# Termination analysis of first order programs

Oleksandr Shturmov

January 26, 2012

$$H(M, x) = \begin{cases} \textit{true} & M \text{ halts on } x, \\ \textit{false} & M \text{ does not halt on } x. \end{cases}$$

$$H(M, x) = \begin{cases} \textit{true} & M \text{ halts on } x, \\ \textit{false} & M \text{ does not halt on } x. \end{cases}$$

$$F(M) = \begin{cases} \textit{true} & H(M, M) \rightsquigarrow \textit{false}, \\ \textit{false} & H(M, M) \rightsquigarrow \textit{true}. \end{cases}$$

Consider $F(F)$.

$$H(M, x) = \begin{cases} \textit{true} & M \text{ halts on } x, \\ \textit{false} & M \text{ does not halt on } x, \\ \textit{unknown} & M \text{ may or may not halt on } x. \end{cases}$$

$$H(M, x) = \begin{cases} \textit{true} & M \text{ halts on } x, \\ \textit{unknown} & M \text{ may or may not halt on } x. \end{cases}$$

“Unfortunately, many have drawn too strong of a conclusion about the prospects of automatic program termination proving and falsely believe we are always unable to prove termination, rather than the more benign consequence that we are unable to always prove termination.”

[Cook et al., 2011]

“The **size-change termination principle** for a first-order functional language with well-founded data is: a program terminates on all inputs if *every infinite call sequence* (following program control flow) would cause an infinite descent in some data values.”

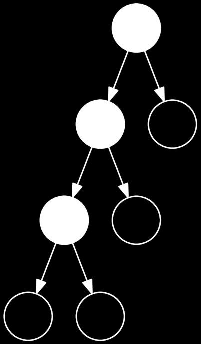
[Lee et al., 2001]

$\Delta$

An untyped, call-by-value, functional first-order language.

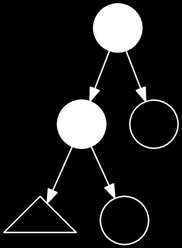
# Δ, values and shapes

$b \in \mathbb{B}$



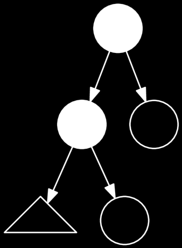
$\succ$

$s \in \mathbb{S}$



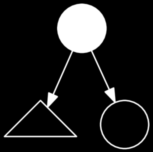
# $\Delta$, shapes and shapes

$s_1 \in \mathbb{S}$



$\succ$

$s_2 \in \mathbb{S}$



# Disjoint shapes

$$s_1 \cap s_2 = \emptyset \quad \text{iff} \quad B_1 \cap B_2 = \emptyset$$

where

$$s_1, s_2 \in \mathbb{S} \land B_1 = \{b \mid b \in \mathbb{B} \land b \succ s_1\} \land B_2 = \{b \mid b \in \mathbb{B} \land b \succ s_2\}$$

Given a shape $s_i \in \mathbb{S}$, we define the **sibling set** $S_i^d$, to be the pairwise disjoint set of shapes disjoint with $s_i$.

```haskell
data Pattern
  = PNil
  | PVariable String
  | PNode Pattern Pattern

getSiblings :: Pattern -> [Pattern]

getSiblings PNil =
  [PNode (PVariable "_") (PVariable "_")]

getSiblings (PVariable _) = []

getSiblings (PNode leftP rightP) =
  let
    leftS = getSiblings leftP
    rightS = getSiblings rightP
    leftInit = map (\s -> PNode leftP s) rightS
    rightInit = map (\s -> PNode s rightP) leftS
  in
    [PNil] ++
      leftInit ++ rightInit ++
      interleaveSiblings name leftS rightS
```

$$
\begin{aligned}
T(1) &= 4 \\
T(n) &= 1 + T(n-1) + T(n-1) + T(n-1) \cdot T(n-1)
\end{aligned}
$$

# Δ, syntax



```
        <program> ::= <clause>+ <expression>
     <expression> ::= <element> ( ‘.’ <expression> ) ?
        <element> ::= ‘0’ | ‘(’ <element> ‘)’ | <name> | <application>
    <application> ::= <name> <expression>*
         <clause> ::= <name> <pattern>* ‘:=’ <expression>‘;’
        <pattern> ::= <pattern-element> ( ‘.’ <pattern> ) ?
<pattern-element> ::= ‘0’ | ‘_’ | ‘(’ <pattern> ‘)’ | <name>
           <name> ::= [‘a’-‘z’] ( [‘-’ ‘a’-‘z’]* [‘a’-‘z’] ) ?
```

