Programmierparadigmen

1 Haskell

- `abs :: Num a => a -> a`: absolute value of the number
- `concat :: [[a]] -> [a]`: accepts a list of lists and concatenates them
- `concatMap :: (a -> [b]) -> [a] -> [b]`: creates a list from a list generating function by application of this function on all elements in a list passed as the second argument
- `div :: Integral a => a -> a -> a`: returns how many times the first number can be divided by the second one.
- `drop :: Int -> [a] -> [a]`: creates a list without the given number of items from the beginning of the second argument.
- `dropWhile :: (a -> Bool) -> [a] -> [a]`: it takes elements from the input list the moment when the condition fails for the first time till the end of the list
- `elem :: Eq a => a -> [a] -> Bool`: true if the list contains the first argument
- `filter :: (a -> Bool) -> [a] -> [a]`: returns a list constructed from members of a list (second argument) fulfilling a condition given by the first argument
- `foldl :: (b -> a -> b) -> b -> [a] -> b`: it takes the second argument and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on.
- `foldr :: (a -> b -> b) -> b -> [a] -> b`: it takes the second argument and the last item of the list and applies the function, then it takes the penultimate item from the end and the result, and so on.
- `fromInteger :: Num a => Integer -> a`: convert from Integer to Num instance
- `head :: [a] -> a`: returns the first item of a list
- `iterate :: (a -> a) -> a -> [a]`: creates an infinite list where the first item is calculated by applying the function on the second argument, the second item by applying the function on the previous result and so on.

- `last :: [a] -> a`: returns the last item of a list
- `length :: [a] -> Int`: returns the number of items in a list
- `map :: (a -> b) -> [a] -> [b]`: returns a list constructed by applying a function (the first argument) to all items in a list passed as the second argument.
- `max :: Ord a => a -> a -> a`: returns the larger of its two arguments
- `maximum :: Ord a => [a] -> a`: returns the maximum value of the list
- `min :: Ord a => a -> a -> a`: returns the smaller of its two arguments
- `negate :: Num a => a -> a`: change the sign of the number
- `null :: [a] -> Bool`: returns True if a list is empty, otherwise False
- `repeat :: a -> [a]`: creates infinite list where all items are the first argument
- `reverse :: [a] -> [a]`: creates new list from original with items in reverse order
- `snd :: (a,b) -> b`: returns the second item in a tuple
- `take :: Int -> [a] -> [a]`: creates a list. The first argument determines how many items should be taken from the list passed as the second argument.
- `tail :: [a] -> [a]`: it accepts a list and returns the list without its first item
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`: output elements are calculated from function and elements of input lists occuring at same position
- `(!!) :: [a] -> Int -> a`: list index operator
- Example of `instance`: `instance Show a => Show (Exp a) where`

2 Prolog

- `append(?List1, ?List2, ?List1AndList2)`: `List1AndList2` is the concatenation of `List1` and `List2`
- `atom(@Term)`: True if `Term` is bound to an atom.
- `delete(+List1, @Elem, -List2)`: Delete matching elements from a list. True when `List2` has all elements from `List1` except for those that unify with `Elem`.
- `integer(@Term)`: True if `Term` is bound to an integer
- `member(?Elem, ?List)`: true if `Elem` is a member of `List`.
- `not(:Goal)`: True if `Goal` cannot be proven.

3 Lambda

- Church-Zahlen: $c_0 = \lambda s.\ \lambda z.\ z,\ c_1 = \lambda s.\ \lambda z.\ s\ z,\ c_2 = \lambda s.\ \lambda z.\ s\ (s\ z),\ \ldots$
- isZero $= \lambda n.\ n\ (\lambda x.\ c_{\text{false}})\ c_{\text{true}}$
- Nachfolgerfunktion: succ $= \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
- Addition: plus $= \lambda m.\ \lambda n.\ \lambda s\ \lambda z.\ m\ s\ (n\ s\ z)$
- Multiplikation: times $= \lambda m.\ \lambda n.\ \lambda s.\ n\ (m\ s)$
- Potenzieren: exp $= \lambda m.\ \lambda n.\ n\ m$
- Boolean: $\mathrm{c}_{\text{true}} = \lambda t.\ \lambda f.\ t,\ \mathrm{c}_{\text{false}} = \lambda t.\ \lambda f.\ f$
- Rekursionsoperator $\mathrm{Y} = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
- Normalreihenfolge: Reduziere linksten äußersten Redex
- Call-by-name: Reduziere linkesten äußersten Redex (nicht falls von λ umgeben)
- Call-by-value: Reduziere linksten Redex (nicht von λ umgeben, Argument ist Wert)

4 Typinferenz

4.1 Normale Regeln

$$\textsc{Const}\ \frac{c \in \text{Const}}{\Gamma \vdash c\ :\ \tau_c} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x\ :\ \tau}$$

$$\textsc{Abs}\ \frac{\Gamma,\ x\ :\tau_1 \vdash t\ :\ \tau_2}{\Gamma \vdash \lambda x.\ t\ :\ \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash t_1\ :\ \tau_2 \to \tau \quad \Gamma \vdash t_2\ :\ \tau_2}{\Gamma \vdash t_1\ t_2\ :\ \tau}$$

4.2 Polymorphismus

$$\textsc{Var}\ \frac{\Gamma(x) = \phi \quad \phi \succeq \tau}{\Gamma \vdash x\ :\ \tau} \qquad \textsc{Let}\ \frac{\Gamma \vdash t_1\ :\ \tau_1 \quad \Gamma,\ x\ :\ \mathrm{ta}(\tau_1, \Gamma) \vdash t_2\ :\ \tau_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2\ :\ \tau_2}$$

