2.2.2 Semantics of Simple Haskell Programs

Dienstag, 31. Mai 2016 14:00

Now we want to define a mapping Val from Haskell-expressions to mathematical objects from the set Dom. To this end, we also want to define the semantics of sub-expressions. However: sub-expressions may contain variables that were defined in the context of this sub-expression

$$\text{let } x = 3 \text{ in } \underbrace{\text{plus } x \ 2}$$

To compute the semantics of this sub-expression, one needs the information of the context that x is 3 and plus might also have been defined previously.

One can only define the semantics of an expression wrt an environment that assigns values to all variables that occur (free) in this expression.

An environment ρ is a function that maps variables to elements of Dom.

If $\rho(x) = 2$ and $\rho(\text{plus})$ is the addition function then the semantics of

plus x 3

is 5.

So our semantics is a function $\text{Val} : \text{Exp} \to \text{Env} \to \text{Dom}$

H-expression   environment

Def 2.2.4 (Environment) (Slide 37)

For a H-program with the domain Dom, an <u>environment</u> is a partial function mapping variables to Dom, which is only defined for finitely many variables: $\rho : Var \to Dom$

Let Env denote the set of all environments. An environment ρ that is only defined on $\underline{var}_1, \dots, \underline{var}_n$ where $\rho(\underline{var}_i) = d_i$ is also denoted $\rho = \{\underline{var}_1, \dots, \underline{var}_n / d_1, \dots, d_n\}$.

We write $\rho_1 + \rho_2$ for the environment that is like ρ_2 whenever ρ_2 is defined. Otherwise, it is like ρ_1:

$$(\rho_1 + \rho_2)\ (\underline{var}) = \begin{cases} \rho_2(\underline{var}), & \text{if } \rho_2(\underline{var}) \text{ is defined} \\ \rho_1(\underline{var}), & \text{otherwise} \end{cases}$$

Let ω be the <u>initial environment</u> which assigns the correct meaning to all pre-defined operations in Haskell.

E.g.: $$(\omega(+))\ x\ y = \begin{cases} x + y, & \text{if } x, y \in \mathbb{Z} \text{ or } x, y \in \mathbb{F} \\ \bot, & \text{otherwise} \end{cases}$$

Semantics of an expression $\underline{exp}$ in an environment ρ will be denoted $Val[\![exp]\!]\rho$

$$Val : Exp \to Env \to Dom$$

To simplify the definition of Val, we only regard a subset of Haskell without pattern matching: "simple

Haskell". In Sect 2.2.3, we will show that every complex H-program can be automatically transformed into a simple H-program.

Restrictions of simple Haskell:

0. No type synonyms, no type classes.
1. Just one declaration, which must be a pattern declaration of the form: <u>var</u> = <u>exp</u>
2. No pre-defined lists.
3. No applications of the form (exp_1 exp_2 .... exp_n) for $n > 2$. But: ((exp_1 exp_2) exp_3 ....) exp_n is allowed.
4. No "case" construct.
5. Only lambda-expressions of the form \ <u>var</u> -> <u>exp</u>
6. No "where" (but "let" is allowed).

Def 2.2.5 (Simple Haskell)

see slide 38

Ex. for a simple Haskell program:

```
fact = \ x -> if x <= 0 then 1 else fact (x-1) * x
```

This simple Haskell program contains the pre-defined variables <=, -, * whose semantics is already defined in the initial environment ω.

Instead of determining the semantics of <u>exp</u>' in the program $\underline{var} = \underline{exp}$, we can determine the semantics of

$$\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp'}$$

in the empty program.

E.g.: if we would like to know the semantics of fact 2 in the fact-program, we can instead compute the semantics of the following expression in the empty program:

let fact = \x -> .... in fact 2

Goal: Define semantics of expressions $\underline{exp}$

$\text{Val} [\![\underline{exp}]\!] \rho$ where ρ is an environment that defines the meaning of all variables that occur free in $\underline{exp}$. (Moreover, $\underline{exp}$ must be syntactically correct and well typed.)

A variable occurring in an expression is free unless it is bounded by a "let" or a lambda ("\").

Def 2.2.6. (Free variables of H-expressions)

For every simple H-expression $\underline{exp}$, we define free($\underline{exp}$) (the set of its free variables) as follows:

- free($\underline{var}$) = {$\underline{var}$}
- free($\underline{constr}$) = free($\underline{integer}$) = free($\underline{float}$) = free($\underline{char}$) = $\emptyset$

- $free((\underline{exp}_1, \ldots, \underline{exp}_n)) = free(\underline{exp}_1) \cup \ldots \cup free(\underline{exp}_n)$
- $free(\underline{exp}_1\ \underline{exp}_2) = free(\underline{exp}_1) \cup free(\underline{exp}_2)$
- $free(\text{if } \underline{exp}_1 \text{ then } \underline{exp}_2 \text{ else } \underline{exp}_3) = free(\underline{exp}_1) \cup free(\underline{exp}_2) \cup free(\underline{exp}_3)$
- $free(\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp}') = (free(\underline{exp}) \cup free(\underline{exp}')) \setminus \{\underline{var}\}$
- $free(\backslash \underline{var} \to \underline{exp}) = free(\underline{exp}) \setminus \{\underline{var}\}$

Now we can define $Val[\![\underline{exp}]\!]\rho$ where ρ must be defined on all variables in $free(\underline{exp})$. (Slide 39+40)

1. $\underline{exp}$ is a variable

$Val[\![\underline{var}]\!]\rho = \rho(\underline{var})$

Ex: If $\rho(x) = 2$, $\rho(plus)$ is the addition fct, then:

$Val[\![x]\!]\rho = \rho(x) = 2$ in $Constructions_0$ in Dom

$Val[\![plus]\!]\rho = \rho(plus) =$ add. fct in Functions in Dom

2. $\underline{exp}$ is a constructor $\underline{constr}_0$ from Con_0

$Val[\![\underline{constr}_0]\!]\rho = \underline{constr}_0$ in $Constructions_0$ in Dom

Ex: $Val[\![5]\!]\rho = 5$ in $Constructions_0$ in Dom

3. $\underline{exp}$ is a constructor $\underline{constr}_n \in Con_n$, where $n > 0$

$Val[\![\underline{constr}_n]\!]\rho = f$ in Functions in Dom

where $f\ d_1 \ldots d_n = (\underline{constr}_n, d_1, \ldots, d_n)$ in $Constructions_n$ in Dom

Ex: data Nats = Zero | Succ Nats

$Val[\![Succ]\!]\rho = f_{Succ}$

where $f_{Succ}\ d = (Succ, d)$ in $Constructions_1$ in Dom

4. $\underline{exp}$ is a tuple $(\underline{exp}_1, \dots, \underline{exp}_n)$, $n \in \{0, 2, 3, 4, \dots\}$

$$Val[\![(\underline{exp}_1, \dots, \underline{exp}_n)]\!]\rho = (Val[\![\underline{exp}_1]\!]\rho, \dots, Val[\![\underline{exp}_n]\!]\rho)$$

in $Tuples_n$ in Dom

5. $\underline{exp}$ is a one-component tuple

$$Val[\![(\underline{exp})]\!]\rho = Val[\![\underline{exp}]\!]\rho$$

6. $\underline{exp}$ is an application $(\underline{exp}_1\ \underline{exp}_2)$

$$Val[\![(\underline{exp}_1\ \underline{exp}_2)]\!]\rho = f(Val[\![\underline{exp}_2]\!]\rho)$$

where $Val[\![\underline{exp}_1]\!]\rho = f$ in Functions in Dom

Ex: $Val[\![Succ\ Zero]\!]\rho = f_{Succ}(\underbrace{Val[\![Zero]\!]\rho}_{Zero})$

$= (Succ, Zero)$ in $Constructions_1$ in Dom

7. $\underline{exp}$ is if $\underline{exp}_1$ then $\underline{exp}_2$ else $\underline{exp}_3$

$$Val[\![\text{if } \underline{exp}_1 \text{ then } \underline{exp}_2 \text{ else } \underline{exp}_3]\!]\rho = \begin{cases} Val[\![\underline{exp}_2]\!]\rho, & \text{if } Val[\![\underline{exp}_1]\!]\rho = True \text{ in } Constructions_0 \text{ in Dom} \\ Val[\![\underline{exp}_3]\!]\rho, & \text{— " — } = False \text{ — " —} \\ \bot, & \text{otherwise} \end{cases}$$

8. $\underline{exp}$ is a let-expression

What is $Val[\![\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp'}]\!] \rho$?

Case 1: $\underline{var}$ does not occur in $\underline{exp}$ (no recursion)

$Val[\![\text{let } x=3 \text{ in plus } x\ y]\!] \rho =$

$Val[\![\text{plus } x\ y]\!] (\rho + \{x/3\}) =$

$\rho(\text{plus})\ \ 3\ \ \rho(y) = 5$

where
$\rho(y) = 2$
$\rho(\text{plus}) = \text{addition}$

$Val[\![\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp'}]\!] \rho = Val[\![\underline{exp'}]\!] (\rho + \{\underline{var} / Val[\![\underline{exp}]\!] \rho\})$

This does not work if $\underline{var} \in free(\underline{exp})$.

Case 2: $\underline{var} \in free(\underline{exp})$

$Val[\![\text{let fact} = \backslash x \to \text{if } x <= 0 \text{ then } 1 \text{ else fact}(x-1) * x \text{ in fact } 2]\!] \rho =$

$Val[\![\text{fact } 2]\!] \rho'$ where ρ' is like ρ for all variables except fact.

The meaning of fact should be the least fixpoint of the following higher-order function ff:

$ff(d) = Val[\![\backslash x \to \text{if } x <= 0 \text{ then } 1 \text{ else fact}(x-1) * x]\!] (\rho + \{\text{fact}/d\})$

So:

$Val[\![\text{let fact} = \backslash x \to \dots \text{fact}(x-1) * x \text{ in fact } 2]\!] \rho =$

$Val[\![\text{fact } 2]\!] (\rho + \{\text{fact} / \text{lfp } ff\})$

where $ff(d) = Val[\![\backslash x \to \text{if } x <= 0 \text{ then } 1 \text{ else fact}(x-1) * x]\!] (\rho + \{\text{fact}/d\})$

In general:

$Val[\![\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp'}]\!] \rho = Val[\![\underline{exp'}]\!] (\rho + \{\underline{var} / lfp\, f\})$

where $f(d) = Val[\![\underline{exp}]\!] (\rho + \{\underline{var} / d\})$

This definition can be used for both recursive and non-recursive declarations.

If $\underline{var} \notin free(\underline{exp})$, then

$f(d) = Val[\![\underline{exp}]\!] (\rho + \{\underline{var}/d\}) = Val[\![\underline{exp}]\!] \rho$

So the only fixpoint of f is $Val[\![\underline{exp}]\!] \rho$.

9. $\underline{exp}$ is a lambda expression

$Val[\![\backslash \underline{var} \to \underline{exp}]\!] \rho = f$ in Functions in Dom

where $f(d) = Val[\![\underline{exp}]\!] (\rho + \{\underline{var} / d\})$

Ex: $Val[\![\backslash x \to plus\ x\ y]\!] \rho = f$

where $f(d) = Val[\![plus\ x\ y]\!] (\rho + \{x/d\})$

$= \rho(plus)\ d\ \rho(y)$

$= d + 2$

$\rho(y) = 2$

$\rho(plus) = addition$

Def 227 (Semantics of Simple Haskell-Programs)

Let Dom be the domain of a simple H-program, let $\underline{var} = \underline{exp}$ be the (only) pattern declaration of the program, let Exp be the set of all simple H-expressions.

Then we define the function $Val : Exp \to Env \to Dom$, See Slide 39 + 40.

The semantics of an expression <u>exp'</u> that does not contain any free variables except <u>var</u> and pre-defined variables of Haskell is defined as:

$$Val[\![\text{let } \underline{var} = \underline{exp} \text{ in } \underline{exp'}]\!]\ \omega$$