# Script generated by TTT

Title: Seidl: Programmoptimierung (21.01.2013)

Date: Mon Jan 21 14:03:33 CET 2013

Duration: 89:07 min

Pages: 45

# (17)

# 4.1 A Simple Functional Language

For simplicity, we consider:

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

# 4.1 A Simple Functional Language

For simplicity, we consider:

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

785

# 4.1 A Simple Functional Language

For simplicity, we consider:

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

#### ... in the Example:

A definition of max may look as follows:

787

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:  $b, c\,e_1\dots e_k, (e_1,\dots,e_k)$ , an operator application or  $\operatorname{fun} x \to e$  we generate the constraint:

$$\llbracket e \rrbracket^{\sharp} \supseteq \{e\}$$

• If  $e \equiv (e_1 \ e_2)$  and  $f \equiv \operatorname{fun} x \to e'$ , then  $\llbracket e \rrbracket^\sharp \ \supseteq \ (f \in \llbracket e_1 \rrbracket^\sharp) \, ? \, \llbracket e' \rrbracket^\sharp \, : \, \emptyset$ 

 $[x]^{\sharp} \supseteq (f \in [e_1]^{\sharp}) ? [e_2]^{\sharp} : \emptyset$ 

...

789

Accordingly, we have for abs:

$$\text{let abs} \ = \ \text{fun} \ x \to \quad \text{let} \ z = (x, -x)$$
 
$$\text{in max} \ z )$$

## 4.2 A Simple Value Analysis

Idea:

For every subexpression  $\ e$  we collect the set  $\ [\![e]\!]^\sharp$  of possible values of  $\ e$  ...

788

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:  $b, c\,e_1\dots e_k, (e_1,\dots,e_k)$ , an operator application or  $\operatorname{fun} x \to e$  we generate the constraint:

$$\llbracket e \rrbracket^{\sharp} \supseteq \{e\}$$

• If  $e \equiv (e_1 e_2)$  and  $f \equiv \mathbf{fun} \ x \rightarrow e'$ , then

•••



• If  $e \equiv \text{let } x_1 = e_1 \text{ in } e_0$  then we generate:

$$\begin{bmatrix} x_1 \end{bmatrix}^{\sharp} \supseteq \begin{bmatrix} e_1 \end{bmatrix}^{\sharp} \\
\begin{bmatrix} e \end{bmatrix}^{\sharp} \supseteq \begin{bmatrix} e_0 \end{bmatrix}^{\sharp}$$

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



790

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

$$\llbracket e \rrbracket^{\sharp} \supseteq (v \in \llbracket e_0 \rrbracket^{\sharp}) ? \llbracket e_i \rrbracket^{\sharp} : \emptyset$$

$$[y_j]^{\sharp} \supseteq (v \in [e_0]^{\sharp})? [e'_i]^{\sharp}: \emptyset$$

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

$$\llbracket e \rrbracket^{\sharp} \supseteq (v \in \llbracket e_0 \rrbracket^{\sharp}) ? \llbracket e_i \rrbracket^{\sharp} : \emptyset$$

$$[\![y_j]\!]^{\sharp} \supseteq (v \in [\![e_0]\!]^{\sharp}) ? [\![e_j']\!]^{\sharp} : \emptyset$$

If  $p_i \equiv y$ , then

$$\llbracket e 
rbracket^{\sharp} \supseteq \llbracket e_i 
rbracket^{\sharp}$$

$$\llbracket y 
rbracket^{\sharp} \supseteq \llbracket e_0 
rbracket^{\sharp}$$

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

$$\llbracket e 
rbracket^\sharp \supseteq \llbracket e_i 
rbracket^\sharp$$

...

