GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

www.uni-frankfurt.de

Lambda Calculus and Extensions as Foundation of Functional Programming

David Sabel and Manfred Schmidt-Schauß

29. September 2015

Lehrerbildungsforum Informatik

Last update: 30. September 2015

Overview

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Overview

- Untyped Lambda Calculus
- Operational Semantics
- Contextual Semantics
- Extension by Data Types

Pure (Untyped) Lambda Calculus

Lambda Calculus

Grammar for expressions $e \in E_\lambda$ of the pure lambda calculus:

$$e, e_i \in E_\lambda ::= x \mid \lambda x.e \mid (e_1\ e_2) \quad \text{where } x \in \mathit{Var}$$

$\lambda x.e$	Abstraction
$(e_1\ e_2)$	Application (of e_1 to e_2)

- We may omit brackets for better readability.
- The body of abstraction extends as far as possible
- $(e_1\ e_2\ e_3\ e_4)$ is reconstructed as $(((e_1\ e_2)\ e_3)\ e_4)$. Bracketing is left-associative.

Pure Lambda Calculus: Examples

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$\lambda x.x$	The identity function
$\lambda x.\lambda y.x$	The function that can be applied to two arguments e_1, e_2 and returns e_1.
$\lambda x.\lambda y.\lambda f.f\ x\ y$	Used as encoding of pairs
$(\lambda x.(x\ x))\ (\lambda x.(x\ x))$	A lambda-expression which is useless as a program: It is nonterminating.

Free and Bound Variables

free variables $FV(e)$ and *bound variables* $BV(e)$ are inductively defined:

$$
\begin{array}{ll}
FV(x) := \{x\}, \text{ if } x \in \mathit{Var} & BV(x) := \emptyset, \text{ if } x \in \mathit{Var} \\
FV((e_1\ e_2)) := FV(e_1) \cup FV(e_2) & BV((e_1\ e_2)) := BV(e_1) \cup BV(e_2) \\
FV(\lambda x.e) := FV(e) \setminus \{x\} & BV(\lambda x.e) := BV(e) \cup \{x\}
\end{array}
$$

Renaming Bound Variables

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

α-renaming:

$$\lambda x.e \xrightarrow{\alpha} \lambda x'.e[x'/x], \text{ if } x' \notin FV(e) \text{ and } x' \notin BV(e)$$

where α-equivalence $=_\alpha$ is the smallest congruence obtained from $\xrightarrow{\alpha}$, i.e. it is inductively defined as

$$\begin{array}{rcl} e_1 & =_\alpha & e_2, \text{ if } e_1 \xrightarrow{\alpha} e_2 \\ e & =_\alpha & e \\ e_1 & =_\alpha & e_2, \text{ if } e_2 =_\alpha e_1 \\ e_1 & =_\alpha & e_3, \text{ if } e_1 =_\alpha e_2 \wedge e_2 =_\alpha e_3 \\ C[e_1] & =_\alpha & C[e_2], \text{ if } e_1 =_\alpha e_2 \end{array}$$

Variable Convention

The Convention on Bound Variables:

In the expressions mentioned:
binders always bind distinct variables,
and bound variables are distinct from free variables.

This can always be achieved by α-renamings.

Example
$\lambda x.x(y\ (\lambda x.\lambda y.y\ x))$: the convention is not satisfied.

It can be satisfied by α-renaming:
$\lambda x.x(y\ (\lambda x.\lambda y.y\ x)) =_\alpha \lambda x.x(y\ (\lambda z.\lambda u.u\ z))$

Substitution

We define a more general notion of substitution:
$e_1[e_2/x]$ denotes the *substitution* of all free occurrences of variable x in e_1 by the expression e_2:

An inductive definition recursing over the expression structure is:

$$
\begin{array}{lcl}
x[e/x] & := & e \\
y[e/x] & := & y, \text{ if } x \neq y \\
(\lambda x.e_1)[e/x] & := & \lambda x.e_1 \\
(\lambda y.e_1)[e/x] & := & \lambda y.(e_1[e/x]), \text{if } x \neq y \\
(e_1\ e_2)[e/x] & := & (e_1[e/x]\ e_2[e/x])
\end{array}
$$

Due to the distinct variable convention:
There is no capture of variables.

Substitution

Examples:

Compute $(x\ \lambda x.x)[\lambda y.y\ /\ x]$

$$\begin{array}{ll} & (\lambda x.x\ x)\ (\lambda y.y\ y) \\ \rightarrow & (x\ x)\ [(\lambda y.y\ y)\ /\ x] \\ = & (\lambda y.y\ y)\ (\lambda y.y\ y) \\ =_\alpha & (\lambda x.x\ x)\ (\lambda y.y\ y) \end{array}$$

Substitution

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Examples:

Compute $(x\ \lambda x.x)[\lambda y.y\ /\ x]$

1. result of renaming: $(x\ \lambda u.u)[\lambda y.y\ /\ x]$
2. result of substitution : $((\lambda y.y)\ \lambda u.u)$

$$
\begin{array}{ll}
 & (\lambda x.x\ x)\ (\lambda y.y\ y) \\
\to & (x\ x)\ [(\lambda y.y\ y)\ /\ x] \\
= & (\lambda y.y\ y)\ (\lambda y.y\ y) \\
=_\alpha & (\lambda x.x\ x)\ (\lambda y.y\ y)
\end{array}
$$

Some Combinators

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Example

$$\begin{array}{lcl} I & := & \lambda x.x \\ K & := & \lambda x.\lambda y.x \\ K_2 & := & \lambda x.\lambda y.y \\ \Omega & := & (\lambda x_1.(x_1\ x_1))\ (\lambda x_2.(x_2\ x_2)) \\ Y & := & \lambda y_1.(\lambda x_1.(y_1\ (x_1\ x_1)))\ (\lambda y_2.(y_2\ (x_2\ x_2))) \end{array}$$

The I-combinator is the identity function.
$K, K2$ are projections, and Ω is non-terminating (diverging).
The Y-combinator is a *fixpoint-combinator*, it has the property that $Y\ e \sim_c e\ (Y\ e)$ holds. It can be used to express recursion.

Operational Semantics

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Small-Step Operational Semantics

of the

Pure Lazy Lambda-Calculus

Beta-Reduction

$$(\lambda x.e_1)\ e_2 \xrightarrow{\beta} e_1[e_2/x]$$

We write $e_1 \xrightarrow{\mathcal{C},\beta} e_2$, if
$e_1 = D[e_1']$, $e_1' \xrightarrow{\beta} e_2'$, and $e_2 = D[e_2']$

Operational Semantics: Normal-order reduction

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Normal order reduction : A deterministic evaluation strategy.
Reduce the outermost-leftmost beta-redex:
A *normal order reduction step* $\xrightarrow{no}$ is any β-reduction which is performed inside a reduction context:

$s \xrightarrow{no} t$ if s is of the form $R[(\lambda x.s_1)\ s_2]$
and $t = R[s_1[s_2/x]]$
for a reduction context R.
Grammar for *reduction contexts*: $R ::= [\cdot] \mid (R\ e)$

$\xrightarrow{no,+}$ transitive closure of $\xrightarrow{no}$

$\xrightarrow{no,*}$ reflexive-transitive closure

$\xrightarrow{no,i}$ exactly i normal order reduction steps.

Normal-order reduction: WHNF

Closed and $\xrightarrow{no}$-irreducible expressions of the (lazy) lambda calculus are exactly the abstractions.

Abstractions are also the weak head normal forms (WHNFs) of the (lazy) lambda calculus.

Converging Expression

Definition

An expression $e \in E_\lambda$ *converges* (or *successfully terminates*) (written as $e{\Downarrow}$) iff there exists a sequence of normal order reduction steps starting with e and ending in a WHNF:
$e{\Downarrow}$ iff $e \xrightarrow{no,*} e'$ where e' is a WHNF.

If $e{\Downarrow}$ does not hold, then we say that e *diverges* and write $e{\Uparrow}$.

Converging Expression

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Definition

An expression $e \in E_\lambda$ *converges* (or *successfully terminates*) (written as $e\Downarrow$) iff there exists a sequence of normal order reduction steps starting with e and ending in a WHNF: $e\Downarrow$ iff $e \xrightarrow{no,*} e'$ where e' is a WHNF.

If $e\Downarrow$ does not hold, then we say that e *diverges* and write $e\Uparrow$.

