Functional Programming SS 2019 Summary - Definitions, Theorems, Lemmas, Algorithms

Defn. 2.1.1: $\sqsubseteq$ on basis domains $x \sqsubseteq y$ iff $x = \bot$ or $x = y$

Defn. 2.1.2: $\sqsubseteq$ on product domains if $D_1, \dots, D_n$ are, $n \geq 2$ are domains, then $D_1 \times \dots \times D_n$ is also a domain. $\sqsubseteq$ is defined as $(x_1, \dots, x_n) \sqsubseteq (y_1, \dots, y_n)$ iff $x_i \sqsubseteq y_i$ $\forall i$ $1 \leq i \leq n$

Lemma 2.1.3: reflexive partial order on product domains if every $\sqsubseteq_{D_i}$ is a reflexive partial order, then so is $\sqsubseteq_{D_1 \times \dots \times D_n}$

Defn. 2.1.4: $\sqsubseteq$ on function domains if D_1 and D_2 are domains and $f, g : D_1 \to D_2$ are functions, then $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ $\forall x \in D_1$
$\bot_{D_1 \to D_2}$ is the smallest element of the function domain

Lemma 2.1.5: reflexive partial order on function domains if $\sqsubseteq_{D_2}$ is a reflexive partial order, then so is $\sqsubseteq_{D_1 \to D_2}$

Defn. 2.1.6: extension and strictness: if $f : A \to B$ is a function, then $f' : D_\bot \to B$ is called an extension of f, s.t. $f'(x) = f(x)$ $\forall x \in A$ and $A_\bot = A \cup \{\bot\}$

A function $f : D_1 \times \dots \times D_n \to D$ is called strict iff $f(d_1, \dots, d_N) = \bot$ whenever any $d_i = \bot_{D_i}$, $1 \leq i \leq n$

Defn. 2.1.7: monotonic functions if $\sqsubseteq_{D_1}$ and $\sqsubseteq_{D_2}$ are partial orders on D_1 and D_2 respectively, a function $f : D_1 \to D_2$ is called monotonic iff $f(x) \sqsubseteq f(y)$ whenever $x \sqsubseteq y$

Lemma 2.1.8: Strictness $\Rightarrow$ monotonicity if $D_1, \dots, D_n$ are flat domains and a function $f : D_1 \times \dots \times D_n \to D$ is strict, then f is also monotonic

Defn. 2.1.9: chain if D is a domain and $\sqsubseteq$ a partial order on it, a non-empty set $\{d_1, d_2, \dots\}$ of countably many elements of D is called a chain if $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \dots$

Defn. 2.1.10: lub let D be a domain with a partial order $\sqsubseteq$ and S be a subset of D. An element d is called an upper bound of S if $d' \sqsubseteq d$ $\forall d' \in S$.
d is called a least upper bound if for every other upper bound e it holds that $d \sqsubseteq e$ (the lub of S is denoted as $\bigsqcup S$)

Lemma 2.1.11: lub for products and functions Let $D_1, \ldots, D_n, D, D'$ be domains

a) Let $S \subseteq D_1 \times \cdots \times D_n$ and for every $1 \le i \le n$ $S_i = \{d_i \mid$ there exist $d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$ with $(d_1, \ldots, d_i, \ldots, d_n) \in S\}$. Then

* $\bigsqcup S$ exists iff $\bigsqcup S_i$ exists $\forall i$ $1 \le i \le n$
* If $\bigsqcup S$, then $\bigsqcup S = (\bigsqcup S_1, \ldots, \bigsqcup S_n)$

b) Let S be a set of functions from D to D' and for every $i \in D$ $S_i = \{f(i) \mid f \in S\}$. Then

* $\bigsqcup S$ exists iff $\bigsqcup S_i$ exists $\forall i \in D$
* If $\bigsqcup S$ exists, then $(\bigsqcup S)(i) = \bigsqcup S_i$

Defn. 2.1.12: cpo a reflexive partial order $\sqsubseteq$ on a domain D is called complete iff

* D has a smallest element (denoted by $\bot_D$)
* for every chain S in D there exists a lub $\bigsqcup S \in D$

Thm. 2.1.13: completeness of the domains Let $D_1, \ldots, D_n$ be domains

(a) the flat domains $\mathbb{Z}_\bot$, $\mathbb{B}_\bot$, $\mathbb{K}_\bot$, and $\mathbb{F}_\bot$ are cpos (since every partial order with a smallest element and finite chains is a cpo)

(b) If $\sqsubseteq_{D_i}$ is complete on D_i $\forall i$ $1 \le i \le n$, then $\sqsubseteq_{D_1 \times \cdots \times D_n}$ is complete on $D_1 \times \cdots \times D_n$

(c) If $\sqsubseteq_{D_2}$ is a cpo on D_2, then $\sqsubseteq_{D_1 \to D_2}$ is a cpo on the set of functions from D_1 to D_2

Defn. 2.1.14: (Scott) continuity Let $\sqsubseteq_{D_1}$ be complete on D_1 and $\sqsubseteq_{D_2}$ be complete on D_2. A function $f: D_1 \to D_2$ is called continuous iff for every chain S in D_1 we have $f(\bigsqcup S) = \bigsqcup \{f(d) \mid d \in S\}$.

