Lecture 10

Strictness analysis

Motivation

The operations and control structures of *imperative* languages are strongly influenced by the way most real computer hardware works.

This makes imperative languages relatively easy to compile, but (arguably) less expressive; many people use *functional* languages, but these are harder to compile into efficient imperative machine code.

Strictness optimisation can help to improve the efficiency of compiled functional code.

Call-by-value evaluation

Strict ("eager") functional languages (e.g. ML)
use a *call-by-value* evaluation strategy:

$$\frac{e_2 \Downarrow v_2 \quad e_1[v_2/x] \Downarrow v_1}{(\lambda x.e_1)\ e_2 \Downarrow v_1}$$

- Efficient in space and time, but
- might evaluate more arguments than necessary.

Call-by-name evaluation

Non-strict ("lazy") functional languages (e.g. Haskell) use a *call-by-name* evaluation strategy:

$$\frac{e_1[e_2/x] \Downarrow v}{(\lambda x.e_1)\ e_2 \Downarrow v}$$

- Only evaluates arguments when necessary, but
- copies (and redundantly re-evaluates) arguments.

Call-by-need evaluation

One simple optimisation is to use *call-by-need* evaluation instead of call-by-name.

If the language has no side-effects, duplicated instances of an argument can be shared, evaluated once if required, and the resulting value reused.

This avoids recomputation and is better than call-by-name, but is still more expensive than call-by-value.

Call-by-need evaluation

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

Using call-by-value:

```
plus(3,4) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(2,5)
          ↦ plus(1,6)
          ↦ plus(0,7)
          ↦ 7
```

Call-by-need evaluation

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

Using call-by-need:

```
plus(3,4) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(3-1,4+1)
          ↦ plus(2-1,4+1+1)
          ↦ plus(1-1,4+1+1+1)
          ↦ 4+1+1+1
          ↦ 5+1+1
          ↦ 6+1
          ↦ 7
```

Replacing CBN with CBV

So why not just replace call-by-name with call-by-value?

Because, while replacing call-by-name with call-by-need never changes the semantics of the original program (in the absence of side-effects), replacing CBN with CBV does.

In particular, the program's *termination* behaviour changes.

Replacing CBN with CBV

Assume we have some nonterminating expression, Ω.

- Using CBN, the expression $(\lambda x.\, 42)\ \Omega$ will evaluate to 42.
- But using CBV, evaluation of $(\lambda x.\, 42)\ \Omega$ will not terminate: Ω gets evaluated first, even though its value is not needed here.

We should therefore try to use call-by-value wherever possible, but not when it will affect the termination behaviour of a program.

Neededness

Intuitively, it will be safe to use CBV in place of CBN whenever an argument is definitely going to be evaluated.

We say that an argument is *needed* by a function if the function will always evaluate it.

- $\lambda x,y.\ x+y$ needs both its arguments.
- $\lambda x,y.\ x+1$ needs only its first argument.
- $\lambda x,y.\ 42$ needs neither of its arguments.

Neededness

These needed arguments can safely be passed by value: if their evaluation causes nontermination, this will just happen sooner rather than later.

Neededness

In fact, neededness is too conservative:

$\lambda x,y,z.$ if x then y else Ω

This function might not evaluate *y*, so only *x* is *needed*.

But actually it's still safe to pass *y* by value: if *y* doesn't get evaluated then the function doesn't terminate anyway, so it doesn't matter if eagerly evaluating *y* causes nontermination.

Strictness

What we really want is a more refined notion:

It is safe to pass an argument by value when *the function fails to terminate whenever the argument fails to terminate.*

When this more general statement holds, we say the function is *strict* in that argument.

$\lambda x,y,z.$ if x then y else Ω

is strict in x and strict in y.

Strictness

If we can develop an analysis that discovers which functions are strict in which arguments, we can use that information to selectively replace CBN with CBV and obtain a more efficient program.

Strictness analysis

We can perform strictness analysis by abstract interpretation.

First, we must define a concrete world of programs and values.

We will use the simple language of *recursion e**quations*, and only consider integer values.

Recursion equations

$$
\begin{aligned}
F_1(x_1, \ldots, x_{k_1}) &= e_1 \\
\ldots &= \ldots \\
F_n(x_1, \ldots, x_{k_n}) &= e_n
\end{aligned}
$$

$$e ::= x_i \mid A_i(e_1, \ldots, e_{r_i}) \mid F_i(e_1, \ldots e_{k_i})$$

where each A_i is a symbol representing a built-in (predefined) function of arity r_i.

Recursion equations

For our earlier example,

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

we can write the recursion equation

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

where *cond*, *eq*, 0, *sub1* and *add1* are built-in functions.

Standard interpretation

We must have some representation of nontermination in our concrete domain.

As values we will consider the integer results of terminating computations, $\mathbb{Z}$, and a single extra value to represent nonterminating computations: $\perp$.

Our concrete domain *D* is therefore $\mathbb{Z}_\perp = \mathbb{Z} \cup \{ \perp \}$.

Standard interpretation

Each built-in function needs a standard interpretation.

We will interpret each A_i as a function a_i on values in D:

$$
\begin{aligned}
cond(\bot, x, y) &= \bot \\
cond(0, x, y) &= y \\
cond(p, x, y) &= x \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
eq(\bot, y) &= \bot \\
eq(x, \bot) &= \bot \\
eq(x, y) &= x =_{\mathbb{Z}} y \quad \text{otherwise}
\end{aligned}
$$

Standard interpretation

The standard interpretation f_i of a user-defined function F_i is constructed from the built-in functions by composition and recursion according to its defining equation.

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

Abstract interpretation

Our abstraction must capture the properties we're interested in, while discarding enough detail to make analysis computationally feasible.

Strictness is all about termination behaviour, and in fact this is all we care about: does evaluation of an expression *definitely not* terminate (as with Ω), or *may* it eventually terminate and return a result?

Our abstract domain $D^{\#}$ is therefore { 0, 1 }.

Abstract interpretation

For each built-in function A_i we need a corresponding strictness function $a_i^{\#}$ — this provides the *strictness interpretation* for A_i.

Whereas the standard interpretation of each built-in is a function on concrete values from D, the strictness interpretation of each will be a function on abstract values from $D^{\#}$ (i.e. 0 and 1).

Abstract interpretation

A formal relationship exists between the standard and abstract interpretations of each built-in function; the mathematical details are in the lecture notes.

Informally, we use the same technique as for multiplication and addition of integers in the last lecture: we define the abstract operations using what we know about the behaviour of the concrete operations.

Abstract interpretation

x	y	$eq^{\#}(x,y)$
0	0	0
0	I	0
I	0	0
I	I	I

Abstract interpretation

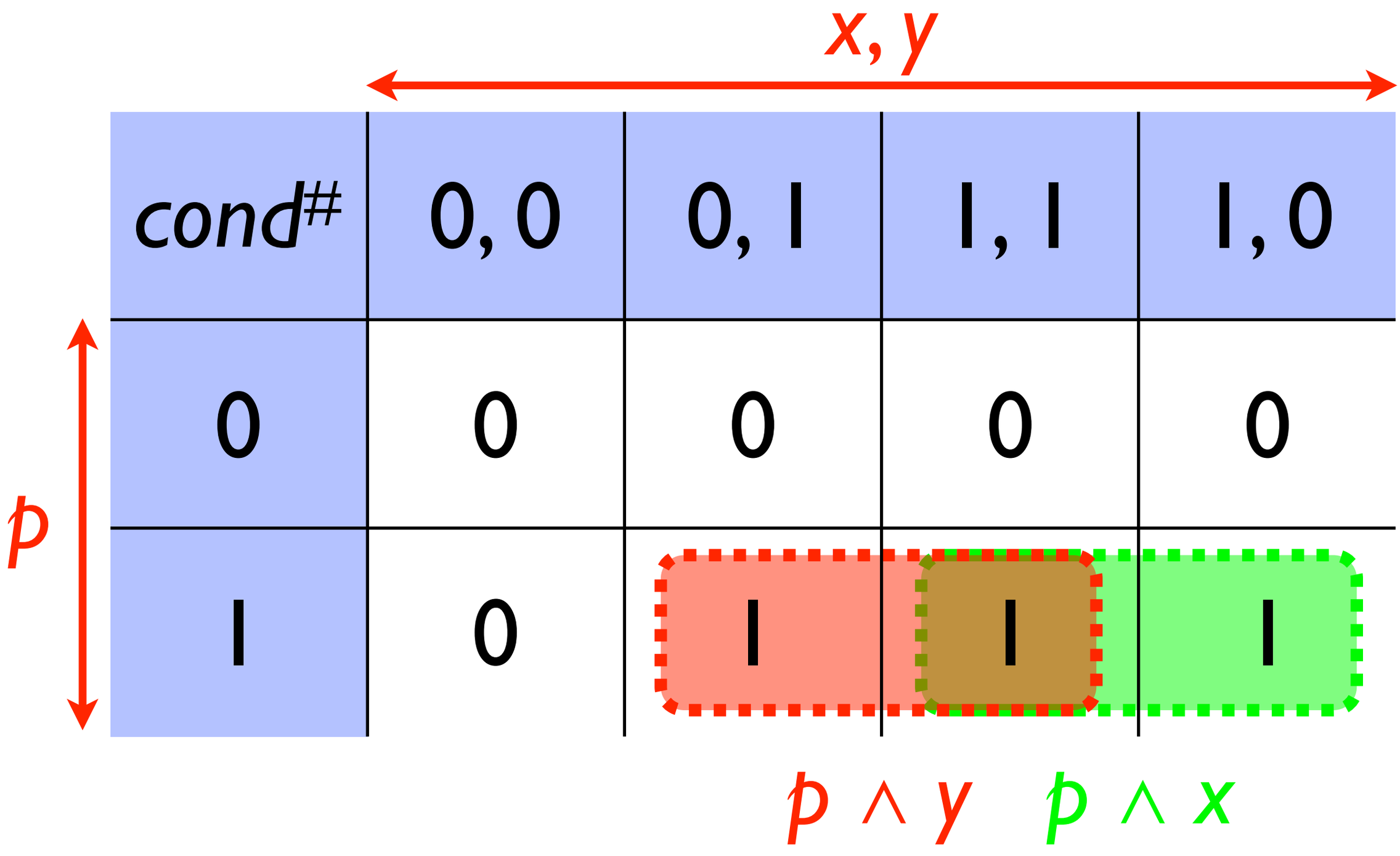
p	x	y	$cond^{\#}(p,x,y)$
0	0	0	0
0	0	I	0
0	I	0	0
0	I	I	0
I	0	0	0
I	0	I	I
I	I	0	I
I	I	I	I

Abstract interpretation

These functions may be expressed more compactly as boolean expressions, treating 0 and 1 from $D^\#$ as *false* and *true* respectively.

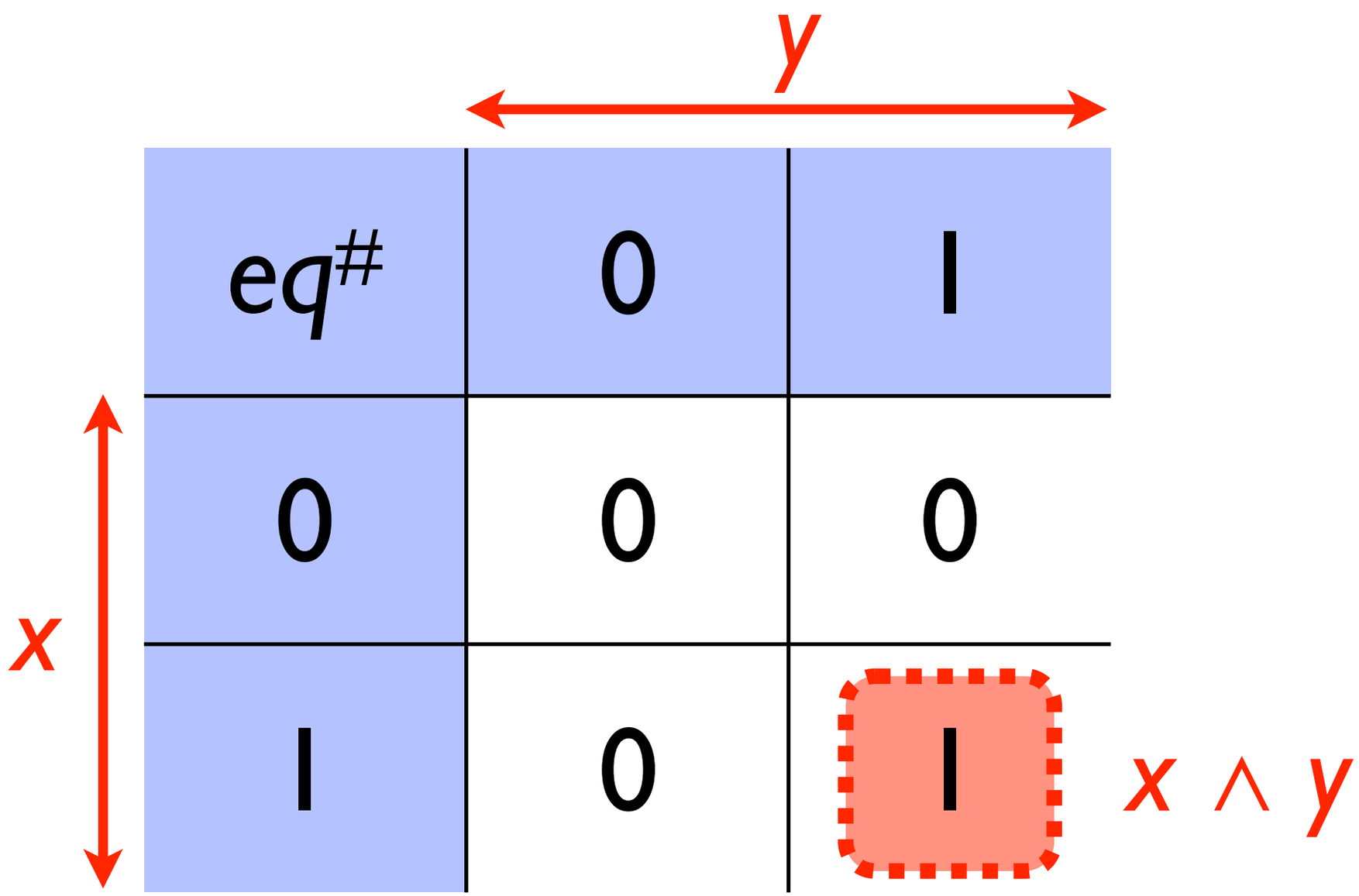
We can use Karnaugh maps (from IA DigElec) to turn each truth table into a simple boolean expression.

Abstract interpretation



$cond^{\#}(p, x, y) = (p \wedge y) \vee (p \wedge x)$

Abstract interpretation



$$eq^{\#}(x, y) = x \wedge y$$

Strictness analysis

So far, we have set up

- a concrete domain, D, equipped with
  - a standard interpretation a_i of each built-in A_i, and
  - a standard interpretation f_i of each user-defined F_i;
- and an abstract domain, $D^{\#}$, equipped with
  - an abstract interpretation $a_i^{\#}$ of each built-in A_i.

Strictness analysis

The point of this analysis is to discover the missing piece: what is the strictness function $f_i^{\#}$ corresponding to each user-defined F_i?

These strictness functions will show us exactly how each F_i is strict with respect to each of its arguments — and that's the information that tells us where we can replace lazy, CBN-style parameter passing with eager CBV.

Strictness analysis

But recall that the recursion equations show us how to build up each user-defined function, by composition and recursion, from all the built-in functions:

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

So we can build up the $f_i^\#$ from the $a_i^\#$ in the same way:

$$plus^\sharp(x, y) = cond^\sharp(eq^\sharp(x, 0^\sharp), y, plus^\sharp(sub1^\sharp(x), add1^\sharp(y)))$$

Strictness analysis

$$plus^\sharp(x, y) = cond^\sharp(eq^\sharp(x, 0^\sharp), y, plus^\sharp(sub1^\sharp(x), add1^\sharp(y)))$$

We already know all the other strictness functions:

$$\begin{aligned} cond^\sharp(p, x, y) &= p \wedge (x \vee y) \\ eq^\sharp(x, y) &= x \wedge y \\ 0^\sharp &= 1 \\ sub1^\sharp(x) &= x \\ add1^\sharp(x) &= x \end{aligned}$$

So we can use these to simplify the expression for *plus*$^\sharp$.

