Script generated by TTT

Title: Seidl: Programmoptimierung (16.01.2012)

Date: Mon Jan 16 12:30:36 CET 2012

Duration: 90:32 min

Pages: 42

3.4 Wrap-Up

We have considered various optimizations for improving hardware utilization.

Arrangement of the Optimizations:

- First, global restructuring of procedures/functions and of loops for better memory behavior ;-)
- Then local restructuring for better utilization of the instruction set and the processor parallelism :-)
- Then register allocation and finally,
- Peephole optimization for the final kick ...

Procedures:	Tail Recursion + Inlining Stack Allocation
Loops:	Iteration Reordering $\to$ if-Distribution $\to$ for-Distribution Value Caching
Bodies:	Life-Range Splitting (SSA) Instruction Selection Instruction Scheduling with $\to$ Loop Unrolling $\to$ Loop Fusion
Instructions:	Register Allocation Peephole Optimization

4 Optimization of Functional Programs

Example:

$$\begin{array}{lll} \textbf{let rec } \mathsf{fac}\ x & = & \textbf{if } x \leq 1 \textbf{ then } 1 \\ & & \textbf{else } x \cdot \mathsf{fac}\ (x-1) \end{array}$$

- There are no basic blocks :-(
- There are no loops :-(
- Virtually all functions are recursive :-((

Strategies for Optimization:

⟹ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

⟹ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

f x = match x with
a → _ _
| b → _ _
| c → _ _
| d → _ _

Strategies for Optimization:

⟹ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

⟹ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

782

Strategies for Optimization:

⟹ Improve specific inefficiencies such as:

- Pattern matching
- Lazy evaluation (if supported ;-)
- Indirections — Unboxing / Escape Analysis
- Intermediate data-structures — Deforestation

⟹ Detect and/or generate loops with basic blocks :-)

- Tail recursion
- Inlining
- **let**-Floating

Then apply general optimization techniques

... e.g., by translation into C ;-)

782

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{array}{lll} \textbf{let } \mathsf{max}\,(x,y) & = & \textbf{if } x>y \textbf{ then } x \\ & & \textbf{else } y \\ \textbf{let } \mathsf{abs}\, z & = & \mathsf{max}\,(z,-z) \end{array}$$

As result of the optimization we expect ...

$$\begin{array}{lll} \textbf{let } \mathsf{max}\,(x,y) & = & \textbf{if } x>y \textbf{ then } x \\ & & \textbf{else } y \\ \textbf{let } \mathsf{abs}\, z & = & \quad\textbf{let } \; x = z \\ & & \textbf{in let } \; y = -z \\ & & \textbf{in } \; \boxed{\begin{array}{l}\textbf{if } x>y \textbf{ then } x \\ \textbf{else } y\end{array}} \end{array}$$

Discussion:

For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!

Warning:

Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

$$\begin{array}{lll} \textbf{let } \mathsf{max}\,(x,y) & = & \textbf{if } x>y \textbf{ then } x \\ & & \textbf{else } y \\ \textbf{let } \mathsf{abs}\, z & = & \mathsf{max}\,(z,-z) \end{array}$$

As result of the optimization we expect ...

783

$$\begin{array}{lll} \textbf{let } \mathsf{max}\,(x,y) & = & \textbf{if } x>y \textbf{ then } x \\ & & \textbf{else } y \\ \textbf{let } \mathsf{abs}\, z & = & \quad\textbf{let } \; x = z \\ & & \textbf{in let } \; y = -z \\ & & \textbf{in } \; \boxed{\begin{array}{l}\textbf{if } x>y \textbf{ then } x \\ \textbf{else } y\end{array}} \end{array}$$

Discussion:

For the beginning, max is just a name. We must find out which value it takes at run-time

$\Longrightarrow$ Value Analysis required !!

784

Warning:

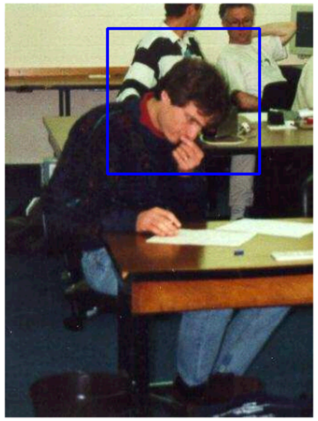
Novel analysis techniques are needed to collect information about functional programs.

Example: Inlining

fun $(x, y) \rightarrow$

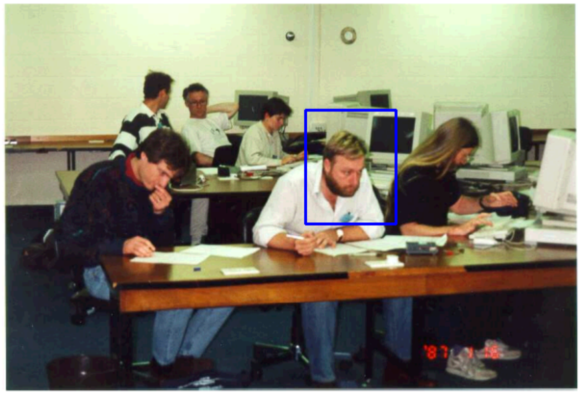
$$
\begin{array}{lll}
\textbf{let}\ \mathsf{max}\ \cancel{(x, y)} & = & \textbf{if}\ x > y\ \textbf{then}\ x \\
 & & \textbf{else}\ y \\
\textbf{let}\ \mathsf{abs}\ z & = & \mathsf{max}\ (z, -z)
\end{array}
$$

As result of the optimization we expect ...



Nevin Heintze in the Australian team
of the Prolog-Programming-Contest, 1998

The complete picture:



4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lcl}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1\, \ldots\, e_k \mid \textbf{fun}\, x \rightarrow e \\
 & & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1\, \Box_2\, e_2) \mid \\
 & & \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0 \mid \\
 & & \textbf{match}\ e_0\ \textbf{with}\ p_1 \rightarrow e_1 \ \mid \ldots \mid\ p_k \rightarrow e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \textbf{let rec}\ x_1 = e_1\ \textbf{and} \ldots \textbf{and}\ x_k = e_k\ \textbf{in}\ e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

f x = match x with
a → ...
| b → ...
| c → ...
| d → ...

4.1 A Simple Functional Language

For simplicity, we consider:

$$
\begin{array}{lll}
e & ::= & b \mid (e_1, \ldots, e_k) \mid c\, e_1 \ldots e_k \mid \textbf{fun}\; x \to e \\
  &   & \mid (e_1\, e_2) \mid (\Box_1\, e) \mid (e_1 \Box_2 e_2) \mid \\
  &   & \textbf{let}\; x_1 = e_1\; \textbf{in}\; e_0 \mid \\
  &   & \textbf{match}\; e_0\; \textbf{with}\; p_1 \to e_1 \mid \ldots \mid p_k \to e_k \\
p & ::= & b \mid x \mid c\, x_1 \ldots x_k \mid (x_1, \ldots, x_k) \\
t & ::= & \textbf{let rec}\; x_1 = e_1\; \textbf{and} \ldots \textbf{and}\; x_k = e_k\; \textbf{in}\; e
\end{array}
$$

where b is a constant, x is a variable, c is a (data-)constructor and $\Box_i$ are i-ary operators.

let max = fun z →
match z with
(x, y) → if x > y
then x
else y

Discussion:

- **let rec** only occurs on top-level.
- Functions are always unary. Instead, there are explicit tuples :-)
- **if**-expressions and case distinction in function definitions is reduced to **match**-expressions.
- In case distinctions, we allow just simple patterns.

  ⟹ Complex patterns must be decomposed ...
- **let**-definitions correspond to basic blocks :-)
- Type-annotations at variables, patterns or expressions could provide further useful information

  — which we ignore :-)

let max = fun z →
match z with
(x, y) → match x > y with
true → x
| false → y

Discussion:

- **let rec** only occurs on top-level.
- Functions are always unary. Instead, there are explicit tuples :-)
- **if**-expressions and case distinction in function definitions is reduced to **match**-expressions.
- In case distinctions, we allow just simple patterns.

  $\Longrightarrow$ Complex patterns must be decomposed ...
- **let**-definitions correspond to basic blocks :-)
- Type-annotations at variables, patterns or expressions could provide further useful information

  — which we ignore :-)

... in the Example:

A definition of `max` may look as follows:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{max}\ =\ \textbf{fun}\ x \to\ \ \textbf{match}\ x\ \textbf{with}\ (x_1, x_2)\ \to\ (\\
\qquad\qquad\qquad\qquad \textbf{match}\ x_1 < x_2\\
\qquad\qquad\qquad\qquad \textbf{with}\ \ \mathsf{True}\ \to\ \ x_2\\
\qquad\qquad\qquad\qquad\ \ \ \mid\ \ \ \mathsf{False}\ \to\ \ x_1\\
\qquad\qquad\qquad\qquad)
\end{array}
$$

Accordingly, we have for `abs` :

max

$$
\begin{array}{l}
\textbf{let}\ \mathsf{abs}\ =\ \textbf{fun}\ x \to\ \ \cancel{\textbf{let}\ z =}\ (x, -x)\\
\qquad\qquad\qquad\qquad\quad \cancel{\textbf{in}\ \mathsf{max}}\ z
\end{array}
$$

4.2 A Simple Value Analysis

Idea:

For every subexpression e we collect the set $[\![e]\!]^\sharp$ 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:

$$\mathbb{V}\ =\ 2^V$$

As usual, we put up a constraint system:

- If e is a value, i.e., of the form: $b, c\, e_1 \ldots e_k, (e_1, \ldots, e_k)$, an operator application or $\textbf{fun}\ x\ \to\ e$ we generate the constraint:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- If $e \equiv (e_1\ e_2)$ and $f \equiv \textbf{fun}\ x\ \to\ e'$, then

$$
\begin{array}{lll}
[\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\ ?\ [\![e']\!]^\sharp\ :\ \emptyset\\
[\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\ ?\ [\![e_2]\!]^\sharp\ :\ \emptyset
\end{array}
$$

...

Accordingly, we have for abs :

$$\begin{array}{lll} \textbf{let } \textsf{abs} = \textbf{fun } x \to & \textbf{let } z = (x, -x) \\ & \textbf{in } \textsf{max}\; z \end{array}$$

$\boxed{\text{fun } y \to x+y}$

$f\,x$

4.2 A Simple Value Analysis

Idea:

For every subexpression e we collect the set $[\![e]\!]^\sharp$ 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:

$$\mathbb{V} = 2^V$$

As usual, we put up a constraint system:

- If e is a value, i.e., of the form: $b, c\, e_1 \ldots e_k, (e_1, \ldots, e_k)$, an operator application or $\textbf{fun } x \to e$ we generate the constraint:

$$[\![e]\!]^\sharp \supseteq \{e\}$$

- If $e \equiv (e_1\, e_2)$ and $f \equiv \textbf{fun } x \to e'$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\, ?\, [\![e']\!]^\sharp : \emptyset \\ [\![x]\!]^\sharp & \supseteq & (f \in [\![e_1]\!]^\sharp)\, ?\, [\![e_2]\!]^\sharp : \emptyset \end{array}$$

...

- 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 ...

- If $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_0$, then we generate:

$$\begin{array}{lll} [\![x_1]\!]^\sharp & \supseteq & [\![e_1]\!]^\sharp \\ [\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- Assume $e \equiv \textbf{match } e_0 \textbf{ with } p_1 \to e_1 \mid \ldots \mid p_k \to e_k$.
  Then we generate for $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp$$

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

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\, ?\, [\![e_i]\!]^\sharp : \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\, ?\, [\![e'_j]\!]^\sharp : \emptyset \end{array}$$

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

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\, ?\, [\![e_i]\!]^\sharp : \emptyset \\ [\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\, ?\, [\![e'_j]\!]^\sharp : \emptyset \end{array}$$

If $p_i \equiv y$, then

$$\begin{array}{lll} [\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\ [\![y]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp \end{array}$$

- 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 ...

- If $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0$, then we generate:

$$
\begin{array}{lll}
[\![x_1]\!]^\sharp & \supseteq & [\![e_1]\!]^\sharp \\
[\![e]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp
\end{array}
$$

- Assume $e \equiv \textbf{match}\ e_0\ \textbf{with}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k$.
  Then we generate for $p_i \equiv b$,

$$[\![e]\!]^\sharp \supseteq [\![e_i]\!]^\sharp$$

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

$$
\begin{array}{lll}
[\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e_i]\!]^\sharp\ :\ \emptyset \\
[\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e'_j]\!]^\sharp\ :\ \emptyset
\end{array}
$$

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

$$
\begin{array}{lll}
[\![e]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e_i]\!]^\sharp\ :\ \emptyset \\
[\![y_j]\!]^\sharp & \supseteq & (v \in [\![e_0]\!]^\sharp)\ ?\ [\![e'_j]\!]^\sharp\ :\ \emptyset
\end{array}
$$

If $p_i \equiv y$, then

$$
\begin{array}{lll}
[\![e]\!]^\sharp & \supseteq & [\![e_i]\!]^\sharp \\
[\![y]\!]^\sharp & \supseteq & [\![e_0]\!]^\sharp
\end{array}
$$

Example The append-Function

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

$$
\begin{array}{l}
\textbf{let rec}\ \mathsf{app} = \textbf{fun}\ x \to \textbf{match}\ x\ \textbf{with} \\
\qquad\qquad\qquad\qquad [\,] \quad \to \textbf{fun}\ y \to y \\
\qquad\qquad\qquad\quad \mid h :: t \to \textbf{fun}\ y \to h :: \mathsf{app}\ t\ y \\
\textbf{in}\ \mathsf{app}\ [1;2]\ [3]
\end{array}
$$

The analysis then results in:

$$
\begin{array}{lll}
[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun}\ x \to \textbf{match} \ldots\} \\
[\![x]\!]^\sharp & = & \{[1;2], [2], [\,]\} \\
[\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun}\ y \to y, \textbf{fun}\ y \to h :: \mathsf{app} \ldots\} \\
[\![y]\!]^\sharp & = & \{[3]\} \\
\ldots & &
\end{array}
$$

$$
\begin{array}{lll}
\ldots & & \\
[\![h]\!]^\sharp & = & \{1, 2\} \\
[\![t]\!]^\sharp & = & \{[2], [\,]\} \\
[\![\mathsf{app}\ t]\!]^\sharp & = & \\
[\![\mathsf{app}\ [1;2]]\!]^\sharp & = & \{\textbf{fun}\ y \to y, \textbf{fun}\ y \to h :: \mathsf{app} \ldots\} \\
[\![\mathsf{app}\ t\ y]\!]^\sharp & = & \\
[\![\mathsf{app}\ [1;2]\ [3]]\!]^\sharp & = & \{[3], h :: \mathsf{app} \ldots\}
\end{array}
$$

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

$\Longrightarrow$ regular tree grammar

Example The append-Function

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

$$
\begin{array}{ll}
\textbf{let rec } \mathsf{app} = \textbf{fun } x \to \textbf{match } x \textbf{ with} & \\
\qquad\qquad [\,] \to \textbf{fun } y \to y & \\
\qquad\quad\ \mid h :: t \to \textbf{fun } y \to h :: \mathsf{app}\; t\; y & \\
\textbf{in } \mathsf{app}\; [1;2]\; [3] &
\end{array}
$$

The analysis then results in:

$$
\begin{array}{lcl}
[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun } x \to \textbf{match} \ldots\} \\
[\![x]\!]^\sharp & = & \{[1;2], [2], [\,]\} \\
[\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun } y \to y, \textbf{fun } y \to h :: \mathsf{app} \ldots\} \\
[\![y]\!]^\sharp & = & \{[3]\} \\
\ldots & &
\end{array}
$$

$$
\begin{array}{lcl}
\ldots & & \\
[\![h]\!]^\sharp & = & \{1,2\} \\
[\![t]\!]^\sharp & = & \{[2], [\,]\} \\
[\![\mathsf{app}\; t]\!]^\sharp & = & \\
[\![\mathsf{app}\; [1;2]]\!]^\sharp & = & \{\textbf{fun } y \to y, \textbf{fun } y \to h :: \mathsf{app} \ldots\} \\
[\![\mathsf{app}\; t\; y]\!]^\sharp & = & \\
[\![\mathsf{app}\; [1;2]\; [3]]\!]^\sharp & = & \{[3], h :: \mathsf{app} \ldots\}
\end{array}
$$

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

$\Longrightarrow$ regular tree grammar

Example The append-Function

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

$$
\begin{array}{ll}
\textbf{let rec } \mathsf{app} = \textbf{fun } x \to \textbf{match } x \textbf{ with} & \\
\qquad\qquad [\,] \to \textbf{fun } y \to y & \\
\qquad\quad\ \mid h :: t \to \textbf{fun } y \to h :: \mathsf{app}\; t\; y & \\
\textbf{in } \mathsf{app}\; [1;2]\; [3] &
\end{array}
$$

The analysis then results in:

$$
\begin{array}{lcl}
[\![\mathsf{app}]\!]^\sharp & = & \{\textbf{fun } x \to \textbf{match} \ldots\} \\
[\![x]\!]^\sharp & = & \{[1;2], [2], [\,]\} \\
[\![\textbf{match} \ldots]\!]^\sharp & = & \{\textbf{fun } y \to y, \textbf{fun } y \to h :: \mathsf{app} \ldots\} \\
[\![y]\!]^\sharp & = & \{[3]\} \\
\ldots & &
\end{array}
$$

794

$A \to B :: [\,] \mid B :: C$

$B \to 1 \mid 2 \qquad C \to 3 :: [\,] \mid B :: C$

$$
\begin{array}{lcl}
\ldots & & \\
[\![h]\!]^\sharp & = & \{1,2\} \\
[\![t]\!]^\sharp & = & \{[2], [\,]\} \\
[\![\mathsf{app}\; t]\!]^\sharp & = & \\
[\![\mathsf{app}\; [1;2]]\!]^\sharp & = & \{\textbf{fun } y \to y, \textbf{fun } y \to h :: \mathsf{app} \ldots\} \\
[\![\mathsf{app}\; t\; y]\!]^\sharp & = & \\
[\![\underbrace{\boxed{\mathsf{app}\; [1;2]\; [3]}}_{A}]\!]^\sharp & = & \{[3], \underbrace{\boxed{h}}_{B} :: \underbrace{\boxed{\mathsf{app} \ldots}}_{C}\}
\end{array}
$$

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

$\Longrightarrow$ regular tree grammar

795

... in the Example:

We obtain for $A = [\![\mathsf{app}\,t\,y]\!]^\sharp$:

$$\begin{array}{lll} A & \to & [3] \mid [\![h]\!]^\sharp :: A \\ [\![h]\!]^\sharp & \to & 1 \mid 2 \end{array}$$

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

$$\begin{array}{lll} \mathcal{L}(h) & = & \{1,2\} \\ \mathcal{L}(\mathsf{app}\,t\,y) & = & \{[a_1;\ldots,a_r;3] \mid r \geq 0, a_i \in \{1,2\}\} \end{array}$$

... in the Example:

We obtain for $A = [\![\mathsf{app}\,t\,y]\!]^\sharp$:

$$\begin{array}{lll} A & \to & [3] \mid [\![h]\!]^\sharp :: A \\ [\![h]\!]^\sharp & \to & 1 \mid 2 \end{array}$$

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

$$\begin{array}{lll} \mathcal{L}(h) & = & \{1,2\} \\ \mathcal{L}(\mathsf{app}\,t\,y) & = & \{[a_1;\ldots,a_r;3] \mid r \geq 0, a_i \in \{1,2\}\} \end{array}$$

796

... in the Example:

We obtain for $A = [\![\mathsf{app}\,t\,y]\!]^\sharp$:

$$\begin{array}{lll} A & \to & [3] \mid [\![h]\!]^\sharp :: A \\ [\![h]\!]^\sharp & \to & 1 \mid 2 \end{array}$$

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

$$\begin{array}{lll} \mathcal{L}(h) & = & \{1,2\} \\ \mathcal{L}(\mathsf{app}\,t\,y) & = & \{[a_1;\ldots,a_r;3] \mid r \geq 0, a_i \in \{1,2\}\} \end{array}$$

796

4.3 An Operational Semantics

Idea:

We construct a Big-Step operational semantics which evaluates expressions w.r.t. an environment :-)

Values are of the form:

$$v ::= b \mid c\,v_1 \ldots c_k \mid (v_1, \ldots, v_k) \mid (\mathbf{fun}\,x \to e, \eta)$$

Examples for Values:

$$\begin{array}{l} c\,1 \\ [1;2] = ::\,1\,(::\,2\,[\,]) \\ (\mathbf{fun}\,x \to x{::}y, \{y \mapsto [5]\}) \end{array}$$

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

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:

$$(e, \eta) \Longrightarrow v$$

Values:

$$(b, \eta) \Longrightarrow b$$

$$(\mathbf{fun}\, x \to e, \eta) \Longrightarrow (\mathbf{fun}\, x \to e, \eta)$$

$$\frac{(e_1, \eta) \Longrightarrow v_1 \quad \ldots \quad (e_k, \eta) \Longrightarrow v_k}{(c\, e_1 \ldots e_k, \eta) \Longrightarrow c\, v_1 \ldots v_k}$$

Operator applications are treated analogously!

$$\frac{(e_1, \eta) \Longrightarrow v_1 \quad \ldots \quad (e_k, \eta) \Longrightarrow v_k}{((e_1, \ldots, e_k), \eta) \Longrightarrow (v_1, \ldots, v_k)}$$

Global Definition:

$$\frac{\begin{array}{l}\mathbf{let\ rec}\ \ldots\ x = e\ \ldots\ \mathbf{in}\ \ldots \\ (e, \emptyset) \Longrightarrow v\end{array}}{(x, \eta) \Longrightarrow v}$$

$$\frac{(e_1, \eta) \Longrightarrow v_1 \quad \ldots \quad (e_k, \eta) \Longrightarrow v_k}{((e_1, \ldots, e_k), \eta) \Longrightarrow (v_1, \ldots, v_k)}$$

Global Definition:

$$\frac{\begin{array}{l}\mathbf{let\ rec}\ \ldots\ x = e\ \ldots\ \mathbf{in}\ \ldots \\ (e, \emptyset) \Longrightarrow v\end{array}}{(x, \eta) \Longrightarrow v}$$

799