3.3 Reducing Haskell to the Lambda Calculus

Goal: Compile (simple) Haskell expression to a λ-term. Then this λ-term can be evaluated by \( \rightarrow^\beta \) implementation of Haskell.

2 tasks:
1. Define a suitable evaluation strategy for \( \rightarrow^\beta \) 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)

\[
(\lambda x. xx) (\lambda x. xx) \rightarrow^\beta (\lambda x. xx) (\lambda x. xx) \rightarrow^\beta
\]

So already \( \rightarrow^\beta \) does not terminate. Moreover, \( \rightarrow^\beta \) can introduce even more sources of non-termination.

Termination can depend on the reduction strategy:

\[
(\lambda x. y) ((\lambda x. xx) (\lambda x. xx))
\]

\[
\frac{\lambda \beta}{\frac{\beta}{\frac{\beta}}}
\]

To implement Haskell, \( \rightarrow^\beta \) should be applied with a leftmost outermost strategy.
Moreover, we should not always evaluate to normal form, but we should stop earlier:

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

\[
\text{Ex:} \quad (1+2) \quad : \quad (\cdot) \quad (1+2) \quad [3]
\]

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

\[
\text{Ex:} \quad x \quad (1+2)
\]

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

\[
\text{Ex:} \quad \lambda x. \quad (1+2)
\]

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

If the topmost symbol is a data constructor, a variable, or a \( \lambda \), 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 \( \lambda \)-term is in Weak Head Normal Form (WHNF) if it is in normal form or it has one of the following forms:
\[ \lambda x. t \quad \text{for any } t \in A \]
\[ c \; t_1 \ldots t_n \quad \text{for any } t_1, \ldots, t_n \in A \text{ and } c \in C \text{ such that} \]
\[ \text{there is no } \delta\text{-rule for } c \text{ (i.e., } c \text{ is a constructor)} \]
\[ x \; t_1 \ldots t_n \quad \text{for any } t_1, \ldots, t_n \in A \text{ and } x \in V \]

Now we can define an evaluation strategy for the \( \lambda \)-calculus that can be used to implement Haskell.

**Def 3.32 (Weak Head Normal Order Reduction)**

The WHNO-reduction on \( \lambda \)-terms is defined as:
\[ t \rightarrow r \quad \text{iff } t \text{ is not in WHNF and } t \rightarrow_{\beta_0} r, \]
where the reduction follows the leftmost-outermost strategy.

Now we solve Task 2: Translate Simple Haskell to the \( \lambda \)-calculus.

Define a function \( \text{Lam} : \text{Exp} \rightarrow A \)

- \( \text{Lam}(\text{var}) = \text{var} \)
- \( \text{Lam}(c) = c, \text{ where } c \in C \cup \text{Con} \)
- \( \text{Lam}(\text{exp}_1 \ldots \text{exp}_n) = \text{tuple} \; \text{Lam}(\text{exp}_1) \ldots \text{Lam}(\text{exp}_n) \quad \text{for } n \in \{0,2,3,\ldots\} \)
- \( \text{Lam}(\text{exp}) = \text{Lam}(\text{exp}) \)
- \( \text{Lam}(\text{exp}_1 \text{, exp}_2) = \text{Lam}(\text{exp}_1) \; \text{Lam}(\text{exp}_2) \)
- \( \text{Lam}(\text{if exp then exp}_1 \text{ else exp}_3) = \text{if Lam}(\text{exp}) \; \text{Lam}(\text{exp}_1) \; \text{Lam}(\text{exp}_3) \)

\[ \]
\[ \text{Let } \text{var} = \text{exp} \text{ in } \text{exp}' \text{ \ where \ var \ does \ not \ occur \ free \ in } \text{exp} \]

If \( \text{Lam} (\text{exp}) = t \) and \( \text{Lam} (\text{exp}') = t' \),
then \( \text{let } \text{var} = \text{exp} \text{ in } \text{exp}' \) should be translated to
\[ t' [\text{var} / t] \]

\[ \text{Ex: let } x = 3 \text{ in } x + 2 \]
\[ \text{Lam: 3 } \quad \text{Lam: } x + 2 \]
\[ \text{would be translated into } \]
\[ 3 + 2 \]
\[ x + 2 [x / 3] \]

\[ \text{Case 2: recursion} \]
\[ \text{let fact} = \lambda x \rightarrow \text{if } x \leq 0 \text{ then } 1 \text{ else } \text{fact} (x - 1) \times x \text{ in } \text{fact 2} \]

According to the semantics of Haskell, this means that
\[ \text{fact} \]
should be the least fixpoint of the function
\[ \lambda x \rightarrow \text{if } x \leq 0 \text{ then } 1 \text{ else } \text{fact} (x - 1) \times x \]

Idea: introduce another constant \( \text{fix } \in \mathcal{E} \) in the
\( \lambda \)-calculus, which computes the least fixpoint.
Then, we could reformulate the above expression in \( \text{a} \)
Non-recursive way:

\[ \text{let } \text{fact } = \text{fix } (\text{fact} \rightarrow x \rightarrow \text{if } x < 0 \text{ then } 1 \text{ else } \text{fact}(x-1) \times x) \text{ in } \text{fact } 2 \]

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

\[ \text{Lam}(\text{fact } 2)[\text{fact} / \text{Lam}(\text{fix } (\text{fact} \rightarrow x \rightarrow \ldots))] = \]

\[ (\text{fact } 2)[\text{fact} / \text{fix } (\lambda \text{fact } x, \text{ if } \ldots)] = \]

\[ (\text{fix } (\lambda \text{fact } x, \text{ if } (x < 0) \land (\text{fact } (x-1) \times x))) \]

So the general rule for the translation of \(\text{let}\)-expressions to \(\lambda\)-terms is:

\[ \text{Lam}(\text{let } \text{var } = \text{exp} \text{ in } \text{exp}') = \text{Lam}(\text{exp}')[\text{var} / (\text{fix } (\lambda \text{var}. \text{Lam}(\text{exp}'))] \]

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

\[ \text{Lam}(\text{let } x = 3 \text{ in } x + 2) = \]

\[ \text{Lam}(x+2)[x / (\text{fix } (\lambda x, \text{Lam}(3)))] = \]

\[ x + 2 \]

\[ x + 2 [x / \text{fix } (\lambda x. 3)] = \]

\[ (\text{fix } (\lambda x. 3)) + 2 \rightarrow^* 3 + 2 \]

We need \(\delta\)-rules to evaluate fix. They should ensure:
\[
\text{fix } t \rightarrow \ast \rightarrow t (\text{fix } t) \rightarrow \ast \rightarrow t (t (\text{fix } t)) \rightarrow \ast \rightarrow t (t (t (\text{fix } t)))
\]

Then:

\[
\text{fix } (\lambda x . 3) \rightarrow \ast (\lambda x . 3) (\text{fix } (\lambda x . 3)) \rightarrow \beta 3
\]

Def 333 (Translation from Simple Haskell into \(\lambda\)-terms)

Law: \(\text{Exp} \rightarrow \text{\(\lambda\)}\) is defined on Slide 57.

To implement Haskell, we now proceed as follows:

1. \(P\) is the \(\text{H}\)-program (seq. of declarations)
2. \(\text{exp}\) is the \(\text{H}\)-expression that should be evaluated in the program \(P\),

then we translate \((\text{let } P \text{ in } \text{exp})\) into a \(\lambda\)-term using Law.

Then this \(\lambda\)-term is evaluated using WHNO.

Def. 334 (Translation of Complex Haskell into \(\lambda\)-terms)

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

This can be made more efficient by translating \(P\) in advance (compilation).
or pre-defined in Haskell.

Let $\mathcal{C}$ be the pre-defined functions in Haskell (e.g., $+$, not, ...) and $\mathcal{Con}$ be the data constructors of the $\lambda$-program.

Then we define

$\mathcal{C} = \mathcal{C}_0 \cup$

$\mathcal{Con} \cup$

{ \text{if, fix, bot} } \cup

{ \text{isa n-tuple } | \text{n} \in \{0, 2, 3, \ldots \} } \cup

{ \text{isa constr } | \text{constr} \in \mathcal{Con} } \cup

{ \text{arg of constr } | \text{constr} \in \mathcal{Con} } \cup

{ \text{sel}_{n,i} | \text{n} \geq 2, 1 \leq i \leq n } \cup

{ \text{tuple}_{n} | \text{n} \in \{0, 2, 3, \ldots \} }$

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

$$\text{Tran}(\mathcal{P}, \text{exp}) = \lambda x \left( (\text{let } \mathcal{P} \text{ in } \text{exp})_\text{tr} \right)$$

The $\lambda$-term $\text{Tran}(\mathcal{P}, \text{exp})$ should now be evaluated by $\text{WHNO} \left( \rightarrow_\beta \text{ until WHNF} \right)$.

Which $\beta$-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 $\delta$-rules of the form

\[ ct_1 \ldots t_n \rightarrow r \]

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

So for example, we can have a rule

\[ \text{isa} (\text{constr } t_1 \ldots t_n) \rightarrow \text{True} \quad \text{for all closed } \lambda\text{-terms} \]

\[ t_1, \ldots, t_n \]

(t_i may now contain left-hand sides of other $\delta$-rules).

This does not destroy confluence of WHNF,

since $\text{constr } t_1 \ldots t_n$

is already in WHNF.

This is needed to implement Haskell correctly:

\[ f \text{ zero } = \text{ zero} \]

\[ f (\text{ succ } x) = \text{ zero} \]

Here, evaluation of $f (\text{ succ bot})$ should terminate.

To this end, we need the $\delta$-rule:

\[ \text{isa} \ (\text{ succ } \text{ bot}) \rightarrow \text{ True} \]

(although there is another $\delta$-rule for bot)

\[ \text{Def 335 } (\delta\text{-rules for Haskell}) \]

Let $\text{Con}$ be the set of data constructors of a Haskell
program, where Con are all constructors of arity n. Let $\delta_0$ be the rules for the pre-defined functions $\mathbf{C}$ of Haskell (so $\delta_0$ contains rules like $1 + 2 \rightarrow 3$, not $\text{True} \rightarrow \text{False}$, ...).

Then $\delta$ is defined as on Slide 58.

For $e \in E$, we need $\delta$-rules which ensure:

$\begin{align*}
\text{if } \text{True} \times y & \rightarrow^* x \\
\text{if } \text{False} \times y & \rightarrow^* y
\end{align*}$

These are no legal $\delta$-rules, since they contain free variables.

Solution:

$\begin{align*}
\text{if } \text{True} & \rightarrow \lambda x. y. x \\
\text{if } \text{False} & \rightarrow \lambda x. y. y
\end{align*}$

These are legal $\delta$-rules.

Now, $\text{if } \text{True} \times y \rightarrow^* (\lambda x. y. x) \times y \rightarrow^\beta x$

For $fix \in E$, we need a $\delta$-rule which ensures:

$fix \times \rightarrow^* x (fix \times)$

This is no legal $\delta$-rule, because $x$ is free.

Solution:

$fix \rightarrow \lambda x. x (fix \times)$

This is a legal $\delta$-rule.

Now:

$fix \times \rightarrow^\beta (\lambda x. x (fix \times)) \times \rightarrow^\beta x (fix \times)$
The set $S$ is infinite, but it can be represented in a finite way and $\rightarrow_S$ can easily be implemented. The set $S$ only depends on the constructors of the program, not on the functions/algorithms in the program.

**Def 336 (Implementing Haskell)**

For a complex $\lambda$-program with the constructors $\text{Con}$, let $\bar{S}$ be the corresponding Delta-Rule-Set (as in Def. 335). Let $P$ be the seq. of pattern- and function declarations. Then the evaluation of $\text{exp}$ in the program is done by $\text{WHNO}$ of $\text{Ir}an (P, \text{exp})$ using the Delta-Rule-Set $\bar{S}$.

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 $\text{exp}$ be as in Def 336, where $P$ and $\text{exp}$ are well typed.
If $\text{ Tran } (P, \exp) \rightarrow^* q$ for a $\lambda$-term $q$ in W+NF, then $\text{Val } I \left( \text{let } P \text{ in } \exp \right) \uparrow_{tr} = \text{Val } I q \uparrow_{tr}.

If $\text{ Tran } (P, \exp)$ leads to a non-terminating W+NO-reduction, then $\text{Val } I \left( \text{let } P \text{ in } \exp \right) \uparrow_{tr} = \bot$.

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).

Here, Val must be extended to $\lambda$-terms in the obvious way.