Examples

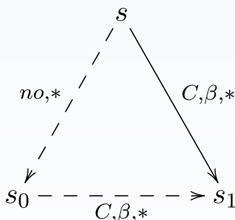
- $(\lambda x.x)\Downarrow$
- $((\lambda x.\lambda y.x)\ (\lambda z.z)\ (\lambda w.w))\Downarrow$:
  $\xrightarrow{no} (\lambda y.\lambda z.z)\ (\lambda w.w) \xrightarrow{no} \lambda z.z.$
- $((\lambda x.x)\ (\lambda y.y))\Downarrow$
- $((\lambda x.x\ x)\ (\lambda y.y\ y))\Uparrow$
- $(x\ (\lambda y.y)\Uparrow$

Standardization

In order to compute WHNFs from an expression, normal-order reduction is sufficient:

Proposition

Let e be an expression, such that $e \xrightarrow{C,\beta,*} e'$ where e' is a WHNF. Then $e{\Downarrow}$.



where s_0, s_1 are abstractions

($\xrightarrow{C,\beta,*}$ is β-reduction at any occurrence)

Semantics: Contextual Equivalence

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Definition *contextual equivalence*

Definition

Let $e_1, e_2 \in E_\lambda$.
Then e_1 and e_2 are *contextually equivalent*, written as $e_1 \sim e_2$ iff for all contexts $C \in C_\lambda$: $C[e_1]\Downarrow \iff C[e_2]\Downarrow$.

Semantics: Contextual Equivalence

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Properties of $\sim_c$

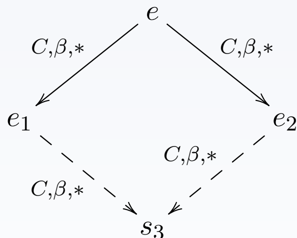
- Contextual preorder is a precongruence (i.e. it is a partial order and $s \leq_c t \implies C[s] \leq_c C[t]$)
- Contextual equivalence is a congruence (i.e. it is an equivalence relation and $s \sim_c t \implies C[s] \sim_c C[t]$)

Consequence: if $s \sim_c t$ it is possible to replace s by t anywhere in a program P to P' and $P \sim_c P'$ holds.

Divergent closed expressions are equivalent:
Let e_1, e_2 be two closed expressions with $e_1\Uparrow$ and $e_2\Uparrow$.
Then $e_1 \sim_c e_2$.

Connection to Rewriting and Confluence

The reduction $\xrightarrow{C,\beta,*}$ is confluen:



The confluence of $\xrightarrow{C,\beta,*}$ and the standardization of normal-order reduction imply:

Theorem

$\xrightarrow{\beta}$ is correct w.r.t. $\sim_c$, i.e. if $e_1 \xrightarrow{C,\beta} e_2$ then $e_1 \sim_c e_2$.

Connection to Rewriting

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$\xleftrightarrow{C,\beta,*}$ is called the conversion relation

- $\xleftrightarrow{C,\beta,*}$ is a congruence.
- $\xleftrightarrow{C,\beta,*} \quad \subseteq \quad \sim_c$.

However : $\xleftrightarrow{C,\beta,*} \quad \neq \quad \sim_c$.

Example:
$(\lambda x.(x\ x\ x)\ (\lambda x.(x\ x\ x))\Uparrow$, hence,
$\Omega \quad \sim_c \quad (\lambda x.(x\ x\ x))\ (\lambda x.(x\ x\ x))$.
But, $\Omega \overset{C,\beta,*}{\not\longleftrightarrow} \quad (\lambda x.(x\ x\ x))\ (\lambda x.(x\ x\ x))$,
since there is no common reduction successor.

Extension by Data Types

$LNAME$

as a Core Language of Haskell

Syntax of $LNAME$

$LNAME$ is
an extension of the lazy lambda calculus
by three constructs:

- **data**: there are primitives for constructing and deconstructing data
- **Sequentialization**: `seq`
- **Supercombinators** $\in \mathcal{F}$: recursive function definitions

Summary: Syntax of $LNAME$

For every $f \in \mathcal{F}$ there is a *definition* of the form $f\ x_1 \ldots\ x_n = e$ where x_i are variables and $FV(e) \subseteq \{x_1, \ldots, x_n\}$.

