.4 Conjunction

$$\frac{\Gamma \vdash \texttt{M} : A \quad \Gamma \vdash \texttt{N} : B}{\Gamma \vdash \texttt{(M, N)} : A \wedge B} \wedge i$$

$$\frac{\Gamma \vdash \texttt{M} : A \wedge B}{\Gamma \vdash \texttt{let (ha, \_) = M in ha} : A} \wedge e_1$$

$$\frac{\Gamma \vdash \texttt{M} : A \wedge B}{\Gamma \vdash \texttt{let (\_, hb) = M in hb} : B} \wedge e_2$$

2.5 Disjunction

$$\frac{\Gamma \vdash \texttt{M} : A}{\Gamma \vdash \texttt{Left M} : A \vee B} \vee i_1$$

$$\frac{\Gamma \vdash \texttt{M} : B}{\Gamma \vdash \texttt{Right M} : A \vee B} \vee i_2$$

$$\frac{\Gamma \vdash \texttt{M} : A \vee B \quad \Gamma, \texttt{ha} : A \vdash \texttt{L} : C \quad \Gamma, \texttt{hb} : B \vdash \texttt{R} : C}{\Gamma \vdash \texttt{match M with Left ha -> L | Right hb -> R} : C} \vee e$$

2.6 Implication

$$\frac{\Gamma, \texttt{ha} : A \vdash \texttt{M} : B}{\Gamma \vdash \texttt{fun ha -> M} : A \to B} \to i$$

$$\frac{\Gamma \vdash \texttt{M} : A \to B \quad \Gamma \vdash \texttt{N} : A}{\Gamma \vdash \texttt{M N} : B} \to e$$

3 Example

We want to prove: $A \vee B \to \neg A \to B$

3.1 As a Proof Tree

$$\dfrac{\dfrac{}{A \vee B, \neg A \vdash A \vee B}\textsc{Hyp} \quad \dfrac{\dfrac{\dfrac{}{\neg A, A \vdash A \to \bot}\textsc{Hyp} \quad \dfrac{}{\neg A, A \vdash A}\textsc{Hyp}}{\neg A, A \vdash \bot} \to e}{\neg A, A \vdash B} \bot e \quad \dfrac{}{\neg A, B \vdash B}\textsc{Hyp}}{A \vee B, \neg A \vdash B} \vee e$$

$$\dfrac{\dfrac{A \vee B, \neg A \vdash B}{A \vee B \vdash \neg A \to B} \to i}{\cdot \vdash A \vee B \to \neg A \to B} \to i$$

Note: we have removed $A \vee B$ after the $\vee e$ to save space, and because we could always re-introduce it using $\vee i_1$ or $\vee i_2$.

3.2 As a Program

```
fun h1 ->  (* →i *)
  fun hna ->  (* →i *)
    match h1 with  (* ∨e *)
      | Left ha -> raise (Contradiction  (* ⊥e *)
          (hna ha))  (* →e *)
      | Right hb -> hb
```

The places where we use `h1`, `hna`, `ha`, or `hb` correspond to a use of the HYP-axiom in the proof.

If we type this in OCaml (after **`open`** `Prop`) we get the type signature:

```
('a, 'b) disj -> ('a -> bot) -> 'b
```

which translates to:

$$\alpha \vee \beta \to \neg\alpha \to \beta$$

Day 2

Simply Typed Lambda Calculus

Christian Clausen

December 30, 2013

1 Syntax

```
τ ::= A                 ATOM(new)
    | τ -> τ            ARROW(new)

t ::= x                 VAR
    | lam x. t          ABS
    | t t               APP
    | t : τ             ANN(new)
    | lam x : τ. t      ANNABS(new)

lam x y : τ. t ⇝ lam x : τ. lam y : τ. t
Lemma: τ Proof: t ⇝ t : τ
```

2 Evaluation

$$\frac{}{x \Downarrow x}\textsc{Var} \qquad \frac{t \Downarrow t'}{\lambda x.t \Downarrow \lambda x.t'}\textsc{Abs}$$

$$\frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[t_2'/x] \Downarrow t'}{t_1\ t_2 \Downarrow t'}\textsc{App}_1 \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}\textsc{App}_2$$

$$\frac{t \Downarrow t'}{t : \tau \Downarrow t'}\textsc{Ann} \qquad \frac{t \Downarrow t'}{\lambda x : \tau.t \Downarrow \lambda x.t'}\textsc{AnnAbs}$$

3 Typing rules

3.1 Type Well-formedness

$$\frac{}{\Gamma \vdash_{wf} A}\textsc{Atom} \qquad \frac{\Gamma \vdash_{wf} \tau_1 \quad \Gamma \vdash_{wf} \tau_2}{\Gamma \vdash_{wf} \tau_1 \to \tau_2}\textsc{Arrow}$$

3.2 Inferable Types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \uparrow \tau}\textsc{Var} \qquad \frac{\Gamma \vdash t_1 \uparrow \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 \downarrow \tau_1}{\Gamma \vdash t_1\ t_2 \uparrow \tau_2}\textsc{App}$$

$$\frac{\Gamma \vdash_{wf} \tau \quad \Gamma \vdash t \downarrow \tau}{\Gamma \vdash t : \tau \uparrow \tau}\textsc{Ann} \qquad \frac{\Gamma \vdash_{wf} \tau_1 \quad \Gamma, x : \tau \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t \uparrow \tau_1 \to \tau_2}\textsc{AnnAbs}$$

3.3 Checkable Types

$$\frac{\Gamma, x : \tau_1 \vdash t \downarrow \tau_2}{\Gamma \vdash \lambda x . t \downarrow \tau_1 \to \tau_2}\textsc{Abs} \qquad \frac{\Gamma \vdash t \uparrow \tau}{\Gamma \vdash t \downarrow \tau}\textsc{Check}$$

4 Examples

$\mathtt{id}_A \equiv$ **lam** x : A. x
$\mathtt{id}_{A \to A} \equiv$ **lam** x : A -> A. x

$\mathtt{id}_A \Downarrow \mathtt{id}_A$
$\mathtt{id}_A \uparrow \mathtt{A} \to \mathtt{A}$

$\mathtt{id}_A$ n $\Downarrow$ n
$\mathtt{id}_A$ n $\uparrow$ A

$\mathtt{id}_{A \to A}\ \mathtt{id}_A \Downarrow \mathtt{id}_A$
$\mathtt{id}_{A \to A}\ \mathtt{id}_A \uparrow \mathtt{A} \to \mathtt{A}$

$\mathtt{id}_A\ \mathtt{id}_A \Downarrow \mathtt{id}_A$
$\mathtt{id}_A\ \mathtt{id}_A \uparrow \lightning$

$\mathtt{id}_{A \to A}\ \mathtt{id}_{A \to A} \Downarrow \mathtt{id}_{A \to A}$

$\texttt{id}_{A \to A}$ $\texttt{id}_{A \to A} \uparrow \lightning$

5 Proofs

```
Lemma:
  A -> B -> A
Proof:
  lam a : A. lam b : B. a

Lemma:
  (A -> B) -> (A -> B)
Proof:
  lam f : A -> B. lam a : A. f a
```

6 Variations

- Fixed set of atom types.
- Extendable set of atom types.
- Infering the set of atom types.
- Support for "holes", where it will tell you what the current type environment is.
- Support for free variables, using this rule:

$$\frac{}{\Gamma \vdash n \downarrow \tau} \textsc{Free Var}$$

Day 3

System F

Christian Clausen

September 30, 2014

1 Syntax

τ ::=	A	Atom
	τ -> τ	Arrow
	α	TypeVar$^{(new)}$
	forall α, τ	Forall$^{(new)}$
t ::=	x	Var
	lam x. t	Abs
	t t	App
	t : τ	Ann
	lam x : τ. t	AnnAbs
	Lam α. t	TypeAbs$^{(new)}$
	t[τ]	TypeApp$^{(new)}$

2 Evaluation

$$\frac{}{x \Downarrow x}\textsc{Var} \qquad \frac{t \Downarrow t'}{\lambda x.t \Downarrow \lambda x.t'}\textsc{Abs}$$

$$\frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[t_2'/x] \Downarrow t'}{t_1\ t_2 \Downarrow t'}\textsc{App}_1 \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}\textsc{App}_2$$

$$\frac{t \Downarrow t'}{t : \tau \Downarrow t'}\textsc{Ann} \qquad \frac{t \Downarrow t'}{\lambda x : \tau.t \Downarrow \lambda x.t'}\textsc{AnnAbs}$$

$$\frac{t \Downarrow t'}{\Lambda\alpha, t \Downarrow t'}\textsc{TypeAbs} \qquad \frac{t \Downarrow t'}{t[\tau] \Downarrow t'}\textsc{TypeApp}$$

3 Typing rules

3.1 Type Well-formedness

$$\frac{}{\Gamma \vdash_{wf} A}\textsc{Atom} \qquad \frac{\Gamma \vdash_{wf} \tau_1 \quad \Gamma \vdash_{wf} \tau_2}{\Gamma \vdash_{wf} \tau_1 \to \tau_2}\textsc{Arrow}$$

$$\frac{\Gamma(\alpha) = *}{\Gamma \vdash_{wf} \alpha}\textsc{TypeVar} \qquad \frac{\Gamma, \alpha : * \vdash_{wf} \tau}{\Gamma \vdash_{wf} \forall\alpha, \tau}\textsc{Forall}$$

3.2 Inferable Types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \uparrow \tau}\textsc{Var} \qquad \frac{\Gamma \vdash t_1 \uparrow \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 \downarrow \tau_1}{\Gamma \vdash t_1\ t_2 \uparrow \tau_2}\textsc{App}$$

$$\frac{\Gamma \vdash_{wf} \tau \quad \Gamma \vdash t \downarrow \tau}{\Gamma \vdash t : \tau \uparrow \tau}\textsc{Ann} \qquad \frac{\Gamma \vdash_{wf} \tau_1 \quad \Gamma, x : \tau \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.t \uparrow \tau_1 \to \tau_2}\textsc{AnnAbs}$$

$$\frac{\Gamma, \alpha : * \vdash t \uparrow \tau}{\Gamma \vdash \Lambda\alpha, t \uparrow \forall\alpha, \tau}\textsc{TypeAbs} \qquad \frac{\Gamma \vdash_{wf} \tau \quad \Gamma \vdash t \uparrow \forall\alpha, \tau'}{\Gamma \vdash t[\tau] \uparrow \tau'[\tau/\alpha]}\textsc{TypeApp}$$

3.3 Checkable Types

$$\frac{\Gamma, x : \tau_1 \vdash t \downarrow \tau_2}{\Gamma \vdash \lambda x.t \downarrow \tau_1 \to \tau_2}\textsc{Abs} \qquad \frac{\Gamma \vdash t \uparrow \tau}{\Gamma \vdash t \downarrow \tau}\textsc{Check}$$

4 Examples

```
id_α ≡ Lam α. lam x : α. x

id_α ⇓ id_α
id_α ↑ ∀α, α → α

id_α [A] n ⇓ n
id_α [A] n ↑ A

id_α [A] id_α ⇓ id_α
id_α [A] id_α ↑ ↯

id_α [forall α, α -> α] id_α ⇓ id_α
id_α [forall α, α -> α] id_α ↑ ∀α, α → α
```

5 Proofs

```
Lemma:
  forall α β, α -> β -> α
Proof:
  Lam α β. lam a : α. lam b : β. a

Lemma:
  forall α β, (α -> β) -> (α -> β)
Proof:
  Lam α β. lam f : α -> β. lam a : α. f a
```

6 Variations

- Implicite type application, using this rule:

$$\frac{\Gamma \vdash t_1 \uparrow \forall \alpha, \tau_1 \quad \Gamma \vdash t_2 \uparrow \tau_2}{\Gamma \vdash t_1\ t_2 \uparrow \tau_1[\tau_2/\alpha]}\textsc{ImplTypeApp}$$

- Hindley-Milner type inference

Hindley-Milner Type Inference

Kristoffer Andersen
kja@cs.au.dk

Christian Clausen
christia@cs.au.dk

March, 2014

1 Introduction

This brief note accompanies a hands-on session on HM-style type inference conducted through the Spare Time Teaching initiative in March, 2014. It is not intended to be self contained. For self-study we refer to the articles in the list of references, and from then on encourage further exploration; the literature behind a whole culture of programming and technology is truly more exhaustive than we could ever hope to represent here.

