

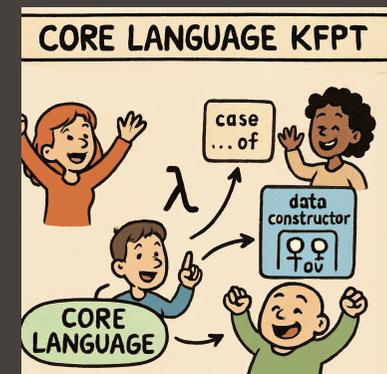
## Introduction

- we modularly extend the call-by-name lambda calculus
- with constructs that make programming easier and
- that fit to non-strict functional programming languages like Haskell
- We add data and case-expressions, recursive functions, the seq-operator and polymorphic types
- naming of the languages: KFP ... defined by Schmidt-Schauß in various lectures on functional programming

## Contents

- The core language KFPT
- The core language KFPTS
- Extension by seq
- Polymorphic types: KFPTSP

## CORE LANGUAGE KFPT



- Extends the lambda calculus with **data constructors** (and data types) and **case**
- **KFPT**: T means typed case
- This typing of case is only syntactic sugar
- I.e., KFPT is still an untyped calculus

**Assumptions:**

- We have a finite set of **types** (these are only names)
- For each type, there exists a finite set of **data constructors**: Notation:  $c_i$ .
- data constructors have a fixed **arity**  $ar(c_i) \in \mathbb{N}_0$

**Examples**

- Type Bool, data constructors: True and False,  $ar(\text{True}) = 0 = ar(\text{False})$ .
- Type Pair, data constructors: Pair,  $ar(\text{Pair}) = 2$ , **Haskell-notation**:  $(a, b)$  for Pair  $a b$
- Type List, data constructors: Nil and Cons,  $ar(\text{Nil}) = 0$  und  $ar(\text{Cons}) = 2$ .

**Haskell-notation**:  $[]$  for Nil and  $:$  (infix) for Cons

**Expr** ::=

- $V$  (variable)
- $\lambda V. \text{Expr}$  (abstraction)
- $(\text{Expr}_1 \text{ Expr}_2)$  (application)
- $(c_i \text{ Expr}_1 \dots \text{ Expr}_{ar(c_i)})$  (constructor application)
- $(\text{case}_T \text{ Expr of } \{\text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_n \rightarrow \text{Expr}_n\})$  (case-expression)

**Pat<sub>i</sub>** ::=  $(c_i V_1 \dots V_{ar(c_i)})$  (pattern for constructor  $i$ )  
 where the variables  $V_i$  are pairwise distinct

**Side conditions:**

- case is labeled with type  $T$
- **Pat<sub>i</sub>  $\rightarrow$  Expr<sub>i</sub>** is a **case-alternative**
- case-alternatives are complete and disjoint for the type:  
for each constructor of type  $T$  there is exactly one case-alternative

- Head of a list:

$$\lambda xs. \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow \perp; (\text{Cons } y \text{ } ys) \rightarrow y\}$$

- Tail of a list:

$$\lambda xs. \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow \perp; (\text{Cons } y \text{ } ys) \rightarrow ys\}$$

$\perp$  is a diverging expression like  $\Omega$

- Test, if a list is empty:

$$\lambda xs. \text{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow \text{True}; (\text{Cons } y \text{ } ys) \rightarrow \text{False}\}$$

## Examples (Cont'd)



- if  $e$  then  $s$  else  $t$ :

$$\text{case}_{\text{Bool}} e \text{ of } \{\text{True} \rightarrow s; \text{False} \rightarrow t\}$$

- projections for pairs:

$$\begin{aligned} \text{fst} &:= \lambda x. \text{case}_{\text{Pair}} x \text{ of } \{\text{Pair } a \ b \rightarrow a\} \\ \text{snd} &:= \lambda x. \text{case}_{\text{Pair}} x \text{ of } \{\text{Pair } a \ b \rightarrow b\} \end{aligned}$$

## case-expressions



- We use  $(\text{case}_T s \text{ of } \text{Alts})$  as abbreviation
- KFPT's case is more restrictive than Haskell's case
- Haskell permits missing alternatives (may lead to runtime errors)
- Haskell permits a default alternative
- Haskell permits nested and overlapping patterns
- All this can be expressed in KFPT using  $\perp$ -alternative and nested case-expressions