Thm. 2.1.15: continuity and monotonicity Let $\sqsubseteq_{D_1}$ and $\sqsubseteq_{D_2}$ be cpos on D_1 and D_2 respectively and $f: D_1 \to D_2$ be a function.

(a) f is continuous iff f is monotonic and for every chain S in D_1 we have $f(\bigsqcup S) \sqsubseteq \bigsqcup f(S)$

(b) If D_1 only has finite chains, then f is continuous iff f is monotonic.

Thm. 2.1.16: continuity of the function domains Let D_1 and D_2 be domains with respective cpos. Then $\langle D_1 \to D_2 \rangle$ is the appropriate function domain and $\sqsubseteq_{D_1 \to D_2}$ is a cpo on $\langle D_1 \to D_2 \rangle$

Thm. 2.1.17: (kleene's) fixpoint theorem let $\sqsubseteq$ be a cpo on a domain D and $f : D \to D$ be a function. Then, f has a least fixpoint s.t $lfp\ f = \bigsqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$

Defn. 2.2.1: domain lift let D be a domain with a relation $\sqsubseteq_D$. The lift of D is the set $\{ d \mid d \in D \} \cup \{ \bot_D \}$. The relation $\sqsubseteq_{D_\bot}$ is defined as $e \sqsubseteq_{D_\bot} e'$ iff $e = \bot_{D_\bot}$ or $e = d$, $e' = d'$ for $d, d' \in D$ and $d \sqsubseteq d'$.

Defn. 2.2.2: coalesced sum of domains let $D_1, \ldots, D_n$ be domains. The coalesced sum of $D_1, \ldots, D_n$ is defined as $D_1 \oplus \cdots \oplus D_n = \{ d^{D_1} \mid d^{D_1} \in D_1, d^{D_1} \neq \bot_{D_1} \} \cup \ldots \cup \{ d^{D_n} \mid d^{D_n} \in D_n, d^{D_n} \neq \bot_{D_n} \} \cup \{ \bot_{D_1 \oplus \cdots \oplus D_n} \}$. The relation $\sqsubseteq_{D_1 \oplus \cdots \oplus D_n}$ is defined as $e \sqsubseteq_{D_1 \oplus \cdots \oplus D_n} e'$ iff $e = \bot_{D_1 \oplus \cdots \oplus D_n}$ or $e = d^{D_i}$, $e' = d'^{D_i}$ and $d \sqsubseteq_{D_i} d'$ for some $1 \leq i \leq n$. Note: the coalesced sum of domains is also a domain.

Defn. 2.2.3: domain of a program for a Haskell program, let Con_n be the set of n-ary constructors ($\mathbb{Z} \cup \mathbb{B} \cup \mathbb{C} \cup \mathbb{F}$ are seen as nullary constructors). The domain of a program is then defined as the smallest domain that satisfies the equation

$$Dom = Functions \oplus Tuples_0 \oplus Tuples_2 \oplus \cdots \oplus Constructions_0 \oplus Constructions_1 \oplus \ldots$$

where

$$Functions = \langle Dom \to Dom \rangle$$
$$Tuples_0 = \{ () \}_\bot$$
$$Tuples_n = (Dom^n)_\bot$$
$$Constructions_n = (Con_n \times Dom^n)_\bot$$

Defn. 2.2.4: environment for a Haskell program with domain Dom, ρ is an environment if ρ is a partial function that maps variables to values of Dom, such that ρ is only defined for finitely many variables. If Var is the set of variables, then ρ is defined as $\rho : Var \to Dom$. The set of all environments in a program is denoted Env.

(An environment ρ with definitions $var_1, \ldots, var_n$ and $\rho(var_i) = d_i$ is also written as
$\rho = \{var_1, \ldots, var_n / d_1, \ldots, d_n\}$)

$\rho_1 + \rho_2$ is the environment that behaves as ρ_2 for all elements defined in ρ_2 and as ρ_1 otherwise.

Let ω be the initial environment that assigns strict extensions to all predefined Haskell operators and is undefined for all other variables.

Defn. 2.2.5: **simple Haskell programs** a simple Haskell program is a program without type synonyms and type classes and without the predefined Haskell lists, whose only declaration obeys the following grammar:

$$
\begin{aligned}
decl &\to var = exp \\
exp &\to var \\
&\mid constr \\
&\mid integer \\
&\mid float \\
&\mid char \\
&\mid (exp_1, \ldots, exp_n),\ n \geq 0 \\
&\mid (exp_1\ exp_2) \\
&\mid \text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \\
&\mid \text{let } var = exp \text{ in } exp' \\
&\mid \backslash var \to exp
\end{aligned}
$$