This note details the study of an object language of study along with a syntax of its types. The following and final section presents Algorithm $\mathcal{W}$.

2 The Language

The object language is the first order Lambda Calculus, presented in Figure 1—with implicit typing—along with a language of types featuring second order quantification, necessary for polymorphic functions. This is presented in Figure 2

t ::=	x	variable
|	t t	application
|	λx.t	abstraction
|	let x = t in t	let-binding

Figure 1: Simply typed lambda calculus with let-bindings

τ ::=	α	type variable
|	atom_i	atomic types
|	$\tau \to \tau$	function types
|	$\forall\alpha.\tau$	type abstraction

Figure 2: Simply typed lambda calculus with let-bindings

3 The Algorithm

Algorithm $\mathcal{W}$ was first presented by Damas and Milner in 1982, a functional programming twin to the work done by Hindley a decade earlier; the type systems using these ideas are also sometimes triple-barelled, as in *Hindley-Milner-Damas*.

The presentation here is taken from Heeren, Haage, & Swierstra who did an excellent and very readable summary in 2002, and is shown in Figure 3. Proofs of soundness and completeness are to be found in the original article.

$$\mathcal{W} :: \mathit{TypeEnvironment} \times \mathit{Expression} \to \mathit{Substitution} \times \mathit{Type}$$

$$\begin{array}{lcl}
\mathcal{W}(\Gamma, \mathsf{x}) & = & ([], \mathit{instantiate}(\sigma)), \text{ where } (\mathsf{x} : \sigma) \in \Gamma \\
\\
\mathcal{W}(\Gamma, \lambda \mathsf{x}.t) & = & \textbf{let } (\mathcal{S}_1, \tau_1) = \mathcal{W}(\Gamma \setminus \mathsf{x} \cup \{\mathsf{x} : \beta\}, t), \text{ fresh } \beta \\
 & & \textbf{in } (\mathcal{S}_1, \mathcal{S}_1\beta \to \tau_1) \\
\\
\mathcal{W}(\Gamma, t_1\ t_2) & = & \textbf{let } (\mathcal{S}_1, \tau_1) = \mathcal{W}(\Gamma, t_1) \\
 & & \quad\ (\mathcal{S}_2, \tau_2) = \mathcal{W}(\mathcal{S}_1\Gamma, t_2) \\
 & & \quad\ \mathcal{S}_3 = \mathit{unify}(\mathcal{S}_2\tau_1, \tau_2 \to \beta), \text{ fresh } \beta \\
 & & \textbf{in } (\mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{S}_3\beta) \\
\\
\mathcal{W}(\Gamma, \mathsf{let\ x} = t_1 \ \mathsf{in}\ t_2) & = & \textbf{let } (\mathcal{S}_1, \tau_1) = \mathcal{W}(\Gamma, t_1) \\
 & & \quad\ (\mathcal{S}_2, \tau_2) = \mathcal{W}(\mathcal{S}_1\Gamma \setminus \mathsf{x} \cup \{\mathsf{x} : \mathit{generalize}(\mathcal{S}_1\Gamma, \tau_1)\}, t_2) \\
 & & \textbf{in } (\mathcal{S}_2 \circ \mathcal{S}_1, \tau_2)
\end{array}$$

Figure 3: Algorithm $\mathcal{W}$

The algorithm makes use of three auxilliary functions, along with the operations on substitutions and environments. *Instantiate* takes a polymorphic type and instantiates all quantified variables with fresh type variables. *Generalize* takes a type and universally quantifies all free variables in that type.

Finally, *unify* finds a substitution that equates two types under that substitution. How exactly to do this can be found in Knight's survey from 1989, but there are a range of classical results.

4 Recommended Exercises

Implement the algorithm in your favourite functional language!

This algorithm is very extensible, witnessed by the multitude of types available in OCaml. Particularly all of the *simple* types are good candidates for experimentation; here we propose a few:

1. Atom types;
   (a) unit, with ()
   (b) bool, with true, false, if t_1 then t_2 else t_3

2. Pair types, $\tau \times \tau$ with corresponding introduction and elimination forms: $\langle t_1, t_2 \rangle$, π_0 and π_1.

3. A fixpoint operator: fix f x.t

4. Sum types: $\tau + \tau$, inl, inr, case t_1 of | inl x -> t_2 | inr x -> t_3

5 References

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '82). ACM, New York, NY, USA, 207-212.

Bastian Heeren, Jurriaan Haaga and Doaitse Swierstra. 2002. Generalizing Hindley-Milner Type Inference Algorithms. Techical Report. Utrecht University, Utrecht, The Netherlands.

Kevin Knight. 1989. Unification: A Multidisciplinary Survey. ACM Computing Surveys.

Day 4

Dependant Types

Christian Clausen

December 30, 2013

1 Syntax

`t ::= x`	VAR
`| lam x. t`	ABS
`| t t`	APP
`| t : t`	ANN$^{(changed)}$
`| lam x : t. t`	ANNABS$^{(changed)}$
`| *`	STAR$^{(new)}$
`| forall` α `: t, t`	FORALL$^{(changed)}$

τ `->` $\tau \rightsquigarrow$ `forall _ :` τ`,` τ

`forall` α`,` $\tau \rightsquigarrow$ `forall` α `: *,` τ

2 Evaluation

$$\frac{}{x \Downarrow x}\textsc{Var} \qquad \frac{t \Downarrow t'}{\lambda x.t \Downarrow \lambda x.t'}\textsc{Abs}$$

$$\frac{t_1 \Downarrow \lambda x.t \quad t_2 \Downarrow t_2' \quad t[t_2'/x] \Downarrow t'}{t_1\ t_2 \Downarrow t'}\textsc{App}_1 \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}\textsc{App}_2$$

$$\frac{t \Downarrow t'}{t : \tau \Downarrow t'}\textsc{Ann} \qquad \frac{t \Downarrow t'}{\lambda x : \tau.t \Downarrow \lambda x.t'}\textsc{AnnAbs}$$

$$\frac{}{* \Downarrow *}\textsc{Star} \qquad \frac{\rho_1 \Downarrow \tau_1 \quad \rho_2 \Downarrow \tau_2}{\forall x : \rho_1, \rho_2 \Downarrow \forall x : \tau_1, \tau_2}\textsc{Forall}$$

3 Typing rules

3.1 Type Well-formedness

Well formedness is much simpler here:

$$\Gamma \vdash_{wf} \tau \quad \text{iff} \quad \Gamma \vdash \tau \downarrow *$$

Therefore, I have chosen to inline this into the following rules.

3.2 Inferable Types

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \uparrow \tau}\textsc{Var} \qquad \frac{\Gamma \vdash t_1 \uparrow \forall x : \tau_1, \tau_2 \quad \Gamma \vdash t_2 \downarrow \tau_1 \quad \tau_2[t_2/x] \Downarrow \tau_3}{\Gamma \vdash t_1\ t_2 \uparrow \tau_3}\textsc{App}$$

$$\frac{\Gamma \vdash \rho \downarrow * \quad \rho \Downarrow \tau \quad \Gamma \vdash t \downarrow \tau}{\Gamma \vdash t : \rho \uparrow \tau}\textsc{Ann} \qquad \frac{\Gamma \vdash \rho_1 \downarrow * \quad \rho_1 \Downarrow \tau_1 \quad \Gamma, x : \tau_1 \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \rho_1.t \uparrow \forall x : \tau_1, \tau_2}\textsc{AnnAbs}$$

$$\frac{}{\Gamma \vdash * \uparrow *}\textsc{Star}^1 \qquad \frac{\Gamma \vdash \rho_1 \downarrow * \quad \rho_1 \Downarrow \tau_1 \quad \Gamma, x : \tau_1 \vdash \rho_2 \downarrow *}{\Gamma \vdash \forall x : \rho_1, \rho_2 \uparrow *}\textsc{Forall}$$

3.3 Checkable Types

$$\frac{\Gamma, x : \tau_1 \vdash t \downarrow \tau_2}{\Gamma \vdash \lambda x.t \downarrow \forall x : \tau_1, \tau_2}\textsc{Abs} \qquad \frac{\Gamma \vdash t \uparrow \tau}{\Gamma \vdash t \downarrow \tau}\textsc{Check}$$

4 Examples

```
id_α ≡ lam α : *. lam x : α. x

id_α ⇓ id_α
id_α ↑ ∀α, α → α

id_α A n ⇓ n
id_α A n ↑ A

id_α A id_α ⇓ id_α
id_α A id_α ↑ ↯

id_α [forall α, α -> α] id_α ⇓ id_α
id_α [forall α, α -> α] id_α ↑ ∀α, α → α
```

5 Proofs

```
lam eq : forall α, α -> α -> *.
lam eq_refl : forall α, forall x : α, eq α x x.
lam eq_ind : forall α,
  forall m : (forall x y : α, eq α x y -> *),
  (forall z : α, m z z (refl α z)) ->
  forall x y : α, forall p : eq α x y, m x y p.

Lemma:
  forall x y, eq x y -> eq y x
Proof:
  [exercise for the reader]

Lemma:
  forall f x y, eq x y -> eq (f x) (f y)
Proof:
  [exercise for the reader]

Lemma:
```

```
  forall P x y, eq x y -> P x -> P y
Proof:
  [exercise for the reader]
```

6 Variations

If we look at some examples, we notice that we can emulate the (non-dependent) inductive definition

```
type nat =
  | Z : nat
  | S : nat -> nat
```

by the following definitions

```
lam nat : *.
lam Z : nat.
lam S : nat -> nat.
lam nat_ind :
  forall P : nat -> *,
  P Z ->
  (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n.
```

Here is another example

```
type list (α : *) =
  | nil : list α
  | cons : α -> list α -> list α
```

can be emulated as

```
lam list : * -> *.
lam nil : forall α, list α.
lam cons : forall α, α -> list α -> list α.
lam list_ind :
  forall α, forall P : list α -> *,
  P (nil α) ->
  (forall hd : α, forall tl : list α,
    P tl -> P (cons α hd tl)) ->
```

```
  forall l : list α, P l.
```

these two examples should give an intuition into how we can extend our language with non-dependent inductive definition.

For this to be really useful, you need to dynamically extend the interpreter with the rules:

$$\frac{}{\texttt{nat\_ind}\ P\ base\ ind\ \texttt{Z} \Downarrow base}\,nat_Z$$

$$\frac{\texttt{nat\_ind}\ P\ base\ ind\ n \Downarrow rec}{\texttt{nat\_ind}\ P\ base\ ind\ (\texttt{S}\ n) \Downarrow ind\ n\ rec}\,nat_S$$

$$\frac{P \Downarrow P' \quad base \Downarrow base' \quad ind \Downarrow ind'}{\texttt{nat\_ind}\ P\ base\ ind\ n \Downarrow \texttt{nat\_ind}\ P'\ base'\ ind'\ n}\,nat$$

$$\frac{}{\texttt{list\_ind}\ P\ base\ ind\ \texttt{nil} \Downarrow base}\,list_{nil}$$

$$\frac{\texttt{list\_ind}\ P\ base\ ind\ n \Downarrow rec}{\texttt{list\_ind}\ P\ base\ ind\ (\texttt{cons}\ a\ b) \Downarrow ind\ a\ b\ rec}\,list_{cons}$$

$$\frac{P \Downarrow P' \quad base \Downarrow base' \quad ind \Downarrow ind'}{\texttt{list\_ind}\ P\ base\ ind\ ls \Downarrow \texttt{list\_ind}\ P'\ base'\ ind'\ ls}\,list_{cons}$$

because then we can actually execute our programs.

6.1 Going Further

The previous section extends trivially to dependent types.