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

$$\llbracket e \rrbracket^{\sharp} \supseteq (v \in \llbracket e_0 \rrbracket^{\sharp}) ? \llbracket e_i \rrbracket^{\sharp} : \emptyset$$

$$[y_j]^{\sharp} \supseteq (v \in [e_0]^{\sharp})? [e'_j]^{\sharp} : \emptyset$$

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

$$\llbracket e \rrbracket^{\sharp} \supseteq (v \in \llbracket e_0 \rrbracket^{\sharp}) ? \llbracket e_i \rrbracket^{\sharp} : \emptyset$$

$$\llbracket y_j \rrbracket^{\sharp} \supseteq (v \in \llbracket e_0 \rrbracket^{\sharp}) ? \llbracket e'_j \rrbracket^{\sharp} : \emptyset$$

If 
$$p_i \equiv y$$
, then

$$\llbracket e 
rbracket^{\sharp} \supseteq \llbracket e_i 
rbracket^{\sharp}$$

$$\llbracket y 
rbracket^{\sharp} \supseteq \llbracket e_0 
rbracket^{\sharp}$$

# Example The append-Function

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

```
let rec app = fun x \to \max x with [] \to \sup y \to y |h :: t \to \sup y \to h :: \mathsf{app}\, t\, y in app [1;2] [3]
```

The analysis then results in:

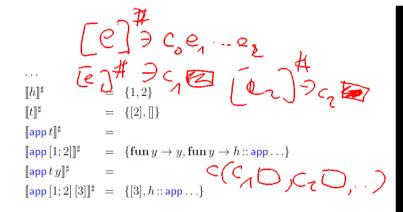
793

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

Values  $c \, e_1 \dots e_k$ ,  $(e_1, \dots, e_k)$  or operator applications  $e_1 \square e_2$  now are interpreted as recursive calls  $c \, \llbracket e_1 \rrbracket^\sharp \dots \llbracket e_k \rrbracket^\sharp$ ,  $(\llbracket e_1 \rrbracket^\sharp, \dots, \llbracket e_k \rrbracket^\sharp)$  or  $\llbracket e_1 \rrbracket^\sharp \square \llbracket e_2 \rrbracket^\sharp$ , respectively.

→ regular tree grammar

794



Values  $c \, e_1 \dots e_k$ ,  $(e_1, \dots, e_k)$  or operator applications  $e_1 \square e_2$  now are interpreted as recursive calls  $c \, \llbracket e_1 \rrbracket^\sharp \dots \llbracket e_k \rrbracket^\sharp$ ,  $(\llbracket e_1 \rrbracket^\sharp, \dots, \llbracket e_k \rrbracket^\sharp)$  or  $\llbracket e_1 \rrbracket^\sharp \square \llbracket e_2 \rrbracket^\sharp$ , respectively.

→ regular tree grammar

... in the Example:

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

$$A \rightarrow [3] \mid \llbracket h \rrbracket^{\sharp} :: A$$
$$\llbracket h \rrbracket^{\sharp} \rightarrow 1 \mid 2$$

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

$$\mathcal{L}(h) = \{1, 2\}$$

$$\mathcal{L}(\mathsf{app}\,t\,y) = \{[a_1; \dots, a_r; 3] \mid r \ge 0, a_i \in \{1, 2\}\}$$

795

# Example The append-Function

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

```
let rec app = fun x \to \max x with [] \to \sup y \to y |h :: t \to \sup y \to h :: \mathsf{app}\, t\, y in app [1;2] [3]
```

The analysis then results in:

793

## ... in the Example:

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

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

$$\mathcal{L}(h) = \{1, 2\}$$

$$\mathcal{L}(\mathsf{app}\,t\,y) = \{[a_1; \dots, a_r; 3] \mid r \ge 0, a_i \in \{1, 2\}\}$$

795

# 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:

$$v := b \mid c v_1 \dots c_k \mid (v_1, \dots, v_k) \mid (\text{fun } x \rightarrow e, \eta)$$

Examples for Values:

$$c1 \\ [1;2] = :: 1 (:: 2 []) \\ (\text{fun } x \to x :: y, \{y \mapsto [5]\})$$

796

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:

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

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

$$(e_1, \eta) \Longrightarrow v_1 \dots (e_k, \eta) \Longrightarrow v_k$$

$$(c e_1 \dots e_k, \eta) \Longrightarrow c v_1 \dots v_k$$

Operator applications are treated analogously!

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

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:

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

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

$$(e_1, \eta) \Longrightarrow v_1 \dots (e_k, \eta) \Longrightarrow v_k$$

$$(c e_1 \dots e_k, \eta) \Longrightarrow (c v_1 \dots v_k)$$

Operator applications are treated analogously!

797

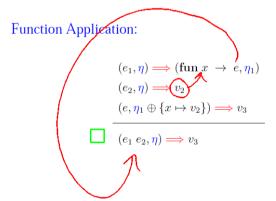
$$\frac{(e_1, \eta) \Longrightarrow v_1 \dots (e_k, \eta) \Longrightarrow v_k}{((e_1, \dots, e_k), \eta) \Longrightarrow (v_1, \dots, v_k)}$$

#### Global Definition:

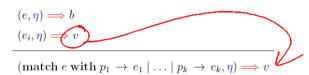
let rec ... 
$$x = e$$
 ... in ...  $(e, \emptyset) \Longrightarrow v$  
$$(x, \eta) \Longrightarrow v$$

798

 $(e_1, h) \Rightarrow \sigma_1 (e_2, h) \Rightarrow \sigma_2 c_1 + c_2 = \sigma$  $(e_1 + e_2, h) \Rightarrow (5)$ 



#### Case Distinction 1:



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

800

#### Case Distinction 2:

$$(e, \eta) \Longrightarrow c \, v_1 \dots v_k$$

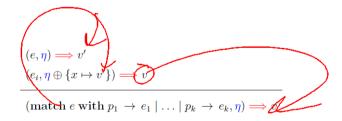
$$(e_i, \eta \oplus \{z_1 \mapsto v_1, \dots, z_k \mapsto v_k\}) \Longrightarrow v$$

$$(\text{match } e \text{ with } p_1 \to e_1 \mid \dots \mid p_k \to e_k, \eta) \Longrightarrow v$$

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

801

# Case Distinction 4:



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

# Local Definitions:

$$(e_1, \eta) \Longrightarrow v_1$$

$$(e_0, \eta \oplus \{x_1 \mapsto v_1\}) \Longrightarrow v_0$$

$$(\text{let } x_1 = e_1 \text{ in } e_0, \eta) \Longrightarrow v_0$$

Variables:

$$(x,\eta) \Longrightarrow \eta(x)$$

804

# 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  $(e, \eta) \Longrightarrow v$ , then  $[v] \Delta \mathcal{L}(e) \dots$
- where [v] is the stripped expression corresponding to v, i.e., obtained by removing all environments, and
- $\bullet \quad v \ \Delta \ L \ \text{iff} \ v \in L \ \text{or} \ L \ \text{has an expression} \ v' \ \text{which evaluates to} \ v.$

## Conclusion:

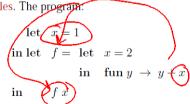
 $\mathcal{L}(e)$  returns a superset of the values to which e is evaluated :-)

805

# 4.4 Application: Inlining

#### Problem:

• global variables. The program:



4.4 Application: Inlining

#### Problem:

• global variables. The program:

$$\begin{array}{lll} & \text{let} & x=1 \\ & \text{in let} & f= & \text{let} & x=2 \\ & & \text{in} & \text{fun } y \, \rightarrow \, y+x \\ & \text{in} & f \, x \end{array}$$

806

... computes something else than:

$$\begin{array}{cccc} & \text{let} & x=1 \\ & \text{in let} & f=& \text{let} & x=2 \\ & & \text{in} & \text{fun } y \, \rightarrow \, y+x \\ & \text{in} & & \text{let} & y=x \\ & & \text{in} & y+x \end{array}$$

• recursive functions. In the definition:

$$foo = fun y \rightarrow foo y$$

foo should better not be substituted :-)

... computes something else than:

• recursive functions. In the definition:

$$foo = fun y \rightarrow foo y$$

foo should better not be substituted :-)

807

... computes something else than:

$$\begin{array}{cccc} & \text{let} & x=1 \\ & \text{in let} & f=& \text{let} & x=2 \\ & & \text{in} & \text{fun } y \, \rightarrow \, y+x \\ & \text{in} & & \text{let} & y=x \\ & & \text{in} & y+x \end{array}$$

• recursive functions. In the definition:

$$foo = foo y$$

foo should better not be substituted :-)

4.4 Application: Inlining

#### Problem:

• global variables. The program:

$$\begin{array}{ll} \text{let} & x=1 \\ \text{in let} & f=\text{ let} & x=2 \\ & \text{in} & \text{fun } y \ \rightarrow \ y+x \\ \text{in} & f \ x \end{array}$$

806

Idea 1:

- → First, we introduce unique variable names.
- → Then, we only substitute functions which are staticly within the scope of the same global variables as the application :-)
- $\rightarrow$   $\;$  For every expression, we determine all function definitions with this property  $\;$  :-)

# 4.4 Application: Inlining

#### Problem:

• global variables. The program:

let 
$$x = 1$$
  
in let  $f = 1$  let  $x = 2$   
in  $f(x) = 1$ 

806

# 4.4 Application: Inlining

#### Problem:

• global variables. The program:

$$\begin{array}{ll} \text{let} & x=1 \\ \text{in let} & f=\text{ let} & x=2 \\ & \text{in} & \text{fun } y \, \rightarrow \, y+x \\ \text{in} & f \, x \end{array}$$

Idea 1:

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

808

## Idea 1:

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

806

#### Idea 1:

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

one

... in the Example:

let 
$$x = 1$$
  
in let  $f = \text{let} \underbrace{x_1 = 2}_{\text{in } \text{fun } y \to y + x_1}$   
in  $f x$ 

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

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

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

•• If  $e \equiv \det x_1 = e_1 \text{ in } e_0$  then:

$$D[e_1] = D$$

$$D[e_0] = D \cup \{x_1\}$$

•• If  $e \equiv \operatorname{fun} x \to e_1$  then:

$$D[e_1] = D \cup \{x\}$$

•• Similarly, for  $e \equiv \operatorname{match}_{\mathbf{c_0}} c x_1 \dots x_k \to e_i \dots$ 

$$D[e_i] = D \cup \{x_1, \dots, x_k\}$$

809

 $\begin{array}{ll} \mathrm{let} & x=1 \\ \mathrm{in} \ \mathrm{let} & x_1=2 \\ \mathrm{in} \ \mathrm{let} & f= \ \mathrm{let} & x_1=2 \\ & \mathrm{in} & \mathrm{fun} \ y \ \rightarrow \ y+x_1 \\ \mathrm{in} & f \ x \end{array}$ 

→ the inner definition becomes redundant !!!

$$\begin{array}{ll} \mathrm{let} & x=1 \\ \mathrm{in} \ \mathrm{let} & \textcolor{red}{x_1}=2 \\ \mathrm{in} \ \mathrm{let} & f=\mathrm{fun} \ y \ \rightarrow \ y+\textcolor{red}{x_1} \\ \mathrm{in} & f \ x \end{array}$$

→ now we can apply inlining:

$$\begin{array}{ll} \text{let} & x=1 \\ \text{in let} & \textbf{\textit{x}}_1=2 \\ \\ \text{in let} & f=\text{fun } y \, \rightarrow \, y+\textbf{\textit{x}}_1 \\ \\ \text{in} & \text{let} & y=x \\ \\ \text{in} & y+\textbf{\textit{x}}_1 \end{array}$$

Removing variable-variable-assignments, we arrive at:

812