Defn. 2.2.6: **free variables in a Haskell expression** for a simple Haskell expression $\underline{exp}$, the set of free variables in $\underline{exp}$, denoted by $free(\underline{exp})$, is defined as follows:

$$free(var) = var$$

$$free(constr) = free(integer) = free(float) = free(char) = \emptyset$$

$$free((exp_1, \ldots, exp_n)) = free(exp_1) \cup \ldots \cup free(exp_n)$$

$$free((exp_1\ exp_2)) = free(exp_1) \cup free(exp_2)$$

$$free(\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3) = free(exp_1) \cup free(exp_2) \cup free(exp_3)$$

$$free(\text{let } var = exp \text{ in } exp') = (free(exp) \cup free(exp')) \setminus \{var\}$$

$$free(\backslash var \to exp) = free(exp) \setminus \{var\}$$

Defn. 2.2.7: Semantics of simple Haskell programs Let Dom be the domain of a simple Haskell program, $var = exp$ the declaration of the program, and Exp the set of simple Haskell expressions. We define the function $Val : Exp \to Env \to Dom$ that assigns a value to every expression in every environment (that is defined on all of its free variables). The semantics of an expression exp' that has no free variables other than var and the ones predefined in Haskell is then defined as

$$Val\ [\![\text{let } var = exp \text{ in } exp']\!]_\rho$$

For an environment ρ, the function Val is defined as follows, where $constr_0 \in Con_0$ and $constr_n \in Con_n$ for $n > 0$.

$$Val\ [\![var]\!]\rho = \rho(var)$$

$$Val\ [\![constr_0]\!]\rho = constr_0 \text{ in } Constructions_0 \text{ in } Dom$$

$$Val\ [\![constr_n]\!]\rho = f \text{ in } Functions \text{ in } Dom, \text{ where}$$
$$f\ d_1 \ldots d_n = (constr_n, d_1, \ldots, d_n) \text{ in } Constructions_n \text{ in } Dom$$

$$Val\ [\![(exp_1, \ldots, exp_n)]\!]_\rho = (Val\ [\![exp_1]\!]_\rho \ldots Val\ [\![exp_n]\!]_\rho)$$
in $Tuples_n$ in Dom, $n = 0$ or $n \geq 2$

$$Val\ [\![(exp)]\!]_\rho = Val\ [\![exp]\!]_\rho$$

$$Val\ [\![(exp_1\ exp_2)]\!]_\rho = f(Val\ [\![exp_2]\!]_\rho) \text{ where } Val\ [\![exp_1]\!]_\rho = f \text{ in Functions in Dom}$$

$$Val\ [\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3]\!]_\rho = \begin{cases} Val\ [\![exp_2]\!]_\rho, \text{ if } Val\ [\![exp_1]\!]_\rho = True \text{ in Constructions 0 in Dom} \\ Val\ [\![exp_3]\!]_\rho \text{ if } Val\ [\![exp_1]\!] = False \text{ in Constructions 0 in Dom} \\ \perp \text{ otherwise} \end{cases}$$

$$Val\ [\![\text{let } var = exp \text{ in } exp']\!]_\rho = Val\ [\![exp']\!](\rho + \{var // f\rho f\}) \text{ where } f : Dom \to Dom \text{ with } f(d) = Val\ [\![exp]\!](\rho + \{var/d\})$$

$$Val\ [\![\backslash var \to exp]\!]_\rho = f \text{ in Functions in Dom where } f(d) = Val\ [\![exp]\!](\rho + \{var/d\})$$

Defn. 2.2.8: predefined functions The following functions are predefined for a Haskell program with constructors Con_n ($n \geq 0$), where k is the maximum arity of the program functions, length of tuples, and number of declarations:

```
bot :: a
isConstr :: type -> Bool
    for every constr_n ∈ Con_n, where
    constr :: type_1 -> ... -> type_n -> type
argOfConstr :: type -> (type_1, ..., type_n)
    for every constr ∈ Con_n, where
    constr :: type_1 -> ... -> type_n -> type
```

$$\text{isa}_{n\text{-tuple}} :: (a_1, \ldots, a_n) \to \text{Bool}$$
for every $n \in \{0, 2, 3, \ldots, k\}$

$$\text{sel}_{n,i} :: (a_1, \ldots, a_n) \to a_i$$
for every $2 \le n \le k$, $1 \le i \le n$

The initial environment w is extended to a new initial environment w_{tr} in which the semantics of the predefined functions is given as follows:

$$w_{tr}(\text{Bot}) = \bot$$

$$w_{tr}(\text{isa}_{\text{constr}})(d) = \begin{cases} \text{True in Constructors}_0 \text{ in Dom} \\ \quad \text{if } d = (\text{constr}, d_1, \ldots, d_n) \text{ in Constructors}_n \text{ in Dom} \\ \text{False in Constructors}_0 \text{ in Dom} \\ \quad \text{if } d = (\text{constr}', d_1, \ldots, d_n) \text{ in Constructors}_n \text{ in Dom} \\ \quad \text{and constr} \neq \text{constr}' \\ \bot \text{ otherwise} \end{cases}$$

$$w_{tr}(\text{arg}_i \text{ of constr})(d) = \begin{cases} (d_1, \ldots, d_n) \text{ in Tuples}_n \text{ in Dom} \\ \quad \text{if } d = (\text{constr}, d_1, \ldots, d_n) \text{ in Constructors}_n \\ \quad \text{in Dom}, \ n \neq 1 \\ d_1 \text{ if } d = (\text{constr}, d_1) \text{ in Constructors}_1 \\ \quad \text{in Dom} \\ \bot \text{ otherwise} \end{cases}$$

$$w_{tr}(\text{isa}_{n\text{-tuple}})(d) = \begin{cases} \text{True in Constructors}_0 \text{ in Dom} \\ \quad \text{if } d = (d_1, \ldots, d_n) \text{ in Tuples}_n \text{ in Dom} \\ \bot \text{ otherwise} \end{cases}$$

$$w_{tr}(\text{sel}_{n,i})(d) = \begin{cases} d_i \text{ if } d = (d_1, \ldots, d_n) \text{ in Tuples}_n \text{ in Dom} \\ \bot \text{ otherwise} \end{cases}$$

Defn. 2.2.9: complex Haskell programs a complex Haskell program is a program without type synonyms, type classes, infix declarations, and without the predefined Haskell lists. Defining equations for functions need to be grouped together and the program can only have pattern declarations in which there is a variable on the left hand side. The program can also not have where blocks, guards, and no patterns with @.

Defn. 2.2.10: variable dependencies, declaration splits let $P = \{var_1 = exp_1; \ldots; var_n = exp_n\}$ be a list of declarations in a complex Haskell program. Let $var_i \gtrsim_P var$ be defined as "var_i depends on var" if $var_i = var$ or a variable var' with $var' \gtrsim_P var$ appears free in exp_i. Otherwise, let $var_i \sim_P var$ if $var_i \gtrsim_P var$ and $var \gtrsim_P var_i$. Let also $var_i \gtrdot_P var$ if $var_i \gtrsim var$ and $var \not\gtrsim var_i$.

We have that $P_1, \ldots, P_k$ with $P_i = \{var_{i,1} = exp_{i,1}; \ldots; var_{i,n_i} = exp_{i,n_i}\}$ is a separation of P if

$$P_1 \uplus \ldots \uplus P_k = P$$

$$P_i \neq \emptyset \quad \forall i\ 1 \leq i \leq k$$

$$var_{i,1} \sim_P \ldots \sim_P var_{i,n_i} \quad \forall i\ 1 \leq i \leq k$$

if $var_{i,j} \gtrsim_P var_{i',j'}$, then $i \geq j$

Defn. 2.2.11: transformation to simple Haskell programs the following rules can be used to transform a complex Haskell program or expression into simple Haskell expressions

(1) Function declarations to pattern declarations

$$\frac{\text{var } pat^1_1 \ldots pat^1_n = exp^1; \ldots; \text{var } pat^k_1 \ldots pat^k_n = exp^k}{\begin{array}{l}\text{var} = \backslash x_1 \ldots x_n \to \text{case } (x_1, \ldots, x_n) \text{ of } \{(pat^1_1, \ldots, pat^1_n) \to exp_1 \\ \qquad \vdots \\ \qquad (pat^k_1, \ldots, pat^k_n) \to exp_k\}\end{array}}$$

where $x_1, \ldots, x_n$ are variables and the above list is maximal, i.e. there are no other function declarations for var.

(2) Conversion of lambda expressions with multiple patterns

$$\frac{\backslash pat_1 \ldots pat_n \to exp \qquad n \geq 2}{\backslash pat_1 \to (\backslash pat_2 \to \ldots (\backslash pat_n \to exp) \ldots)}$$

(3) conversion of lambda patterns to case

$$\frac{\backslash pat \to exp}{\backslash var \to \text{case } var \text{ of } pat \to exp}$$

(4) translation of case to match

$$\frac{\begin{array}{l}\text{case } exp \text{ of } \{pat_1 \to exp_1; \\ \qquad \vdots \\ \qquad pat_n \to exp_n\}\end{array}}{\begin{array}{l}\text{match } pat_1\ exp\ exp_1 \\ \quad (\text{match } pat_2\ exp\ exp_2 \\ \qquad \ldots \\ \qquad (\text{match } pat_n\ exp\ exp_n\ bot) \ldots)\end{array}}$$

(5) match of variables

$$\frac{\text{match } var\ exp\ exp_1\ exp_2}{(\lambda var \to exp_1)\ exp}$$

(6) match of the joker pattern

$$\frac{\text{match } \_\ exp\ exp_1\ exp_2}{exp_1}$$

(7) match of constructors

$$\frac{\text{match } (constr\ pat_1 \ldots pat_n)\ exp\ exp_1\ exp_2}{\text{if } (\text{isa}_{constr}\ exp) \text{ then } (\text{match } (pat_1, \ldots, pat_n)\ \text{arg}_{constr}\ exp)\ exp_1\ exp_2) \text{ else } exp_2}$$

(8) match of the empty tuple

$$\frac{\text{match } ()\ exp\ exp_1\ exp_2}{\text{if } (\text{isa}_{0\text{-tuple}}\ exp) \text{ then } exp_1 \text{ else } exp_2}$$