6.2 Going Even Further

Consider the code:

```
record pair (α β : *) =
{ fst : α ;
  snd : β ; }
```

This can be expanded to:

```
type pair (α β : *) =
| construct_pair : α -> β -> pair α β

let fst α β (p : pair α β) =
```

```
  pair_ind α β (lam _ . α) (lam x y . x) p in
let snd α β (p : pair α β) =
  pair_ind α β (lam _ . β) (lam x y . y) p in
```

We could then support code like:

```
let swap α β (p : pair α β) =
{ p.snd ;
  p.fst ; }
```

We could even support this:

```
let swap α β (p : pair α β) =
{ snd = p.fst ;
  fst = p.snd ; }
```

By expanding it to:

```
let swap α β (p : pair α β) =
  pair_construct (snd p) (fst p)
```

We could even go as far as:

```
let dup_fst α (p : pair α α) =
{ p with snd = p.fst }
```

Using everything that we have discussed we have a dependently typed language, with an almost convenient notation for complex math. If we assume notation for `and` (`/\`) and `eq` (=), we have:

```
let set α = α -> * in
let set_in α (x : α) (G : set α) = G x in
let subset α (G H : set α) =
  forall x, G x -> H x in
let union α (G H : set α) =
  [exercise for the reader] in
let intersection α (G H : set α) =
  [exercise for the reader] in

type exists α (P : α -> *) =
| ex_intro :
  forall x, P x -> exists α P
```

```
record Group α (G : set α) (comp : α -> α -> α) (e : α) =
{ closed : forall a b,
    set_in a G -> set_in b G ->
    set_in (comp a b) G ;
  assoc : forall a b c,
    set_in a G -> set_in b G -> set_in c G ->
    comp (comp a b) c = comp a (comp b c) ;
  e_in_G : set_in e G ;
  neutral_l : forall g, set_in g G -> comp e g = g ;
  neutral_r : forall g, set_in g G -> comp g e = g ;
  inverse_l : forall g, set_in g G -> exists α
    (fun h => set_in h G /\ comp h g = e) ;
  inverse_r : forall g, set_in g G -> exists α
    (fun h => set_in h G /\ comp g h = e) ; }

Lemma e_unique: forall G comp e, Group G comp e ->
  forall e', G e' ->
  (forall g, G g -> comp e' g = g) ->
  (forall g, G g -> comp g e' = g) ->
  e = e'
Proof:
  [exercise for the reader]
```

Happy proving!

Folding Data

Richard Möhn

2014-05-16

0. Outline

1. What are folds for? Make something out of some data structure. Which one? Let's take the list. We already know there is a fold. How to make it? Refer to [1].

2. What is a list?

```
List α ::= Nil
         | Cons α (List α)
```

3. So, the fold:

```
foldList : ... → List α → γ
```

4. A closer look at the list:

```
List α : * (Lattice: [Val] - [Type] - * ← only one)
Nil    : List α
Cons   : α → List α
```

Want to make γ out of List α. Have to make it out of every occurence of List α.

```
γ : *
e : γ
f : α → γ
```

Therefore:

```
foldList : (α → γ) → γ → List α → γ
```

Note: This is always the same procedure. Less hand-waving explanations derived from category theory.

5. Have the ingredients. How the fold? In the natural way!

```
foldList f e Nil          = e
foldList f e (Cons x xs) = f x (foldList f e xs)
```

Trace foldList f e (Cons x1 (Cons x2 (Cons x3 Nil))).

6. foldList applies f from right to left:

```
(  (  (  (    ))))
[0, 1, 2, 3] e
```

What about left-associative functions? foldLeft! (Two ways to define the fold on lists! Christian and Ratatouille.)

```
foldLeft e f Nil          = e
foldLeft e f (Cons x xs) = foldLeft (f e x) f xs
```

...applies f from left to right.

```
     e [0, 1, 2, 3]
((((    )  )  )  )
```

7. foldList in the editor. Defining and demonstrating functions:

   - length (by induction - program calculation!)

   - sum

8. Folds also on other data types: natural numbers!

```
Nat ::= Z
      | S Nat

Nat : *
Z   : Nat
S   : Nat → Nat

foldNat : ... Nat → γ

γ : *
e : γ
f : γ → γ
```

Therefore:

```
foldNat : (γ → γ) → γ → γ

foldNat f e Z     = e
foldNat f e (S n) = f (foldNat f e n)
```

Trace foldNat f e (S (S (S n))).

9. foldNat in the editor. Examples:

   - plus

   - fib

10. Compare the traces from (5) and (7). What we had last week in lambda calculus!

    - plus λ m n . m succ n

    - fib λ n . Π_1 (n f $\langle 0, 1 \rangle$) Even sliding window!

11. Can also define folds on non-recursive data structures like pairs. Later, dear.

12. Folds structurally recursive. Terminate if data structure is finite. Take always as many steps as the data structure. What about uncertain things like Collatz' conjecture? Evil direct recursion again?

    No! Unfolds! Make something into a data structure instead of data structure into something.

```
unfoldList : ... → γ → List α
```

13. How to define an unfold? Define an unconstructor first.

    (And defining Either first:

```
Either α β ::= Left α
             | Right β

uncoList Nil          = Left "Nothing"
uncoList (Cons x xs) = Right (x, List xs)

uncoList : List α → Either String (α * List α)
```

    Again, substitute List α with γ. (Can only grow the list piecewise from the original data item.)

```
f : γ → Either String (α * γ)
```

    And the unfold with it:

```
unfoldList : (γ → Either String (α * γ)) → γ → List α
```

14. Defining the unfold in the natural way:

```
unfoldList f s =
   case f s of
       Left _         -> []
       Right (a, s') -> a : (unfoldList f s')
```

15. unfoldList in the editor. Show:

    - collatz

16. Big example.

    - Things in types that are not uppercase are polymorphic.

17. Conclusion:

    - Here, folds for avoiding explicit recursion. Much more to it:

    - Dependently typed folds.

    - Can derive laws like fold fusion.

- Make algebraic manipulation of programs easier.

- Whole thing about program calculation. Drawing on category theory. Funny names and symbols.

- What I wanted to show: useful thing for everyday programming.

18. Not so many exercises anymore:

    - (application) Define the multiplication and factorial function as folds.
    - (application) Define the map function on lists as a fold.
    - (application) Define the reverse function on lists as a fold.
    - (application) Implement foldNat in Scheme and use it for writing a function for exponentiating matrices. (The utility functions required for this shouldn't use explicit recursion as well, of course.)
    - (transfer, frolic) Derive the fold and the unfold on your favourite kind of tree and play around with them.
    - (transfer) Derive the folds on Pair ((a * b)) and Either. Which well-known functions do you obtain? [1]
    - (making connections) While looking at the fold on Pair with one eye, look at what Mikkel said about pairs in the lambda calculus talk with the other.
    - (♡) Make unparseArith output nicely indented code.
    - (♡) Can the compiler from Arith to list of BCIs written as an unfold? If yes, do it. If no, prove it.
    - (program calculation) Define a tail-recursive append as a fold. (Hint: Use foldList f e = append as a specification and calculate your way to the result without using the second parameter of append.) [1, SPOILER!]
    - (♡) Solve the challenges involving folds!

19. References:

    1 Cezar Ionescu: Advanced Functional Programming. Freie Universitt Berlin, 2013. (http://www.pik-potsdam.de/members/ionescu/advanced-functional-programming, 2014-05-12)

2 Olivier Danvy: dProgSprog 2014, Lecture Notes for Week 18. (http://users-cs.au.dk/danvy/dProgSprog/Lecture-notes/week-18.html, 2014-05-12)

3 Erik Meijer, Maarten Fokkinga, Ross Paterson: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: Proceedings of the 5th ACM conference on Functional programming languages and computer architecture, Springer-Verlag New York, 1991, Pages 124-144

4 Richard Bird, Oege de Moor: Algebra of Programming. Prentice Hall Europe, 1996

Semantics Overview

Christian Clausen
christia@cs.au.dk

May 23, 2014

Semantics are used for several different purposes, the most common are: to give meaning to (or prove properties about) programs, and to prove properties about languages. The definition of a semantics is: a set of constructions, and a set of inference rules on these constructions. Coincidentally, this is also the definition of a logic, thus a semantics is also a logic, and vice versa.

I have chosen to keep the defined languages as similar as possible, so that the semantics will be easier to compare. By the same argument, I have generally chosen very small (but Turing complete[1]) languages.

Thanks to Kristoffer Andersen, for comments and ideas for this presentation.

Disclaimer There is so much material about semantics out there, and not all of it agrees on the names and categories, this is my view. Many of them are also very closely related, and some even express the exact same language.

[1] for the dynamic part anyway

1 Operational Semantics aka Dynamic aka Execution

1.1 Big-step aka Natural

Grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$

Semantics:

$$\frac{}{x \Downarrow x} \qquad \frac{t \Downarrow t'}{\lambda x.\ t \Downarrow \lambda x.\ t'}$$

$$\frac{t_1 \Downarrow \lambda x.\ t_1' \quad t_2 \Downarrow t_2' \quad t_1'[t_2/x] \Downarrow t'}{t_1\ t_2 \Downarrow t'} \qquad \frac{t_1 \Downarrow t_1' \quad t_2 \Downarrow t_2'}{t_1\ t_2 \Downarrow t_1'\ t_2'}$$

Normalizer:

$$\texttt{eval}\ t$$

1.2 Denotational aka Mathematical

Grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$

Semantics:

$$\begin{aligned} [\![x]\!]\rho &\triangleq \rho(x) \\ [\![\lambda x.\ t]\!]\rho &\triangleq \texttt{VLam (fun x -> } [\![t]\!]\rho[x \mapsto \texttt{x}]) \\ [\![t_1\ t_2]\!]\rho &\triangleq f\ ([\![t_2]\!]\rho) \\ &\quad \text{if } [\![t_1]\!]\rho = \texttt{VLam}\ f \end{aligned}$$

Normalizer:

$$\texttt{lift}\ t\ \texttt{[]}$$

1.3 Reduction Semantics

1.3.1 Small-step aka Structural

Grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$
$$v ::= x \mid \lambda x.\ v$$

Semantics:

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

$$\frac{t \to t'}{\lambda x.\ t \to \lambda x.\ t'} \qquad \frac{}{(\lambda x.\ v_1)\ v_2 \to v_1[v_2/x]}$$

Normalizer:

$$\texttt{fix step}\ t$$

1.3.2 Evaluation Context

Grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$
$$v ::= x \mid \lambda x.\ v$$
$$E ::= \bullet \mid E\ t \mid v\ E \mid \lambda x.\ E$$

Semantics:

$$E[\![(\lambda x.\ v_1)\ v_2]\!] \triangleq v_1[v_2/x]$$

Normalizer:

$$\texttt{repeat (recompose} \circ \texttt{contract} \circ \texttt{decompose)}$$

1.3.3 Rewriting System

Grammar:

$$t ::= S \mid K \mid t\ t$$

Semantics:

$$S\ x\ y\ z = x\ z\ (y\ z)$$
$$K\ x\ y = x$$

Normalizer:

```
repeat (rewrite S_eq || rewrite K_eq)
```

1.3.4 Transition System aka Abstract Machines

CEK, by Felleisen et al.

Grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$
$$v ::= [x, t, e]$$
$$k ::= \texttt{stop} \mid \texttt{fun}(v,\ k) \mid \texttt{arg}(t,\ e,\ k)$$

Semantics:

$$t \to \langle t, \cdot, \texttt{stop} \rangle$$
$$\langle x, e, k \rangle \to_e \langle k, e(x) \rangle$$
$$\langle \lambda x.\ t, e, k \rangle \to_e \langle k, [x, t, e] \rangle$$
$$\langle t_1\ t_2, e, k \rangle \to_e \langle t_1, e, \texttt{arg}(t_2,\ e,\ k) \rangle$$
$$\langle \texttt{arg}(t,\ e,\ k), v \rangle \to_c \langle t, e, \texttt{fun}(v,\ k) \rangle$$
$$\langle \texttt{fun}([x, t, e],\ k), v \rangle \to_c \langle t, e[x \mapsto v], k \rangle$$
$$\langle \texttt{stop}, v \rangle \to v$$

Normalizer: it is normalizing.

2 Static

2.1 Relational

2.1.1 Simple Types

Grammar:

$$t ::= x \mid \lambda x : \tau.\ t \mid t\ t$$
$$\tau ::= A \mid \tau \to \tau$$

Semantics:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \uparrow \tau} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ t \uparrow \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 \uparrow \tau_2}$$

2.1.2 Linear Types

Grammar:

$$t ::= x \mid \lambda x : \tau.\ t \mid t\ t$$
$$\tau ::= A \mid \tau \multimap \tau$$

Semantics:

$$\frac{}{\{(x, \tau)\} \vdash x \uparrow \tau} \qquad \frac{\Gamma \cup \{(x, \tau_1)\} \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ t \uparrow \tau_1 \multimap \tau_2}$$

$$\frac{\Gamma_1 \vdash t_1 : \tau_1 \multimap \tau_2 \quad \Gamma_2 \vdash t_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash t_1\ t_2 \uparrow \tau_2}$$

Note: $\uplus$ denotes the disjoint union of environments.