Strictness analysis

$$
\begin{aligned}
plus^\sharp(x,y) &= cond^\sharp(eq^\sharp(x,0^\sharp), y, plus^\sharp(sub1^\sharp(x), add1^\sharp(y))) \\
&= eq^\sharp(x,0^\sharp) \wedge (y \vee plus^\sharp(sub1^\sharp(x), add1^\sharp(y))) \\
&= eq^\sharp(x,1) \wedge (y \vee plus^\sharp(x,y)) \\
&= x \wedge 1 \wedge (y \vee plus^\sharp(x,y)) \\
&= x \wedge (y \vee plus^\sharp(x,y))
\end{aligned}
$$

Strictness analysis

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

This is a recursive definition, and so unfortunately doesn't provide us with the strictness function directly.

We want a definition of *plus*$^\#$ which satisfies this equation — actually we want the *least fixed point* of this equation, which (as ever!) we can compute iteratively.

Algorithm

```
for i = 1 to n do f#[i] := λ⃗x.0
while (f#[] changes) do
   for i = 1 to n do
      f#[i] := λ⃗x.e_i#
```

$e_i^{\#}$ means "e_i (from the recursion equations) with each A_j replaced with $a_j^{\#}$ and each F_j replaced with `f#[j]`".

Algorithm

We have only one user-defined function, *plus*, and so only one recursion equation:

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

We initialise the corresponding element of our `f#[ ]` array to contain the always-0 strictness function of the appropriate arity:

`f#[1]` := $\lambda x,y.\, 0$

Algorithm

On the first iteration, we calculate $e_1^{\#}$:

- The recursion equations say

  $e_1 = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$

  - The current contents of `f#[ ]` say $f_1^{\#}$ is $\lambda x,y.\ 0$
  - So:

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, (\lambda x,y.\ 0)\ (sub1^{\#}(x), add1^{\#}(y)))$$

Algorithm

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, (\lambda x,y.\ 0)\ (sub1^{\#}(x), add1^{\#}(y)))$$

Simplifying:

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, 0)$$

Using definitions of $cond^{\#}$, $eq^{\#}$ and $0^{\#}$:

$$e_1^{\#} = (x \wedge 1) \wedge (y \vee 0)$$

Simplifying again:

$$e_1^{\#} = x \wedge y$$

Algorithm

So, at the end of the first iteration,

$$\texttt{f\#[1]} \;:=\; \lambda x,y.\, x \wedge y$$

Algorithm

On the second iteration, we recalculate $e_1^{\#}$:

- The recursion equations still say

  $e_1 = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$

  - The current contents of `f#[ ]` say $f_1^{\#}$ is $\lambda x,y.\ x \wedge y$
  - So:

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, (\lambda x,y.\ x \wedge y)\ (sub1^{\#}(x), add1^{\#}(y)))$$

Algorithm

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, (\lambda x,y.\ x \wedge y)\ (sub1^{\#}(x), add1^{\#}(y)))$$

Simplifying:

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, sub1^{\#}(x) \wedge add1^{\#}(y))$$

Using definitions of $cond^{\#}$, $eq^{\#}$, $0^{\#}$, $sub1^{\#}$ and $add1^{\#}$:

$$e_1^{\#} = (x \wedge 1) \wedge (y \vee (x \wedge y))$$

Simplifying again:

$$e_1^{\#} = x \wedge y$$

Algorithm

So, at the end of the second iteration,

$$\texttt{f\#[1]} := \lambda x,y.\, x \wedge y$$

This is the same result as last time, so we stop.

Algorithm

$$plus^{\#}(x, y) = x \wedge y$$

Optimisation

So now, finally, we can see that

$$plus^{\#}(1, 0) = 1 \wedge 0 = 0$$

and

$$plus^{\#}(0, 1) = 0 \wedge 1 = 0$$

which means our concrete *plus* function is strict in its first argument and strict in its second argument: we may always safely use CBV when passing either.

Summary

- Functional languages can use CBV or CBN evaluation
- CBV is more efficient but can only be used in place of CBN if termination behaviour is unaffected
- Strictness shows dependencies of termination
- Abstract interpretation may be used to perform strictness analysis of user-defined functions
- The resulting strictness functions tell us when it is safe to use CBV in place of CBN