Cheatsheet Virtual Machines

This cheatsheet is only useful, if you followed the lecture & understand the material!!!
Please don't go into the exam with this cheatsheet if you haven't done the exercises/ followed the lecture!!!
Some parts of the lecture (without code) are left out!

code$_L$ (X) ⇒ loadc (p X) (G) | loadrc (p X) (L) | loads/stores X
(e1[e2])p ⇒ code$_R$ e1 p code$_R$ e2 p loadc |t| mul add
(e.c) ⇒ code$_L$ (e)p loadc (pt c) add pt → {0,1,2,3,...}
*c) ⇒ code$_R$ (e)p (e → a)p ⇒ code$_R$ (e)p loadc (pt a) add

CMA

Locals ⇒ {L → 1, ... L → n}
Globals ⇒ {G → 1, ... G → n}
Parameters ⇒ {L → -3, ..., L → -n}

---

code$_R$ (x) ⇒ loadc (p x) load

(x=e) ⇒ code$_R$ (e)p code$_L$ (x)p store
(;) ⇒ pop (if e is an array) ⇒ code$_L$ (e)
(malloc (e) p ⇒ code$_R$ e p new
(&e)p ⇒ code$_L$ (e)p
(f) ⇒ loadc _f (g()p) ⇒ loadc 0 mark code$_R$ g p call
(g(e1,...,en)p) ⇒ code$_R$ (en) ... code$_R$ (e1) p mark code$_R$ g p call slide(n-1)
(e) ⇒ e is a struct as parameter code$_L$ (e)p move k

---

code

(if (e) s1 else s2) ⇒ code$_R$ (e)p jumpz A code(s1)p jump B A: code(s2)p B:
(while (e) s' ⇒ A: code$_R$ (e)p jumpz B code s'p jump A B:
(for (e1; e2; e3) s' ⇒ code$_R$ e1 pop A: code$_R$ (e2) p jumpz B code s'p code$_R$ e3 p pop jump A B:
switch (e) {case 0: ss0 break; ... case k-1: ssk-1 break; default ssk}
⇒ code$_R$ e p check 0 k B C0: code ss0 p jump D ... jump Ck-1 D
(free (e);) ⇒ code$_R$ e pop
(f(specs){ss}) ⇒ _f: enter q alloc k code ss pf return
(return) ⇒ code$_R$ e p storer -3 return
(programm) ⇒ enter k+4 alloc k+1 mark loade _main call slide k halt

q = max stack + locals
*global(rad)

---

code$_B$ (e) p sd (constant) ⇒ loadc b (variables) ⇒ code$_V$ e getbasic

MaMa

---

code$_V$ (e) p sd (constant) ⇒ loadc b mkbasic (e1 op e2) code$_B$ e1 code$_B$ e2 op mkbasic
(variable) ⇒ getvar x eval getvar x
(let g1=e1 in ... gn=en in e2) ⇒ code$_C$ e1 ... code$_C$ en code$_V$ e2 slide n use code$_V$ instead
(fun x0...xn-1 → e) ⇒ getvar zi mkvec g mkfunval A jump B A: targ k code$_V$ e sd=0 return B
unbound vars zi ie {0,...,g} slide k; apply (outer supply) mkvec 0 wrap popvar (outer supply) B: ...
(e' e0 ... en-1) e' is a function application ⇒ mark A code$_C$ en-1 code$_C$ e0 code$_V$ e' apply A: code$_V$ instead
↪ if last call ⇒ no mark A, but move r k
(let rec g1=e1 and ... and gn=en in e2) ⇒ alloc n code$_C$ e1 rewrite n code$_C$ en rewrite 1 code$_V$ e2 slide n
eval) ⇒ mark 0 pushloc 3 apply 0 (tuples) ⇒ code$_C$ (e0) code$_C$ ek-1 mkvec k (#j e) code$_V$ e get j eval
(let g0 ... gk-1) = e1 in e2 ⇒ code$_V$ e1 getvec k code$_V$ e2 slide k [C] nil
(e1 :: e2) ⇒ code$_C$ e1 code$_C$ e2 cons
(match e0 with [] → e1 | h::t → e2) ⇒ code$_V$ e0 tlist A code$_V$ e1 jump B A: code$_V$ e2 slide 2 B: ...
(try e1 with x → e2) ⇒ try A code$_V$ e1 restore B A: code$_V$ e2 slide 1 B: ... p' = p @ {x → (L, sd+1)}
(raise e) ⇒ code$_V$ e raise

CBV
CON

G → {0, ..., n}
L → Locals {0, ..., n}
Parameters {-1, ..., -n}

---

code$_C$ (e) p sd ⇒ getvar zi mkvec g mkclos A jump B A: code$_V$ e sd=0 update B: ...
code$_C$ (basic) ⇒ code$_V$ (basic) code$_C$ (X) = getvar x (reject cyclic and recordes)
code$_C$ (function) ⇒ code$_V$ (function) code (Programm) ⇒ code$_V$ e halt

---

code$_A$ (atom) ⇒ putatom a (X) ⇒ putvar (p X) (X) ⇒ putref (p X) (_) ⇒ putanon
(f (t1, ..., tn)) ⇒ code$_A$ t1 ... code$_A$ tn putstruct f/n

WiM

Literals code$_G$ (p (t1, ..., tk) p) = mark B code$_A$ t1 p ... code$_A$ tk p call p/k B: ...
Last call lastmark code$_A$ t1 p ... code$_A$ tk p lastcall p/k

L → {1, ..., n}

↳ if also last clause & last literal is the only one ⇒ code$_{A+1}$ p ... code$_{A+k}$ p move m h; jump p|k ← args for clause
(stack height)

Unification 1 (X̄ = t)p = put X̄ p code$_A$ t p unify

Unification 2 (X̄ = t)p = put X̄ p code$_U$ t p

local vars ↓ m

put Xp ⇒ putvar (pX) put _p = putanon put X̄p = putref (pX) (!) ⇒ prune pushenv notV(X) ∈ p(X), !, fail notV(X) ∈

(X̄ is t) ⇒ put X̄ code$_A$ t (code$_E$ t) evalexp (makeint) unify (e1 + e2) code$_G$(e1) code$_G$(e2) putstruct(+)/2

---

code$_U$ (a)p ⇒ uatom a (X) ⇒ uvar (pX) (_) ⇒ pop (X̄) ⇒ uref (pX)

f(t1, ..., tn)p ⇒ ustruct f/n A son1 code$_U$ t1 p ... son n code$_U$ tn p up B A: check ivars(f(t1,..,tn))

code$_A$ f(t1,...,tn)p bind B: ... only once per var → check(t1) ... check(tn)

Clauses code$_C$ r ⇒ pushenv m code$_G$ g1 ... code$_G$ gn popenv

Predicates rr ⇒ q/k: setbtp try A1 ... try A(f-1) delbtp jump A; A1: code$_C$ r1 ... A(f): code$_C$ rf

---

Program (rr1 ... rrn ? g) ⇒ init A pushenv d code$_G$ g halt d A: no code$_P$ rr1 ... code$_P$ rrn

---

Trimming order variables formal params first, then from right to left in goals, trim m at every non-last goal

---

Clause Indexing code$_P$ rr = p/k: putref 1 getMode index p|k tabins rr A1: code$_C$ r1 ... Am: code$_C$ rm

NIL, CONS, ELSE

---

code$_E$ (z) ⇒ loadc z (X̄) ⇒ evalref (pX) (t1 + t2) code$_E$ t1 code$_E$ t2 add

---

global(g,a)	local(l,a)	attribute(A,a)	virtual(V,b)	pot-virtual(W,a)	static(S,a)	constructor(C,a)
_main	-3 ... -n	1 ... n	0 ... n	1 ... n	1 ... n	_int_list

Attributes/memberfunctions loadr -3 loadc a add | loadm q storem q

Objects in CMA

---

code$_R$ e1.f(e2, ..., en) ⇒ code$_R$ en ... code$_R$ e2 code$_L$ e1 mark loadc _f call (slide m (non-virtual) / loadi 6 add load (virtual)) rcall 6

new C(e2, ..., en) = load |C|+1 new code$_R$ en ... code$_R$ e2 loads m mark loadc _C call pop m+1 (m=param)

---

code f(t2 x2, ..., tn xn){ss} ⇒ enter q alloc m code ss p1 return (defining member-fun)

defining constructor virtual _C q = max stack + m m = space for local vars

C x(e2, ..., en); = code$_R$ en ... code$_R$ e2 code$_L$ x mark loadc _C call pop m+1

---

code$_R$ self() ⇒ self (while) ⇒ A: code$_R$ e jumpz B code s yield jump A **Threaded CMA**

(create(e0,e1)) ⇒ code$_R$ e0 code$_R$ e1 initStack initThread (exit(e)) ⇒ code$_R$ e exit term next

(join(e)) ⇒ code$_R$ e join finalize (newMutex()) ⇒ newMutex (lock(e);) ⇒ code$_R$ e lock

(unlock(e);) ⇒ code$_R$ e unlock (newCondVar()) ⇒ newCondVar (signal(e);) ⇒ code$_R$ e signal

(wait(e0,e1);) ⇒ code$_R$ e1 code$_R$ e0 wait dup unlock next lock (broadcast(e);) ⇒ code$_R$ e broadcast

---

Semaphore

```
typedef struct { Mutex *me; CondVar *cv; int count; } Sema;
Sema* newSema(int n) { Sema *s; s=(Sema*)malloc(sizeof(Sema)); s->me=newMutex(); s->cv=newCondVar();
                       s->count=n; return(s); }
void Down(Sema *s) { Mutex *me; me=s->me; lock(me); while(s->count<=0) wait(s->cv,me); s->count--; unlock(me); }
void Up(Sema *s) { Mutex *me; me=s->me; lock(me); s->count++; if(s->count<=1) signal(s->cv); unlock(me); }
```