2.2 Axiomatic

Q language, by E. Ernst.

Grammar:

$$s ::= \texttt{skip} \mid x\ \texttt{:=}\ e \mid s\texttt{;}\ s \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ e\ \texttt{do}\ s$$

Semantics:

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \qquad \frac{}{\{P[e/x]\}\ x\ \texttt{:=}\ e\ \{P\}}$$

$$\frac{\{P\}\ s_1\ \{R\} \quad \{R\}\ s_2\ \{Q\}}{\{P\}\ s_1\texttt{;}\ s_2\ \{Q\}} \qquad \frac{\{P \wedge e\}\ s_1\ \{Q\} \quad \{P \wedge \neg e\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}}$$

$$\frac{\{P \wedge e\}\ s\ \{P\}}{\{P\}\ \texttt{while}\ e\ \texttt{do}\ s\ \{P \wedge \neg e\}} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ s\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

$$\frac{\{P\}\ s\ \{Q_1\} \quad \{P\}\ s\ \{Q_2\}}{\{P\}\ s\ \{Q_1 \wedge Q_2\}} \qquad \frac{\{P_1\}\ s\ \{Q\} \quad \{P_2\}\ s\ \{Q\}}{\{P_1 \vee P_2\}\ s\ \{Q\}}$$

A Substitution

Here I show a couple of different approaches to implement substitution ($t'[t/x]$).

A.1 Capture Avoiding Substitution

$$
\begin{aligned}
x[t/x] &\triangleq t \\
y[t/x] &\triangleq y \\
(t_1\ t_2)[t/x] &\triangleq t_1[t/x]\ t_2[t/x] \\
(\lambda x.\ t')[t/x] &\triangleq \lambda x.\ t' \\
(\lambda y.\ t')[t/x] &\triangleq \lambda y.\ t'[t/x] \\
&\qquad \text{if } y \text{ is not free in } t
\end{aligned}
$$

A.2 De Bruijn

The idea behind De Bruijn indecies is to store the reference to the lambda that bound the variable instead of the variable. Thus, $\lambda x.\ x$ would be $\lambda 0$, and $\lambda f.\ \lambda g.\ \lambda x.\ f\ x\ (g\ x)$ becomes $\lambda\lambda\lambda 2\ 0\ (1\ 0)$. We can thus define an algorithm to go from the usual representation to De Bruijn representation:

$$
\begin{aligned}
[\![x]\!]_\rho &\triangleq \text{indexOf}(x, \rho) \\
[\![\lambda x.\ t]\!]_\rho &\triangleq \lambda [\![t]\!]_{x::\rho} \\
[\![t_1\ t_2]\!]_\rho &\triangleq [\![t_1]\!]_\rho\ [\![t_2]\!]_\rho
\end{aligned}
$$

Now substitution $((\lambda M)\ N)$ happens in three steps, for all occurences:

1. Find a variable that refer to the λ, call it x.
2. Decrement the free variables in M.
3. Add x to all free variables in N.

B Environment Styles

B.1 Ordered Structure

Notation examples:

$$\frac{}{\Gamma_1, x : \tau, \Gamma_2 \vdash x \uparrow \tau} \quad \frac{\Gamma, x : \tau_1 \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ t \uparrow \tau_1 \to \tau_2}$$

Implementation:

```
let empty = []
let put x tau Gamma = (x, tau) :: Gamma
let rec get x Gamma =
  match Gamma with
    | [] -> raise Not_found
    | (k, v) :: Gamma' ->
      if k = x
      then v
      else get x Gamma'
```

B.2 Set

Notation examples:

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x \uparrow \tau} \quad \frac{\Gamma \cup \{(x, \tau_1)\} \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ t \uparrow \tau_1 \to \tau_2}$$

In practice, this would be like the ordered structure.

B.3 Function

Notation examples:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma[x \mapsto \tau_1] \vdash t \uparrow \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ t \uparrow \tau_1 \to \tau_2}$$

Implementation:

```
let empty = fun x' -> raise Not_found
let put x tau Gamma = fun x' ->
```

```
  if x = x'
  then tau
  else Gamma x'
let get x Gamma = Gamma x
```

Assembling Assembly

Christian Clausen
christia@cs.au.dk

May 30, 2014

General:

$$\begin{aligned} \mathit{prog} &::= \mathit{instr}^* \\ \mathit{loc} &\in \{n \in \mathbb{N} \mid n < \text{len}(\mathit{prog})\} \end{aligned}$$

1 3-Counter Machine

1.1 Formal Semantics

$$\begin{aligned} \mathit{var} &\in \{\texttt{ax}, \texttt{bx}, \texttt{cx}\} \\ \mathit{instr} &::= \texttt{inc } \mathit{var} \\ &\quad \mid \texttt{dec } \mathit{var} \\ &\quad \mid \texttt{zero } \mathit{var} \texttt{ then } \mathit{loc} \texttt{ else } \mathit{loc} \\ \mathit{state} &::= \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathit{loc} \end{aligned}$$

$$\frac{P_{pc} = \texttt{inc ax}}{\langle ax, bx, cx, pc\rangle \to \langle ax+1, bx, cx, pc+1\rangle}$$

$$\frac{P_{pc} = \texttt{dec ax}}{\langle ax, bx, cx, pc\rangle \to \langle ax-1, bx, cx, pc+1\rangle}$$

$$\frac{P_{pc} = \texttt{zero ax then } m \texttt{ else } n \quad ax = 0}{\langle ax, bx, cx, pc\rangle \to \langle ax, bx, cx, m\rangle}$$

$$\frac{P_{pc} = \texttt{zero ax then } m \texttt{ else } n \quad ax \neq 0}{\langle ax, bx, cx, pc\rangle \to \langle ax, bx, cx, n\rangle}$$

2 IJVM

2.1 Formal Semantics

$$
\begin{aligned}
\mathit{method} &::= \texttt{.method}\ \mathit{method}\ \mathit{directive}^*\ \mathit{instr}^+ \\
\mathit{directive} &::= \texttt{.args}\ \mathbb{N} \\
&\quad \mid \texttt{.locals}\ \mathbb{N} \\
&\quad \mid \texttt{.define}\ \mathit{var}\ \texttt{=}\ \mathbb{N} \\
\mathit{instr} &::= \texttt{bipush}\ \mathbb{Z} \mid \texttt{pop} \\
&\quad \mid \texttt{dup} \mid \texttt{swap} \\
&\quad \mid \texttt{iadd} \mid \texttt{isub} \\
&\quad \mid \texttt{iand} \mid \texttt{ior} \\
&\quad \mid \texttt{goto}\ \mathit{loc} \\
&\quad \mid \texttt{ifeq}\ \mathit{loc} \mid \texttt{iflt}\ \mathit{loc} \\
&\quad \mid \texttt{if\_icmpeq}\ \mathit{loc} \\
&\quad \mid \texttt{iinc}\ \mathit{var}\texttt{,}\ \mathbb{Z} \\
&\quad \mid \texttt{istore}\ \mathit{var} \mid \texttt{iload}\ \mathit{var} \\
&\quad \mid \texttt{invokevirtual}\ \mathit{method} \mid \texttt{ireturn} \\
&\quad \mid \texttt{ldc\_w}\ \mathit{expr} \\
&\quad \mid \texttt{nop} \\
\mathit{state} &::= (\mathit{loc} \times (\mathbb{N} \to \mathbb{Z}) \times \mathbb{Z}^*)^*
\end{aligned}
$$

$$\frac{P_{pc} = \texttt{nop}}{(pc, l, s) :: sf \to (pc + 1, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{bipush}\ \ c}{(pc, l, s) :: sf \to (pc + 1, l, c :: s) :: sf}$$

$$\frac{P_{pc} = \texttt{pop}}{(pc, l, v :: s) :: sf \to (pc + 1, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{dup}}{(pc, l, v :: s) :: sf \to (pc + 1, l, v :: v :: s) :: sf}$$

$$\frac{P_{pc} = \texttt{swap}}{(pc, l, a :: b :: s) :: sf \to (pc + 1, l, b :: a :: s) :: sf}$$

$$\frac{P_{pc} \in \{\texttt{iadd}, \texttt{isub}, \texttt{iand}, \texttt{ior}\}}{(pc, l, a :: b :: s) :: sf \to (pc + 1, l, [\![P_{pc}]\!](a, b) :: s) :: sf}$$

$$\frac{P_{pc} = \texttt{goto}\ \ loc}{(pc, l, s) :: sf \to (loc, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{ifeq}\ \ loc \quad n = 0}{(pc, l, n :: s) :: sf \to (loc, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{ifeq}\ \ loc \quad n \neq 0}{(pc, l, n :: s) :: sf \to (pc + 1, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{iflt}\ \ loc \quad n < 0}{(pc, l, n :: s) :: sf \to (loc, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{iflt}\ \ loc \quad n \not< 0}{(pc, l, n :: s) :: sf \to (pc + 1, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{istore}\ \ var}{(pc, l, n :: s) :: sf \to (pc + 1, l[var \mapsto n], s) :: sf}$$

$$\frac{P_{pc} = \texttt{iload}\ \ var}{(pc, l, s) :: sf \to (pc + 1, l, l(var) :: s) :: sf}$$

$$\frac{P_{pc} = \texttt{invokevirtual}\ \ method \quad locals = instanciate(method, \vec{v})}{(pc, l, \vec{v} :: s) :: sf \to (method_{loc}, locals, \cdot) :: (pc, l, s) :: sf}$$

$$\frac{P_{pc} = \texttt{ireturn}}{(pc, l, v :: s) :: (pc', l', s') :: sf \to (pc' + 1, l', v :: s') :: sf}$$

3 Our Language

3.1 Formal Semantics

$$
\begin{aligned}
var &\in \{\texttt{ax}, \texttt{bx}, \texttt{cx}, \texttt{pc}\} \\
instr &::= \texttt{inc } var \\
&\mid \texttt{dec } var \\
&\mid \texttt{zero } var\texttt{, } loc\texttt{, } loc \\
&\mid \texttt{push } var|loc|\mathbb{Z} \\
&\mid \texttt{pop } var \\
state &::= \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times loc \times \mathbb{Z}^*
\end{aligned}
$$

$$
\frac{P_{pc} = \texttt{inc ax}}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax+1, bx, cx, pc+1, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{dec ax}}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax-1, bx, cx, pc+1, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{zero ax then } loc_1 \texttt{ else } loc_2 \quad ax = 0}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax, bx, cx, loc_1, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{zero ax then } loc_1 \texttt{ else } loc_2 \quad ax \neq 0}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax, bx, cx, loc_2, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{push ax}}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax, bx, cx, pc, ax :: stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{push } m}{\langle ax, bx, cx, pc, stack\rangle \to \langle ax, bx, cx, pc, m :: stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{pop ax}}{\langle ax, bx, cx, pc, a :: stack\rangle \to \langle a, bx, cx, pc, stack\rangle}
$$

3.2 Comparison with IJVM

See Figures 1 through 6. Notice: We can save two of our three registers when calling a method as is clear from Figure 6.

```
.method method
.args 3
.define a = 1
.define b = 2
.locals 1
.define c = 3

iload a
iload b
```

Figure 1: IJVM

```
:method
pop cx
pop ax
pop bx
inc cx
inc cx
push cx
```

Figure 2: BotCode

```
ireturn
```

Figure 3: IJVM

```
pop cx
push ax
push cx
pop pc
```

Figure 4: BotCode

```
// push arguments
invokevirtual method
```

Figure 5: IJVM

```
push ax
push bx
# push arguments
push pc
zero ax, method, method
pop cx
pop bx
pop ax
push cx
```

Figure 6: BotCode

3.3 How to use it

We now have two types of functions:

- One for stack calculations, ex.:

$$call :: a :: b :: s \rightarrow a \cdot b :: s$$

- One for register calculations, ex.:

$$\langle ax, bx, cx \rangle \rightarrow \langle bx, ax \cdot bx, ? \rangle$$

```
# MUL REC BEGIN
# call :: a :: b :: s -> a * b :: s
:mul_rec
pop cx
pop ax
pop bx
inc cx
inc cx
push cx
zero ax, mul_rec_done, mul_rec_loop
:mul_rec_loop
push bx
push bx
dec ax
push ax
push pc
zero ax, mul_rec, mul_rec
push pc
zero ax, add, add
pop ax
pop cx
push ax
push cx
pop pc
:mul_rec_done
pop cx
push 0
```

```
push cx
pop pc
# MUL REC END

# MUL REG BEGIN
# <a, b, ?> -> <b, a * b, ?>
:mul_reg
push ax
push bx
push 0
pop bx
pop ax
pop cx
zero cx, mul_reg_done, mul_reg_loop
:mul_reg_loop
push ax
push pc
zero ax, add_reg, add_reg
pop ax
dec cx
zero cx, mul_reg_done, mul_reg_loop
:mul_reg_done
pop cx
inc cx
inc cx
push cx
pop pc
# MUL REG END

# MUL CONST BEGIN
# call :: a :: b :: s -> a * b :: s
:mul
pop cx
pop ax
pop bx
inc cx
inc cx
push cx
```

```
push pc
zero ax, mul_reg, mul_reg
pop ax
push cx
push ax
pop pc
# MUL CONST BEGIN

# MUL ITER BEGIN
# call :: a :: b :: s -> a * b :: s
:mul_iter
pop cx
inc cx
inc cx
pop ax
pop bx
push cx     # ret :: s
push 0      # 0 :: ret :: s
zero ax, mul_zero, mul_gen
:mul_zero
push pc
zero ax, swap, swap
pop pc
:mul_gen
push mul_loop
push bx
dec ax
zero ax, mul_zero, mul_gen
:mul_loop           # b :: [b :: mul_loop] ^ a :: 0 :: ret :: s
push pc
zero ax, add, add   # b + b :: mul_loop :: [b :: mul_loop] ^ (a - 1)
push pc
zero ax, swap, swap # mul_loop :: b + b :: [b :: mul_loop] ^ (a - 1)
pop pc
# MUL ITER END
```

4 Peephole Optimizations

See Figures 7 through 8.

```
zero var, loc1, loc2
...
:loc1
zero var, loc3, loc4
```

Figure 7: Unoptimized

```
zero var, loc3, loc2
...
:loc1
zero var, loc3, loc4
```

Figure 8: Optimized

5 Exercises

1. Write a `swap` function:

$$call :: a :: b :: s \to b :: a :: s$$

2. Write a `dup` function:

$$call :: n :: s \to n :: n :: s$$

3. Write a non-recursive `add` function:

$$call :: a :: b :: s \to a + b :: s$$

4. Write a register `add` function:

$$\langle ax, bx, cx \rangle \to \langle ?, ax + bx, cx \rangle$$

5. Write `and` and `or` functions:

$$call :: b_1 :: b_2 :: s \to b_1 \wedge b_2 :: s$$

$$call :: b_1 :: b_2 :: s \to b_1 \vee b_2 :: s$$

6. If we chose `true` to be 0 and `false` to be anything else, then prove that `and = add`. Is there a function for `or`? What happens if we set `true = 1, false = 0`, do the functions exist?

7. Write an `is_neg` function:

$$call :: n :: s \rightarrow is\_neg_n :: s$$

8. Write a recursive fibonacci function:

$$call :: n :: s \rightarrow fib_n :: s$$

9. Write a non-recursive fibonacci function:

$$call :: n :: s \rightarrow fib_n :: s$$

10. Which of the two fibonacci functions is fastest? Why?

11. Write a recursive factorial function:

$$call :: n :: s \rightarrow n! :: s$$

12. Write a non-recursive factorial function:

$$call :: n :: s \rightarrow n! :: s$$

13. Write a non-recursive factorial function using the loop technique:

$$call :: n :: s \rightarrow n! :: s$$

14. Which of the three factorial functions is fastest? Why?

Bits and Pieces

Christian Clausen
christia@cs.au.dk

June 6, 2014

We assume that all variables refer to N-bit integers. But for the first three sections, we are going to use $N = 3$.

Precedence: $x + x \gg x \oplus x \,\&\, x \mid x$

1 Turning [last/tail] [on/off]

x	$x+1$	$x-1$	tail off $x \,\&\, x+1$	last off $x \,\&\, x-1$	last on $x \mid x+1$	tail on $x \mid x-1$
000	001	111	000	000	001	111
001	010	000	000	000	011	001
010	011	001	010	000	011	011
011	100	010	000	010	111	011
100	101	011	100	000	101	111
101	110	100	100	100	111	101
110	111	101	110	100	111	111
111	000	110	000	110	111	111

2 Some unary operators

x	$-x$	$\neg x$	$x^<$	$x^>$	$x^{\neq}$	$x^{\gg}$
000(0)	000	111	000	000	000	000
001(1)	111	110	000	001	001	000
010(2)	110	101	000	001	001	000
011(3)	101	100	000	001	001	000
100(-4)	100	011	001	001	001	111
101(-3)	011	010	001	000	001	111
110(-2)	010	001	001	000	001	111
111(-1)	001	000	001	000	001	111

$x^<$ and $x^>$ can be implemented as $x \ggg (N-1)$ and $-x \ggg (N-1)$ respectively. $x^{\neq}$ can be implemented as $x^< \mid x^>$. The operator $x^{\gg}$ can be calculated by: $x \gg (N-1)$; but if we don't have $\gg$, we can also define it as: $-(x^<)$.

3 Dividing with 2

Math	$\lfloor x/2 \rfloor$	$x \div 2$	$\lceil x/2 \rceil$
Bits	$x + 0 \gg 1$	$x + x^< \gg 1$	$x + 1 \gg 1$
000(0)	000	000	000
001(1)	000	000	001
010(2)	001	001	001
011(3)	001	001	110
100(-4)	110	110	110
101(-3)	110	111	111
110(-2)	111	111	111
111(-1)	111	000	000

Notice the overflow in $3 + 1 \gg 1$, we will return to this later.

4 General

$$
\begin{aligned}
x + y \ll c &= (x \ll c) + (y \ll c) \\
x + y \gg c &= (x \gg c) + (y \gg c) && \text{No overflow} \\
x + y &= ((x \,\&\, y) \ll 1) + (x \oplus y) \\
x + y &= x \oplus y && x = 0 \vee y = 0
\end{aligned}
$$

5 Average

$$
\begin{aligned}
\lfloor (x + y)/2 \rfloor &= x + y \ggg 1 && x, y \geq 0 \\
\lfloor (x + y)/2 \rfloor &= x + y \gg 1 && x, y \text{ different signs}
\end{aligned}
$$

5.1 Unsigned

$$
\begin{aligned}
(x + y) \div 2 &= \lfloor (x + y)/2 \rfloor \\
&= ((x \,\&\, y) \ll 1) + (x \oplus y) \gg 1 \\
&= (((x \,\&\, y) \ll 1) \gg 1) + ((x \oplus y) \gg 1) && \text{No overflow} \\
&= (x \,\&\, y) + ((x \oplus y) \gg 1)
\end{aligned}
$$

Note that while this result works for any x, y signed or unsigned. If we are in a signed environment, the first equal doesn't apply.

Remember the overflow from section 3, which we got from adding one and dividing with two? We can see that as the average of 1 and x, thus we can use this formula.

6 Comparing

6.1 With 0

$$
\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}
$$

$$
\text{sign}(x) = x^{\gg} \mid x^{>}
$$

6.2 With y

$$\mathrm{cmp}(x,y) = \begin{cases} -1 & \text{if } x < y \\ 0 & \text{if } x = y \\ 1 & \text{if } x > y \end{cases}$$

$$\begin{aligned} \mathrm{cmp}(x,y) &= \mathrm{sign}(x-y) && \text{No overflow} \\ &= (x \geq y) - (x \leq y) && \text{(Type error)} \\ &= (x > y) - (x < y) && \text{(Type error)} \\ &= (y < x) - (x < y) && \text{(Type error)} \end{aligned}$$

6.3 Less than

$$\begin{aligned} x < y &\Leftrightarrow y - x > 0 && \text{No overflow} \\ &\Leftrightarrow (y \gg 1) - (x \gg 1) > 0 && \text{Not too close} \\ &\Leftrightarrow (y \gg 1) - (x \gg 1) + (\neg x \,\&\, y \,\&\, 1) > 0 \\ &= ((y \gg 1) - (x \gg 1) + (\neg x \,\&\, y \,\&\, 1))^{>} \end{aligned}$$

7 Max and Min

$$\mathrm{doz}(x,y) = \begin{cases} x - y & x > y \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \max(x,y) &= \begin{cases} x & x > y \\ y & \text{otherwise} \end{cases} \\ &= \begin{cases} y + (x-y) & x > y \\ y + 0 & \text{otherwise} \end{cases} \\ &= y + \mathrm{doz}(x,y) \end{aligned}$$

$$\min(x,y) = x - \mathrm{doz}(x,y)$$

$$\mathrm{doz}(x,y) = (x-y) \,\&\, -(y < x)$$

8 Absolute Values

$$\begin{aligned}\mathrm{abs}(x-y) &= \mathrm{doz}(x,y) + \mathrm{doz}(y,x)\\ \mathrm{abs}(x) &= (x \oplus x^{\gg}) - x^{\gg}\end{aligned}$$

9 Exchanging Values

$$\begin{aligned}x \leftarrow x \oplus y; y \leftarrow x \oplus y; x \leftarrow x \oplus y;\\ t \leftarrow x \oplus y; ...; x \leftarrow x \oplus t; y \leftarrow y \oplus t;\end{aligned}$$

$$\begin{aligned}next(x,a,b) &= \begin{cases} a & x = b \\ b & x = a \end{cases}\\ &= x \oplus (a \oplus b)\\ &= a \,\&\, -((x \oplus a)^{\neq})\\ &\mid b \,\&\, -((x \oplus b)^{\neq})\\ next(x,a,b,c) &= \begin{cases} a & x = c \\ b & x = a \\ c & x = b \end{cases}\\ &= a \,\&\, -((x \oplus a)^{\neq}) \,\&\, -((x \oplus b)^{\neq})\\ &\mid b \,\&\, -((x \oplus b)^{\neq}) \,\&\, -((x \oplus c)^{\neq})\\ &\mid c \,\&\, -((x \oplus c)^{\neq}) \,\&\, -((x \oplus a)^{\neq})\end{aligned}$$

10 Exponentiation

$$\begin{aligned} x^{11} &= x^{1011_2} \\ &= x \cdot x^{1010_2} \\ &= x \cdot (x^{101_2})^2 \\ &= x \cdot (x \cdot x^{100_2})^2 \\ &= x \cdot (x \cdot (x^{10_2})^2)^2 \\ &= x \cdot (x \cdot ((x^{1_2})^2)^2)^2 \\ &= x \cdot (x \cdot ((x \cdot x^{0_2})^2)^2)^2 \\ &= x \cdot (x \cdot ((x \cdot 1)^2)^2)^2 \end{aligned}$$

```
exp(b, unsigned e):
  r = 1;
  while(true){
    if(e & 1 == 1) r *= b;
    e >>= 1;
    if(e == 0) return r;
    b *= b;
  }

powerOf2(unsigned e):
  return 1 << e;
```

11 Logarithms

```
pop(unsigned b):
  b = (b & 0x55555555) + (b >>>  1 & 0x55555555);
  b = (b & 0x33333333) + (b >>>  2 & 0x33333333);
  b = (b & 0x0F0F0F0F) + (b >>>  4 & 0x0F0F0F0F);
  b = (b & 0x00FF00FF) + (b >>>  8 & 0x00FF00FF);
  b = (b & 0x0000FFFF) + (b >>> 16 & 0x0000FFFF);
  return b;

nlz(unsigned b):
  b |= b >>> 1;
  b |= b >>> 2;
```

```
  b |= b >>> 4;
  b |= b >>> 8;
  b |= b >>> 16;
  return pop(~b);

log2(unsigned b):
  return 31 - nlz(b);
log2(unsigned b):
  b |= b >>> 1;
  b |= b >>> 2;
  b |= b >>> 4;
  b |= b >>> 8;
  b |= b >>> 16;
  return pop(b) - 1;
```

12 Exercises

1. Find functions (expressions possibly using x and y) for each colomn:

x	y																
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

2. In Spare Time Teaching, we have a tradition of talking about De Morgan, and let's not make an exception of it here. We all know the usual De Morgan equations; but what about the rest:

$$\begin{aligned}
\neg(a \mid b) &= \neg a \,\&\, \neg b \\
\neg(a \,\&\, b) &= \neg a \mid \neg b \\
\neg(a \oplus b) &= ? \\
\neg(a + b) &= ? \\
\neg(a - b) &= ? \\
\neg(-a) &= ?
\end{aligned}$$

3. Assuming that you have a primitive instruction $x^{=} = \begin{cases} 1 & x = 0 \\ 0 & \text{otherwise} \end{cases}$.

Make an expression to determine if an unsigned integer is a power of two (2^n) or zero, using primitive 3 instructions.