Syntax

Grammar for expressions of $LNAME$
where $x, x_i \in$ *Var*, $f \in \mathcal{F}$, and $c_{T,i} \in K_T$:

$$
\begin{array}{lcl}
e, e_i \in E_{LNAME} & ::= & x \mid \lambda x.e \mid f \mid (e_1\ e_2) \mid (c_{T,i}\ e_1\ \ldots\ e_{\mathrm{ar}(c_{T,i})}) \\
 & & \mid \texttt{case}_T\ e\ \texttt{of}\ \{pat_{T,1} \to e_1; \ldots; \\
 & & \qquad\qquad pat_{T,|T|} \to e_{|T|}\} \\
 & & \mid \texttt{seq}\ e_1\ e_2 \\
pat_{T,i} & ::= & (c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})})
\end{array}
$$

Reductions rules of $LNAME$

β	$(\lambda x.e_1)\ e_2 \ \rightarrow\ e_1[e_2/x]$
(case)	$(\mathtt{case}_T\ (c_{T,i}\ e'_1\ \ldots\ e'_{\mathrm{ar}(c_{T,i})})\ \mathtt{of}$ $\{\ldots;(c_{T,i}\ x_{i,1}\ \ldots\ x_{i,\mathrm{ar}(c_{T,i})}) \rightarrow e_i;\ldots\})$ $\rightarrow\ e_i[e'_1/x_{i,1},\ldots,e'_{\mathrm{ar}(c_{T,i})}/x_{i,\mathrm{ar}(c_{T,i})}]$
(seq)	$\mathtt{seq}\ v\ e\ \rightarrow\ e$, if v is a WHNF.
(SCβ)	$f\ e_1\ \ldots\ e_n\ \rightarrow\ e[e_1/x_1,\ldots,e_n/x_n]$, if $f\ x_1 \ldots\ x_n = e$ is the definition of $f \in \mathcal{F}$.

Syntax of $LNAME$

Extension by Data

- There is a finite nonempty set of *type constructors* $\mathcal{T}$, where for every $T \in \mathcal{T}$ there are pairwise disjoint finite nonempty sets of data constructors $D_T = \{c_{T,1}, \ldots c_{T,|T|}\}$.
- Every constructor has a fixed *arity* (a non-negative integer) denoted by $\mathrm{ar}(T)$ or $\mathrm{ar}(c_{T,j})$,
- The data constructor $c_{T,i}$ of arity n is applied to n expressions to form a *constructor application* $(c_{T,i}\ e_1 \ldots e_{\mathrm{ar}(c_{T,i})})$.
- Case-expressions for analysing and deconstructing data.

Examples

type constructor $Bool$ (of arity 0) with data constructors `True` and `False`

type constructor $List$ (of arity 1) with data constructors `Nil` (of arity 0) and `Cons` (of arity 2).

A list of three elements:

`(Cons True (Cons False (Cons True Nil)))`.

Recursive Definition of Functions

Extension We assume that there is a set $\mathcal{F}$ of *function symbols*, also called *supercombinators*
and that for every $f \in \mathcal{F}$ there is a *definition* of f of the form

$$f\ x_1\ \ldots\ x_n = e$$

where x_i are variables and e is an expression
s.t. $FV(e) \subseteq \{x_1, \ldots, x_n\}$.

The names from F are treated as constants, and so may also occur in the defining expressions e on the right hand side.

The number n in a definition $f\ x_1\ \ldots\ x_n = e$ is called the *arity* of f and written as $\mathrm{ar}(f)$.

Recursive Definition of Supercombinators

Examples map, head $\in \mathcal{F}$

map $\in \mathcal{F}$ is a recursive supercombinator:

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

head $\in \mathcal{F}$ is a non-recursive supercombinator:

$$\text{head}\ xs = \texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \Omega; (\texttt{Cons}\ y\ ys) \to y\}$$

WHNFs of $LNAME$

Definition

WHNFs of $LNAME$ are of the form

- A *functional weak head normal form* (FWHNF) in $LNAME$ is any abstraction and any expression of the form $(f\ e_1\ \ldots\ e_m)$ where $f \in F$ and $\mathrm{ar}(f) > m$.
- A *constructor weak head normal form* (CWHNF) is any expression of the form $(c_{T,i}\ e_1\ \ldots\ e_{\mathrm{ar}(c_{T,i})})$.
- There are $\xrightarrow{no}$-irreducible closed expressions that are not WHNFs, for example (`True True`);
  These are ruled out in typed calculi.