(9) match of a non-empty tuple

$$\frac{\text{match } (pat_1, \ldots, pat_n)\ exp\ exp_1\ exp_2 \qquad n \geq 2}{\begin{array}{l}\text{if } (\text{isa}_{n\text{-tuple}}\ exp) \\ \text{then match } pat_1\ (\text{sel}_{n,1}\ exp) \\ \qquad [\text{match } pat_2\ (\text{sel}_{n,2}\ exp) \\ \qquad \ldots\ (\text{match } pat_n\ (\text{sel}_{n,n}\ exp)\ exp_1\ exp_2) \ldots \\ \qquad exp_2) \\ \qquad exp_2 \\ \text{else } exp_2\end{array}}$$

(10) split of declarations

$$\frac{\texttt{let } P \texttt{ in } \underline{exp}}{\texttt{let } P_1 \texttt{ in let } P_2 \texttt{ in } \cdots \texttt{ let } P_k \texttt{ in } \underline{exp}}$$

where $P_1, \ldots, P_k$ is a separation of P and $k \geq 2$

(11) conversion of multiple declarations into a single declaration

$$\frac{\{ \underline{var_1} = \underline{exp_1} ; \cdots ; \underline{var_n} = \underline{exp_n} \}}{(\underline{var_1}, \ldots, \underline{var_n}) = (\underline{exp_1}, \ldots, \underline{exp_n})}$$

where $n \geq 2$, $var_i \neq var_j$ for $i \neq j$ and $var_1 \sim_p \cdots \sim_p var_n$

(12) declaration of multiple variables

$$\frac{\texttt{let } (\underline{var_1}, \ldots, \underline{var_n}) = \underline{exp} \texttt{ in } exp'}{\texttt{let } \underline{var} = \texttt{match } (\underline{var_1}, \ldots, \underline{var_n}) \texttt{ with } (sel_{n,1}\ \underline{var}, \ldots, sel_{n,n}\ \underline{var}) \rightarrow \underline{exp} \texttt{ end} \\ \texttt{in match } (\underline{var_1}, \ldots, \underline{var_n}) \texttt{ with } (sel_{n,1}\ \underline{var}, \ldots, sel_{n,n}\ \underline{var}) \rightarrow exp' \texttt{ end}}$$

where $n \geq 2$ and $\underline{var}$ is a new variable

Thm 2.2.12: termination, confluence, and outcome of the transformation

let $\underline{exp}$ be a complex Haskell outcome. Then

(a) every application of the rules in Defn. 2.2.11 terminates, i.e. there is an expression $\underline{exp}_{tr}$ into which $\underline{exp}$ can be translated

(b) with the exception of rule (10), all rules are confluent, i.e. their repeated application

is well-defined.

(c) The expression expr is a simple Haskell expression

Defn. 2.2.13: Semantics of complex Haskell programs for a complex Haskell program, let P be a sequence of pattern and function declarations. The semantics of a Haskell expression exp that has no free variables other than the ones defined in P and the ones predefined in Haskell is defined as

$$\mathrm{Val}\,[\![(\mathtt{let}\ P\ \mathtt{in}\ exp)_{tr}]\!]_{\omega_{tr}}$$

Defn. 3.1.1. lambda terms let C be a set of constants and V a countably infinite set of variables. The set of lambda terms is defined as the smallest set so that:

* $C \subseteq \Lambda$
* $V \subseteq \Lambda$
* $(t_1\ t_2) \in \Lambda$ if $t_1, t_2 \in \Lambda$
* $\lambda x.t \in \Lambda$ if $x \in V$ and $t \in \Lambda$

Defn. 3.1.2: free variables in a lambda term for every lambda term $t \in \Lambda$ we define $free(t) \subseteq V$ as the set of its free variables as follows:

* $free(c) = \emptyset$ for every $c \in C$
* $free(x) = \{x\}$ for every $x \in V$
* $free(t_1\ t_2) = free(t_1) \cup free(t_2)$ for every $t_1, t_2 \in \Lambda$
* $free(\lambda x.t) = free(t) \setminus \{x\}$ for every $x \in V$, $t \in \Lambda$

A term t is called closed if $\text{free}(t) = \emptyset$.

Defn. 3.1.3: substitution on lambda terms for every $r, t \in \Lambda$ and every $x \in V$, we define $r[x/t]$ as follows:

* $x[x/t] = t$
* $y[x/t] = y$ for every $y \in V$ with $y \neq x$
* $c[x/t] = c$ for every $c \in C$
* $(r_1\ r_2)[x/t] = (r_1[x/t]\ r_2[x/t])$ for every $r_1, r_2 \in \Lambda$
* $(\lambda x.\ r)[x/t] = \lambda x.\ r$
* $(\lambda y.\ r)[x/t] = \lambda y.(r[x/t])$ if $y \neq x$ and $y \notin \text{free}(t)$
* $(\lambda y.\ r)[x/t] = \lambda y'.(r[y/y'][x/t])$ if $y \neq x$, $y \in \text{free}(t)$, $y' \notin \text{free}(r) \cup \text{free}(t)$

Defn. 3.2.1: α-reduction the relation $\to_\alpha \subseteq \Lambda \times \Lambda$ is the smallest relation with

* $\lambda x.\ t \to_\alpha \lambda y.\ t[x/y]$ if $y \notin \text{free}(t)$
* if $t_1 \to_\alpha t_2$, then also $(t_1\ r) \to_\alpha (t_2\ r)$, $(r\ t_1) \to_\alpha (r\ t_2)$, and $\lambda y.\ t_1 \to_\alpha \lambda y.\ t_2$ for all $r \in \Lambda$, $y \in V$

Defn. 3.2.2: β-reduction the relation $\to_\beta \subseteq \Lambda \times \Lambda$ is the smallest relation with

* $(\lambda x.\ t)\ r \to_\beta t[x/r]$
* if $t_1 \to_\beta t_2$, then also $(t_1\ r) \to_\beta (t_2\ r)$,

$(\tau\, t_1) \to_\beta (\tau\, t_2)$, and $(\lambda y.t_1) \to_\beta (\lambda y.t_2)$ for all $\tau \in \Lambda, y \in V$.

Defn. 3.2.3: transitive-reflexive closure, normal form, confluence

Let $\to$ be a relation over some set.

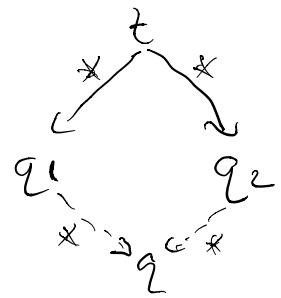
(a) The relation $\to^*$ (the transitive-reflexive closure of $\to$) is the smallest relation so that for all $t_1, t_2, t_3 \in N$, it holds that

* if $t_1 \to t_2$, then $t_1 \to^* t_2$
* if $t_1 \to t_2 \to^* t_3$, then $t_1 \to^* t_3$
* $t_1 \to^* t_1$

(b) An object $q \in N$ is called a $\to$-normal form iff there is no object $q' \in N$ with $q \to q'$. The object q is called a normal form of an object t iff $t \to^* q$ and q is a normal form.

(c) The relation $\to$ is called confluent if for all $t, q_1, q_2 \in N$ we have:

if $t \to^* q_1$ and $t \to^* q_2$, then there is a $q \in N$ with $q_1 \to^* q$ and $q_2 \to^* q$



Lemma 3.2.4: confluence implies a single normal form

Let $\to$ be a confluent relation over a set N. Then every object $t \in N$ has at most one normal form.

Thm. 3.2.5: confluence of the λ calculus with β rules

$\to_\beta$ is confluent, i.e. if $t \to_\beta^* q_1$ and $t \to_\beta^* q_2$, then there are $q_1', q_2' \in \Lambda$ with $q_1 \to_\beta^* q_1'$, $q_2 \to_\beta^* q_2'$, and $q_1' \to_\alpha^* q_2'$

Defn. 3.2.6: δ-reduction a collection of rules δ of the form $c\, t_1 \ldots t_n \to r$ with $c \in C, t_1, \ldots, t_n \in \Lambda$ is called a delta ruleset if the following conditions are fulfilled:

* $t_1, \ldots, t_n, r$ are closed λ-terms
* the t_i are in $\to_\beta$-normal form and contain no left-hand side of a rule from δ
* in δ there are no two different rules $c\, t_1 \ldots t_n \to r$ and $c\, t_1 \ldots t_m \to r'$ with $m \geq n$

For such a set δ we define the relation $\to_\delta$ as the smallest relation with

* $\ell \to_\delta r$ for all $\ell \to r \in \delta$
* if $t_1 \to_\delta t_2$, then also $(t_1\, r) \to_\delta (t_2\, r)$, $(r\, t_1) \to_\delta (r\, t_2)$, and $\lambda y.t_1 \to_\delta \lambda y.t_2$

for all $r \in \Lambda, \forall \in V$

The combination of β- and δ- reduction is denoted by $\to_{\beta\delta}$, i.e. we have $\to_{\beta\delta} = \to_\beta \cup \to_\delta$

Thm. 3.2.7: confluence of the λ calculus with β- and δ-rules $\to_{\beta\delta}$ is confluent, i.e. if $t \to^*_{\beta\delta} q_1$ and $t \to^*_{\beta\delta} q_2$, then there are $q, q' \in \Lambda$ with $q_1 \to^*_{\beta\delta} q$, $q_2 \to^*_{\beta\delta} q'$, and $q \to^*_\alpha q'$

Defn. 3.3.1: weak head normal form a term is in weak head normal form (WHNF) iff it is a normal form or has any of the following forms:

* $\lambda x.t$ for an arbitrary $t \in \Lambda$
* $c\, t_1 \dots t_n$ for arbitrary $t_1, \dots, t_n \in \Lambda$ and every $c \in \mathbb{C}$ for which there are no rules in δ (i.e. c is a constructor)
* $x\, t_1 \dots t_n$ for arbitrary $t_i \in \Lambda$ and $x \in V$

Defn. 3.3.2: weak head normal order reduction the WHNO reduction on λ terms is defined as $t \to r$ iff t is not in WHNF and $t \to_{\beta\delta} r$, where the reduction is performed using a leftmost outermost strategy

Defn. 3.3.3: translation of simple Haskell to λ terms Let Exp be the set of simple Haskell expressions, let C_0 be the symbols of the predefined operators in Exp (eg. +, mod, sqrt, etc.) and let Λ be the set of λ terms over the constants $C = C_0 \cup \{Con \cup \{tuple_n \mid n=0 \text{ or } n \geq 2\} \cup \{if, fix\}$.