# Δ, sample programs

```
reverse 0 := 0
reverse left.right := (reverse right).(reverse left)

reverse input
```

```
fibonacci n = fibonacci-aux (normalize n) 0 0

fibonacci-aux 0 x y := 0
fibonacci-aux 0.0 x y := y
fibonacci-aux 0.n x y := fibonacci-aux n y (add x y)

fibonacci input
```

```
ackermann 0 n := 0.n
ackermann a.b 0 := ackermann (decrease a.b) 0.0
ackermann a.b c.d :=
  ackermann (decrease a.b) (ackermann a.b (decrease c.d))

ackermann input input
```

| **Description** | **Instance** | **Finite list** | **Space** |
| --- | --- | --- | --- |
| Expression | $x$ | $X$ | $\mathbb{X}$ |
| Element (of an expression) | $e$ | $E$ | $\mathbb{E}$ |
| Function | $f$ | $F$ | $\mathbb{F}$ |
| Clause | $c$ | $C$ | $\mathbb{C}$ |
| Pattern | $p$ | $P$ | $\mathbb{P}$ |
| Value (think "binary") | $b$ | $B$ | $\mathbb{B}$ |
| Name (think "variable") | $v$ | $V$ | $\mathbb{V}$ |
| Program ($p$ was taken) | $r$ | $R$ | $\mathbb{R}$ |
| Shape | $s$ | $S$ | $\mathbb{S}$ |

# Δ, syntax



```
        <program> ::= <clause>+ <expression>
     <expression> ::= <element> ( '.' <expression> ) ?                                  x
        <element> ::= '0' | '(' <element> ')' | <name> | <application>                 e
    <application> ::= <name> <expression>*                                       ⟨v, X⟩
         <clause> ::= <name> <pattern>* ':=' <expression>';'                 c = ⟨v, P, x⟩
        <pattern> ::= <pattern-element> ( '.' <pattern> ) ?                             p
<pattern-element> ::= '0' | '_' | '(' <pattern> ')' | <name>                            p
           <name> ::= ['a'-'z'] ( ['-' 'a'-'z']* ['a'-'z'] ) ?                          v
```

# Δ, syntax 

<program> ::= <clause>$^{+}$ <expression>

<expression> ::= <element> ( ‘.’ <expression> ) ? $\quad x$

<element> ::= ‘0’ | ‘(’ <element> ‘)’ | <name> | <application> $\quad e$

<application> ::= <name> <expression>$^{*}$ $\quad \langle v, x\rangle$

<clause> ::= <name> <pattern>$^{*}$ ‘:=’ <expression>‘;’ $\quad c = \langle v, p, x\rangle$

<pattern> ::= <pattern-element> ( ‘.’ <pattern> ) ? $\quad p$

<pattern-element> ::= ‘0’ | ‘_’ | ‘(’ <pattern> ‘)’ | <name> $\quad p$

<name> ::= [‘a’-‘z’] ( [‘-’ ‘a’-‘z’]$^{*}$ [‘a’-‘z’] ) ? $\quad v$

# Functions in $\Delta$



$$f = \langle v, C \rangle \quad \text{s.t.} \quad \forall \ \langle v', \_, \_ \rangle \in C \ \left( v' = v \right)$$

Pattern matching is ensured **exhaustive** at compile time, i.e.

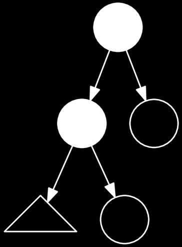
$$\forall\, b \in \mathbb{B}\ \exists\, c \in C\ c \succ b.$$

$$\text{WLOG, } c = \langle p, x \rangle.$$

Patterns and shapes



```
f (_.0).0 := ...
```



```
ENSURE-EXHAUSTIVE(c : C)
P_siblings = GET-SIBLINGS(c)
C' = [c]
for c' ∈ C
    (P_success, P_fail) = MATCH-CLAUSE-TO-SIBLINGS(c, P_s)
    for p ∈ P_success
        c'' = CLONE(c')
        MERGE-PATTERN(c'', p)*
        C' = c' : C'
    P_siblings = P_fail
return C'
```

**Invariants:**

- $P_{siblings}$ is always a list of siblings that wasn't matched by any forthcoming clause.
- $P_{success}$ and $P_{fail}$ are always sibling lists.

Demo

# Programs in $\Delta$

$$r = \langle F, x \rangle .$$

WLOG, $r = \langle C, x \rangle$.

# Observed mistakes

List indexing

- Lists are sometimes 0-indexed rather than 1-indexed.
- $\forall \{i \mid 0 \geq i < |P|\}$ should obviously be $\forall \{i \mid 0 < i \leq |P|\}$.

$\Cup$

# References


[Cook et al., 2011] B. Cook, A. Podelski & A. Rybalchenko, *Proving program termination*, Communications ACM Vol. 54(5), 2011, 88–98.

[Lee et al., 2001] Chin Soon Lee, Neil D. Jones & Amir M. Ben-Amram, *The size-change principle for program termination*, POPL '01, 81–92.