## Example: Haskell vs. KFPT



Haskell:  $\text{case } [] \text{ of } \{[] \rightarrow []; (x:(y:ys)) \rightarrow [y]\}$

KFPT:

$$\begin{aligned} \text{case}_{\text{List}} \text{Nil} \text{ of } \{ &\text{Nil} \rightarrow \text{Nil}; \\ &(\text{Cons } x \ z) \rightarrow \text{case}_{\text{List}} z \text{ of } \{ &\text{Nil} \rightarrow \perp; \\ &(\text{Cons } y \ ys) \rightarrow (\text{Cons } y \ \text{Nil}) \\ &\} \\ &\} \end{aligned}$$

## Free and Bound Variables in KFPT



In addition to the lambda calculus: in a case-alternative

$$(c_i \ x_1 \ \dots \ x_{ar(c_i)}) \rightarrow s$$

the variables  $x_1, \dots, x_{ar(c_i)}$  are bound with scope  $s$

## Free Variables in KFPT



$$\begin{aligned}
 FV(x) &= x \\
 FV(\lambda x.s) &= FV(s) \setminus \{x\} \\
 FV(s \ t) &= FV(s) \cup FV(t) \\
 FV(c \ s_1 \ \dots \ s_{ar(c)}) &= FV(s_1) \cup \dots \cup FV(s_{ar(c)}) \\
 FV(\text{case}_T \ t \ \text{of} \\
 &\quad \{(c_1 \ x_{1,1} \ \dots \ x_{1,ar(c_1)}) \rightarrow s_1; \\
 &\quad \dots \\
 &\quad (c_n \ x_{n,1} \ \dots \ x_{n,ar(c_n)}) \rightarrow s_n\}) \\
 &= FV(t) \cup \left( \bigcup_{i=1}^n (FV(s_i) \setminus \{x_{i,1}, \dots, x_{i,ar(c_i)}\}) \right)
 \end{aligned}$$

## Bound Variables in KFPT



$$\begin{aligned}
 BV(x) &= \emptyset \\
 BV(\lambda x.s) &= BV(s) \cup \{x\} \\
 BV(s \ t) &= BV(s) \cup BV(t) \\
 BV(c \ s_1 \ \dots \ s_{ar(c)}) &= BV(s_1) \cup \dots \cup BV(s_{ar(c)}) \\
 BV(\text{case}_T \ t \ \text{of} \\
 &\quad \{(c_1 \ x_{1,1} \ \dots \ x_{1,ar(c_1)}) \rightarrow s_1; \\
 &\quad \dots \\
 &\quad (c_n \ x_{n,1} \ \dots \ x_{n,ar(c_n)}) \rightarrow s_n\}) \\
 &= BV(t) \cup \left( \bigcup_{i=1}^n (BV(s_i) \cup \{x_{i,1}, \dots, x_{i,ar(c_i)}\}) \right)
 \end{aligned}$$

## Example



$$s := ((\lambda x. \text{case}_{\text{List}} \ x \ \text{of} \ \{\text{Nil} \rightarrow x; \text{Cons} \ x \ xs \rightarrow \lambda u. (x \ \lambda x. (x \ u))\}) \ x)$$

$$FV(s) = \{x\} \ \text{und} \ BV(s) = \{x, xs, u\}$$

$\alpha$ -equivalent expression:

$$s' := ((\lambda x_1. \text{case}_{\text{List}} \ x_1 \ \text{of} \ \{\text{Nil} \rightarrow x_1; \text{Cons} \ x_2 \ xs \rightarrow \lambda u. (x_2 \ \lambda x_3. (x_3 \ u))\}) \ x)$$

$$FV(s') = \{x\} \ \text{und} \ BV(s') = \{x_1, x_2, xs, x_3, u\}$$

## Notations



As in the lambda calculus (with the new definition of  $FV$  and  $BV$ )

- Open and closed expressions
- $\alpha$ -renaming and -equivalence
- Distinct variable convention
- Substitution  $s[t/x]$

## Parallel Substitution

- $s[t_1/x_1, \dots, t_n/x_n]$ : substitutes  $x_i$  with  $t_i$  in parallel  
(where for all  $i$ :  $BV(s) \cap FV(t_i) = \emptyset$ )
- $s[t_1/x_1, \dots, t_n/x_n]$  is **not** the same as  $s[t_1/x_1] \dots [t_n/x_n]$ !