We define the function $Lam : Exp \to \lambda$ that translates any simple Haskell expression to a λ term as follows:

$$Lam(var) = var$$
$$Lam(c) = c$$
$$Lam((exp_1, .., exp_n)) = tuple_n\ Lam(exp_1) .. Lam(exp_n)$$
$$Lam((exp)) = Lam(exp)$$
$$Lam((exp_1\ exp_2)) = (Lam(exp_1)\ Lam(exp_2))$$
$$Lam(\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3) =$$
$$\quad \text{if } Lam(exp_1)\ Lam(exp_2)\ Lam(exp_3)$$
$$Lam(\text{let } var = exp \text{ in } exp') =$$
$$\quad Lam(exp')[var / fix(\lambda var.\ Lam(exp))]$$
$$Lam(\backslash var \to exp) = \lambda var.\ Lam(exp)$$

Defn. 3.3.4: translation of Haskell to the λ calculus Let P be the sequence of function and pattern declarations in a complex Haskell program and let exp be a complex expression without any free variables other than the ones defined in P or predefined in Haskell. Let C_0 be the symbols of the predefined operators in Haskell (i.e. +, not, sqrt, etc.) and C_{on} the constructors in the program. We then define

$$C = C_0 \cup C_{on} \cup \{bot, if, fix\} \cup$$

$$\{isa_{n\text{-tuple}} \mid n \in \{0,2,3,\ldots\}\} \cup$$
$$\{isa_{constr} \mid constr \in Con\} \cup$$
$$\{arg\text{-}of_{constr} \mid constr \in Con\} \cup$$
$$\{sel_{n,i} \mid n \geq 2,\ 1 \leq i \leq n\} \cup$$
$$\{tuple_n \mid n = 0 \text{ or } n \geq 2\}$$

The translation of exp then results in a λ term over the set of constants C and is defined as

$$Tran(P, exp) = Lam((let\ P\ in\ exp)_{tr})$$

Defn. 3.3.5: δ-rules for Haskell programs Let Con be the set of constructors in a Haskell program, where Con_n is the set of constructors of arity n. Let δ_0 be the rules for the predefined Haskell operators, i.e. δ_0 contains rules such as plus 1 2 → 3, not True → False, etc. The set δ is then defined as follows:

$$\delta = \delta_0 \cup$$
$$\{ bot \to bot$$
$$if\ True \to \lambda x y.x$$
$$if\ False \to \lambda x y.y$$
$$fix \to \lambda f.f(fix\ f)\} \cup$$
$$\{isa_{n\text{-tuple}}(tuple_n\ t_1 \ldots t_n) \to True \mid n \in \{0,2,3,\ldots\}, t_i \in \Lambda \text{ and closed}\}$$
$$\{isa_{constr}(constr\ t_1 \ldots t_n) \to True \mid constr \in Con_n, t_i \in \Lambda \text{ and closed}\}$$

$\{ isacon_{str}\ (constr'\ t_1 \ldots t_n) \to False \mid constr' \in Con_m, constr = constr' \}$

$\{ argof_{constr}\ (constr\ t_1 \ldots t_n) \to tuple_n\ t_1 \ldots t_n \mid constr \in Con_n \}$

$\{ argof_{constr}\ (constr\ t) \to t \mid constr \in Con_1 \}$

$\{ sel_{n,i}\ (tuple_n\ t_1 \ldots t_n) \to t_i \mid n \geq 2,\ 1 \leq i \leq n \}$

Defn. 3.3.6: *implementation of Haskell* For a complex Haskell program with constructors Con, let δ be the set of associated rules. Let P be the sequence of pattern and function declarations in the program and let $\underline{exp}$ be a complex expression that has no free variables other than the ones defined in P or predefined in Haskell. The evaluation of $\underline{exp}$ in P is then performed through WHNO-reduction using the delta rules in δ on the λ term $Tran(P, \underline{exp})$.

