2.2.3 Semantics of Complex Haskell Programs

Mittwoch, 1. Juni 2016 11:30

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

Ch2 • definition of semantics: The semantics of a complex program is defined to be the semantics of the corresponding simple program

Ch3 • implementing Haskell: First compile complex to simple Haskell, then execute it.

Ch4 • 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 41)

• bot should be a pre-defined constant with semantics $\bot$

could be defined in Haskell by:

```
bot :: a
bot = bot
```

• isa$_{constr}$ pre-defined function which checks whether its argument is built with the data constructor constr

constructor $\overset{\cup}{constr}$

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

```
data List a = Nil | Cons a (List a)

isa_Cons :: (List a) -> Bool
isa_Cons (Cons x y) = True
isa_Cons Nil        = False
```

- $arg\,of_{constr}$: if the argument is built with constr, then $arg\,of_{constr}$ returns the tuple of its arguments

```
arg of_Cons :: (List a) -> (a, List a)
arg of_Cons (Cons x y) = (x, y)
```

↑ Can easily be defined in Haskell but not in Simple Haskell

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

Can easily be implemented in non-simple Haskell:

```
isa_n-tuple :: (a_1, .., a_n) -> Bool
isa_n-tuple (x_1, .., x_n) = True
```

```
isa_0-tuple :: () -> Bool
isa_0-tuple () = True
```

Note: $isa_{n\text{-}tuple}$ bot is well typed, but undefined (Semantics is $\perp$).

- $sel_{n,i}$: selects the i-th component of an n-tuple

  Can easily be defined in non-simple Haskell:

  e.g.: $sel_{3,2} :: (a_1, a_2, a_3) \to a_2$

  $sel_{3,2}\ (x_1, x_2, x_3) = x_2$

Now we can translate every complex H-program/expression into an equivalent simple ——— " ———
that uses these finitely many pre-defined functions.

Def 228 (Pre-defined Functions)

For a Haskell program with the constructors 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 arities of functions in our program, and the number of declarations in our program.

The initial environment ω is extended to a new initial environment ω_{tr}, where the semantics of the new pre-defined functions is defined as follows:

$\omega_{tr}(bot) = \bot$

$$\omega_{tr}(isa_{constr})(d) = \begin{cases} True \text{ in } Constructions_0 \text{ in } Dom, \\ \quad \text{if } d = (constr, d_1, \ldots, d_n) \text{ in } Constructions_n \text{ in } Dom \\ False \text{ in } Constructions_0 \text{ in } Dom, \\ \quad \text{if } d = (constr', d_1, \ldots, d_m) \text{ in } Constructions_m \text{ in } Dom \\ \quad \text{and } constr \neq constr' \\ \bot, \text{ otherwise} \end{cases}$$

$$\omega_{tr}(argof_{constr})(d) = \begin{cases} (d_1, \ldots, d_n) \text{ in } Tuples_n \text{ in } Dom, \\ \quad \text{if } d = (constr, d_1, \ldots, d_n) \text{ in } Constructions_n \text{ in } Dom \\ \quad \text{and } n \neq 1 \\ \ldots \end{cases}$$

$\omega_{tr}(\text{arg}_{\underline{\text{constr}}})(d) =$ $\begin{cases} \text{and } n \neq 1 \\ d_1, \text{ if } d = (\underline{\text{constr}}, d_1) \text{ in Constructions}_1 \text{ in Dom} \\ \bot, \text{ otherwise} \end{cases}$

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

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

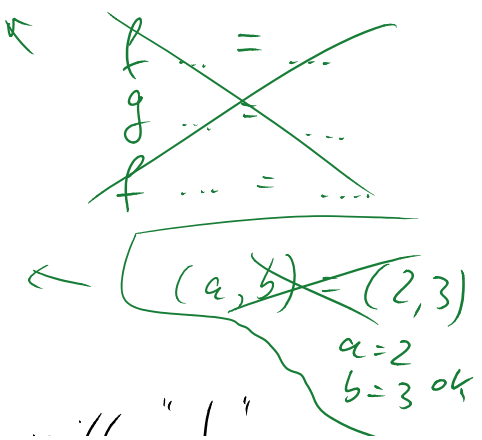
Now we want to introduce our transformation from complex to simple Haskell.

Full Haskell $\supseteq$ Complex Haskell $\supseteq$ Simple Haskell

Def 2.2.9 (Complex Haskell)

A <u>complex Haskell program</u> 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



⑦ no "where", and

⑧ no conditional right-hand sides with "|".

a=2
b=3 ok

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.

$$\frac{\underline{var}\ pat^1_1 \ldots pat^1_n = exp^1\ ;\ \ldots\ ;\ \underline{var}\ pat^k_1 \ldots pat^k_n = exp^k}{\underline{var} = \backslash x_1 \ldots x_n \to case\ (x_1,\ldots,x_n)\ of\ \{(pat^1_1,\ldots,pat^1_n) \to exp^1\ ; \ldots ; (pat^k_1,\ldots,pat^k_n) \to exp^k\}}$$

if $x_1,\ldots,x_n$ are fresh variables ($n>0$), and these are all defining equations for var

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

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

Rule (3): Lambdas may only have a variable as their argument, not a general pattern. ⇒ Transform such lambda expressions into case-constructs

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

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.

match pat exp exp_1 exp_2

means:
if pat matches exp (with matcher σ), then the result $\sigma(exp_1)$, otherwise the result is exp_2.

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

$$\frac{\text{case } exp \text{ of } \{pat_1 \to exp_1; \ \vdots \ pat_n \to exp_n\}}{\text{match } pat_1 \ exp \ exp_1 \ (\text{match } pat_2 \ exp \ exp_2 \ \ddots \ (\text{match } pat_n \ exp \ exp_n \ bot) \ldots)}$$

The remaining rules are used to remove "match" for the different forms of patterns.

(Slide 43)

Rule (9) match for non-empty tuples

$$\frac{\text{match } (pat_1, \ldots, pat_n)\ exp\ exp_1\ exp_2}{\text{if } (isa_{n\text{-tuple}}\ exp) \text{ then match } pat_1\ (sel_{n,1}\ exp)\ (\text{match } pat_2\ (sel_{n,2}\ exp) \ldots (\text{match } pat_n\ (sel_{n,n}\ exp)\ exp_1\ exp_2) \ldots)\ exp_2) \text{ else } exp_2} \quad \text{if } n \geq 2$$

On Slide 43, we simplified "if $isa_{n\text{-tuple}}\ (exp_1, \ldots, exp_n)$ then exp' else exp''" to exp'.

(Slide 44)

Rule (5): match for variables

$$\frac{\text{match } var\ exp\ exp_1\ exp_2}{(\backslash var \rightarrow exp_1)\ exp}$$

← exp_1, but all free occurrences of var in exp_1 are replaced by exp

Rule (6) match for Joker pattern

$$\frac{\texttt{match}\ \_\ \underline{exp}\ \underline{exp_1}\ \underline{exp_2}}{\underline{exp_1}}$$

(Slide 45)

Rule (7) match for constructors

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

(Slide 46)

Rule (8) match for empty tuple

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

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 57)

Def 2.2.13 (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 exp that does not contain any free variables except the pre-defined variables of Haskell and the variables defined in P is:

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