## Definition

The reduction rules  $(\beta)$  and  $(\text{case})$  in KFPT are defined as:

$$(\beta) \quad (\lambda x. s) t \rightarrow s[t/x]$$

$$(\text{case}) \quad \text{case}_T (c \ s_1 \ \dots \ s_{ar(c)}) \ \text{of} \ \{ \dots; (c \ x_1 \ \dots \ x_{ar(c)}) \rightarrow t; \dots \} \\ \rightarrow t[s_1/x_1, \dots, s_{ar(c)}/x_{ar(c)}]$$

If  $r_1 \rightarrow r_2$  with  $(\beta)$  or  $(\text{case})$ , then  $r_1$  **directly reduces to**  $r_2$

$$\begin{aligned} & (\lambda x. \text{case}_{\text{Pair}} x \ \text{of} \ \{ (\text{Pair } a \ b) \rightarrow a \}) (\text{Pair True False}) \\ \xrightarrow{\beta} & \text{case}_{\text{Pair}} (\text{Pair True False}) \ \text{of} \ \{ (\text{Pair } a \ b) \rightarrow a \} \\ \xrightarrow{\text{case}} & \text{True} \end{aligned}$$

Contexts = expression with a hole  $[\cdot]$  at expression position

$$\begin{aligned} \text{Ctxt} ::= & [\cdot] \mid \lambda V. \text{Ctxt} \mid (\text{Ctxt Expr}) \mid (\text{Expr Ctxt}) \\ & \mid (c_i \ \text{Expr}_1 \ \dots \ \text{Expr}_{i-1} \ \text{Ctxt} \ \text{Expr}_{i+1} \ \text{Expr}_{ar(c_i)}) \\ & \mid (\text{case}_T \ \text{Ctxt} \ \text{of} \ \{ \text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_n \rightarrow \text{Expr}_n \}) \\ & \mid (\text{case}_T \ \text{Expr} \ \text{of} \ \{ \text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_i \rightarrow \text{Ctxt}; \dots; \text{Pat}_n \rightarrow \text{Expr}_n \}) \end{aligned}$$

If  $C[s] \rightarrow C[t]$  with  $s \xrightarrow{\beta} t$  or  $s \xrightarrow{\text{case}} t$ , then  $s$  is a **redex** of  $C[s]$

## Definition

Reduction contexts in KFPT are defined as:

$$\text{RCtxt} ::= [\cdot] \mid (\text{RCtxt Expr}) \mid (\text{case}_T \ \text{RCtxt} \ \text{of} \ \text{Alts})$$

**Definition**

If  $r_1$  directly reduces to  $r_2$ , then a **call-by-name reduction step in KFPT** is  $R[r_1] \xrightarrow{\text{name}} R[r_2]$  for every reduction context  $R$ .

Notation:

- We use  $\xrightarrow{\text{name}}$ , but also  $\xrightarrow{\text{name},\beta}$  and  $\xrightarrow{\text{name},\text{case}}$ .
- $\xrightarrow{\text{name},+}$  is the transitive closure of  $\xrightarrow{\text{name}}$
- $\xrightarrow{\text{name},*}$  is the reflexive-transitive closure of  $\xrightarrow{\text{name}}$

$$\begin{aligned} (\lambda x.x) ((\lambda y.y) (\lambda z.z)) &\rightarrow x[(\lambda y.y) (\lambda z.z)/x] \\ &= (\lambda y.y) (\lambda z.z) \end{aligned}$$

is a call-by-name reduction

$$\begin{aligned} (\lambda x.x) ((\lambda y.y) (\lambda z.z)) &\rightarrow (\lambda x.x) (y[(\lambda z.z)/y]) \\ &= (\lambda x.x) (\lambda z.z) \end{aligned}$$

is **not** a call-by-name reduction

**Definition**

A KFPT-expression  $s$  is a

- **normal form** (NF), if  $s$  does not contain any  $(\beta)$ - or (case)-redex.
- **head normal form** (HNF), if  $s$  is a constructor application or an abstraction  $\lambda x_1, \dots, \lambda x_n. s'$  where  $s'$  is either a variable, a constructor application or of the form  $(x s'')$  (where  $x$  is a variable).
- **functional weak head normal form** (FWHNF) if  $s$  is an abstraction.
- **constructor weak head normal form** (CWHNF) if  $s$  is a constructor application  $(c s_1 \dots s_{ar(c)})$ .
- **weak head normal form** (WHNF), if  $s$  is an FWHNF or a CWHNF.

A call-by-name evaluation (a sequence of call-by-name reduction steps) **ends successfully** if a **WHNF** is reached

**Definition**

We define **convergence** of KFPT-expression  $s$ :

$$s \Downarrow \iff \exists \text{ WHNF } t : s \xrightarrow{\text{name},*} t$$

If  $\neg s \Downarrow$ , then  $s$  diverges, written as  $s \Uparrow$ .

We say  $s$  **has a WHNF** (a FWHNF, CWHNF, resp), if  $s$  reduces to a WHNF (a FWHNF, CWHNF, resp.) using  $\xrightarrow{\text{name},*}$

## Dynamic Typing

Call-by-name reduction stops without a WHNF in the following cases:

- a free variable is on a reduction position, i.e. the expression is of the form  $R[x],o$
- a **dynamic type error** occurs

### Definition (Dynamic typing rules)

A KFPT-expression  $s$  is **directly dynamically untyped** if:

- $s = R[\text{case}_T (c\ s_1 \dots s_n) \text{ of } Alts]$  and  $c$  is **not** of type  $T$
- $s = R[\text{case}_T \lambda x.t \text{ of } Alts]$
- $s = R[(c\ s_1 \dots s_{ar(c)})\ t]$

$s$  is **dynamically untyped**

$\iff \exists t : s \xrightarrow{\text{name},*} t \wedge t$  is directly dynamically untyped

## Examples

- $\text{case}_{\text{List}} \text{True}$  of  $\{\text{Nil} \rightarrow \text{Nil}; (\text{Cons } x\ xs) \rightarrow xs\}$  is directly dynamically untyped
- $(\lambda x. \text{case}_{\text{List}} x \text{ of } \{\text{Nil} \rightarrow \text{Nil}; (\text{Cons } x\ xs) \rightarrow xs\}) \text{True}$  is dynamically untyped
- $(\text{Cons True Nil}) (\lambda x.x)$  is directly dynamically untyped
- $(\text{case}_{\text{Bool}} x \text{ of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False}\})$  is not (directly) dynamically untyped
- $(\lambda x. \text{case}_{\text{Bool}} x \text{ of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow \text{False}\}) (\lambda y.y)$  is dynamically untyped

## Irreducible Closed Expressions

### Proposition

A closed KFPT-expression  $s$  is irreducible (w.r.t. call-by-name evaluation) iff one of the following conditions is true:

- $s$  is WHNF or
- $s$  is directly dynamically untyped.

There are divergent closed expressions that are not dynamically untyped:

$$\Omega := (\lambda x.x\ x) (\lambda x.x\ x)$$

## Searching the Call-by-Name-Redex: Labeling

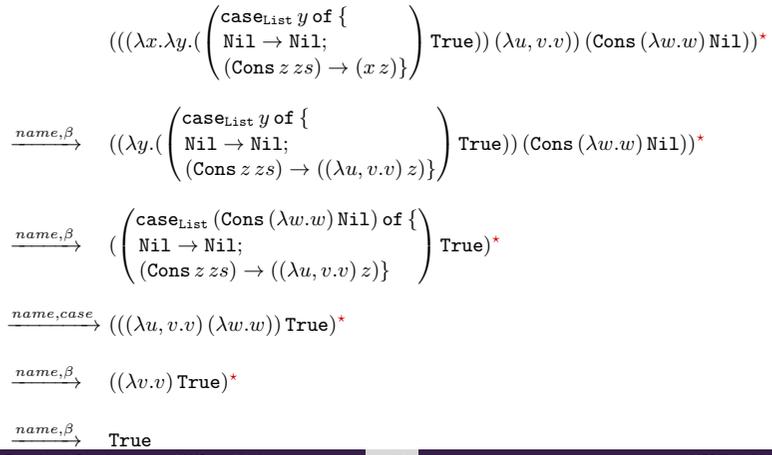
For expression  $s$ , start with  $s^*$  and apply the shifting rules exhaustively:

- $(s\ t)^* \Rightarrow (s^*\ t)$
- $(\text{case}_T s \text{ of } Alts)^* \Rightarrow (\text{case}_T s^* \text{ of } Alts)$

Cases after labeling:

- Label is at an abstraction, subcases:
  - $(\lambda x.s')^*$ , then a FWHNF is detected, no reduction applicable
  - $C[(\lambda x.s')^* s'']$ , then reduce the application with  $(\beta)$
  - $C[\text{case}_T (\lambda x.s')^* \dots]$ , then the expression is directly dynamically untyped
- Label is at a constructor application
  - $(c \dots)^*$ , then a CWHNF is detected, no reduction applicable
  - $C[(c \dots)^* s']$ , the expression is directly dynamically untyped
  - $C[(\text{case}_T (c \dots)^* alts)]$ , reduce with (case) if  $c$  belongs to type  $T$ , otherwise directly dynamically untyped
- Label is at a variable: no reduction applicable

# Example



# Representing Expressions as Termgraphs



A node for every syntactic construct of the expression:

- variable = a leaf
- abstraction  $\lambda x. s$ 

where  $\boxed{s}$  is the tree for  $s$
- application  $(s \ t)$ 

where  $\boxed{s}, \boxed{t}$  are the trees for  $s$  and  $t$

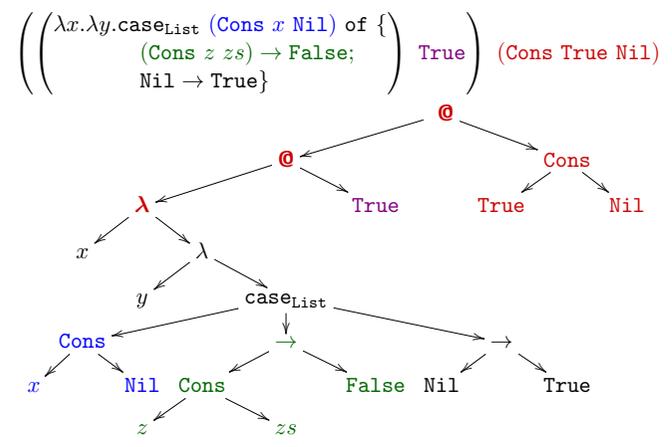
# Representing Expressions as Termgraphs (Cont'd)



- constructor application:  $(c \ s_1 \ \dots \ s_n)$ 

where  $\boxed{s_i}$  are the trees for  $s_i$
- case-expressions:  $n + 1$  children,  $\text{case}_T \ s$  of  $\{\text{Alt}_1; \dots; \text{Alt}_n\}$
- case-alternative  $\text{Pat} \rightarrow t$

# Example



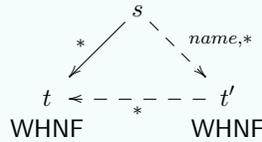
CBN-redex: search always left, until a variable, an abstraction, or a constructor application is found

## Properties of the Call-by-Name Reduction

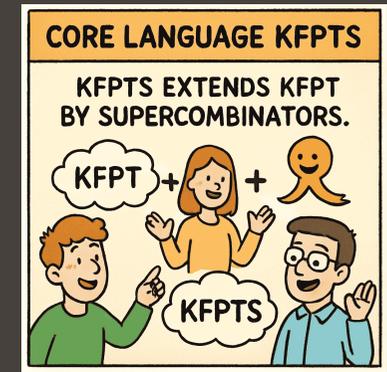
- The call-by-name reduction is deterministic, i.e. for every KFPT-expression  $s$ , there is at most one expression  $t$  with  $s \xrightarrow{\text{name}} t$ .
- A WHNF is irreducible w.r.t. call-by-name evaluation.

### Theorem

Let  $s$  be a KFPT-expression. If  $s \xrightarrow{*} t$  with ( $\beta$ )- and (case)-reductions (applied in arbitrary contexts), where  $t$  is a WHNF, then there exists a WHNF  $t'$ , such that  $s \xrightarrow{\text{name},*} t'$  and  $t' \xrightarrow{*} t$



## CORE LANGUAGE KFPTS



## Recursive Supercombinators: KFPTS

- Next extension: from KFPT to KFPTS
- 'S' means **supercombinators**
- supercombinators are names (constants) for functions
- supercombinators may be **recursive** functions

We assume a set of supercombinator names  $\mathcal{SC}$ .

