Goal: Present an automatic translation from complex to simple Haskell. This is used for

A2 definition of semantics: The semantics of a complex program is defined to be the semantics of the corresponding simple program.

A3 implementing Haskell: First compile complex to simple Haskell, then execute it.

A4 type-checking: First compile complex to simple Haskell, then type-check it.

Main task: develop a translation that gets rid of pattern matching.

Idea: Extend the set of pre-defined functions by some extra functions for pattern matching. Define semantics of these functions directly.

Pre-defined functions: (Slide 44)
- bot should be a pre-defined constant with semantics 1
  could be defined in Haskell by:
  \[
  \text{bot :: a} \\
  \text{bot = bot}
  \]
- isa constr pre-defined function which checks whether its argument is built with the data constructor constr.
constructor \texttt{constr}

Could be defined easily in Haskell, but not in Simple Haskell:

\[
data \text{List}\ a = \text{Nil} \mid \text{Cons} \ a \ (\text{List}\ a)
\]

\[
is\ a :: (\text{List}\ a) \rightarrow \text{Bool}
\]

\[
is\ \text{Cons} \ (\text{Cons} \ x \ y) = \text{True}
\]

\[
is\ \text{Cons} \ \text{Nil} = \text{False}
\]

\* \text{arg of constr} : \text{if the argument is built with constr, then arg of constr returns the type of its arguments}

\[
\text{arg of Cons} :: (\text{List}\ a) \rightarrow (a, \text{List}\ a)
\]

\[
\text{arg of Cons} \ (\text{Cons} \ x \ y) = (x, y)
\]

\* \text{isa n-tuple} (for all \( n \in \{0, 2, 3, \ldots\} \)) to recognize tuples of \( n \) components

Can easily be implemented in non-simple Haskell:

\[
is\ a_{-n-tuple} :: (a_1, \ldots, a_n) \rightarrow \text{Bool}
\]

\[
is\ a_{-n-tuple} \ (x_1, \ldots, x_n) = \text{True}
\]

\[
is\ (\_o-tuple) :: () \rightarrow \text{Bool}
\]

\[
is\ (\_o-tuple) = \text{True}
\]

\textbf{Note:} \text{isa n-tuple} \ (\text{Semantics is } \bot).

FP 2016 Seite 2
\( \text{sel}_{n,i} \): selects the \( i \)-th component of an \( n \)-tuple

Can easily be defined in non-simple Haskell:

\[
\text{e.g.: } \, \text{Sel}_{3,2} \, \colon (a_1, a_2, a_3) \rightarrow a_2
\]

\[
\text{Sel}_{3,2} \, (x_1, x_2, x_3) = x_2
\]

Now we can translate every complex \( H \)-program/expression into an equivalent simple one that uses these pre-defined functions.

**Def 2.28 (Pre-defined Functions)**

For a Haskell program with the constructors \( \text{Con}_n \,(n \geq 0) \) the function on Slide 41 are pre-defined. Here, \( k \) is the maximum of the lengths of tuples in our program, the names of functions in our program, and the number of declarations in our program.

The initial environment \( \omega_0 \) is extended to a new initial environment \( \omega_T^r \) where the semantics of the new pre-defined functions is defined as follows:

\[
\omega_T^r (\text{bot}) = 1
\]

\[
\omega_T^r (\text{isa}_\text{constr})(d) = \begin{cases} 
\text{True in Constructions}_n \text{ in Dom,} \\
\text{if } d = (\text{constr}, d_1, \ldots, d_n) \text{ in Constructions}_n \text{ in Dom} \\
\text{False in Constructions}_0 \text{ in Dom,} \\
\text{if } d = (\text{constr}, d_1, \ldots, d_m) \text{ in Constructions}_m \text{ in Dom and constr} \neq \text{constr'} \\
1, \text{ otherwise} \\
\end{cases}
\]

\[
\omega_T^r (\text{argof}\text{constr})(d) = \begin{cases} 
\text{(d_1, \ldots, d_n) in Tuples}_n \text{ in Dom,} \\
\text{if } d = (\text{constr}, d_1, \ldots, d_n) \text{ in Constructions}_n \text{ in Dom and } n \neq 1 \\
1, \text{ otherwise} \end{cases}
\]
\[ w_{\forall x} (\text{Constr}) (d) = \begin{cases} d_n, & \text{if } d = (\text{Constr}, d_n) \text{ in } \text{Constructions} \text{ in } \text{Dom} \\ \perp, & \text{otherwise} \end{cases} \]

\[ w_{\forall x} (\text{lsu}_{n,tuple}) (d) = \begin{cases} \text{True in } \text{Constructions} \text{ in } \text{Dom}, & \text{if } d = (d_1, \ldots, d_n) \text{ in } \text{Tuples} \text{ in } \text{Dom} \\ \perp, & \text{otherwise} \end{cases} \]

\[ w_{\forall x} (\text{sel}_{n,i}) (d) = \begin{cases} d_i, & \text{if } d = (d_1, \ldots, d_n) \text{ in } \text{Tuples} \text{ in } \text{Dom} \\ \perp, & \text{otherwise} \end{cases} \]

Now, we want to introduce our transformation from complex to simple Haskell.

**Full Haskell \supset Complex Haskell \supset Simple Haskell**

**Def 2.2.9 (Complex Haskell)**

A complex Haskell program is a program

1. without type synonyms
2. without type classes
3. without infix declarations
4. without pre-defined lists.
5. Moreover, defining equations for the same function should be beside each other.
6. no pattern declarations except declarations with a variable on the left-hand side,
7. no "where" and
8. ...
(7) no "where", and
(8) no conditional right-hand sides with "|".

Transformation consists of 12 rules that are applied repeatedly to a complex H-expression. The order of the application of the rules does not matter, their repeated application always terminates, and the result is a simple H-expression.

We introduce the individual rules and illustrate them with the append program.

( Slide 42 )

Rule (1): Transform a sequence of function declarations into a single pattern declaration with a variable on the LHS.

\[
\text{Var } \text{pat}_1^\text{a} \ldots \text{pat}_n^\text{a} = \text{exp}^\text{a} ; \ldots ; \text{Var } \text{pat}_1^\text{k} \ldots \text{pat}_n^\text{k} = \text{exp}^\text{k}
\]

\[
\text{Var } \backslash x_1 \ldots x_n \to \text{Case } (x_1, \ldots, x_n) \text{ of } \{ \left( \text{pat}_1^\text{a}, \ldots, \text{pat}_n^\text{a} \right) \to \text{exp}^\text{a} ; \ldots ; \left( \text{pat}_1^\text{k}, \ldots, \text{pat}_n^\text{k} \right) \to \text{exp}^\text{k} \} \]

if \( x_1, \ldots, x_n \) are fresh variables (\( n > 0 \)), and these are all defining equations for \( \text{Var} \).

Rule (2): Lambdas may only have one argument in Simple Haskell

\[
\text{\backslash } \text{pat}_1 \ldots \text{pat}_n \to \text{exp}
\]

\[
\text{\backslash } \text{pat}_1 \to (\text{\backslash } \text{pat}_2 \to \ldots (\text{\backslash } \text{pat}_n \to \text{exp}) \ldots)
\]

if \( n \geq 2 \).
Rule (3): Lambdas may only have a variable as their argument, not a general pattern. \[\Rightarrow\] Transform such lambda expressions into case-constructs:

\[
\begin{align*}
\text{\textbackslash{} pat} & \rightarrow \text{\textbackslash{} exp} \\
\text{\textbackslash{} var} & \rightarrow \text{case \ var \ of \ pat \ \textbackslash{} \rightarrow \text{\textbackslash{} exp}}
\end{align*}
\]

If pat is no variable and var is a fresh variable.

Now we want to remove all case-constructs. To this end, we first transform them into "match"-constructs. This is only needed as an intermediate step. Our transformation will eliminate "match" again afterwards.

\[
\text{match \ pat \ exp \ exp_1 \ exp_2}
\]

Means:
- If pat matches exp (with matcher $\sigma$), then the result is $\sigma(\text{exp}_1)$.
- Otherwise the result is $\text{exp}_2$.

Rule (4): Translating "case" into "match"

\[
\text{case \ exp \ of \ \{ \text{pat}_1 \ \rightarrow \text{exp}_1; \\
\text{\ldots} \\
\text{pat}_n \ \rightarrow \text{exp}_n \}\}
\]

\[
\text{match \ pat_1 \ exp \ exp_1} \\
\text{\ldots} \\
\text{\textbackslash{} (match \ pat_n \ exp \ exp_n \ \textbackslash{} \rightarrow \text{\textbackslash{} (match \ pat_n \ exp \ exp_n \ \textbackslash{} \rightarrow \text{\textbackslash{} \ldots}))}
\]
The remaining rules are used to remove "match" for the different forms of patterns.

Rule (3) match for non-empty tuples

\[
\begin{align*}
\text{match } (\text{pat}_1, \ldots, \text{pat}_n) \text{ exp } \text{exp}_1 \text{ exp}_2 \\
\text{if (is trope } \text{exp}) \\
\text{then match } \text{pat}_1 (\text{sel}_{n,1} \text{exp}) \\
(\text{match } \text{pat}_2 (\text{sel}_{n,2} \text{exp}) \\
\ldots \\
(\text{match } \text{pat}_n (\text{sel}_{n,n} \text{exp}) \text{exp}_1 \text{exp}_2) \\
\text{exp}_2)
\end{align*}
\]

else \text{exp}_2

On Slide 43, we simplified "if is trope (exp_1, \ldots, exp_n) then exp_1 \text{ else } \text{exp}_2 " to \text{exp}.

Rule (5): match for variables

\[
\begin{align*}
\text{match var exp exp}_1 \text{exp}_2 \\
(\text{var }\rightarrow \text{exp}_1) \text{ exp}
\end{align*}
\]

Rule (6) match for joker pattern

\[
\begin{align*}
\text{exp}_1, \text{but all free occurrences of var in exp}_1 \text{are replaced by exp}_2
\end{align*}
\]
match = \( \exp \ \exp_1 \ \exp_2 \)

\[\exp_1\]

(Slide 45)

Rule (7) match for constructors

\[
\text{match } (\text{constr } \text{pat}_1 \ldots \text{pat}_n) \ \exp \ \exp_1 \ \exp_2 \\
\]

\[
\text{if } (\text{isa constr } \exp) \ \text{then } (\text{match } (\text{pat}_1, \ldots, \text{pat}_n) \ \text{arg of constr } \exp) \ \exp_1 \ \exp_2 \ \text{else } \exp_2
\]

(Slide 46)

Rule (8) match for empty tuple

\[
\text{match } () \ \exp \ \exp_1 \ \exp_2
\]

\[
\text{if } (\text{isa o-type } \exp) \ \text{then } \exp_1 \ \text{else } \exp_2
\]

Rules (1) - (9): transform complex into simple Haskell

Rule (10) - (12): needed for programs with several functions (including mutual recursion)

Def 2.2.11: (see Slide 52)

Thm 2.2.12: (see Slide 52)
Def 2.2.43  (Semantics of Complex H-Programs)

For a complex H-program, let \( P \) be the sequence of its function+pattern declarations.

The semantics of an expression \( \text{exp} \) that does not contain any free variables except the pre-defined variables of Haskell and the variables defined in \( P \) is:

\[
\text{Val} \ II \ (\text{let } P \ \text{in} \ \text{exp}) \ \text{II} \ \text{W} \ \text{tr}
\]