4.1 A Simple Functional Language

For simplicity, we consider:

\[ c ::= b | (c_1, \ldots, c_k) | c \cdot c_1 \ldots c_k | \text{fun } x \rightarrow c \]

\[ | (c_1, c_2) | ([c_1] | (c_1 \sqcup c_2) | \]

\[ \text{let } x_1 = c_1 \text{ in } c_0 | \]

\[ \text{match } c_0 \text{ with } p_1 \rightarrow c_1 | \ldots | p_k \rightarrow c_k \]

\[ p ::= b | x \cdot \{ x_1 \ldots x_k \cdot \{ x_1, \ldots, x_k \} \]

\[ t ::= \text{let rec } x_1 = c_1 \text{ and } \ldots \text{ and } x_k = c_k \text{ in } e \]

where \( b \) is a constant, \( x \) is a variable, \( c \) is a (data-)constructor and \( \sqcup_i \) are \( i \)-ary operators.
... in the Example:

A definition of `max` may look as follows:

```plaintext
let max = fun x -> match x with (x1, x2) -> {
  match x1 < x2
  with True -> x2
  | False -> x1
}
```

Accordingly, we have for `abs`:

```plaintext
let abs = fun x -> let z = (x, -x) in max z
```

### 4.2 A Simple Value Analysis

#### Idea:

For every subexpression `e` we collect the set `\([e]^{\widetilde{\nu}}\)` of possible values of `e`...

Let \( V \) denote the set of occurring (classes of) constants, functions as well as applications of constructors and operators. As our lattice, we choose:

\[ V = 2^V \]

As usual, we put up a constraint system:

- If `e` is a value, i.e., of the form: \( b, e_1, \ldots, e_k, (e_1, \ldots, e_k) \), an operator application or \( \text{fun } x \to e \) we generate the constraint:
  \[ [e]^{\widetilde{\nu}} \supseteq \{ e \} \]

- If `e` \( \equiv (e_1 e_2) \) and \( f \equiv \text{fun } x \to e' \), then
  \[
  [e]^{\widetilde{\nu}} \supseteq (f \in [e_1]^{\widetilde{\nu}}) \wedge [e_2]^{\widetilde{\nu}} : 0 \\
  [e]^{\widetilde{\nu}} \supseteq (f \in [e_1]^{\widetilde{\nu}}) \wedge [e_2]^{\widetilde{\nu}} : 0
  \]

Let \( V \) denote the set of occurring (classes of) constants, functions as well as applications of constructors and operators. As our lattice, we choose:

\[ V = 2^V \]

As usual, we put up a constraint system:

- If `e` is a value, i.e., of the form: \( b, e_1, \ldots, e_k, (e_1, \ldots, e_k) \), an operator application or \( \text{fun } x \to e \) we generate the constraint:
  \[ [e]^{\widetilde{\nu}} \supseteq \{ e \} \]

- If `e` \( \equiv (e_1 e_2) \) and `f` \( \equiv \text{fun } x \to e' \), then
  \[
  [e]^{\widetilde{\nu}} \supseteq (f \in [e_1]^{\widetilde{\nu}}) \wedge [e_2]^{\widetilde{\nu}} : 0 \\
  [e]^{\widetilde{\nu}} \supseteq (f \in [e_1]^{\widetilde{\nu}}) \wedge [e_2]^{\widetilde{\nu}} : 0
  \]

...
If $c \equiv \text{let } x = e \text{ in } e_0$ then we generate:

\[
[x_1]^I \supseteq [e_1]^I \\
[e]^I \supseteq [e_0]^I
\]

Analogously for $t \equiv \text{letrec } x_1 = e_1 \ldots x_k = e_k \text{ in } e_0$:

\[
[x_1]^I \supseteq [e_1]^I \\
[t]^I \supseteq [e_0]^I
\]

If $p_i \equiv c y_1 \ldots y_k$ and $v \equiv e e'_1 \ldots e'_k$ is a value, then

\[
[e]^I \supseteq (v \in [e_0]^I)?[e_1]^I : \emptyset \\
[y_j]^I \supseteq (v \in [e_0]^I)?[e'_j]^I : \emptyset
\]

If $p_i \equiv (y_1, \ldots, y_k)$ and $v \equiv (e'_1, \ldots, e'_k)$ is a value, then

\[
[e]^I \supseteq (v \in [e_0]^I)?[e_1]^I : \emptyset \\
[y_j]^I \supseteq (v \in [e_0]^I)?[e'_j]^I : \emptyset
\]

If $p_i \equiv y$, then

\[
[e]^I \supseteq [e_0]^I \\
[y]^I \supseteq [e_0]^I
\]

int-values returned by operators are described by the unevaluated expression:

Operator applications might return Boolean values or other basic values. Therefore, we do replace tests for basic values by non-deterministic choice ...

Assume $e \equiv \text{match } e_0 \text{ with } p_1 \rightarrow e_1 | \ldots | p_k \rightarrow e_k$.
Then we generate for $p_i \equiv b$ (basic value),

\[
[e]^I \supseteq [e_0]^I
\]

...
Example

The `append` Function

Consider the concatenation of two lists. In OCaml, we would write:

```ocaml
let rec append = fun x -> match x with
  | []      -> fun y -> y
  | h :: t   -> fun y -> h :: append t y
in append [1; 2] [3]
```

The analysis then results in:

```
[append]^2 = \{fun x -> match \ldots\}
[x]^2 = \{[1; 2], [2; []]\}
[match \ldots]^2 = \{fun y -> y, fun y -> h :: append \ldots\}
[y]^2 = \{[3]\}
... 
```

Values \(c e_1 \ldots e_k\) or operator applications \(e_1 \& e_2\)
now are interpreted as recursive calls \(c [e_1]^2 \ldots [e_k]^2, ([e_1]^2, \ldots, [e_k]^2)\)
or \([e_1]^2 \& [e_2]^2\), respectively.

---

... in the Example:

We obtain for \(A = [append t y]^2\):

```
A  \rightarrow [3]  \mid [h^2 :: A]
[h]^2 \rightarrow 1  \mid 2
```

Let \(\mathcal{L}(e)\) denote the set of terms derivable from \([e]^2\) w.r.t. the regular tree grammar. Thus, e.g.,

\(\mathcal{L}(h) = \{1, 2\}\)
\(\mathcal{L}([append t y]) = \{[a_1; \ldots; a_r; 3] \mid r \geq 0, a_i \in \{1, 2\}\}\)
Example

The \textit{append}-Function

Consider the concatenation of two lists. In \texttt{Ocam}, we would write:

\begin{verbatim}
let rec app = fun x -> match x with
    | []      -> fun y -> y
    | h::t    -> fun y -> h :: app t y

in app [1;2] [3]
\end{verbatim}

The analysis then results in:

\begin{align*}
[\text{app}^2] &= \{\text{fun } x \to \text{match } \ldots\} \\
[\text{x}^2] &= \{[1;2], [2], []\} \\
[\text{match } \ldots] &= \{\text{fun } y \to y, \text{fun } y \to \text{h :: app } \ldots\} \\
[\text{y}^2] &= \{[3]\}
\end{align*}

... in the Example:

We obtain for $A = [\text{app } t y]^2$:

\begin{align*}
A & \rightarrow [3] \mid [h]^2 :: A \\
[h]^2 & \rightarrow 1 \mid 2
\end{align*}

Let $\mathcal{L}(c)$ denote the set of terms derivable from $[c]^2$ w.r.t. the regular tree grammar. Thus, e.g.,

\begin{align*}
\mathcal{L}(h) &= \{1, 2\} \\
\mathcal{L}(\text{app } t y) &= \{[a_1; \ldots; a_3; 3] \mid r \geq 0, a_i \in \{1, 2\}\}
\end{align*}

4.3 An Operational Semantics

Idea:

We construct a \texttt{Big-Step} operational semantics which evaluates expressions w.r.t. an environment $\cdot$:

Values are of the form:

\begin{align*}
v &::= b \mid c v_1 \ldots c v_k \mid (v_1, \ldots, v_k) \mid (\text{fun } x \to c, \eta)
\end{align*}

Examples for Values:

\begin{align*}
c 1 \\
[1;2] &= :: 1 :: 2 [] \\
(\text{fun } x \to x : y, \{y \mapsto [5]\})
\end{align*}

Expressions are evaluated w.r.t. an environment $\eta : \text{Vars } \rightarrow \text{Values}$.