4. How could you make $x^=$ if you didn't have it?

5. There are 3 kinds of bit shifts: Logical ($\ll$, $\lll$), arithmetic ($\gg$), and rotating ($\overset{r}{\ll}$, $\overset{r}{\gg}$). Here is an example: $b_4b_3b_2b_1b_0 \overset{r}{\gg} 2 = b_1b_0b_4b_3b_2$. Come up with expressions that are equivalent to $\overset{r}{\ll}$, and $\overset{r}{\gg}$.

6. Come up with an unsigned version of less than ($\overset{u}{<}$).

7. Explain when $\overset{u}{<}$ would be useful?

8. If our machine has `doz` as a primitive, then we can code $<$ in four instructions (counting duplicated subexpressions once), how?

9. One could argue that using zero as false and one as true is aesthetically pleasing, but then another might argue that using the bitstring of only ones for true and the bitstring of zeroes is more useful, or faster. Come up with expressions to convert both ways. How many instructions are necessary?

10. What would happen if we called `powerOf2` with 31? What about a number greater than 31? What would happen if we called `exp` with 2 and something greater than 31?

11. What would happen if we accidentally used $\gg$ instead of $\lll$ in the `pop` function? What about in `nlz`?

12. Our version of `pop` uses 25 (including assignment) instructions, can you reduce it without using branches? Hints:

    - Are all the masks necessary?
    - ♡ There is a solution using 20.

Introduction to Type Classes (in Haskell)

Richard Möhn

2014-06-13

0. Outline

1,2	Intro
3-8	Eq
9-b	Ord etc.
c-11	ADTs
12-15	Package
16-18	Multi
1a,1b	F & M
1c	Schluss
20,21	Bonus

1. Type classes central to Haskell. Means: not just cool feature, but part of how it works.

2. Note: not using idiomatic Haskell for didactic reasons. (Mental overload and so on.) cf. Danvy and Scheme.

3. Have the member function:

```
member e l = foldl f False l
    where
    f a x = (e == x) || a
```

   Works for integers, floats, characters, etc., therefore polymorphic. When we manually provide the type, type-checking fails:

```
member :: a -> [a] -> Bool
```

Following advice, we add something and it works:

```
member :: Eq a => a -> [a] -> Bool
```

Also the type inferred by Haskell.

4. Eq a type constraint. - Constrains the polymorphism of a.

5. What about non-built-in types? member doesn't work for Nat. Define universal member:

```
lmember eq e l = foldl f False l
    where
    f a x = (eq e x) || a
```

Works on Nat with

```
eq = eqNat (see code, derivation/proof: exercise)
```

6. Clumsy. We can of course use Haskell's built-in mechanism. But we could do it like that or implement it with a preprocessor on top of Hindley-Milner [1]. What is Haskell's built-in mechanism? - Type constraints are requirement that the actual type put in place of the variable be member of a type class.

```
member :: Eq a => a -> [a] -> Bool
```

- What is put in for `a` has to be a member of type class `Eq`.

7. What is a type class? Declared like this:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

Function signatures and default implementations.

8. How do I make a type member of a type class? - Implement some functions:

```
instance Eq Nat where
    (==) = eqNat
```

(/=) covered by default implementation.

9. What if we want to sort?

```
sort l = foldr insert [] l

insert x sl = paraList f [x] sl
    where
    f y (ys, lwx) = if x > y then
                        y:lwx
                    else
                        x:y:ys
```

Works for numbers and other stuff, but not for Nat. - Type:

```
sort :: Ord a => [a] -> [a]
```

Because of ¿, what is put in for `a` has to be member of type class `Ord`.

a. Implementing Ord:

```
class (Eq a) => Ord a where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT
```

```
x <= y = compare x y /= GT
x <  y = compare x y == LT
x >= y = compare x y /= LT
x >  y = compare x y == GT

-- Note that (min x y, max x y) = (x,y) or (y,x)
max x y | x <= y    = y
        | otherwise = x
min x y | x <= y    = x
        | otherwise = y
```

The default declarations allow a user to create an `Ord` instance either with a type-specific compare function or with type-specific `==` and `<=` functions [2].

Note the constraint `Eq a`. - Sufficient to implement (`<=`). See code.

b. Many other built-in type classes, eg. `Num`, `Integral`, `Enum` with hierarchy and built-in types implementing them. See [2].

c. Might use type classes to specify abstract data types. Note: this is not a good example. You shouldn't use type classes for specifying abstract data types.

```
class Queue q where
    empty :: q a
    push  :: a -> q a -> q a
    peek  :: q a -> a
    pop   :: q a -> q a
```

and implement them like this:

```
instance Queue [] where
    empty          = []
    push x l       = l
    peek (x:x':xs) = x'
    pop (x:xs)     = xs ++ [x]
```

This type-checks, but doesn't make sense.

10. Type classes usually accompanied by set of laws the implementations must fulfill, like:

```
(ia)  pop  (push x empty) = empty
(iia) peek (push x empty) = x
```

    For LIFO:

```
(ib)  pop  (push x q) = q
(iib) peek (push x q) = x
```

    For FIFO:

```
(ic)  peek (push x_n ... (push x_1 (push x_0 empty))...) = x_0
(iic) pop  (push x_n ... (push x_1 (push x_0 empty))...)
         = (push x_n ... (push x_1 empty)...)
```

```
(ib)  and (ic)  imply (ia).
(iib) and (iic) imply (iia).
```

    Above implementation violates these laws.

11. Better implementation (LIFO):

```
instance Queue [] where
    empty       = []
    push x l    = x:l
    peek (x:xs) = x
    pop (x:xs)  = xs
```

    Proving (ib):

```
  pop (push x l)
= pop (push x l)
= pop (x:l)
= l
```

Proving (iib):

```
  peek (push x l)
= peek (x:l)
= x
```

12. We can have more fancy type classes:

```
class Package p where
    cmap :: (a -> b) -> p a -> p b
```

and implement them like this:

```
instance Package [] where
    cmap f []     = []
    cmap f (x:xs) = cmap f xs
```

It type-checks, but doesn't make sense.

13. Therefore add laws:

```
(i)  cmap id q = id q
(ii) cmap (f . g) q = cmap f (cmap g q)
```

Above implementation violates these laws.

14. Better implementation:

```
instance Package [] where
    cmap f l = map f l
```

We prove (i) by induction on l:

- base case:

```
cmap id [] = map id [] = [] = id []
```

- induction hypothesis:

```
cmap id xs = map id xs = id xs
```

- induction case:

```
cmap id (x:xs) =! id (x:xs)

LHS = map id (x:xs)
    = id x : (map id xs)
    = id x : (cmap id xs)
  IH= id x : (id xs)
    = x:xs
    = id (x:xs)
```

Proof for (ii) is left as an exercise.

15. Fancy implementation:

```
instance Package ((->) t) where
    cmap fab fta = \ vt -> fab (fta vt)
```

Proving (i):

```
cmap id fta = \ vt -> id (fta vt)
            = \ vt -> fta vt
            = fta
```

Proving (ii):

```
LHS = cmap (f . g) fta
    = \ vt -> (f . g) (fta vt)
    = \ vt -> f (g (fta vt))

RHS = cmap f (cmap g fta)
    = cmap f (\ vtg -> g (fta vtg))
    = \ vtf -> f ((\ vtg -> g (fta vtg)) vtf)
    = \ vtf -> f (g (fta vtf))
    = \ vt  -> f (g (fta vt)) = RHS
```

16. Even fancier type class (names are not meant to convey any intuition):

```
class Multi m where
    pack  :: a -> m a
    rpmap :: m a -> (a -> m b) -> m b
```

    With laws:

```
(i)   rpmap (pack a) f = f a
(ii)  rpmap vma pack = vma
(iii) rpmap vma (\ x -> rpmap g (f x)) = rpmap (rpmap f vma) g
```

17. Useful implementation:

```
instance Multi Maybe where
    pack x     = Just x
    rpmap mx f = case mx of
                      Nothing -> Nothing
                      Just x  -> f x
```

    Proof of law-abidingness is left as an exercise.

18. Application: minus. Predecessor often like this:

```
pred Z     = Z
pred (S n) = n
```

    Minus (monus) as a fold:

```
monus n1 n2 = foldNat pred n1 n2
```

    Alternative predecessor:

```
pred' Z     = Nothing
pred' (S n) = Just n
```

Minus not so pretty anymore:

```
minus n1 n2 = foldNat f e n2
    where
    e    = Just n1
    f mn = case mn of
               Nothing -> Nothing
               Just n  -> pred' n
```

Use membership of Maybe in Multi:

```
minus' n1 n2 = foldNat f e n2
    where
    e    = pack n1
    f mn = rpmap mn pred'
```

Also `Maybe` chaining see code.

19. Package often called `Functor` and defined like this:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Laws are the same.

1a. `Multi` is often called Monad and defined like this:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

Laws are the same. Haskell has some extra stuff.

1b. Functor and Monad can be implemented by a host of things. No general intuition.

1c. Conclusion:

- We have seen the type class mechanism in Haskell.
- It can be used for any Hindley-Milner type system (preprocessor) [1].
- We have seen some examples of how to use it ourselves.
- Main aim: introduce functors and monads through the back door. - Look at definitions and don't get confused by others' intuitions and whine.

20. Exercises:

    - Remove "deriving Show" from the definition of Nat and give your own implementation of Show.
    - Make Nat implement Enum.
    - Define the data type `LInteger` as a pair of `Nat`s and make it implement `Num`. (Or even `Integral` if you feel good.)
    - Prove that the list implementation of `Package` fulfills the second law for implementations of `Package`.
    - Prove that the `Maybe` implementation of `Multi` fulfills the laws for implementations of `Multi`.
    - Solve the exercises in [5]. You might notice a similarity to the type classes we defined. Prove that your solutions also abide by the respective laws.
    - Write a monad tutorial.

      Addenda to exercises about folds:

    - Prove that eqNat implements equality for Nats.
      Hint: Specify equality on Nat using a recursive function eqNatRec and prove that $\forall n_1, n_2 \in \mathbb{N}, eqNat\ n_1\ n_2 = eqNatRec\ n_1\ n_2$ by induction on n_1.
    - Using program calculation, derive insert.
      Hint: Specify insertion of an element into a sorted list as a recursive function insertRec. Using the principle of induction, find insert as a function that fulfills `insert x l = insertRec x l` for any sorted list `l`.

21. References:

1 Philip Wadler, Stephen Blott: How to make ad-hoc polymorphism less ad hoc. In: 16th Symposium on Principles of Programming Languages, Austin, Texas, January 1989

2 Simon Marlow (ed.): Haskell 2010 Language Report. (http://www.haskell.org/onlinerep 2014-06-12)

3 Cezar Ionescu: Advanced Functional Programming. Freie Universitt Berlin, 2013. (http://www.pik-potsdam.de/members/ionescu/advanced-functional-programming, 2014-05-12)

4 Mike Vanier: Yet Another Monad Tutorial. 2010-07-25. (http://mvanier.livejournal.com 2014-06-12)

5 Dan Burton: 20 intermediate exercises. 2013-03-04 (https://www.fpcomplete.com/user/ intermediate-exercises, 2014-06-14)

Combinator Gymnastics

Christian Clausen

August 21, 2014

1 Reminder: Pairs and Tuples

```
mk_pair ≡ λ a b c . c a b
π₁² ≡λ p . p (λ a b . a)
π₂² ≡λ p . p (λ a b . b)
<x, y> ⇝ mk_pair a b = λ c . c x y
mk_tuple³ ≡λ a b c d . d a b c
π₁³ ≡λ p . p (λ a b c . a)
π₂³ ≡λ p . p (λ a b c . b)
π₃³ ≡λ p . p (λ a b c . c)
mk_tuple⁴ ≡λ a b c d e . e a b c d
π₁⁴ ≡λ p . p (λ a b c d . a)
...
```

Pretty regular structure. Can we make a function `mk_mk_tuple` N?

