Fun with strictness and groundness analysis

Lee Naish
Computer Science and
Software Engineering
University of Melbourne

Outline

Static analysis of strictness and groundness

Worst case groundness analysis

Another class of Boolean functions

Worst case strictness analysis

Further work

Strictness and groundness analysis

In lazy functional programming languages, some arguments of some functions are "strict" (always needed to evaluate the function)

Knowing which arguments are definitely strict allows more efficient object code to be produced, so the better compilers perform strictness analysis

In logic programming languages, knowing which arguments are definitely ground (completely instantiated) when a predicate succeeds is also of interest

Abstract interpretation is used to perform both analyses

The abstract domains are often classes of Boolean functions

Abstracting strictness

Consider a function which finds the minimum of two natural numbers (note arg. order in recursive call):

```
f x y = if x==0 then x
        else if y==0 then y
        else 1 + (f (y-1) (x-1))
```

Each function can be abstracted to a Boolean function:

$$
\begin{aligned}
f\ x\ y = x \wedge true \wedge (x \vee \\
& (y \wedge true \wedge (y \vee \\
& (true \wedge (f\ (y \wedge true)\ (x \wedge true))))))
\end{aligned}
$$

We can execute this if $\wedge$ and $\vee$ are non-strict, and consider all cases where the result is $true$ ($f\ true\ true$ and $f\ true\ false$)

Abstracting strictness (cont.)

Recursion is a problem in general (as is executing a function 2^N times)

We want the least fixedpoint or strongest function which satisfies the equation (a weaker function is also a correct approximation)

Kleene iteration can be used to find a fixedpoint

Eg, $f\ x\ y = false$; $f\ x\ y = x$; $f\ x\ y = x$

So `f` is strict in just the first argument

The functions used are *monotonic* — CNF/DNF have no negative literals

Abstracting groundness

Consider `append/3`, normalised:

```
append(X1, X2, X3) :-
    X1 = [], X2 = X3.
append(X1, X2, X3) :-
    X1 = [H|T], X3 = [H|U], append(T, X2, U).
```

The calls/constraints can be abstracted to Boolean constraints:

$$
\begin{aligned}
\mathit{append}(X_1, X_2, X_3) \leftarrow & \exists H \exists T \exists U \\
& (X_1 \leftrightarrow \mathit{true} \wedge X_2 \leftrightarrow X_3 \\
& \vee X_1 \leftrightarrow (H \wedge T) \wedge X_3 \leftrightarrow (H \wedge U) \wedge \mathit{append}(T, X_2, U))
\end{aligned}
$$

Find a (preferably strongest) boolean function which satisfies this, eg $\mathit{append}(X_1, X_2, X_3) = (X_1 \wedge X_2) \leftrightarrow X_3$

Abstracting groundness (cont.)

Sometimes *positive* functions are used (a conjunction of *clauses*, $V_i \wedge \ldots \wedge V_j \rightarrow V_k \vee \ldots \vee V_l$, where RHS is non-empty)

Sometimes just *definite* functions are used (a conjunction of definite clauses, $V_i \wedge \ldots \wedge V_j \rightarrow V_k$)

Pos allows disjunctive information to be expressed whereas Def does not

In practice, most of the code we write does not require disjunctive information

Worst case groundness analysis

Groundness inference with both Pos and Def can take $2^N - 1$ steps (the height of the lattice with N variables)

For Pos the program size can be $O(N)$

For Def the program size can be $O(N^2)$

For Def its unknown if it can be $O(N)$

Exponential iterations — $N = 4$ case

```
p(X1, X2, X3, cn) :- p(X1, X2, X3, X4).
p(X2, X3, cn, X4) :- p(X2, X3, X4, cn).
p(X3, cn, X4, X4) :- p(X3, X4, cn, cn).
p(cn, X4, X4, X4) :- p(X4, cn, cn, cn).
p(X4, X4, X4, X4).
```

Start by assuming the only possibility it that all arguments are ground — $X_1 \wedge X_2 \wedge X_3 \wedge X_4$ or $\{1111\}$

Successive iterations add 0000, 0001, 0010, 0011, ... to the set of models of the approximation function

The search space of the program includes many redundant answers, not to mention infinite branches

Another class of Boolean functions

Suppose we have a Mercury or Ground/Directionally Typed Prolog predicate with arguments X and Y

The following groundness/type descriptions are of interest: $X \wedge Y$, $X \rightarrow Y$, $Y \rightarrow X$, $X \leftrightarrow Y$

$X \vee Y$ is not of interest; it is in Pos but not in Def

X and Y are in Def but they are not of interest either

$IMon$ is the set of functions which are conjunctions of implications $V_1 \wedge \ldots \wedge V_k \rightarrow V_{k+1} \wedge \ldots \wedge V_N$ which have (a permutation of) all N variables

Equivalently, functions of the form $M \rightarrow V_1 \wedge \ldots \wedge V_N$ where M is in Mon

$IMon$ is isomorphic to $Mon \setminus \{false\}$

Size of various classes

For $N = 0, Size = 1$ and $N = 1, Size = 2$

N	2	3	4	5	6
Pos	8	128	32,768	2,147,483,648	922,337,203,685,477,580
Def	7	61	2,480	1,385,552	75,973,751,474
$IMon$	5	19	167	7,580	7,828,353
$EPos$	5	15	52	203	877
$ICon$	4	8	16	32	64

Maximum ascending chain length is 2^N for Pos, Def and $IMon$ and $N + 1$ for $EPos$ and $ICon$

All the approximations in the worst case code for groundness are in $IMon$

Worst case strictness analysis

First, tweek the logic program so it can be made deterministic and its clearer what its doing

We add an argument which is a bit string (head = LSB; arguments have been reversed and renamed)

```
p([1|S],       1,B,C,D) :- p([0|S],       Z,B,C,D).
p([0,1|S],     Z,1,C,D) :- p([1,0|S],     1,Z,C,D).
p([0,0,1|S],   Z,Z,1,D) :- p([1,1,0|S],   1,1,Z,D).
p([0,0,0,1|S], Z,Z,Z,1) :- p([1,1,1,0|S], 1,1,1,Z).
p([0,0,0,0],   Z,Z,Z,Z).
```

`p(S,A,B,C,D)` returns all 2^N answers and terminates

`p([1,1,1,1],1,1,1,1)` has a derivation of length 2^N with all groundness patterns

Worst case strictness analysis (cont.)

Ascending the lattice of $IMon$ functions is like *descending* the lattice of Mon functions

The analysis should count *down*, adding 1111, 0111, 1011, 0011, ... (recall bit order reversal); base case should have all arguments strict

So:

- Swap ones and zeros in bit strings,
- Replace `Z` by 1 (or another constant), and simultaneously
- Replace 1 on LHS by `_` and 1 on RHS by $\perp$ (we lose some symmetry of the LP version)

Worst case strictness analysis (cont.)

```
f (0:s)         _ b c d = f (1:s)         1 b c d
f (1:0:s)       1 _ c d = f (0:1:s)       z 1 c d
f (1:1:0:s)     1 1 _ d = f (0:0:1:s)     z z 1 d
f (1:1:1:0:s) 1 1 1 _ = f (0:0:0:1:s) z z z 1
f ([1,1,1,1]) 1 1 1 1 = 1
z = z -- undefined/bottom
```

Very similar to LP version with LHS and RHS swapped

Compare evaluating `f [0,0,0,0] z z z z` with `p([1,1,1,1],A,B,C,D)`

Variations of code possible, eg replace ones by variables on LHS and their sum on RHS; `f [1,0,1,0] (2^0) (2^1) (2^2) (2^3) = 5`

Reverse counting in Prolog

Swap LHS and RHS, base case and top level goal/answer

```
p([0|S],       Z,B,C,D) :- p([1|S],       1,B,C,D).
p([1,0|S],     1,Z,C,D) :- p([0,1|S],     Z,1,C,D).
p([1,1,0|S],   1,1,Z,D) :- p([0,0,1|S],   Z,Z,1,D).
p([1,1,1,0|S], 1,1,1,Z) :- p([0,0,0,1|S], Z,Z,Z,1).
p([1,1,1,1],   1,1,1,1).
```

Analysis with Def and Pos take $O(2^N)$ steps

Analysis with $IMon$ doesn't, due to closure properties of the domain

Def and Pos have $(2^N)!$ chains of length 2^N; $IMon$ and Mon have just $N!$

Further work

Are there programs of size N which take $O(2^N)$ iterations for groundness inference using Def?

What about groundness using $IMon$/strictness using Mon?

Is there a more precise characterisation of the relationship between groundness and strictness analysis?

Is other FP work useful for groundness/mode analysis of higher order logic programs?