3.3 Reducing Haskell to the Lambda Calculus

Dienstag, 14. Juni 2016 15:00

Goal: Compile (simple) Haskell expression to a λ-term.
Then this λ-term can be evaluated by $\to_{\beta\delta}$
↷ implementation of Haskell.

2 tasks:

1. Define a suitable evaluation strategy for $\to_{\beta\delta}$ that corresponds to the desired evaluation of Haskell.
2. Define an automatic translation from (simple) Haskell expressions to λ-terms.

λ-calculus is confluent, but what about termination? (Slide 56)

$$(\lambda x.\, x\,x)\,(\lambda x.\, x\,x) \to_\beta (\lambda x.\, x\,x)\,(\lambda x.\, x\,x) \to_\beta \dots$$

So already $\to_\beta$ does not terminate. Moreover, $\to_\delta$ can introduce even more sources of non-termination.

Termination can depend on the reduction strategy:

$$(\lambda x.\, y)\,((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))$$

$\to_\beta\ y$

$\to_\beta\ (\lambda x.\, y)\,((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))$

$\to_\beta \dots$

To implement Haskell, $\to_{\beta\delta}$ should be applied with a leftmost outermost strategy.

Moreover, we should not always evaluate to normal form, but we should stop earlier:

If we reached $f\ t_1 \ldots t_n$ and there is no rule to evaluate f, then the arguments $t_1, .., t_n$ should not be evaluated further.

Ex: $(1+2) : [\,]$ $\quad\left(\text{i.e. } (:)\ (1+2)\ [\,]\right)$

Since $:$ is a data constructor (i.e., there is no rule for $:$), we do not evaluate $(1+2)$ further.

Ex: $x\ (1+2)$

Since a variable x cannot be evaluated further, we do not evaluate $(1+2)$ further either.

Ex: $\lambda x.\ (1+2)$

Again, $(1+2)$ is not evaluated further.

If the topmost symbol is a data constructor, a variable, or a λ, then we stop the evaluation, since a further evaluation would not change this topmost symbol anymore.

Def 3.3.1 (Weak Head Normal Form)
A λ-term is in Weak Head Normal Form (WHNF) iff it is in normal form or it has one of the following forms:

- $\lambda x.t$ for any $t \in \Lambda$
- $c\ t_1 \ldots t_n$ for any $t_1, \ldots, t_n \in \Lambda$ and $c \in \mathcal{C}$ such that there is no δ-rule for c (i.e., c is a constructor)
- $x\ t_1 \ldots t_n$ for any $t_1, \ldots, t_n \in \Lambda$ and $x \in \mathcal{V}$

Now we can define an evaluation strategy for the λ-calculus that can be used to implement Haskell.

Def 332 (Weak Head Normal Order Reduction)
The WHNO-reduction on λ-terms is defined as:
$t \to r$ iff t is not in WHNF and $t \xrightarrow[\beta\delta]{} r$, where the reduction follows the leftmost-outermost strategy.

Now we solve Task 2: Translate simple Haskell to the λ-calculus. (Slide 57)

Define a function $\mathcal{L}am : Exp \to \Lambda$ (set of simple Haskell-expressions)

- $\mathcal{L}am(var) = var$
- $\mathcal{L}am(c) = c$, where $c \in \mathcal{C}_0 \cup Con$ (pre-defined functions (+, map, ...); constructors (including the ones for Int, Float, ...))
- $\mathcal{L}am((exp_1, \ldots, exp_n)) = tuple_n\ \mathcal{L}am(exp_1) \ldots \mathcal{L}am(exp_n)$ for $n \in \{0,2,3,\ldots\}$ ($tuple_n$: new constant that must be contained in $\mathcal{C}$)
- $\mathcal{L}am((exp)) = \mathcal{L}am(exp)$
- $\mathcal{L}am(exp_1\ exp_2) = \mathcal{L}am(exp_1)\ \mathcal{L}am(exp_2)$
- $\mathcal{L}am(\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3) = \text{if}\ \mathcal{L}am(exp_1)\ \mathcal{L}am(exp_2)\ \mathcal{L}am(exp_3)$ (if: new constant in $\mathcal{C}$)

- $\mathcal{L}am\ (\backslash var \to exp) = \lambda var.\ \mathcal{L}am\ (exp)$

If remains to translate expressions with "let" (i.e., with local declarations).

Case 1: no recursion

let var = exp in exp' where var does not occur free in exp

If $\mathcal{L}am\ (exp) = t$ and $\mathcal{L}am\ (exp') = t'$,
then let var = exp in exp' should be translated to

$$t'[var/t]$$

Ex: let x = 3 in x+2 would be translated into 3+2

$\mathcal{L}am: 3$ $\quad$ $\mathcal{L}am: x+2$

$x+2\ [x/3]$

Case 2: recursion

let fact = \x -> if x<=0 then 1 else fact (x-1) * x in fact 2

According to the semantics of Haskell, this means that fact should be the least fixpoint of the function

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

Idea: introduce another constant fix $\in \mathcal{C}$ in the λ-calculus, which computes the least fixpoint

Then, we could reformulate the above expression in a

non-recursive way:

let fact = fix (\fact -> \x -> if x<=0 then 1 else fact(x-1) * x) in fact 2

Now this expression can be translated into the following λ-term (using the translation for non-recursive declarations in Case 1):

$$\mathcal{L}am(fact\ 2)[fact/\ \mathcal{L}am(fix(\backslash fact \to \backslash x \to \ldots))] =$$

$$(fact\ \ 2)[fact\ /\ fix(\lambda fact\ x.\ if\ \ldots)] =$$

$$(fix\ (\lambda fact\ x.\ if\ (x \leq 0)\ 1\ \ (fact\ (x-1) * x)))\ 2$$

So the general rule for the translation of let-expressions to λ-terms is:

$$\mathcal{L}am(let\ \underline{var} = \underline{exp}\ in\ \underline{exp'}) = \mathcal{L}am(\underline{exp'})[\underline{var}/(fix\ (\lambda \underline{var}.\ \mathcal{L}am(\underline{exp})))]$$

This translation can be used for both recursive and non-recursive declarations:

$$\mathcal{L}am(let\ x=3\ in\ x+2) =$$

$$\underbrace{\mathcal{L}am(x+2)}_{x+2}[x/(fix\ (\lambda x.\ \underbrace{\mathcal{L}am(3)}_{3}))] =$$

$$x+2\ [x/\ fix\ (\lambda x.3)] =$$

$$(fix\ (\lambda x.3)) + 2 \to^* 3+2$$

We need δ-rules to evaluate fix. They should ensure:

$fix\ t \to^* t\ (fix\ t) \to^* t\ (t\ (fix\ t)) \to^* t\ (t\ (t\ (fix\ t)))$ ...

$\hat{=} \bot$ $\quad \hat{=} t(\bot)$ $\quad \hat{=} t(t(\bot))$ $\quad \hat{=} t^3(\bot)$

Then: $fix\ (\lambda x.3) \to^* (\lambda x.3)\ (fix\ (\lambda x.3)) \to_\beta 3$

Def 333 (Translation from simple Haskell into λ-terms)

$\mathcal{L}am : Exp \to \Lambda$ is defined on Slide 57.

To implement Haskell, we now proceed as follows:

if P is the H-program (seq. of declarations)

and exp is the H-expression that should be evaluated in the program P,

then we translate $(let\ P\ in\ exp)_{tr}$ into a λ-term using $\mathcal{L}am$.

Then this λ-term is evaluated using WHNO.

translation from complex to simple Haskell

This can be made more efficient by translating P in advance (compilation).

Def. 334 (Translation of complex Haskell into λ-terms)

Let P be the sequence of pattern and function declarations of a complex H-program, let exp be a complex H-expression that does not contain free variables except those defined in P

or pre-defined in Haskell.
Let $\mathcal{C}_0$ be the pre-defined functions in Haskell (e.g., +, not, ...)
let Con be the data constructors of the H-program.
Then we define

$$\mathcal{C} = \mathcal{C}_0 \cup$$
$$Con \cup$$
$$\{if, fix, bot\} \cup$$
$$\{isa_{n\text{-}tuple} \mid n \in \{0,2,3,\ldots\}\} \cup$$
$$\{isa_{\underline{constr}} \mid \underline{constr} \in Con\} \cup$$
$$\{argof_{\underline{constr}} \mid \underline{constr} \in Con\} \cup$$
$$\{sel_{n,i} \mid n \geq 2,\ 1 \leq i \leq n\} \cup$$
$$\{tuple_n \mid n \in \{0,2,3,\ldots\}\}$$

The translation of $\underline{exp}$ in the program P is a λ-term over the constants $\mathcal{C}$ that is defined as:

$$Tran(P, \underline{exp}) = \mathcal{L}am((let\ P\ in\ \underline{exp})_{tr})$$

The λ-term $Tran(P, \underline{exp})$ should now be evaluated by WHNO ($\rightarrow_{B\delta}$ until WHNF).
Which δ-rules should we use? They should evaluate the constants in $\mathcal{C}$ appropriately.
Since we only evaluate until we reach WHNF,

we can also allow δ-rules of the form

$$c\ t_1 \ldots t_n \to r$$

where $t_1, \ldots, t_n$ are in WHNF.

So for example, we can have a rule

$$isa_{constr}\ (Constr\ t_1 \ldots t_n) \to True \quad \text{for all closed } \lambda\text{-terms } t_1, \ldots, t_n$$

(t_i may now contain left-hand sides of other δ-rules).

This does not destroy confluence of WHNO, since $Constr\ t_1 \ldots t_n$ is already in WHNF.

This is needed to implement Haskell correctly:

```
f Zero = Zero
f (Succ x) = Zero
```

Here, evaluation of $f\ (Succ\ bot)$ should terminate.

To this end, we need the δ-rule:

$$isa_{Succ}\ (Succ\ bot) \to True$$

(although there is another δ-rule for bot)

Def 335 (δ-rules for Haskell)

Let Con be the set of data constructors of a Haskell

program, where Con_n are all constructors of arity n.
Let δ_0 be the rules for the pre-defined functions $\mathcal{C}_0$ of Haskell (so δ_0 contains rules like $1+2 \to 3$, not $True \to False$, ...).

Then δ is defined as on Slide 58.

For $if \in \mathcal{C}$, we need δ-rules which ensure

$if\ True\ x\ y \to^* x$
$if\ False\ x\ y \to^* y$

} These are no legal δ-rules, since they contain free variables.

Solution: $if\ True \to \lambda x\,y.\,x$
$if\ False \to \lambda x\,y.\,y$

} These are legal δ-rules.

Now: $if\ True\ x\ y \to_\delta (\lambda x\,y.\,x)\ x\ y \to_\beta^2 x$

For $fix \in \mathcal{C}$, we need a δ-rule which ensures:

$fix\ x \to^* x\ (fix\ x)$ ← This is no legal δ-rule, because x is free.

Solution: $fix \to \lambda x.\ x\ (fix\ x)$ ← This is a legal δ-rule

Now: $fix\ x \to_\delta (\lambda x.\ x\ (fix\ x))\ x \to_\beta x\ (fix\ x)$

The set δ is infinite, but it can be represented in a finite way and $\rightarrow_\delta$ can easily be implemented. The set δ only depends on the constructors of the program, not on the functions/algorithms in the program.

Def 336 (Implementing Haskell)
For a complex H-program with the constructors Con, let δ be the corresponding Delta-Rule-Set (as in Def. 335). Let P be the seq. of pattern and function declarations. Then the evaluation of exp in the program is done by WHNO of Tran(P, exp) using the Delta-Rule-Set δ.

One can now show that this implementation of Haskell is correct w.r.t. the denotational semantics of Ch. 2:

Thm 337 (Correctness of Implementation)
Let P and exp be as in Def 336, where P and exp are well typed.

WHNO with the δ rules as in Def 3.3.5

If $Tran(P, \underline{exp}) \to^* q$ for a λ-term q in WHNF, then $Val[\![(\text{let } P \text{ in } \underline{exp})_{tr}]\!]\omega_{tr} = Val[\![q]\!]\omega_{tr}$.

Here, Val must be extended to λ-terms in the obvious way.

If $Tran(P, \underline{exp})$ leads to a non-terminating WHNO-reduction, then $Val[\![(\text{let } P \text{ in } exp)_{tr}]\!]\omega_{tr} = \underline{\bot}$.

So our implementation realizes undefinedness by non-termination. Of course, this could be changed (e.g., one could return an error message for incompletely defined functions).