2 Basis

For closed terms. By induction on t:

$$\begin{aligned}
\mathcal{A}[\![t_1\ t_2]\!] &= \mathcal{A}[\![t_1]\!]\ \mathcal{A}[\![t_2]\!] \\
\mathcal{A}[\![\lambda x.x]\!] &= I \\
\mathcal{A}[\![\lambda x.\lambda y.t]\!] &= \mathcal{A}[\![\lambda x.\mathcal{A}[\![\lambda y.t]\!]]\!] \\
\mathcal{A}[\![\lambda x.t]\!] &= K\ \mathcal{A}[\![t]\!] \quad \text{if } t \text{ doesn't use } x \\
\mathcal{A}[\![\lambda x.t_1\ t_2]\!] &= S\ \mathcal{A}[\![\lambda x.t_1]\!]\ \mathcal{A}[\![\lambda x.t_2]\!]
\end{aligned}$$

```
I = λ x . x
K = λ x y . x
S = λ f g x . f x (g x) = λ f g x . (f x) (g x)
```

We now have a 3-point basis for the closed lambda terms.

However, we know that:

```
S K I  →β  λ x . K x (I x)  →β  λ x . x = I
or
S K S  →β  λ x . K x (S x)  →β  λ x . x = I
...
S K M  →β λx . K x (M x)  →β λx . x = I
therefore:
S K K  →β  λ x . K x (K x)  →β  λ x . x = I
```

And thus we have a 2-point basis.

2.1 Example

```
A[λ a b c . c a b] = A[λ a . A[λ b c . c a b]]
= A[λ a . A[λ b . A[λ c . c a b]]]
= A[λ a . A[λ b . S A[λ c . c a] A[λ c . b]]]
= A[λ a . A[λ b . S (S A[λ c . c] A[λ c . a]) (K b)]]
= A[λ a . A[λ b . S (S I (K a)) (K b)]]
= A[λ a . S A[λ b . S (S I (K a))] A[λ b . K b]]
= A[λ a . S (K S (S I (K a))) (S A[λ b . K] A[λ b . b])]
= A[λ a . S (K S (S I (K a))) (S (K K) I)]
= S A[λ a . S (K S (S I (K a)))] A[λ a . (S (K K) I)]
= S (S A[λ a . S] A[λ a . K S (S I (K a))]) (K (S (K K) I))
= S (S (K S) (S A[λ a . K] A[λ a . S (S I (K a))])) (K (S (K K) I)
= S (S (K S) (S (K K) (S A[λ a . S] A[λ a . S I (K a)]))) (K (S (K
= S (S (K S) (S (K K) (S (K S) (S A[λ a . S] A[λ a . I (K a)]))))
= S (S (K S) (S (K K) (S (K S) (S (K S) (S A[λ a . I] A[λ a . K a]
= S (S (K S) (S (K K) (S (K S) (S (K S) (S (K I) (S A[λ a . K] A[λ
= S (S (K S) (S (K K) (S (K S) (S (K S) (S (K I) (S (K K) I))))))
```

3 More Detailed Basis

Again for closed terms. By induction on t:

$$\begin{aligned}
\mathcal{B}[\![t_1\ t_2]\!] &= \mathcal{B}[\![t_1]\!]\ \mathcal{B}[\![t_2]\!] \\
\mathcal{B}[\![\lambda x.x]\!] &= I \\
\mathcal{B}[\![\lambda x.\lambda y.t]\!] &= \mathcal{B}[\![\lambda x.\mathcal{B}[\![\lambda y.t]\!]]\!] \\
\mathcal{B}[\![\lambda x.t]\!] &= K\ \mathcal{B}[\![t]\!] \quad \text{if } t \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.t_1\ t_2]\!] &= B\ \mathcal{B}[\![t_1]\!]\ \mathcal{B}[\![\lambda x.t_2]\!] \quad \text{if } t_1 \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.t_1\ t_2]\!] &= C\ \mathcal{B}[\![\lambda x.t_1]\!]\ \mathcal{B}[\![t_2]\!] \quad \text{if } t_2 \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.t_1\ t_2]\!] &= S\ \mathcal{B}[\![\lambda x.t_1]\!]\ \mathcal{B}[\![\lambda x.t_2]\!]
\end{aligned}$$

```
B = λ f g x . f (g x)
C = λ f g x . (f x) g = λ f g x . f x g
```

We now have a 5-point basis for the closed lambda terms.

We can add an η rule before the "B-rule":

$$\mathcal{B}[\![\lambda x.t\ x]\!] = \mathcal{B}[\![t]\!] \quad \text{if } t \text{ doesn't use } x$$

3.1 Example

```
B[λ a b . b a] = B[λ a . B[λ b . b a]]
= B[λ a . C B[λ b . b] a]
= B[λ a . C I a]
= C I

B[λ a b c . c a b] = B[λ a . B[λ b c . c a b]]
= B[λ a . B[λ b . B[λ c . c a b]]]
= B[λ a . B[λ b . C B[λ c . c a] b]]
= B[λ a . B[λ b . C (C B[λ c . c] a) b]]
= B[λ a . B[λ b . C (C I a) b]]
= B[λ a . C (C I a)]
= B C B[λ a . C I a]
= B C (C I)

B[λ a b c d . d a b c] = B (B C) (B C (C I))
```

$$\langle n, t\rangle \xrightarrow{f} \langle S\ n, n\ B\ C\ t\rangle$$

```
f ≡ λ p . <S (π²₁ p), (π²₁ p) B C (π²₂ p)>
mk_mk_tuple ≡ λ n . π²₂ (n f <0, I>)
```

4 Exercises

Warm-up Prove that that $\langle a, b\rangle$, π_1^2, and π_2^2 are mutually correct.
(Hint: $\pi_1^2\langle a, b\rangle = a$, and $\pi_2^2\langle a, b\rangle$ =?)

1. Apply the $\mathcal{A}$ algorithm to $\omega = \lambda x.x\ x$.
2. Apply the $\mathcal{A}$ algorithm to $\Omega = \omega\ \omega$.
3. Apply the $\mathcal{A}$ algorithm to $Y_{Turring} = A\ A$, where $A = \lambda xy.y\ (x\ x)$.
4. ♡ Discover a 1-point basis.
5. Apply the $\mathcal{B}$ algorithm to $\omega = \lambda x.x\ x$.
6. Apply the $\mathcal{B}$ algorithm to $\Omega = \omega\ \omega$.
7. Apply the $\mathcal{B}$ algorithm to $Y_{turring} = A\ A$, where $A = \lambda xy.y\ (x\ x)$.
8. Apply $\mathcal{B}$ to π_2^2.
9. Apply $\mathcal{B}$ to π_3^3.
10. Apply $\mathcal{B}$ to π_4^4.
11. Implement a lambda term `last N =` π_N^N.
12. Apply $\mathcal{B}$ to π_1^2.
13. Apply $\mathcal{B}$ to π_1^3.
14. Apply $\mathcal{B}$ to π_1^4.
15. Implement a lambda term `fst N =` π_1^N.

16. If you haven't done so already, implement the $\mathcal{B}$ algorithm in a programming language.

17. Run your algorithm on:

    (a) π_1^3, π_2^3, π_3^3.

    (b) π_1^4, π_2^4, π_3^4, π_4^4.

    (c) π_1^5, π_2^5, π_3^5, π_4^5, π_5^5.

18. ♡ Implement a lambda term $\pi\ N\ n = \pi_n^N$.

19. Implement the $\mathcal{A}$ algorithm in a programming language.

20. Run your algorithm on:

    (a) π_1^3, π_2^3, π_3^3.

    (b) π_1^4, π_2^4, π_3^4, π_4^4.

    Is the result as structured? As meaningful?

21. When a function does not have a fixed number of arguments, it is called variadic. Implement a variadic version of `K N x` $\mathtt{y}_1$ $\mathtt{y}_2$ `...` $\mathtt{y}_n$ that will always return the first argument.

22. ♡ Implement a variadic version of `S N` $\mathtt{f}_1$ $\mathtt{f}_2$ `...` $\mathtt{f}_n$ `x` that will pass the last argument to all the previous.

23. ♡ Implement a variadic function `V N M` $\mathtt{f}_1$ $\mathtt{f}_2$ `...` $\mathtt{f}_m$ `...` $\mathtt{f}_n$ `x` that will pass the last argument to $\mathtt{f}_m$.

Introduction to Abstract Interpretation

Christian Clausen, 20081015

September 4, 2014

1 Operational Semantics

$$
\begin{aligned}
var \in Var &= \{\texttt{X}, \texttt{Y}, \texttt{Z}\} \\
instr ::= &\ \texttt{inc}\ var \\
&| \ \texttt{dec}\ var \\
&| \ \texttt{zero}\ var\ \texttt{then}\ m\ \texttt{else}\ n \\
&| \ \texttt{push}\ m \\
&| \ \texttt{pop}\ var \\
&| \ \texttt{stop}
\end{aligned}
$$

$$
\frac{P_{pc} = \texttt{inc X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X+1, Y, Z, pc+1, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{dec X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X-1, Y, Z, pc+1, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{zero X then}\ m\ \texttt{else}\ n \quad X = 0}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, m, stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{zero X then}\ m\ \texttt{else}\ n \quad X \neq 0}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, n, stack\rangle}
$$

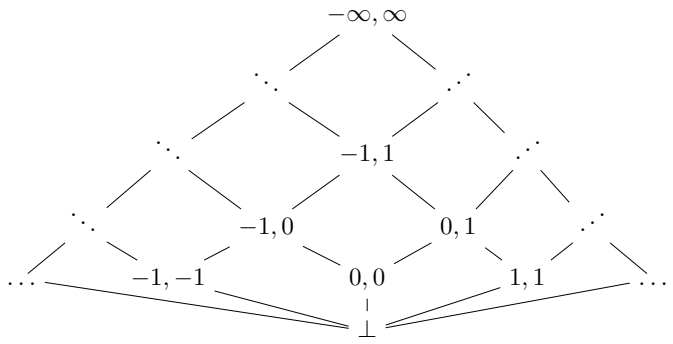
$$
\frac{P_{pc} = \texttt{push X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, pc+1, X :: stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{push}\ m}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, pc+1, m :: stack\rangle}
$$

$$
\frac{P_{pc} = \texttt{pop X}}{\langle X, Y, Z, pc, a :: stack'\rangle \to \langle a, Y, Z, pc+1, stack'\rangle}
$$

2 Abstract Domain

$Interval :=$



2.1 Join and Meet

$$
\begin{aligned}
X \sqcup \bot &= X \\
\bot \sqcup Y &= Y \\
[a, b] \sqcup [c, d] &= [\min(a, c), \max(b, d)] \\
X \sqcap \bot &= \bot \\
\bot \sqcap Y &= \bot \\
[a, b] \sqcap [c, d] &= \begin{cases} [\max(a, c), \min(b, d)] & \text{if } \max(a, c) \leq \min(b, d) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

3 Galois Connection

$$
\begin{aligned}
\wp(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times PC \times Stacks) &\leftrightarrows PC \to \wp(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times Stacks) \\
&\leftrightarrows PC \to \wp(Stacks) \\
&\underset{\alpha}{\overset{\gamma}{\leftrightarrows}} PC \to Interval
\end{aligned}
$$

where the last step is achieved by:

$$
\begin{aligned}
\alpha(\emptyset) &= \bot \\
\alpha(S) &= [\min(\text{len}(S)), \max(\text{len}(S))] \\
\gamma(\bot) &= \emptyset \\
\gamma([a, b]) &= \{stack \mid a \leq \text{len}(stack) \leq b\}
\end{aligned}
$$

4 Abstract Operators

$$
\begin{aligned}
-- &: \wp(Stacks) \to \wp(Stacks) \\
\texttt{=0} &= \lambda S.S \\
\texttt{<>0} &= \lambda S.S \\
\texttt{+1} &= \lambda S.S \\
\texttt{-1} &= \lambda S.S \\
\texttt{push} &= \lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\} \\
\texttt{pop} &= \lambda S.\{s' \mid s \in S \wedge s = n :: s'\}
\end{aligned}
$$

The first four operators are identical thus we will treat them at once. We have:

$$
\begin{aligned}
\alpha \circ (\lambda S.S) \circ \gamma(\bot) &= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.S) \circ \gamma([a,b]) &= \alpha(\{s \mid a \leq \text{len}(s) \leq b\}) \\
&= [a,b]
\end{aligned}
$$

For `push`, we have:

$$
\begin{aligned}
\alpha \circ (\lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\}) \circ \gamma(\bot) &= \alpha(\{n :: s \mid s \in \emptyset \wedge n \in \mathbb{N}\}) \\
&= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\}) \circ \gamma([a,b]) &= \alpha(\{n :: s \mid s \in \{s \mid a \leq \text{len}(s) \leq b\} \wedge n \in \mathbb{N}\}) \\
&= \alpha(\{n :: s \mid a \leq \text{len}(s) \leq b \wedge n \in \mathbb{N}\}) \\
&= [1+a, 1+b]
\end{aligned}
$$

For `pop`, we have:

$$
\begin{aligned}
\alpha \circ (\lambda S.\{s' \mid s \in S \wedge s = n :: s'\}) \circ \gamma(\bot) &= \alpha(\{s' \mid s \in \emptyset \wedge s = n :: s'\}) \\
&= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.\{s' \mid s \in S \wedge s = n :: s'\}) \circ \gamma([a,b]) &= \alpha(\{s' \mid s \in \{s \mid a \leq \text{len}(s) \leq b\} \wedge s = n :: s'\}) \\
&= \alpha(\{s' \mid a \leq \text{len}(s) \leq b \wedge s = n :: s'\}) \\
&= [a-1, b-1]
\end{aligned}
$$

5 Widening and Narrowing

$$
\begin{aligned}
X \bigtriangledown \bot &= X \\
\bot \bigtriangledown Y &= Y \\
[a,b] \bigtriangledown [c,d] &= \left[\begin{cases} -\infty & c < a \\ a & a \geq c \end{cases}, \begin{cases} \infty & d > b \\ b & d \leq b \end{cases} \right]
\end{aligned}
$$

$$X \triangle \bot = \bot$$
$$\bot \triangle Y = \bot$$
$$[a,b] \triangle [c,d] = \left[\begin{cases} c & a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & b = \infty \\ b & \text{otherwise} \end{cases} \right]$$

6 Abstract Transfer Function

$$\begin{aligned} T^{\#}(S) := {} & [0 \mapsto [0,0]] \\ & \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{push}\ m}} [pc+1 \mapsto push(S(pc))] \right) \\ & \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{pop}\ var}} [pc+1 \mapsto pop(S(pc))] \right) \\ & \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{zero}\ var,\ pc',\ pc''}} [pc' \mapsto S(pc)] \cup [pc'' \mapsto S(pc)] \right) \\ & \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{inc}\ var \vee P_{pc} = \texttt{dec}\ var}} [pc+1 \mapsto S(pc)] \right) \end{aligned}$$

where $f \cup g$ is the pointwise join; $f \cup g = \lambda pc.f(pc) \sqcup g(pc)$.

Logic Programming

Mathias Vorreiter Pedersen

September 19, 2014

Prolog

These notes will introduce the basics of logic programming using the Prolog programming language.

Datatypes

Prolog has 1 datatype called a *term*. A term is either

1. An *atom*
2. A *number*, which can either be an integer or a floating point number.
3. A *variable*
4. A *compound term*

In the subset of Prolog which we will be using an atom is a sequence of symbols, starting with a lower case letter, or the empty list, which is written as [ ].

A variable is any sequence of symbols that start with an upper case letter like A, B1, MyVariable etc.

Finally a compound term (or complex term) is a term of the form $\mathbf{f}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_\text{N})$, where $\mathbf{f}$ is an atom (usually called the *functor*) and $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_\text{N}$ are terms (usually called *arguments*). A nonempty list $[\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_\text{N}]$ is also a compound term.

Syntax

A Prolog program is a list of Horn clauses ending with a dot. A horn clause is either a fact, which is a compound term or a rule of the form

`head` ($\texttt{arg}_1, \texttt{arg}_2, \ldots, \texttt{arg}_\text{N}$) :- $\texttt{fact}_1, \texttt{fact}_2, \ldots, \texttt{fact}_\text{N}$

where head is an atom and $\texttt{arg}_1, \texttt{arg}_2, \ldots, \texttt{arg}_\text{N}, \texttt{fact}_2, \ldots, \texttt{fact}_\text{N}$ are terms.

We say that the list of horn clauses defines a *knowledge base* on which we can perform queries. An example knowledge base is

```
append([], L, L).
append([H | T], L, [H | L2]) :- append(T, L, L2).

reverse([], L, L).
reverse([H | T], L, R) :- reverse(T, [H | L], R).

reverse(L, R) :- reverse(L, [], R).
```

Unification

Unification is the process of finding variable bindings such that a goal is satisfied. Unification can be viewed as a recursive procedure on two terms. Two terms `t1` and `t2` unify iff:

1. $\texttt{t}_1$ is an atom with symbol $\texttt{a}_1$, $\texttt{t}_2$ is an atom with symbol $\texttt{a}_2$ and $\texttt{a}_1 = \texttt{a}_2$.
2. $\texttt{t}_1$ is an unbound variable and $\texttt{t}_2$ is a term. The same is true if $\texttt{t}_2$ is an unbound variable and $\texttt{t}_1$ is a term.
3. $\texttt{t}_1$ is a bound variable, $\texttt{t}_2$ is a term and the value bound to $\texttt{t}_1$ unifies with $\texttt{t}_2$. The same is true if $\texttt{t}_2$ is an unbound variable and $\texttt{t}_1$ is a term.
4. $\texttt{t}_1$ is a compound term with functor $\texttt{f}_1$ and term list $\texttt{terms}_1$, and $\texttt{t}_2$ is a compound term with functor $\texttt{f}_1$ and term list $\texttt{terms}_2$, and $\texttt{f}_1 = \texttt{f}_2$ and each pair $(\texttt{t}_1, \texttt{t}_2)$ in $\texttt{zip}(\texttt{terms}_1, \texttt{terms}_2)$ unifies. Note that this requires that $|\texttt{terms}_1| = |\texttt{terms}_2|$.

Unification in Prolog is achieved using the `=/2` goal. An example follows

```
1 ?- a = a.
true.

2 ?- X = a.
X = a.

3 ?- f(X, g(Y, a)) = f(h(a), g(Z, a)).
X = h(a),
Y = Z.
```

Or the last goal without syntactic sugar

```
1 ?- =(f(X, g(Y, a)), f(h(a), g(Z, a))).
X = h(a),
Y = Z.
```

Backtracking

Backtracking is the process of "undoing" choices made during unification. An example follows.

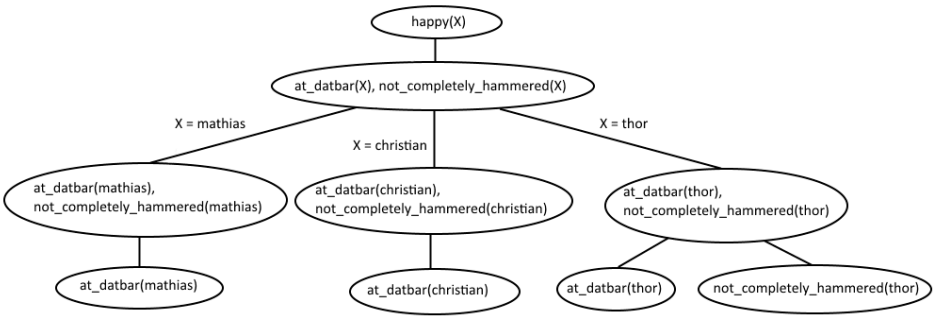
```
happy(X):-
        at_datbar(X),
        not_completely_hammered(X).

at_datbar(mathias).
at_datbar(christian).

not_completely_hammered(thor).
at_datbar(thor).
```

This causes Prolog to derive the following proof tree. First we unify the variable X with the atom "mathias". This choice fails since we can't prove `not_completely_hammered(mathias)`. This causes

Prolog to backtrack to the point where it chose the latest binding of X. Repeating this process two more times it succeeds with X = "thor."



This process is confirmed by SWI-Prolog, which performs the following operations, which corresponds exactly to the proof tree.

```
[trace] 1 ?- happy(X).
   Call: (6) happy(_G2852) ? creep
   Call: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(mathias) ? creep
   Call: (7) not_completely_hammered(mathias) ? creep
   Fail: (7) not_completely_hammered(mathias) ? creep
   Redo: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(christian) ? creep
   Call: (7) not_completely_hammered(christian) ? creep
   Fail: (7) not_completely_hammered(christian) ? creep
   Redo: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(thor) ? creep
   Call: (7) not_completely_hammered(thor) ? creep
   Exit: (7) not_completely_hammered(thor) ? creep
   Exit: (6) happy(thor) ? creep
X = thor.
```

Cuts

Sometimes we (as logic programmers) know more than what Prolog is able to infer. For instance if there is only 1 correct binding of variables that cause a clause to succeed. Take for instance the following implementation of the max function.

```
max(A, B, B) :- B >= A.
max(A, B, A) :- A > B.
```

which is certainly correct, but is inefficient. This can be seen in the context of the following knowledge base.

```
f(X) :- max(X, 20, X), p(X).
p(31).
```

That is, f(X) is true iff X is greather than or equal to 20, and p(X) is true. Tracing `f(20)` we see

```
[trace] 1 ?- f(20).
   Call: (6) f(20) ? creep
   Call: (7) max(20, 20, 20) ? creep
   Call: (8) 20>=20 ? creep
   Exit: (8) 20>=20 ? creep
   Exit: (7) max(20, 20, 20) ? creep
   Call: (7) p(20) ? creep
   Fail: (7) p(20) ? creep
   Redo: (7) max(20, 20, 20) ? creep
   Call: (8) 20>20 ? creep
   Fail: (8) 20>20 ? creep
   Fail: (7) max(20, 20, 20) ? creep
   Fail: (6) f(20) ? creep
false.
```

Notice the `Redo`? That's Prolog trying to find another way of satisfying `max(X, 20, X)`, by trying the other clause `A > B`. But we know that this cannot be satisfied, since `B >= A` already succeeded!

Thus we need a way to tell Prolog that it should not try other clauses once the first succeeds. The `!/0` (cut) goal does exactly that. It always succeeds, and it tells Prolog to never try alternatives for the current goal. Rewriting max as

```
max(A, B, B) :- B >= A, !.
max(A, B, A) :- A > B.
```

and tracing `f(20)` again we see

```
[trace] 1 ?- f(20).
   Call: (6) f(20) ? creep
   Call: (7) max(20, 20, 20) ? creep
   Call: (8) 20>=20 ? creep
   Exit: (8) 20>=20 ? creep
   Exit: (7) max(20, 20, 20) ? creep
   Call: (7) p(20) ? creep
   Fail: (7) p(20) ? creep
   Fail: (6) f(20) ? creep
false.
```

which is a large performance gain!

Negation as failure

Let's say Vincent likes burgers.

```
likes(vincent,X) :- burger(X)
```

That is, vincent likes X iff X is a burger. But what if Vincent really dislikes Big Kahuna burgers? We need to specify that vincent likes burgers that are not from Big Kahuna Burgers. Let's use our new cut goal can help us here, combined with `fail/0`, which always fails the current goal.

```
likes(vincent,X) :- big_kahuna_burger(X),!,fail.
likes(vincent,X) :- burger(X)
```

Since Prolog examine the rules from top to bottom, we always hit the rule containing `big_kahuna_burger(X)` first. In that case we prevent Prolog from trying any other choices for X, and force it to fail with this choice. In other words, Prolog will always fail `likes(vincent,X)` if `big_kahuna_burger(X)` is satisfied. Perfect!

Let's encapsulate this pattern in a rule, called neg (for negation duh!)

```
neg(Goal) :- Goal, !, fail.
neg(Goal).
```

and we can now write `likes/2` as

```
likes(vincent,X) :- burger(X), neg(big_kahuna_burger(X)).
```