Thm. 3.3.7: *correctness of the implementation* Let P and $\underline{exp}$ be defined as in Defn. 3.3.6, such that both P and $\underline{exp}$ are type correct. If $Tran(P, \underline{exp}) \to^* q$ for a λ term q in WHNF, then $Val[\![(let\ P\ in\ exp)_{tr}]\!]_{w_{tr}} = Val[\![q]\!]w_{tr}$. If the WHNO-reduction of $Tran(P, exp)$ does not terminate, then $Val[\![(let\ P\ in\ exp)_{tr}]\!]w_{tr} = \underline{\perp}$

Defn 4.11: type schemas and type assumptions a type schema is formed with the following grammar:

$$\begin{aligned}
\text{typeschema} \to\ & (\text{tycstr typeschema}_1, \ldots, \text{typeschema}_n), n \geq 0 \\
& \mid (\text{typeschema}_1 \to \text{typeschema}_2) \\
& \mid (\text{typeschema}_1, \ldots, \text{typeschema}_n), n \geq 0 \\
& \mid \text{var} \\
& \mid \forall \text{var}.\ \text{typeschema}
\end{aligned}$$

A typeschema is thus a type type, but where type variables can be universally quantified. For a type schema τ with free variables $a_1, \ldots, a_n$, $\forall\tau$ is the type schema $\forall a_1, \ldots, a_n.\ \tau$

A type assumption A is a (potentially partial) function on $V \cup C$ in the set of type schemas. A type assumption A with $A(x_i) = \tau_i$ $(1 \leq i \leq n)$ that is undefined on other arguments is also written as $\{x_1 :: \tau_1, \ldots, x_n :: \tau_n\}$.

The initial type assumption A_0 is defined as follows, where constr is a user-defined constructor in a data type declaration:

$A_0(x) = \forall a.\ a$ for all $x \in V$

$A_0(c) =$ the predefined type in Haskell where all free variables are universally quantified, for all $c \in C_0$

$A_0(\text{constr}) = \forall(\text{type}_1 \to \cdots \to \text{type}_n \to \text{tycstr}(a_1 \ldots a_m))$

$A_0(\text{Bot}) = \forall a.\, a$

$A_0(\text{if}) = \forall a.\, \text{Bool} \to a \to a \to a$

$A_0(\text{fix}) = \forall a.\, (a \to a) \to a$

$A_0(\text{isa}_{\text{constr}}) = \forall((\text{tyconstr } a_1 \ldots a_m) \to \text{Bool})$

$A_0(\text{arg of}_{\text{constr}}) = \forall((\text{tyconstr } a_1 \ldots a_m) \to (\text{type}_1 \ldots \text{type}_n))$

$A_0(\text{isa}_{n\text{-tuple}}) = \forall a_1 \ldots a_n.\, (a_1, \ldots, a_n) \to \text{Bool}$

$A_0(\text{sel}_{n,i}) = \forall a_1 \ldots a_n.\, (a_1, \ldots, a_n) \to a_i$

$A_0(\text{tuple}_n) = \forall a_1 \ldots a_n.\, a_1 \to \ldots a_n \to (a_1, \ldots, a_n)$

Here

data tyconstr $a_1 \ldots a_m = \ldots \mid$ constr type$_1 \ldots$ type$_n \mid \ldots$

For two type assumptions A and A', we define $A + A'$ as $(A + A')(x) = A'(x)$ if $A'(x)$ is defined and as $A(x)$ otherwise.

Defn. 4.2.1. **shallow type schemas** a type schema is called shallow iff it is of the form $\forall a_1 \ldots a_n.\, T$ and T does not contain any quantifiers (i.e. T is a type)

Defn. 4.2.2. **unification, most general unifier** let Θ be a substitution (i.e. a mapping of type variables on types, where only infinitely many type variables are not mapped to themselves). Θ is a unifier of two types T_1 and T_2

if $T_1\theta = T_2\theta$. A substitution θ' is called a most general unifier (mgu) of T_1 and T_2 if the following conditions are met:

* θ' is a unifier of T_1 and T_2
* for every unifier θ of T_1 and T_2 there is a substitution δ with $\theta = \theta'\delta$

We then write $\theta' = mgu(T_1, T_2)$

Defn. 4.2.3: **type inference algorithm W** for every type assumption A and every λ term $t \in \Lambda$, $W(A,t)$ calculates a pair of a substitution on type variables and a type or the calculation fails due to a unification error.

* $W(A + \{c :: \forall a_1, \ldots, a_n . T\}) = (id, T[a_1/b_1, \ldots, a_n/b_n]), b_1, \ldots, b_n$ new vars
* $W(A, \lambda x.t) = (\theta, b\theta \to T)$ where $W(A + \{x :: b\}, t) = (\theta, T)$, b is a new var
* $W(A, (t_1\ t_2)) = (\theta_1\theta_2\theta_3, b\theta_3)$ where
  $W(A, t_1) = (\theta_1, T_1)$
  $W(A\theta_1, t_2) = (\theta_2, T_2)$
  $\theta_3 = mgu(T_1\theta_2, T_2 \to b)$, b is a new var.

Thm. 4.2.4: **correctness of the W algorithm** Let t be a λ term over the constants C from Defn. 3.3.4, where t is type correct according to the W algorithm (i.e. it holds that $W(A_0, t) = (\theta, T)$ for a type T and a substitution θ). Let $t \to_\beta^* t'$,

where δ is the delta set defined in Defn. 3.3.5. Then, $w(A_\sigma, t') = (\theta', \tau')$ for a substitution θ'.