Example: supercombinator `length`

```
length xs = caseList xs of {
  Nil → 0;
  (Cons y ys) → (1 + length ys)}
```

## Syntax of KFPTS

$$\begin{aligned} \text{Expr} ::= & V \mid \lambda V. \text{Expr} \mid (\text{Expr}_1 \text{ Expr}_2) \\ & \mid (c_i \text{ Expr}_1 \dots \text{Expr}_{ar(c_i)}) \\ & \mid (\text{case}_T \text{ Expr of } \{\text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_n \rightarrow \text{Expr}_n\}) \\ & \mid \text{SC where } \text{SC} \in \mathcal{SC} \end{aligned}$$

$\text{Pat}_i ::= (c_i V_1 \dots V_{ar(c_i)})$  where the variables  $V_i$  are pairwise distinct

## Syntax of KFPTS (Cont'd)



For every supercombinator  $SC$  exists a **supercombinator definition**:

$$SC V_1 \dots V_n = \mathbf{Expr}$$

where

- $V_i$  are pairwise distinct variables
- $\mathbf{Expr}$  is a KFPTS-expression
- $FV(\mathbf{Expr}) \subseteq \{V_1, \dots, V_n\}$
- $ar(SC) = n \geq 0$ : arity of the supercombinator

Example: definition of supercombinator  $map$ :

$$map f xs = \mathbf{case}_{\text{List}} xs \text{ of } \{\text{Nil} \rightarrow \text{Nil}; (\text{Cons } y \ ys) \rightarrow \text{Cons } (f y) (map f ys)\}$$

## KFPTS-Program



A **KFPTS-program** consists of

- a set of types and data constructors,
- a set of supercombinator definitions,
- and a KFPTS-expression  $s$ .

**Side condition:**

All supercombinators that occur in the right-hand sides of the definitions and in  $s$  are defined.

## Operational Semantics



**Reduction contexts:**

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt} \mathbf{Expr}) \mid \mathbf{case}_T \mathbf{RCtxt} \text{ of } \mathbf{Alts}$$

**Reduction rules**  $(\beta)$ ,  $(\text{case})$  and  $(\text{SC-}\beta)$ :

$$(\beta) \quad (\lambda x.s) t \rightarrow s[t/x]$$

$$(\text{case}) \quad \mathbf{case}_T (c s_1 \dots s_{ar(c)}) \text{ of } \{\dots; (c x_1 \dots x_{ar(c)}) \rightarrow t; \dots\} \\ \rightarrow t[s_1/x_1, \dots, s_{ar(c)}/x_{ar(c)}]$$

$$(\text{SC-}\beta) \quad (SC s_1 \dots s_n) \rightarrow e[s_1/x_1, \dots, s_n/x_n], \\ \text{if } SC x_1 \dots x_n = e \text{ is the definition of } SC$$

**Call-by-name reduction:**

$$s \rightarrow t \text{ with } (\beta)\text{-, } (\text{case})\text{- or } (\text{SC-}\beta) \\ \hline R[s] \xrightarrow{\text{name}} R[t]$$

## KFPTS: WHNFs and Dynamic Typing



WHNFs

- WHNF = CWHNF or FWHNF
- CWHNF = constructor application  $(c s_1 \dots s_{ar(c)})$
- FWHNF = abstraction or  $SC s_1 \dots s_m$  with  $ar(SC) > m$

directly dynamically untyped:

- rules as in KFPT:  $R[(\mathbf{case}_T \lambda x.s \text{ of } \dots)]$ ,  $R[(\mathbf{case}_T (c s_1 \dots s_n) \text{ of } \dots)]$ , if  $c$  is not of type  $T$  and  $R[(c s_1 \dots s_{ar(c)}) t]$
- new rule:  
 $R[\mathbf{case}_T (SC s_1 \dots s_m) \text{ of } \mathbf{Alts}]$  is directly dynamically untyped if  $ar(SC) > m$ .

Labeling and shifting is same as in KFPT:

- $(s\ t)^* \Rightarrow (s^*\ t)$
- $(\text{case}_T\ s\ \text{of}\ \text{Alts})^* \Rightarrow (\text{case}_T\ s^*\ \text{of}\ \text{Alts})$

New cases:

- A supercombinator is labeled with  $\star$ :
  - Enough arguments are present: reduce using  $(SC-\beta)$
  - Too few arguments without a surrounding context: WHNF
  - Too few arguments in context  $C[(\text{case}_T\ [\cdot]\ \dots)]$ : directly dynamically untyped

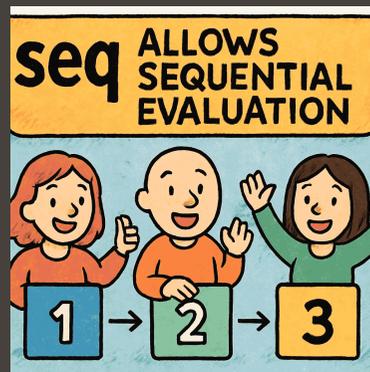
Assume that the supercombinators *map* and *not* are defined as:

$$\begin{aligned} \text{map}\ f\ xs &= \text{case}_{\text{List}}\ xs\ \text{of}\ \{\text{Nil} \rightarrow \text{Nil}; \\ &\quad (\text{Cons}\ y\ ys) \rightarrow \text{Cons}\ (f\ y)\ (\text{map}\ f\ ys)\} \\ \text{not}\ x &= \text{case}_{\text{Bool}}\ x\ \text{of}\ \{\text{True} \rightarrow \text{False}; \text{False} \rightarrow \text{True}\} \end{aligned}$$

Evaluation:

$$\begin{aligned} &\text{map}\ \text{not}\ (\text{Cons}\ \text{True}\ (\text{Cons}\ \text{False}\ \text{Nil})) \\ \xrightarrow{\text{name}, SC-\beta} &\text{case}_{\text{List}}\ (\text{Cons}\ \text{True}\ (\text{Cons}\ \text{False}\ \text{Nil}))\ \text{of}\ \{ \\ &\quad \text{Nil} \rightarrow \text{Nil}; \\ &\quad (\text{Cons}\ y\ ys) \rightarrow \text{Cons}\ (\text{not}\ y)\ (\text{map}\ \text{not}\ ys)\} \\ \xrightarrow{\text{name}, \text{case}} &\text{Cons}\ (\text{not}\ \text{True})\ (\text{map}\ \text{not}\ (\text{Cons}\ \text{False}\ \text{Nil})) \end{aligned}$$

EXTENSION BY SEQ



The seq-Operator

Haskell has the binary operator *seq* with semantics:

$$\text{seq}\ a\ b = \begin{cases} b & \text{if } a \downarrow \\ \perp & \text{if } a \uparrow \end{cases}$$

With *seq*, one can define:

$$f\ \$!\ x = \text{seq}\ x\ (f\ x)$$

(makes sense if call-by-need is used instead of call-by-name)

Let  $f\ x = t$  be a supercombinator definition, then  $f\ \$!\ s$  only returns  $t[s/x]$  if  $s \downarrow$ , otherwise  $(f\ s) \uparrow$ .

Operationally:

- first evaluate  $s$  and
- then perform  $(SC-\beta)$
- mimics call-by-value instead of call-by-name evaluation

## Using seq for Space-Optimization

A naive implementation of computing the faculty:

$$fac\ x = \text{if } x = 0 \text{ then } 1 \text{ else } x * (fac\ (x - 1))$$

- Call-by-name evaluation of  $fac\ n$  generates  $n * (n - 1) * \dots * 1$
- Space consumption:  $O(n)$

End-recursive variant:

$$\begin{aligned} fac\ x &= facER\ x\ 1 \\ facER\ x\ y &= \text{if } x = 0 \text{ then } y \text{ else } facER\ (x - 1)\ (x * y) \end{aligned}$$

- does not solve the space-problem

With sharing and seq: constant space:

```
fac x = facER x 1
where facER 0 y = y
      facER x y = let x' = x-1
                  y' = x*y
                  in seq x' (seq y' (facER x' y'))
```

## Extension by seq

seq is not encodable in KFPT and KFPTS

We denote with

- KFPT+seq the extension of KFPT with seq
- KFPT+seq the extension of KFPTS with seq

We omit the formal definitions, but illustrate the extension:

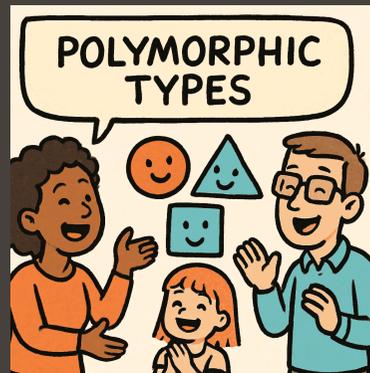
The new reduction rule is:

$$\text{seq } v\ t \rightarrow t, \text{ if } v \text{ is aWHNF}$$

A new shifting rule is:

$$(\text{seq } s\ t)^* \rightarrow (\text{seq } s^*\ t)$$

## POLYMORPHIC TYPES



## Types

- Type constructors are names like Bool, List, Pair,...
- If the arity  $ar(TC) > 0$ , then they are applied to types, e.g. List Bool
- Type constructor of arity 0 are called **base types**

### Definition

The **syntax of polymorphic types** is

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1 \dots \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$$

where  $TV$  is a type variable,  $TC$  is a type constructor of arity  $n$

- $T_1 \rightarrow T_2$  is a **function type**
- Types without type variables are called **monomorphic types**.
- **Polymorphic types** may have type variables
- With **KFPTSP** we denote **polymorphically typed** KFPTS

## Examples

```

True  :: Bool
False :: Bool
not   :: Bool → Bool
map   :: (a → b) → [a] → [b]
(λx.x) :: (a → a)

```

hsrm\*

## Simplified Typing Rules

- For the application:

$$\frac{s :: T_1 \rightarrow T_2, t :: T_1}{(s\ t) :: T_2}$$

- Instantiation:

$$\frac{s :: T \quad \text{if } T' = \sigma(T), \text{ where } \sigma \text{ is a type substitution,}}{s :: T' \text{ that replaces type variables with types.}}$$

- For case-expressions:

$$\frac{s :: T_1, \quad \forall i : Pat_i :: T_1, \quad \forall i : t_i :: T_2}{(\text{case}_T\ s\ \text{of } \{Pat_1 \rightarrow t_1; \dots; Pat_n \rightarrow t_n\}) :: T_2}$$

Note that these rules are not completely correct (will be corrected in the next chapter)!

hsrm\*

## Example

```

and := λx, y. caseBool x of {True → y; False → False}
or  := λx, y. caseBool x of {True → True; False → y}

```

With rule for application:

$$\frac{\text{and} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}, \text{True} :: \text{Bool}}{\text{(and True) :: Bool} \rightarrow \text{Bool}}, \text{False} :: \text{Bool}}{\text{(and True False) :: Bool}}$$

hsrm\*

## Example

$$\frac{\text{Cons} :: a \rightarrow [a] \rightarrow [a], \text{True} :: \text{Bool}}{\text{Cons} :: \text{Bool} \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]}, \frac{\text{Nil} :: [a]}{\text{Nil} :: [\text{Bool}]}}{\text{True} :: \text{Bool}, \frac{(\text{Cons True}) :: [\text{Bool}] \rightarrow [\text{Bool}]}{(\text{Cons True Nil}) :: [\text{Bool}]}, \text{Nil} :: [\text{Bool}]}}{\text{False} :: \text{Bool}, \text{Nil} :: [\text{Bool}]}}{\text{case}_{\text{Bool}}\ \text{True}\ \text{of } \{\text{True} \rightarrow (\text{Cons True Nil}); \text{False} \rightarrow \text{Nil}\} :: [\text{Bool}]}$$

hsrm\*

## Example

hsrm\*

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]}{\text{map} :: (\text{Bool} \rightarrow \text{Bool}) \rightarrow [\text{Bool}] \rightarrow [\text{Bool}], \text{not} :: \text{Bool} \rightarrow \text{Bool}}$$
$$\frac{}{(\text{map not}) :: [\text{Bool}] \rightarrow [\text{Bool}]}$$

## Conclusion

hsrm\*

Core Language	Description
KFPT	Extension of the call-by-name lambda calculus with weakly typed case and data constructors seq is not encodable.
KFPTS	Extension of KFPT by recursive supercombinators
KFPTSP	Restriction of KFPTS to well-typed expressions using a polymorphic type system
KFPT+seq	Extension of KFPT with the seq-operator
KFPTS+seq	Extension of KFPTS with the seq-operator
KFPTSP+seq	Restriction of KFPTS+seq to well-typed expressions using a polymorphic type system