Termination Analysis of a subset of CoreML

William Blum

william.blum@comlab.ox.ac.uk

Oxford University Computing Laboratory

BCTCS Nottingham

Outline

First order programs

Untyped functional language recursion, if-then-else, primitive operators, single data type

Call-by-value evaluation semantics:

$\mathcal{E}[\![\mathtt{f}]\!]\ \vec{x} = v$ $\mathtt{f}$ evaluates to $\mathtt{v}$ on input $\vec{x}$,

$\mathcal{E}[\![\mathtt{f}]\!]\ \vec{x} = \perp$ $\mathtt{f}$ does not terminate on input $\vec{x}$.

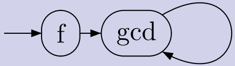
Exact call semantics: a computation is described by a state transition sequence.

Finite approximation of the call semantics: the control flow graph.

Example

```
f(x) = gcd(x,18)
gcd(x,y) =
 if y == 0 then x
 else gcd(y, x mod y)
```

$f, 4 \to gcd, (4, 18) \to$
$gcd, (18, 2) \to$
$gcd, (2, 2) \to gcd, (2, 0)$



Termination

Characterization of termination

P terminates on all input values
$\Longleftrightarrow$ Infinite state transition sequences are invalid computations.

- *What is an invalid computation?*
  For instance: a computation in which some positive integer variable decreases infinitely...
- *The Size-Change Principle* proves that for any computation corresponding to an infinite path in the control flow, the value of some well-founded variable decreases infinitely.

Size-change graphs (SCG)

Definition: A SCG describes a program call. It consists of a source set of vertices, a target set of vertices and a set of labeled arcs.

The SCG $\left(\begin{array}{ccc} x & \xrightarrow{=} & x \\ & & y \end{array}\right)$ describes the call from `f` to `gcd`.

Safety: $\xrightarrow{\downarrow}$ arcs denote decreases in parameter value, $\xrightarrow{=}$ arcs denote non increase in parameter value.

Example: consider the call "`gcd(`$\underbrace{\texttt{y}}_{\texttt{x}}$, $\underbrace{\texttt{x mod y}}_{\texttt{y}}$`)`":

$$\left(\begin{tikzcd} x \arrow[dr, "="] & x \\ y \arrow[ur, "="] \arrow[r, "\downarrow"] & y \end{tikzcd}\right) \quad \left(\begin{tikzcd} x \arrow[dr, "="] & x \\ y \arrow[r, "="] & y \end{tikzcd}\right) \quad \left(\begin{tikzcd} x \arrow[r, "="] & x \\ y \arrow[r, "\downarrow"] & y \end{tikzcd}\right) \quad \left(\begin{tikzcd} x & x \\ y & y \end{tikzcd}\right)$$

Only one of these SCG is not safe for this call.

Composition of size-change graphs

$$\text{If } f \xrightarrow{G_1} g \text{ and } g \xrightarrow{G_2} h \text{ then } f \xrightarrow{G_1;G_2} h$$

$$\overbrace{\begin{tikzcd} a \arrow[r, "="] \arrow[dr, "="] & x \arrow[r, "\downarrow"] \arrow[dr, "="] & u \\ & y \arrow[r, "\downarrow"] & v \end{tikzcd}}^{G_1 \qquad G_2} \equiv \overbrace{\begin{tikzcd} a \arrow[r, "\downarrow"] \arrow[dr, "="] & u \\ & v \end{tikzcd}}^{G_1;G_2}$$

If $\mathcal{G}$ is a set of size-change graphs then $\overline{\mathcal{G}}$ denotes the composition closure of $\mathcal{G}$.

Size-change termination (SCT)

Definition Consider $\mathcal{G}$ a set of size-change graphs.
A program $\mathtt{P}$ is $\mathcal{G}$-SCT if

- $\mathcal{G}$ safely describes $\mathtt{P}$ (for every reachable call c there is a corresponding SCG $G_c \in \mathcal{G}$)
- for all infinite computation $cs = \langle c_0 c_1 \ldots \rangle$, any sequence of size-change graphs $G_{c_0} G_{c_1} \ldots$ (describing safely the calls of cs) has an infinite descending thread.

$$\begin{tikzcd}
x & u \arrow[dr, "="] & u & x \arrow[r, "="] & \cdots \\
y & v & v \arrow[ur, "\downarrow"] & y & \cdots
\end{tikzcd}$$

We assume that data-types are well-founded.

Theorem

If $\mathtt{P}$ is $\mathcal{G}$-SCT then $\mathtt{P}$ terminates for all input values

Deciding Size-Change Termination

$\mathcal{G}$-SCT characterization [Jones et al. 2001]

$\texttt{P}$ is **not** $\mathcal{G}$-SCT

$$\Longleftrightarrow$$

$$\exists \texttt{f} \xrightarrow{G} \texttt{f} \in \overline{\mathcal{G}} \text{ such that } \begin{pmatrix} G; G = G \\ \forall \texttt{x} \in gb(\texttt{f}) : \texttt{x} \xrightarrow{\downarrow} \texttt{x} \notin G \end{pmatrix}$$

Hence $\mathcal{G}$-SCT is decidable. And it is PSPACE-complete (see [1])

The language $\mathcal{L}_{ml}$

Grammar:

`e ::= x, f`	value identifiers
`  | true | false`	boolean constants
`  | if e then e else e`	conditional
`  | n`	integer constants (n $\in \mathbb{N}$)
`  | e = e`	integer equality
`  | succ e | pred e`	successor and predecessor
`  | fun (x:ty) -> e`	function abstraction
`  | fun f=(x:ty) -> e`	recursively defined function
`  | e e`	function application
`  | let x = e in e`	local variable definition

A program is a single closed expression.

Data types: ground values + higher-order functions.

Semantics of $\mathcal{L}_{ml}$ (environment based)

Canonical expressions: $\mathbb{N} \cup \mathbb{B} \cup \{\mathtt{e} \mid \mathtt{e} \text{ is an abstraction}\}$

$$
\begin{aligned}
\mathit{State} &= \{\mathtt{e} : \rho \mid \mathtt{e} \in \mathit{subexp}(\mathtt{P}), \rho \in \mathit{Env}, \mathit{fv}(\mathtt{e}) \subseteq \mathit{dom}(\rho)\} \\
\mathit{Value} &= \{\mathtt{e} : \rho \in \mathit{State} \mid \mathtt{e} \text{ canonical}\} \\
\mathit{Env} &= \{\rho : X \to \mathit{Value} \mid X \text{ finite set of variables}\}
\end{aligned}
$$

Let $s \in \mathit{State}$, $v \in \mathit{Value}$ and $\rho \in \mathit{Env}$

Call-by-value evaluation semantics "$s \Downarrow v$"

$$\frac{}{\mathtt{v} : \rho \Downarrow \mathtt{v} : \rho}(\mathtt{v} \text{ canonical})$$

Run-time errors "$s \oslash$"

$$(\text{ErrOp1})\frac{\mathtt{e} : \rho \Downarrow 0}{\mathtt{pred\ e} : \rho \oslash}$$

Call semantics "$s \rightarrow s'$"

$$(\text{CallG})\frac{\mathtt{e}_1 : \rho \Downarrow \mathtt{fun\ \ (x:ty)->e}_0 : \rho_0 \qquad \mathtt{e}_2 : \rho \Downarrow \mathtt{v}_2}{\mathtt{e}_1\mathtt{e}_2 : \rho \xrightarrow[c]{} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{v}_2]}$$

Graph generation

- Two SCG generated per call: G^{+} describing higher-order values and G^{0} for ground type values.
- The *free variables* of an expression correspond to the *input parameters* in the first-order case.
- We define well-founded notions of size for higher-order and ground type expressions.
- We extend the semantic rules to generate safe SCG:

$$\text{(ValueG)}\frac{}{\mathtt{v} \Downarrow \mathtt{v},\, id^{=}_{\mathtt{e}} | id^{=}_{\mathtt{e}}}(\mathtt{v} = \mathtt{e} : \rho \text{ in canonical form})$$

$$\text{(CallG)}\ \frac{\mathtt{e}_1 : \rho \Downarrow \mathtt{fun\ \ (x:ty)->e}_0 : \rho_0, G_1 | G_1^{+} \qquad \mathtt{e}_2 : \rho \Downarrow \mathtt{v}_2, G_2 | G_2^{+}}{\mathtt{e}_1\mathtt{e}_2 : \rho \xrightarrow[c]{} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{v}_2], CallGr^{0}_{x}(G_1, G_2) | CallGr^{+}_{x}(G_1^{+}, G_2^{+})}$$

Finite approximation of the call semantics

We need a "control flow graph" for ML programs
Solution:

- drop the ρ components of the states
- abstract integers by a single symbol "$?^{\mathtt{int}}$".

We obtain a finite abstraction of the computation.
The set of vertices of the control flow graph (i.e control points) is:

$$\mathcal{P} = \textit{subexp}(\mathtt{P}) \cup \{?^{\mathtt{int}}\}$$

The size-change principle

What do we have:

- Termination characterized by infinite call sequences
- Well-founded order on the data values
- Finite approximation of call semantics
- We can compute two safe sets of size-change graphs describing the calls (by applying the semantic rules exhaustively).

Hence the SCP can be applied! (twice)

Results

- Counter example

```
let rec counter x =
 if x = 0 then counter (succ x) else 1
in counter 7;;
```

  is terminating but not SCT.
- Ackerman's function: SCT relatively to ground-type values.
- Function computing the minimum of two numbers:
  - is SCT if we use the native representation of integers provided by $\mathcal{L}_{ml}$,
  - is not SCT if we use Church numeral to encode integers.

Conclusion

- The Size-Change Principle from Neil D. Jones et al.
  - based on a finite approximation of the call semantics,
  - and a safe description of the calls.
- Extension to a higher-order functional language
  - detects decrease on ground-type values as well as higher-order values
  - allows local definition `let`
  - handles recursion natively (no need to define a Y combinator)
  - handles numbers natively
- Further direction
  sequential composition, storage location and references, tuples list, user defined structures, for and while loop structures.

Bibliography

- Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram.
  The Size-Change Principle for Program Termination.
  Principles of Programming Languages, pp. 81-92. Volume 28 of Principles of Programming Languages. ACM press 2001
- Neil D. Jones and Nina Bohr
  Termination Analysis of the Untyped λ-Calculus.
  Rewriting Techniques and Applications.,Proceedings (V van Oostrom, ed.), pp. 1-23. Volume 3091 of LNCS. Springer-Verlag 2004.
- Andrew M. Pitts
  Operational Semantics and Program Equivalence.
  Applied Semantics., pp. 378-412. Volume 2395 of LNCS. Springer-Verlag 2002