4.3 Typinferenz für Let

$$\frac{\Gamma \vdash t_1\ :\ \alpha_i \quad \Gamma' \vdash t_2\ :\ \alpha_j}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2\ :\ \alpha_k}$$

- Sammle Constraints aus linkem Teilbaum in C_{let}
- Berechne σ_{let} von C_{let}
- Berechne $\Gamma' := \sigma_{\text{let}}(\Gamma),\ x\ :\ \mathrm{ta}(\sigma_{\text{let}}(\alpha_i),\ \sigma_{\text{let}}(\Gamma))$.
- Benutze Γ' in rechtem Teilbaum, sammle Constraints in C_{body}
- Constraints: $C'_{\text{let}} \cup C_{\text{body}} \cup \{\alpha_j = \alpha_k\}$ ($C'_{\text{let}} := \{\alpha_n = \sigma_{\text{let}}(\alpha_n) \mid \sigma_{\text{let}}$ definiert für $\alpha_n\}$)

4.4 Robinson-Algorithmus

```
if C == ∅ then []
else let {θ_l = θ_r} ∪ C' = C in
  if θ_l == θ_r  then unify(C')
  else if θ_l ==Y and Y ∉ FV(θ_r) then unify([Y ⇒ θ_r] C') ∘ [Y ⇒ θ_r]
  else if θ_r ==Y and Y ∉ FV(θ_l) then unify([Y ⇒ θ_l] C') ∘ [Y ⇒ θ_l]
  else if θ_l == f(θ_l^1, ..., θ_l^n) and θ_r == f(θ_r^1, ..., θ_r^n)
    then unify(C' ∪ {θ_l^1 = θ_r^1, ..., θ_l^n = θ_r^n} )
  else fail
```

Allgemeinster Unifikator: σ mgu, falls $\forall$ Unifikator γ $\exists$ Substitution δ. $\gamma = \delta \circ \sigma$.

5 MPI

- Datatypes: `MPI_INT`, `MPI_CHAR`
- `int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`: gathers data from all tasks and distribute the combined data to all tasks.
- `int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`: sends data from all to all processes.
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`: rank of caller in `comm`.
- `int MPI_Comm_size(MPI_Comm comm, int *size)`: number of processes in `comm`.
- `int MPI_Finalize(void)`: terminates MPI execution environment.
- `int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`: gathers together values from a group of processes.
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`: Begins nonblocking receive.
- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`: begins a nonblocking send.
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`: Blocking receive for message.
- `int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`: reduces values on all processes to a single value (using operations such as `MPI_SUM`, `MPI_MAX` or `MPI_MIN`).

- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`: performs a blocking send.
- `int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`: sends data from one process to all other processes in a communicator.

6 Java

- `IntStream mapToInt(ToIntFunction<? super T> mapper)`: returns `IntStream` after applying function to elements
- `ExecutorService::newSingleThreadExecutor()`
- `ExecutorService::newFixedThreadPool(int n)`
- `ExecutorService::newCachedThreadPool()`
- `ExecutorService::execute(Runnable runnable)`
- `ExecutorService::shutdown()`
- `Future<String> future = exeService.submit(() ->  return "Hi"; );`
- `String s = future.get();`
- Coffman conditions: mutual exclusion, hold and wait, no preemption, circular wait
- Amdahl's Law: $S(n) = \frac{1}{(1-p) + \frac{p}{n}}$

6.1 Akka

- `Actor::createReceive()`: called if message is received, creates `Receive` object
- `ReceiveBuilder receiveBuilder()`: builder for `Receive` objects
- `ReceiveBuilder::match(Class<P> type, UnitApply<P> apply)`: handles messages of specified type by using apply method (method expecting object type `P`)
- `RecieveBuilder::match(Class<P> type, TypedPredicate<P> predicate, UnitApply<P> apply)`: works like above, also checks predicate
- `matchAny(UnitApply<Object> apply)`: handles messages of any type with `apply`
- `Actor::preStart() / postStop()`: executed before actor start/after stop
- `Actor::preRestart() / postRestart()`: executed before/after actor is restarted
- `Actor::getSelf()`: delivers reference (`ActorRef`) to the actor

- `Actor::getContext()`: delivers `ActorRefFactory` for creating new actors
- `Actor::getSender()`: reference to actor sending the currently processed message
- `Props::create(Class<?> actorType, Object... parameters)`: `Props` for creating an actor of given type
- `ActorRefFactory::actorOf(Props props)`: returns `ActorRef` for created actor (implemented by `ActorSystem` and actor context)
- `ActorSystem::create(String)`: returns an `ActorSystem`
- `ActorSystem::terminate()`: terminates actor system
- `ActorRef::tell(Object msg, ActorRef sender)`: asynchronous message send
- `Patterns:ask(ActorRef trg, Object msg, Timeout tout)`: returns `Future<?>`
- `Await::result(Future<T> future, Duration duration)`: wait for result
- `ActorRefFactor::stop(ActorRef actorToStop)`: stops actor
- `PoisonPill.getInstance()`: stop actor by sending posion pill

```
public Receive createReceive() {
  return receiveBuilder()
    .match(String.class,
      message -> message.equals("printHello"),
      message -> System.out.println("Hello World!"))
    .matchAny(message -> unhandled(message))
    .build();
}
```

7 Compilerbau

- Indizmenge von $A \to \alpha$: $\text{First}_k(\alpha\text{Follow}_k(A))$
- SLL(k)-Bedingung: $\forall A \to \alpha \mid \beta : \text{First}_k(\alpha\text{Follow}_k(A)) \cap \text{First}_k(\beta\text{Follow}_k(A)) = \emptyset$
- Linksfaktorisierung: $X \to \gamma\alpha \mid \gamma\beta \Rightarrow X \to \gamma X',\ X' \to \alpha \mid \beta$