Haskell functions

app (++)	notElem item list
chr int (toChar)	nub list (no dups)
drop count list	null list
dropWhile func list	odd number
elem item list (isIn)	partition func list
even number	repeat str (endless)
filter func list	replicate cnt el
flatten [[a]]	reverse list
fold op init list	show str
foldr op init list	snd tuple
fst tuple	sort list
head list	span tester list
init list (w/o last)	splitAt leftCnt list
iterate func init	tail list
last list	take count list
length list	takeWhile func list
map func list	zip list1 list2
not bool	zipWith func l1 l2

data Maybe **t** = Nothing | Just **t**

** (pow floats)	log x	pi
^ (pow ints)	max x y	product list
abs x	maximum l	round x
all sum	min x y	signum x
and list	minimum l	sin x
cos x	mod x y	sqrt x
div x y	negate x	subtract x y
exp x	not bool	sum list
floor x	or list	tan x

Prolog functions

append(L1, L2, Res)	atom(X)
atomic(X)	call(X)
delete(LX, X, L)	even(X)
fail	freezeAll([X1, X2,..], T)
indexOf(El, Ls, Idx)	integer(X)
X is *NumericExpr*	member(X, List)
nat(N) (nat. Num)	not(X)
odd(X)	permute(Ls, Perm)
qsort(Ls, SortedLs)	rev(Ls, RevLs)
split TODO	subseq(SubL, SupL)
sqrt(X, $\sqrt{X}$)	var(Uninstanciated)

Type inference

$$CONST: \frac{c \in Const}{\Gamma \vdash c\ :\tau_c}$$

$$VAR: \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$ABS: \frac{\Gamma, x: \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

$$APP: \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$LET: \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let X} = t_1\ in\ t_2 : \tau_2}$$

$$VAR_{LET} = \frac{\Gamma(x) = \tau' \quad \tau' \succcurlyeq \tau}{\Gamma \vdash x : \tau}$$

$$ABS_{LET} = \frac{\Gamma, x :\ \tau_1 \vdash t :\ \tau_2 \quad \tau_1\ kein\ Typschema}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

$$ta(\tau, \Gamma) = \forall \alpha_i.\tau \textbf{ mit } \alpha_i \in FV(\tau) \backslash FV(\Gamma)$$

Churchnumbers, recursion operator

$$c_0 = \lambda s.\lambda z.z \quad c_n = \lambda s.\lambda z.s^n\ z$$

$$plus = \lambda m.\lambda n.\lambda\ s.\lambda z.m\ s\ (n\ s\ z)$$

$$times = \lambda m.\lambda n.\lambda s.n\ (m\ s)$$

$$exp = \lambda m.\lambda n.n\ m$$

$$if\ = boolexpr$$

$$b_1 \&\&\ b_2 = b_1 b_2 c_{false}$$

$$isZero = \lambda n.n\ (\lambda x.c_{false})\ c_{true}$$

$$c_{true} = \lambda t.\lambda f.t \quad c_{false} = \lambda t.\lambda f.f$$

$$succ\ c_n = TODO$$

$$pred = \lambda n.n\ \varphi(\lambda f.f\ c_0\ c_0)\ c_{true}$$

$$\varphi = \left(\lambda x.\lambda f.f\ (x\ c_0)\ (succ\ (x\ c_0))\right)$$

$$Y = \lambda f.\left(\lambda x.f\ (x\ x)\right)\left(\lambda x.f\ (x\ x)\right)$$

$$f\ (Y\ f) = Y\ f$$

Funktional *E* anwenden: ~*E* = *Y E*

MPI SEND/RECEIVE

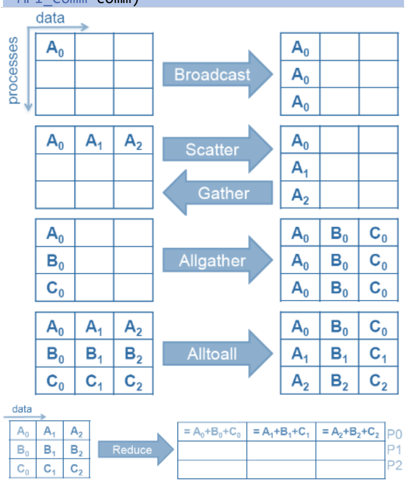
```
int MPI_Send( void* buffer, int count,
MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buffer, int count,
MPI_Datatype datatype,
int source, int tag, MPI_Comm comm,
MPI_Status* status)
MPI_{Send, Ssend, Bsend, Rsend}
```

MPI SEND/RECEIVE

```
int MPI_Send( void* buffer, int count,
MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
int MPI_Recv( void* buffer, int count,
MPI_Datatype datatype,
int source, int tag, MPI_Comm comm,
MPI_Status* status)
```

MPI SEND/RECEIVE

```
int MPI_Bcast( void* buffer, int count,
MPI_Datatype t,
int root, MPI_Comm comm)
int MPI_Scatter( void* sendbuf, int
sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype
recvtype,
int root, MPI_Comm comm)
int MPI_Gather( void* sendbuf, int
sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype
recvtype,
int root, MPI_Comm comm)
int MPI_Allgather( void* sendbuf, int
sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype
recvtype,
MPI_Comm comm)
int MPI_Alltoall( void *sendbuf, int
sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype
recvtype, MPI_Comm comm)
int MPI_Reduce( void* sendbuf, void*
recvbuf, int count,
MPI_Datatype type, MPI_Op op, int root,
MPI_Comm comm)
```



Reducers: MPI_{LAND, BAND, LOR, BOR, MAX, MIN, SUM, PROD, MINLOC, MAXLOC}

Example: Matrix multiplication

```
void mMult(int argc, char* argv[], int
a[n][n], int b[n][n], int c[n][n]) {
  int procs;
  int rank;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int from = rank * n/procs;
  int to = (rank+1) * n/procs;
  MPI_Bcast(b, n*n, MPI_INT, 0,
    MPI_COMM_WORLD);
  MPI_Scatter(a, n*n/procs, MPI_INT, a[from],
    n*n/procs, MPI_INT, 0, MPI_COMM_WORLD);
  int i,j,k;
  for (i = from; i < to; ++i) {
    for (j = 0; j < n; ++j) {
      c[i][j] = 0;
      for (k = 0; k < n; ++k)
        c[i][j] += a[i][k] * b[k][j];
  } } // end all for loops

  MPI_Gather(c[from], n*n/procs, MPI_INT, c,
     n*n/procs, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Finalize();
}
```

JML

@requires	@ensures
==>, <==>, <=!=>	\result, \old(abc)

@invariant (class conditions)

@assignable abc (fields that can be written by method. @assignable \nothing = @pure)

@signals (Exception e) signalCondition

{\forall, \exists} declar, range-expr, body

$$Precondition_{Super} \Rightarrow Precondition_{Sub}$$

$$Postcond_{Sub} \Rightarrow Postcond_{Super}$$

$$Invariants_{Sub} \Rightarrow Invariants_{Super}$$

Amdahl

$\mathrm{S(n)} = \dfrac{1}{(1-\mathrm{p}) + \frac{\mathrm{p}}{\mathrm{n}}}$	$S(n) = \dfrac{T(1)}{T(n)}$
p = parallelizable percentage of progr.	T(n) = Execution time with n processors

Coffman conditions

Mutual Exclusion	Hold and wait
No preemtion	Circular wait

Actors

```
pc A extends AbstractActor {
  @Ov p Receive createReceive() {
    return receiveBuilder()
      .match(MsgClass.class (e.g. String),
             λMatchTester, λMatchHandler)
      .matchAny(λMatchHandler).build();
} }
ActorSystem as=ActorSystem.create(„name“);
ActorRef a
  = as.actorOf(Props.create(A.class));
a.tell(„message“, ActorRef.noSender());
as.terminate();
```

Actormethods: preStart(), postStop(), getSelf(), getSender(), sending PoisonPill .getInstance(), ActorRefFactory.stop()

Java Bytecode

public int fib(int steps) { **int last0 = 1;** **int last1 = 1;** **while (--steps** **> 0) {** **int t = last0** **+ last1;** **last1 = last0;** **last0 = t;** **}** **return last0;** **}**	**iconst 1** // Konstante 1 **istore 2** // in last0 schreiben **iconst 1** // Konstante 1 **istore 3** // in last1 schreiben loop begin: **iinc** 1 -1 // steps-- **iload 1** // steps laden **ifle** after loop // Falls <=0, jump **iload 2** // last0 laden **iload 3** // last1 laden **iadd** // addiere (last0+last1) **istore** 4 // in t (Var 4) schreiben **iload 2** // last0 (Var 2) laden **istore 3** // in last1 schreiben **iload** 4 // t (Var 4) laden **istore 2** // in last0 schreiben **goto** loop begin //springe after loop: **iload 2** // last0 (Var 2) laden **ireturn** // Verlassen mit Wert

public class Test {
int bar() {
return foo(42);
}
int foo(**int** i) {
return i;
}
}
	int bar();
	aload_0
	bipush 42
	invokevirtual #foo.test
	ireturn
	int foo(**int**);
	iload_1
	ireturn

public void arr() {
int[] array
= **new int**[10];
array[7] = 42;
}
	bipush 10 // Konstante 10
	newarray int
	astore 1 // in (var 1) speichern
	aload 1 // (var 1) laden
	bipush 7 // Konstante 7
	bipush 42 // Konstante 42
	iastore // *Wert (42) auf array index (7) von array("array") schreiben*
	return

if (!(x<y && !(y<3))) {..} **else** {..}	**iload** 1 **iload** 2 if_icmplt leftTrueLabel **goto thenLabel** leftTrueLabel: **iload** 2 ldc 3 if_icmplt thenLabel **goto** elseLabel thenLabel: ... **goto** afterLabel elseLabel: ... afterLabel: