Programming Language Foundations

04 Functional Core Languages

Prof. Dr. David Sabel
Wintersemester 2024/25

Last update: November 27, 2024

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

Hochschule RheinMain

CORE LANGUAGE KFPT

Core Language KFPT

Hochschule RheinMain

- 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

Data Constructors

Assumptions:

- We have a finite set of types (these are only names)

Data Constructors

Hochschule RheinMain

Assumptions:

- We have a finite set of types (these are only names)

Examples

- Type `Bool`,
- Type `Pair`,
- Type `List`,

Data Constructors

Hochschule RheinMain

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.

Examples

- Type `Bool`,
- Type `Pair`,
- Type `List`,

Data Constructors

Hochschule RheinMain

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.

Examples

- Type `Bool`, data constructors: `True` and `False`
- Type `Pair`, data constructors: `Pair`
- Type `List`, data constructors: `Nil` and `Cons`

Data Constructors

Hochschule RheinMain

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`
- Type `Pair`, data constructors: `Pair`
- Type `List`, data constructors: `Nil` and `Cons`

Data Constructors

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`,
  $ar(\texttt{True}) = 0 = ar(\texttt{False})$.
- Type `Pair`, data constructors: `Pair`,
  $ar(\texttt{Pair}) = 2$.
- Type `List`, data constructors: `Nil` and `Cons`,
  $ar(\texttt{Nil}) = 0$ und $ar(\texttt{Cons}) = 2$.

Data Constructors

Hochschule RheinMain

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

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

Syntax of KFPT

$$
\begin{array}{lll}
\mathbf{Expr} ::= & & \\
& V & \text{(variable)} \\
| & \lambda V.\mathbf{Expr} & \text{(abstraction)} \\
| & (\mathbf{Expr}_1\ \mathbf{Expr}_2) & \text{(application)} \\
| & &
\end{array}
$$

Syntax of KFPT

Hochschule RheinMain

$\mathbf{Expr} ::=$

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

$\mathbf{Pat}_i ::= (c_i\ V_1 \ldots V_{ar(c_i)})$ (pattern for constructor i)

where the variables V_i are pairwise distinct

Side conditions:

- case is labeled with type T
- $\mathbf{Pat}_i \rightarrow \mathbf{Expr}_i$ 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

Examples

- Head of a list:

$$\lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; (\mathtt{Cons}\ y\ ys) \to y\}$$

Examples

Hochschule RheinMain

- Head of a list:

$$\lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; (\mathtt{Cons}\ y\ ys) \to y\}$$

- Tail of a list:

$$\lambda xs.\mathtt{case}_{\mathtt{List}}\ xs\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; (\mathtt{Cons}\ y\ ys) \to ys\}$$

$\bot$ is a diverging expression like Ω

Examples

Hochschule RheinMain

- Head of a list:

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

- Tail of a list:

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to ys\}$$

$\bot$ is a diverging expression like Ω

- Test, if a list is empty:

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

Examples (Cont'd)

- $\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t$:

$$\texttt{case}_{\texttt{Bool}}\ e\ \texttt{of}\ \{\texttt{True} \to s; \texttt{False} \to t\}$$

- projections for pairs:

$$\begin{array}{lcl} \mathit{fst} & := & \lambda x.\texttt{case}_{\texttt{Pair}}\ x\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to a\} \\ \mathit{snd} & := & \lambda x.\texttt{case}_{\texttt{Pair}}\ x\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to b\} \end{array}$$

case-expressions

Hochschule RheinMain

- We use $(\texttt{case}_T\ s\ \texttt{of}\ Alts)$ 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 $\bot$-alternative and nested case-expressions

Example: Haskell vs. KFPT

Haskell: `case [] of {[] -> []; (x:(y:ys)) -> [y]}`

KFPT:

$$\begin{array}{l}\texttt{case}_{\texttt{List}}\ \texttt{Nil of}\ \{\texttt{Nil} \to \texttt{Nil};\\ \qquad (\texttt{Cons x z}) \to \texttt{case}_{\texttt{List}}\ \texttt{z of}\ \{\texttt{Nil} \to \bot;\\ \qquad\qquad (\texttt{Cons y ys}) \to (\texttt{Cons y Nil})\\ \qquad\qquad \}\\ \qquad \}\end{array}$$

Free and Bound Variables in KFPT

Hochschule RheinMain

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

$$(c_i\ x_1\ \ \ldots\ \ x_{ar(c_i)}) \to s$$

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

Free Variables in KFPT

Hochschule RheinMain

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

Bound Variables in KFPT

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

Example

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

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

α-equivalent expression:

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

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

Notations

As in the lambda calculus (with the new definition of FV andBV)

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

Operational semantics

Hochschule RheinMain

Parallel Substitution

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

Operational semantics

Hochschule RheinMain

Parallel Substitution

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

Definition

The reduction rules (β) and (case) in KFPT are defined as:

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

$$(\text{case}) \qquad \texttt{case}_T\ (c\ s_1\ \ldots\ s_{ar(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\} \to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$$

If $r_1 \to r_2$ with (β) or (case), then r_1 directly reduces to r_2

Example

$$(\lambda x.\texttt{case}_{\texttt{Pair}}\ x\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to a\})\ (\texttt{Pair True False})$$

$$\xrightarrow{\beta}\ \texttt{case}_{\texttt{Pair}}\ (\texttt{Pair True False})\ \texttt{of}\ \{(\texttt{Pair}\ a\ b) \to a\}$$

$$\xrightarrow{case}\ \texttt{True}$$

Contexts in KFPT

Hochschule RheinMain

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

$$
\begin{array}{rl}
\mathbf{Ctxt} ::= & [\cdot] \mid \lambda V.\mathbf{Ctxt} \mid (\mathbf{Ctxt}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{Ctxt}) \\
\mid & (c_i\ \mathbf{Expr}_1\ \ldots \mathbf{Expr}_{i-1}\ \mathbf{Ctxt}\ \mathbf{Expr}_{i+1}\ \mathbf{Expr}_{ar(c_i)}) \\
\mid & (\mathtt{case}_T\ \mathbf{Ctxt}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
\mid & (\mathtt{case}_T\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_i \to \mathbf{Ctxt}; \ldots, \mathbf{Pat}_n \to \mathbf{Expr}_n\})
\end{array}
$$

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

Call-by-Name Evaluation

Hochschule RheinMain

Definition

Reduction contexts in KFPT are defined as:

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

Call-by-Name Evaluation (Cont'd)

Hochschule RheinMain

Definition

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

Notation:

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

Examples

Hochschule RheinMain

$$
\begin{array}{rcl}
(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) & \to & x[(\lambda y.y)\ (\lambda z.z)/x] \\
& = & (\lambda y.y)\ (\lambda z.z)
\end{array}
$$

is a call-by-name reduction

$$
\begin{array}{rcl}
(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) & \to & (\lambda x.x)\ (y[(\lambda z.z)/y]) \\
& = & (\lambda x.x)\ (\lambda z.z)
\end{array}
$$

is not a call-by-name reduction

Normal Forms

Hochschule RheinMain

Definition

A KFPT-expression s is a

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

Convergence

Definition

We define convergence of KFPT-expression s:

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

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 $\xrightarrow{name,*}$

Dynamic Typing

Hochschule RheinMain

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

Dynamic Typing

Hochschule RheinMain

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:

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

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:

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

s is dynamically untyped

$$\iff$$

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

Examples

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

Irreducible Closed Expressions

Hochschule RheinMain

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:

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

Searching the Call-by-Name-Redex: Labeling

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

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

Searching the Call-by-Name-Redex: Labeling

Hochschule RheinMain

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

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

Cases after labeling:

- Label is at an abstraction, subcases:
  - $(\lambda x.s')^\star$, then a FWHNF is detected, no reduction applicable
  - $C[((\lambda x.s')^\star\ s'')]$, then reduce the application with (β)
  - $C[\texttt{case}_T\ (\lambda x.s')^\star\ \ldots)]$, then the expression is directly dynamically untyped

Searching the Call-by-Name-Redex: Labeling

Hochschule RheinMain

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

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

Cases after labeling:

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

Searching the Call-by-Name-Redex: Labeling

Hochschule **RheinMain**

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

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

Cases after labeling:

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

Example

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\, y\, \texttt{of}\, \{ \\ \texttt{Nil} \to \texttt{Nil}; \\ (\texttt{Cons}\, z\, zs) \to (x\, z)\}\end{pmatrix}\, \texttt{True}))\, (\lambda u, v.v))\, (\texttt{Cons}\, (\lambda w.w)\, \texttt{Nil}))^{\star}$$

Example

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\,y\,\texttt{of}\,\{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\,z\,zs) \to (x\,z)\}\end{pmatrix}\,\texttt{True}))\,(\lambda u,v.v))^{\star}\,(\texttt{Cons}\,(\lambda w.w)\,\texttt{Nil}))$$

Example

Hochschule RheinMain

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{ \\ \texttt{Nil} \to \texttt{Nil}; \\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u,v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u,v.v)\ z)\}\end{array}\right)\ \texttt{True}))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^{\star}$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

Example

Hochschule RheinMain

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})^{\star}$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \ \texttt{Nil} \to \texttt{Nil};\\ \ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \ \texttt{Nil} \to \texttt{Nil};\\ \ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \ \texttt{Nil} \to \texttt{Nil};\\ \ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)^{\star}\ \texttt{True})$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\; y \;\textsf{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to (x\; z)\}\end{array}\right)\; \texttt{True}))^{\star}\; (\lambda u, v.v))\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\; y \;\textsf{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{array}\right)\; \texttt{True}))^{\star}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil})^{\star} \;\textsf{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{array}\right)\; \texttt{True})$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)\ (\lambda w.w))\ \texttt{True})^{\star}$$

Example

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \textsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case}\ (((\lambda u, v.v)\ (\lambda w.w))^{\star}\ \texttt{True})$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{name,case}\ (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

Example

Hochschule RheinMain

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u,v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u,v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta}\ (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u,v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case}\ (((\lambda u,v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{name,\beta}\ ((\lambda v.v)\ \texttt{True})^{\star}$$

Example

Hochschule RheinMain

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u,v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u,v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u,v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u,v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{name,\beta} ((\lambda v.v)^{\star}\ \texttt{True})$$

Example

$$(((\lambda x.\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{name,\beta} ((\lambda v.v)^{\star}\ \texttt{True})$$

$$\xrightarrow{name,\beta} \texttt{True}$$

Representing Expressions as Termgraphs

Hochschule RheinMain

A node for every syntactic construct of the expression:

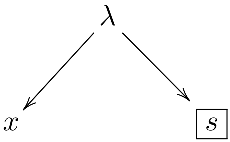
- variable = a leaf

Representing Expressions as Termgraphs

Hochschule **RheinMain**

A node for every syntactic construct of the expression:

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



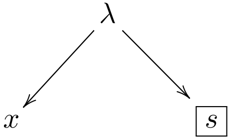
where $\boxed{s}$ is the tree for s

Representing Expressions as Termgraphs

Hochschule RheinMain

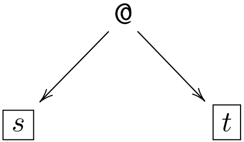
A node for every syntactic construct of the expression:

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



where $\boxed{s}$ is the tree for s

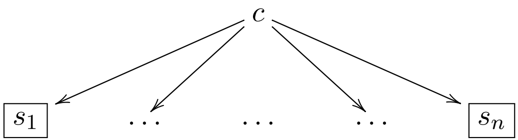
- application $(s\ t)$



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

Representing Expressions as Termgraphs (Cont'd)

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

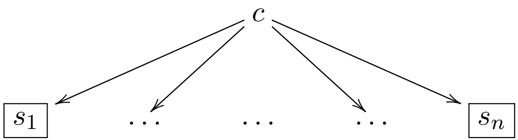


where $\boxed{s_i}$ are the trees for s_i

Representing Expressions as Termgraphs (Cont'd)

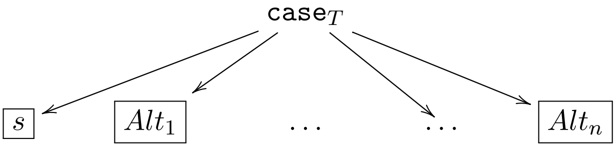
Hochschule RheinMain

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



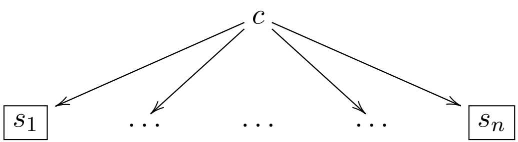
  where $\boxed{s_i}$ are the trees for s_i

- case-expressions: $n+1$ children, $\texttt{case}_T\ s$ `of` $\{Alt_1; \dots; Alt_n\}$



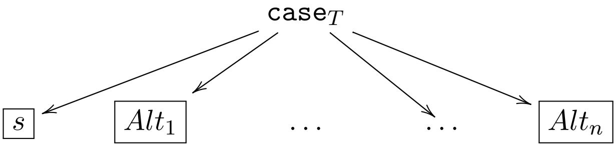
Representing Expressions as Termgraphs (Cont'd)

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

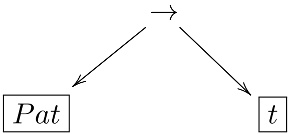


where $\boxed{s_i}$ are the trees for s_i

- case-expressions: $n+1$ children, $\texttt{case}_T\ s\ \texttt{of}\ \{Alt_1; \ldots; Alt_n\}$



- case-alternative $Pat \to t$

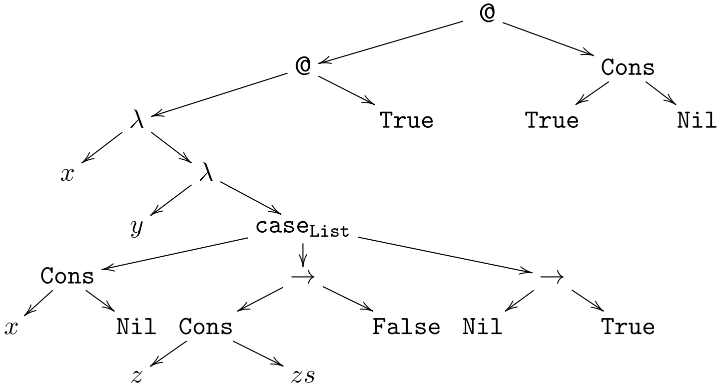


Example

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad\qquad (\texttt{Cons}\ z\ zs) \rightarrow \texttt{False};\\ \qquad\qquad \texttt{Nil} \rightarrow \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$

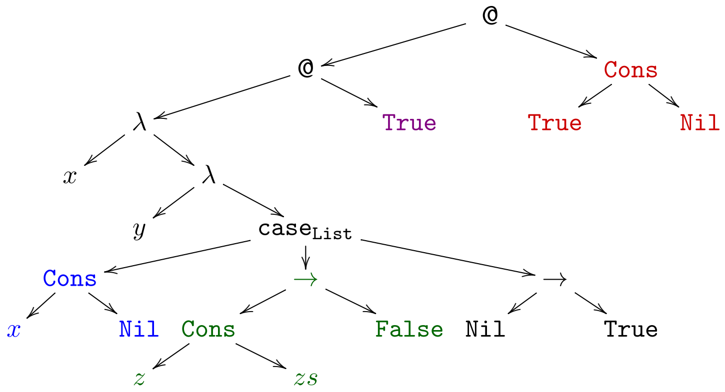
Example

$$\left(\left(\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \left\{\begin{array}{l}(\texttt{Cons}\ z\ zs) \rightarrow \texttt{False};\\ \texttt{Nil} \rightarrow \texttt{True}\}\end{array}\right.\right)\ \texttt{True}\right)\ (\texttt{Cons True Nil})$$



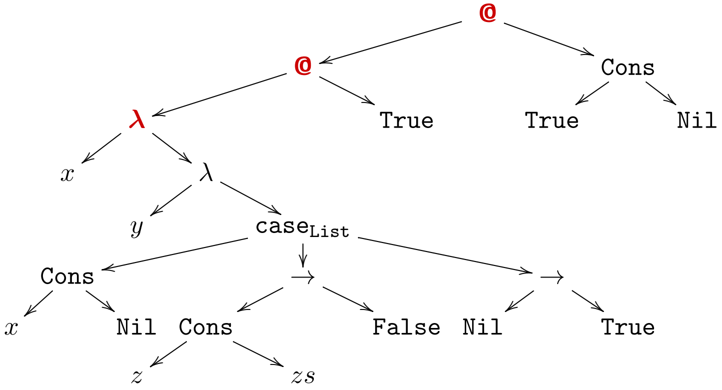
Example

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \quad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \quad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons True Nil})$$



Example

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad\qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad\qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons True Nil})$$



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

Properties of the Call-by-Name Reduction

Hochschule RheinMain

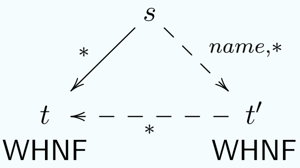
- The call-by-name reduction is deterministic, i.e. for every KFPT-expression s, there is at most one expression t with $s \xrightarrow{name} t$.
- A WHNF is irreducible w.r.t. call-by-name evaluation.

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 $s \xrightarrow{name} t$.
- A WHNF is irreducible w.r.t. call-by-name evaluation.

Theorem

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

$$\begin{array}{l}\texttt{length}\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\\ \qquad\qquad\qquad \texttt{Nil} \to 0;\\ \qquad\qquad\qquad (\texttt{Cons}\ y\ ys) \to (1 + \texttt{length}\ ys)\}\end{array}$$

Syntax of KFPTS

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

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

Syntax of KFPTS (Cont'd)

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

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

where

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

Example: definition of supercombinator map:

$$map\ f\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\}$$

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:

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

Operational Semantics

Hochschule RheinMain

Reduction contexts:

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

Reduction rules (β), (case) and (SC-β):

(β) $\quad (\lambda x.s)\ t \to s[t/x]$

(case) $\quad \mathtt{case}_T\ (c\ s_1\ \ldots\ s_{ar(c)})\ \mathtt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\}$
$\qquad \to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$

(SC-β) $\quad (SC\ s_1\ \ldots\ s_n) \to e[s_1/x_1, \ldots, s_n/x_n]$,
if $SC\ x_1\ \ldots\ x_n = e$ is the definition of SC

Operational Semantics

Hochschule RheinMain

Reduction contexts:

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

Reduction rules (β), (case) and (SC-β):

(β) $\quad (\lambda x.s)\ t \to s[t/x]$

(case) $\quad \texttt{case}_T\ (c\ s_1\ \ldots\ s_{ar(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\}$
$\quad \to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$

(SC-β) $\quad (SC\ s_1\ \ldots\ s_n) \to e[s_1/x_1, \ldots, s_n/x_n]$,
if $SC\ x_1\ \ldots\ x_n = e$ is the definition of SC

Call-by-name reduction:

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

KFPTS: WHNFs and Dynamic Typing

Hochschule **RheinMain**

WHNFs

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

directly dynamically untyped:

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

Labeling Algorithm

Hochschule RheinMain

Labeling and shifting is same as in KFPT:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_T\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_T\ s^\star\ \texttt{of}\ Alts)$

New cases:

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

Example

Assume that the supercombinators map and not are defined as:

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

Example

Assume that the supercombinators map and not are defined as:

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

Evaluation:

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

EXTENSION BY SEQ

The seq-Operator

Hochschule RheinMain

Haskell has the binary operator seq with semantics:

$$(\texttt{seq}\ a\ b) = \begin{cases} b & \text{if } a\Downarrow \\ \bot & \text{if } a\Uparrow \end{cases}$$

With seq, one can define:

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

(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 $(SC\text{-}\beta)$
- mimics call-by-value instead of call-by-name evaluation

Using seq for Space-Optimzation

Hochschule RheinMain

A naive implementation of computing the faculty:

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

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

End-recursive variant:

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

- 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

Hochschule RheinMain

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:

$$\texttt{seq}\ v\ t \quad \to \quad t, \text{ if } v \text{ is aWHNF}$$

A new shifting rule is:

$$(\texttt{seq}\ s\ t)^\star \quad \to \quad (\texttt{seq}\ s^\star\ t)$$

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

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

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

- $T_1 \to T_2$ 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

$$
\begin{array}{lcl}
\texttt{True} & :: & \texttt{Bool} \\
\texttt{False} & :: & \texttt{Bool} \\
\texttt{not} & :: & \texttt{Bool} \to \texttt{Bool} \\
\texttt{map} & :: & (a \to b) \to [a] \to [b] \\
(\lambda x.x) & :: & (a \to a)
\end{array}
$$

Simplified Typing Rules

- For the application:

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

- Instantiation:

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

- For case-expressions:

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

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

Example

$$\begin{aligned}
\texttt{and} &:= \lambda x, y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to y; \texttt{False} \to \texttt{False}\} \\
\texttt{or} &:= \lambda x.y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to y\}
\end{aligned}$$

With rule for application:

$$\frac{\dfrac{\texttt{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \texttt{True} :: \texttt{Bool}}{(\texttt{and True}) :: \texttt{Bool} \to \texttt{Bool}}, \texttt{False} :: \texttt{Bool}}{(\texttt{and True False}) :: \texttt{Bool}}$$

Example

$$
\dfrac{\begin{array}{c}\texttt{True} :: \texttt{Bool},\\ \texttt{False} :: \texttt{Bool}\end{array},\quad
\dfrac{\dfrac{\dfrac{\texttt{Cons} :: a \to [a] \to [a]}{\texttt{Cons} :: \texttt{Bool} \to [\texttt{Bool}] \to [\texttt{Bool}]},\ \texttt{True} :: \texttt{Bool}}{(\texttt{Cons True}) :: [\texttt{Bool}] \to [\texttt{Bool}]},\quad \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{(\texttt{Cons True Nil}) :: [\texttt{Bool}]},\quad
\dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}
{\texttt{case}_{\texttt{Bool}}\ \texttt{True of}\ \{\texttt{True} \to (\texttt{Cons True Nil}); \texttt{False} \to \texttt{Nil}\} :: [\texttt{Bool}]}
$$

Example

$$\dfrac{\dfrac{\texttt{map} :: (a \to b) \to [a] \to [b]}{\texttt{map} :: (\texttt{Bool} \to \texttt{Bool}) \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map}\ \ \texttt{not}) :: [\texttt{Bool}] \to [\texttt{Bool}]}$$

Conclusion

Hochschule RheinMain

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