The \texttt{Big-Step} operational semantics provides rules to infer the value to which an expression is evaluated w.r.t. a given environment, i.e., deals with statements of the form:

\begin{align*}
(c, \eta) \Rightarrow v
\end{align*}

Values:

\begin{align*}
(b, \eta) \Rightarrow b \\
(\text{fun } x \to c, \eta) \Rightarrow (\text{fun } x \to c, \eta) \\
(c_1, \eta) \Rightarrow c_1 \ldots (c_k, \eta) \Rightarrow c_k
\end{align*}

Operator applications are treated analogously!
Expressions are evaluated w.r.t. an environment $\eta : \text{Vars} \rightarrow \text{Values}$.

The Big Step operational semantics provides rules to infer the value to which an expression is evaluated w.r.t. a given environment, i.e., deals with statements of the form:

$$(c, \eta) \Rightarrow v$$

**Values:**

$$(b, \eta) \Rightarrow b$$

$$(\text{fun } x \rightarrow e, \eta) \Rightarrow (\text{fun } x \rightarrow e, \eta)$$

$$(e_1, \eta) \Rightarrow v_1, \ldots, (e_k, \eta) \Rightarrow v_k$$

$$(c_1 \ldots c_k, \eta) \Rightarrow c_1 \ldots c_k$$

Operator applications are treated analogously!

**Function Application:**

$$(c_1, \eta) \Rightarrow (\text{fun } x \rightarrow e, \eta_1)$$

$$(c_2, \eta) \Rightarrow v_2$$

$$(e, \eta_1 \oplus \{x \mapsto v_2\}) \Rightarrow v_3$$

$$(c_1, c_2, \eta) \Rightarrow v_3$$

$$(e_1, \eta) \Rightarrow \delta_1, (e_2, \eta) \Rightarrow \delta_2, v_1 + v_2 \Rightarrow$$

$$(e_1 + e_2, \eta) \Rightarrow \delta$$

**Global Definition:**

let rec ... $x = e$ ... in ...

$$(e, \emptyset) \Rightarrow v$$

$$(x, \eta) \Rightarrow v$$
Case Distinction 1:

\[(e, \eta) \Rightarrow b\]
\[(e_i, \eta) \Rightarrow v_i\]

\[(\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k, \eta) \Rightarrow v\]

if \( p_i \equiv b \) is the first pattern which matches \( b \) :-)

Case Distinction 2:

\[(e, \eta) \Rightarrow c v_1 \ldots v_k\]
\[(e_i, \eta \oplus \{ z_1 \mapsto v_1, \ldots, z_k \mapsto v_k \}) \Rightarrow v\]

\[(\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k, \eta) \Rightarrow v\]

if \( p_i \equiv c z_1 \ldots z_k \) is the first pattern which matches \( c v_1 \ldots v_k \) :-)

Case Distinction 4:

\[(e, \eta) \Rightarrow v'\]
\[(e_i, \eta \oplus \{ x \mapsto v \}) \Rightarrow v\]

\[(\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k, \eta) \Rightarrow v\]

if \( p_i \equiv x \) is the first pattern which matches \( v' \) :-)

Local Definitions:

\[(e_1, \eta) \Rightarrow v_1\]
\[(e_0, \eta \oplus \{ x_1 \mapsto v_1 \}) \Rightarrow v_0\]

\[(\text{let } x_1 = e_1 \text{ in } e_0, \eta) \Rightarrow v_0\]

Variables:

\[(x, \eta) \Rightarrow \eta(x)\]
4.4 Application: Inlining

Problem:

- global variables. The program:

\[
\begin{align*}
&\text{let } x = 1 \\
&\text{in let } f = \text{let } x = 2 \\
&\text{in fun } y \rightarrow y + x \\
&\text{in } f x
\end{align*}
\]

... computes something else than:

\[
\begin{align*}
&\text{let } x = 1 \\
&\text{in let } f = \text{let } x = 2 \\
&\text{in fun } y \rightarrow y + x \\
&\text{in let } y = x \\
&\text{in } y + x
\end{align*}
\]

- recursive functions. In the definition:

\[
\text{foo } = \text{fun } y \rightarrow \text{foo } y
\]

foo should better not be substituted :-(

Conclusion:

\[\mathcal{L}(e)\] returns a superset of the values to which \(e\) is evaluated :-)

Correctness of the Analysis:

For every \((e, \eta)\) occurring in a proof for the program, it should hold:

- If \(\eta(x) = v\), then \([v] \Delta \mathcal{L}(x)\).
- If \((e, \eta) \rightarrow v\), then \([v] \Delta \mathcal{L}(e)\)...

where \([v]\) is the stripped expression corresponding to \(v\), i.e., obtained by removing all environments, and

\(v \Delta L \text{ iff } v \in L \text{ or } L \) has an expression \(v'\) which evaluates to \(v\).
... computes something else than:

```
let x = 1
in let f = let x = 2
        in fun y → y + x
```

- recursive functions. In the definition:
  
  ```
  foo = fun y → foo y
  ```

  foo should better not be substituted  ::)

### 4.4 Application: Inlining

**Problem:**

- global variables. The program:
  
  ```
  let x = 1
  in let f = let x = 2
       in fun y → y + x
  ```

  ```
  in f x
  ```

**Idea 1:**

- First, we introduce unique variable names.
- Then, we only substitute functions which are statically within the scope of the same global variables as the application  ::)
- For every expression, we determine all function definitions with this property  ::)
4.4 Application: Inlining

Problem:

- global variables. The program:

```plaintext
let x = 1
in let y = let x = 2
  in fun y -> y + x
  in f x
```

Idea 1:

→ First, we introduce unique variable names.
→ Then, we only substitute functions which are statically within the scope of the same global variables as the application.
→ For every expression, we determine all function definitions with this property. :}

Idea 1:

→ First, we introduce unique variable names.
→ Then, we only substitute functions which are statically within the scope of the same global variables as the application.
→ For every expression, we determine all function definitions with this property. :}
Idea 1:

- First, we introduce unique variable names.
- Then, we only substitute functions which are statically within the scope of the same global variables as the application.
- For every expression, we determine all function definitions with this property.

Let $D = D[e]$ denote the set of definitions which statically arrive at $e$.

- If $e \equiv \text{let } x_1 = e_1 \text{ in } e_0$ then:
  
  $D[e_1] = D$
  
  $D[e_0] = D \cup \{x_1\}$

- If $e \equiv \text{fun } x \rightarrow e_1$ then:
  
  $D[e_1] = D \cup \{x\}$

- Similarly, for $e \equiv \text{match } e_0 \cdot e_1 \ldots e_k \rightarrow e_i \ldots$
  
  $D[e_i] = D \cup \{x_1, \ldots, x_k\}$

In all other cases, $D$ is propagated to the sub-expressions unchanged.

... in the Example:

$$
\begin{align*}
\text{let } & x = 1 \\
\text{in let } & f = \text{let } x_1 = 2 \\
& \quad \text{in fun } y \rightarrow y + x_1 \\
& \quad \text{in } f x
\end{align*}
$$

... the application $f x$ is not in the scope of $x_1$

$\implies$ we first duplicate the definition of $x_1$.

$\implies$ the inner definition becomes redundant!!!
let $x = 1$

in let $x_1 = 2$

in let $f = \text{fun } y \rightarrow y + x_1$

in \hspace{1cm} $f \; x$

\[ \rightarrow \text{ now we can apply inlining:} \]

\[ \text{let } x = 1 \]

\[ \text{in let } x_1 = 2 \]

\[ \text{in let } f = \text{fun } y \rightarrow y + x_1 \]

\[ \text{in } \begin{array}{l}
\text{let } y = x \\
\text{in } y + x_1
\end{array} \]

Removing variable-variable-assignments, we arrive at: