

Strictness Analysis

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

choose **always** needs its first argument

Consider the following function:

```
choose b x y  
= if b then x then y
```

When we evaluate the following application:

```
choose (not foo) (fun 3 5) (bar True)
```

- The value of the first argument will *always* be needed to produce the result.
- The values of the last two arguments may or may not be needed.

Strictness

The denotational equivalent of this property *strictness*.

A function f defined by the equation

$$f\ x\ y\ z = e$$

is *strict in x* when

$$f\ \perp\ y\ z = \perp$$

for all values of y and z .

(Similarly define strict in y and z .)

Strictness of *choose*

choose b x y
= **if** b **then** x **then** y

choose \perp x y = \perp **strict** in its first argument

choose *True* \perp y = \perp

choose *False* \perp y = y **non-strict** in its second argument

choose *True* x \perp = x

choose *False* x \perp = \perp **non-strict** in its third argument

Consider evaluating the following using **call-by-name**:

choose (not foo) (fun 3 5) (bar True)

- It is generally cheaper to evaluate an expression and store its result, than to store a representation (e.g. a closure) of the expression for possible later evaluation.
- As *choose* is strict in its first argument, we can evaluate *(not e1)* to a boolean value before performing the substitution.
- As we know this result will be needed by *choose* anyway, it doesn't matter if the evaluation diverges before we perform the substitution.
- Strictness analysis tells us when its safe to evaluate then arguments of a function before it is called.

The domain of values

We will call the result of a computation a *value*.

$$Val = \{ \perp \} + \mathbb{B} + \mathbb{Z} + (\mathbb{Z} \rightarrow \mathbb{Z}) + \dots$$

Strictness analysis is usually done after type checking, so we can put integers, booleans, functions etc in the same domain.

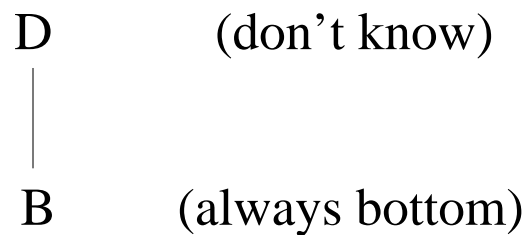
Our domain of values is *lifted*, meaning that it contains a \perp and:

$$\begin{aligned} \forall v \in Val. \perp \sqsubseteq v \\ \perp \sqsubseteq \perp \end{aligned}$$

Abstract domain for strictness analysis

Define an abstract domain to represent whether a value is bottom, or maybe not bottom.

$$AbsVal = \{B, D\}$$



Note that $B \sqsubseteq D$.

We will treat *AbsVal* as a simple lattice with **meet** (\wedge) and **join** (\vee) operators.

Mapping *Value* onto *AbsVal*

$$Val = \{ \perp \} + \mathbb{B} + \mathbb{Z} + (\mathbb{Z} \rightarrow \mathbb{Z}) + \dots$$

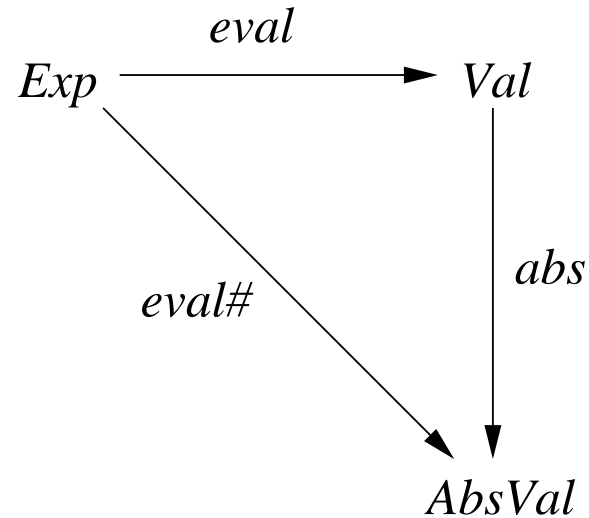
$$AbsVal = \{B, D\}$$

The function *abs* abstracts away from a value, only remembering whether it was a \perp or not. Everything that isn't a \perp , or *might be* a \perp is “dont know” (D).

$$abs : Val \rightarrow AbsVal$$

$$abs\ v = \begin{cases} B & \text{if } v = \perp \\ D & \text{otherwise} \end{cases}$$

Abstract interpretation of expressions

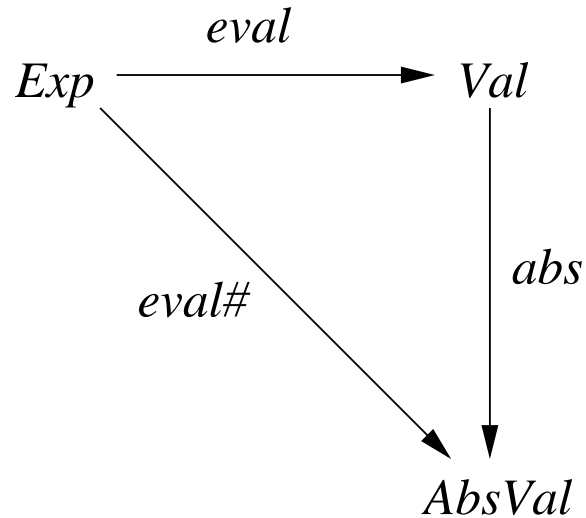


- $eval$ is a standard interpreter for evaluating expressions;
- $eval^\#$ is an abstract interpreter.

If $eval^\#(e) = B$ then $eval(e)$ does not terminate.

If $eval^\#(e) = D$ then $eval(e)$ *may* terminate.

Abstract interpretation of expressions



- To determine whether an expression e evaluates to \perp , we could compute $abs(eval\ e)$... but that could take a long time.
- Instead we'll use $eval^\#$ to make a safe approximation statically (at compile time), without actually evaluating it.

Safe Approximation!

If $eval^\sharp(e) = B$ then $eval(e)$ does not terminate.

If $eval^\sharp(e) = D$ then $eval(e)$ *may* terminate.

- If $eval^\sharp$ says that the evaluation doesn't terminate, then it really doesn't terminate.
- The analysis will be incomplete, so $eval^\sharp$ may not be able to determine that a terminating expression actually does.
- It is safe (but not useful) to return D for *all expressions*.

Defining $eval^\sharp$

For literal constants c we have:

$$eval^\sharp(c) = abs(eval(c)) = D$$

For conditional expressions:

As we won't know what the value of the boolean will be, we can only approximate the strictness information.

To be certain that **if** b **then** e_1 **else** e_2 does not terminate, we need to know that either $b = \perp$ or both $e_1 = \perp$ and $e_2 = \perp$.

$$eval^\sharp(\mathbf{if} \mathit{b} \mathbf{then} \mathit{e}_1 \mathbf{else} \mathit{e}_2) = eval^\sharp(\mathit{b}) \wedge (eval^\sharp(\mathit{e}_1) \vee eval^\sharp(\mathit{e}_2))$$

Analysing functions

Suppose we define a function in **Fun**:

$$f(x, y, z) = e$$

The interpreter will calculate the corresponding (actual) function:

$$f(x, y, z) = eval(e)$$

The aim of our analysis is to construct a corresponding function:

$$f^\#(a, b, c) = eval^\#(e)$$

Their respective types are:

$$(Val, Val, Val) \rightarrow Val$$

and

$$(AbsVal, AbsVal, AbsVal) \rightarrow AbsVal$$

Functions — (ctd)

The pre-defined functions have known strictness properties.

The binary arithmetic, boolean, and relational operators are all strict in both arguments. For example:

$$a +^{\#} b = a \wedge b$$

In other words:

$$e_1 + e_2 = \perp \text{ if either } e_1 = \perp \text{ or } e_2 = \perp$$

For example:

$$B +^{\#} D = B \wedge D = B$$

The list constructor function $(:)$ evaluates neither of its arguments so it is non-strict:

$$(a :^{\#} as) = D$$

So $(a :^{\#} as)$ always terminates, irrespective of whether a or as do.

Discovering the strictness properties of functions

For a function f defined in **Fun** as:

$$f(x, y, z) = e$$

we have discussed the process of constructing an abstract version:

$$f^\sharp(a, b, c) = eval^\sharp(e)$$

Recall that f is strict in x iff $f(\perp, y, z) = \perp$ for all y, z . Similarly for y and z .

In terms of the abstract function f^\sharp we can deduce that:

f is strict in x if $f^\sharp(B, D, D) = B$

f is strict in y if $f^\sharp(D, B, D) = B$

f is strict in z if $f^\sharp(D, D, B) = B$

Example:

Consider the definition:

$$f(p, q, r) = \text{if } p == 0 \text{ then } q \text{ else } q + r$$

Construct $f^\#$:

$$\begin{aligned} f^\#(p, q, r) &= \text{eval}^\#(\text{if } p == 0 \text{ then } q \text{ else } q + r) \\ &= \text{eval}^\#(p == 0) \wedge (\text{eval}^\#(q) \vee \text{eval}^\#(q + r)) \\ &= (p ==^\# D) \wedge (q \vee (q +^\# r)) \\ &= (p \wedge D) \wedge (q \vee (q \wedge r)) \end{aligned}$$

Example:

...

$$f^\sharp(p, q, r) = (p \wedge D) \wedge (q \vee (q \wedge r))$$

Now to infer the strictness properties of f :

$$f^\sharp(B, D, D) = (B \wedge D) \wedge (D \vee (D \wedge D)) = B$$

$$f^\sharp(D, B, D) = (D \wedge D) \wedge (B \vee (B \wedge D)) = B$$

$$f^\sharp(D, D, B) = (D \wedge D) \wedge (D \vee (D \wedge B)) = D$$

So f is strict in p and q , and f *may not* be strict in r .

Recursive definitions

Unfortunately, the algorithm above does not work for recursive definitions.

In general, recursive definitions rely on a *conditional* to terminate, for example:

$$fact(n) = \mathbf{if } n == 0 \mathbf{ then } 1 \mathbf{ else } n \times fact(n - 1)$$

That effect is lost in the abstraction of the conditional to:

$$if^\sharp(a, b, c) = a \wedge (b \vee c)$$

because both arms of the conditional are evaluated in every case.

Time for a fixpoint

Our only recourse is to calculate the *least fixpoint* solution of the equation, using Kleene's technique.

For a definition of the form:

$$f^\sharp = F(f^\sharp)$$

calculate:

$$f^\sharp = \lim_{n \rightarrow \infty} F^n(\perp)$$

Recursion — (ctd)

Assuming a definition like:

$$f^\sharp(x, y, z) = F(f^\sharp)(x, y, z)$$

The successive approximations to f^\sharp are calculated as follows:

$$f_0^\sharp(x, y, z) = B$$

$$f_1^\sharp(x, y, z) = F(f_0^\sharp)(x, y, z)$$

$$f_2^\sharp(x, y, z) = F(f_1^\sharp)(x, y, z)$$

...

$$f_k^\sharp(x, y, z) = F(f_{k-1}^\sharp)(x, y, z)$$

The abstract domain is finite, so there can be only a finite number of distinct f_k^\sharp .

Example:

Consider the definition:

$$g(x, y, z) = \mathbf{if } x == 0 \mathbf{ then } z \mathbf{ else } g(x - 1, z, y + z)$$

Construct g^\sharp (simplified by identity $a \wedge D = a$) (**exercise**):

$$g^\sharp(x, y, z) = x \wedge (z \vee g^\sharp(x, z, y \wedge z))$$

Now the fixpoint calculation, simplifying as I go. (Hold on tight ...)

$$g_0^\sharp(x, y, z) = B$$

$$g_1^\sharp(x, y, z) = x \wedge (z \vee g_0^\sharp(x, z, y \wedge z)) = x \wedge (z \vee B) = x \wedge z$$

$$g_2^\sharp(x, y, z) = x \wedge (z \vee g_1^\sharp(x, z, y \wedge z)) = x \wedge (z \vee (x \wedge (y \wedge z)))$$

$$\begin{aligned} g_3^\sharp(x, y, z) &= x \wedge (z \vee g_2^\sharp(x, z, y \wedge z)) \\ &= x \wedge (z \vee (x \wedge ((y \wedge z) \vee (x \wedge (z \wedge (y \wedge z)))))) \\ &= x \wedge (z \vee (x \wedge (y \wedge z))) \quad \text{— fixpoint!} \end{aligned}$$

Example: (continued)

Is g strict in z ?

$$\begin{aligned}g_3^\sharp(D, D, B) &= D \wedge (B \vee (D \wedge (B \wedge D))) \\ &= B\end{aligned}$$

So g is strict in z .

(I checked it with the analyser program.)

Exercise:

g is also strict in x but not y . Check these claims.

Exercise:

Download and experiment with the interpreter and strictness analyser. Develop your own function definitions, work out their strictness properties and check with the analyser.