Control structures, third lecture

Declarative programming: getting rid of control?

Xavier Leroy

2024-02-08

Collège de France, chair of software sciences
`xavier.leroy@college-de-france.fr`

Declarative programming

A movement that appears in the 1960's (LISP, APL) and grows throughout the 1970's (Prolog) and 1980's (purely functional languages, reactive languages, . . .).

Idea: programs should describe what needs to be computed much more than each and every step of the computation.

This results in languages that avoid exposing memory (mutable state) and control (sequencing of computations) to programmers.

The hope is not only to facilitate the writing of programs, but also their execution in parallel.
(More declarative = fewer dependencies = more parallelism?)

Spreadsheets:
expressions with sharing

A language of expressions with sharing

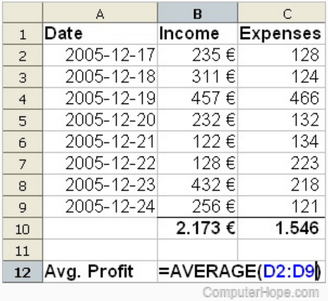
Arithmetic expressions with variables:

$$\begin{array}{rll} e ::= & 0 \mid 1.2 \mid 3.1415 \mid \ldots & \text{constants} \\ & \mid x \mid y \mid z \mid \ldots & \text{variables} \\ & \mid f(e_1, \ldots, e_n) & \text{operations } (+, -, \times, /, \Sigma, \text{etc}) \end{array}$$

Programs are sets of equations *variable* $=$ *expression*:

$$p ::= \{x_1 = e_1; \ldots; x_n = e_n\}$$

Spreadsheets



	A	B	C
1	Date	Income	Expenses
2	2005-12-17	235 €	128
3	2005-12-18	311 €	124
4	2005-12-19	457 €	466
5	2005-12-20	232 €	132
6	2005-12-21	122 €	134
7	2005-12-22	128 €	223
8	2005-12-23	432 €	218
9	2005-12-24	256 €	121
10		**2.173 €**	**1.546**
11			
12	Avg. Profit	=AVERAGE(D2:D9)	

ComputerHope.com

A visual display of a set of equations:

$$B2 = 235 \quad C2 = 128 \quad \dots \quad B9 = 256 \quad C9 = 121$$

$$D2 = B2 - C2 \quad \dots \quad D9 = B9 - C9$$

$$B10 = SUM(B2, \dots, B9) \qquad C10 = SUM(C2, \dots, C9) \qquad B12 = AVG(D2, \dots, D9)$$

The acyclicity condition

To guarantee that programs can always be evaluated, we rule out equations $x = e$ where e depends on x, directly or indirectly:

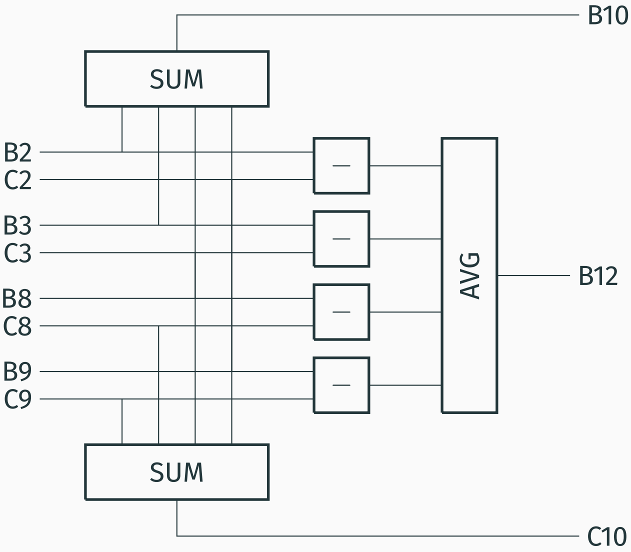
$$\{x = x^2 - 1\} \text{ ✘} \qquad \{x = y + 1;\ y = x - 1\} \text{ ✘}$$

This acyclicity condition holds if and only if we can write the program as an ordered list

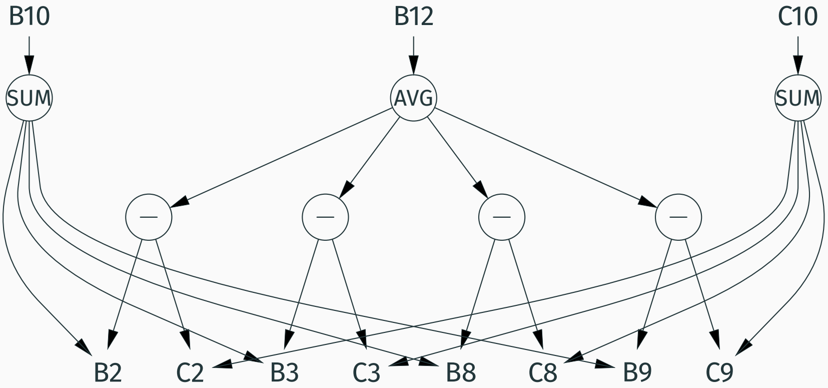
$$x_1 = e_1; \ldots; x_n = e_n$$

where the only variables that can occur in e_i are the x_j with $j < i$.

Alternate presentation: combinational circuits



Alternate presentation: the dependency DAG



Evaluation rules

We reduce the program by repeatedly applying the following rules anywhere in the right-hand sides of equations:

$$\begin{aligned} x &\to e && \text{if } x = e \text{ is an equation} \\ f(v_1, \ldots, v_n) &\to v && \text{if } v = f^*(v_1, \ldots, v_n) \end{aligned}$$

v ranges over numbers. f^* denotes the semantics of operator f, for instance $+^*$ is floating-point addition.

Evaluation stops when the program is in normal form $x_1 = v_1; \ldots x_n = v_n$.

Example:

$$\begin{aligned} \{x = 1; y = x + x\} &\to \{x = 1; y = 1 + x\} \\ &\to \{x = 1; y = 1 + 1\} \to \{x = 1; y = 2\} \end{aligned}$$

Reduction sequences

We can apply reduction rules in different orders:

$$\begin{tikzcd}
 & \{x = 2; y = x + 2\} \arrow[r] & \cdots \\
\{x = 1 + 1; y = x + 2\} \arrow[ur] \arrow[dr] & & \\
 & \{x = 1 + 1; y = (1 + 1) + 2\} \arrow[r] & \cdots
\end{tikzcd}$$

All reduction sequences terminate on the same normal form (confluence property).

Evaluation strategies

Some reduction sequences are more costly than others!

Example: $\{x_0 = 1; x_1 = x_0 + x_0; \ldots; x_n = x_{n-1} + x_{n-1}\}$.

"Call by name" strategy: we substitute before evaluating

$$\{x_0 = 1; x_1 = 1 + 1; x_2 = (1 + 1) + (1 + 1); x_3 = x_2 + x_2; \ldots\}$$

Some intermediate states have size $\mathcal{O}(2^n)$.

"Call by value" strategy: we evaluate before substituting

$$\{x_0 = 1; x_1 = 2; x_2 = 4; x_3 = x_2 + x_2; \ldots\}$$

All intermediate states have size $\mathcal{O}(n)$.

An optimal strategy

Substitute variables by values only:

$$x \to v \qquad\qquad \text{if } x = v \text{ is an equation}$$

If the program is represented as an ordered list, this strategy can be implemented as an evaluation function:

$$\begin{aligned} \texttt{eval}(\varepsilon) &= \varepsilon \\ \texttt{eval}(x = e; p) &= (x = v; \texttt{eval}(p[x \leftarrow v])) \quad \text{where } \; v = \texttt{eval}(e) \end{aligned}$$

Adding non-strict operators

A conditional expression does not need to evaluate all its arguments: it is “non-strict”.

$$\begin{aligned} \texttt{if0}(0, e, e') &\to e \\ \texttt{if0}(v, e, e') &\to e' \quad \text{if } v \neq 0 \end{aligned}$$

The “call-by-value” strategy can perform useless computations.

Example:

$$\{x = e;\ \ y = E;\ \ z = \texttt{if0}(x - x, x, y)\}$$

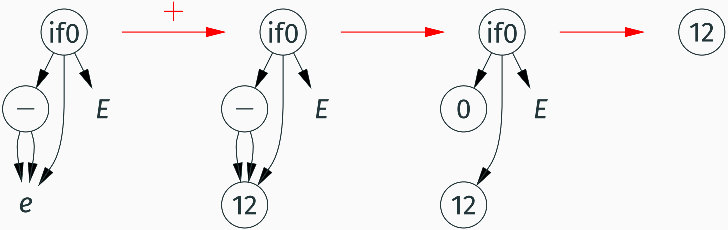
No need to evaluate the costly expression E to find the value of z.

→ Let’s use a “call-by-need” / “lazy” strategy.

Lazy evaluation

A kind of call-by-name with memoization: if $x = e$,
e is evaluated the first time x's value is needed;
its value is reused the next times x's value is needed.

Easy to express on the dependency DAG, using graph rewriting,
where operation nodes are progressively replaced by their values.



Reactive programming

Computing over streams of values

A stream: a sequence of values $v(0), v(1), \ldots, v(t), \ldots$ indexed by discrete time t.

Arithmetic operations extend pointwise to streams, e.g.

	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
x	x_0	x_1	x_2	x_3	x_4	x_5
1	1	1	1	1	1	1
$x + 1$	$x_0 + 1$	$x_1 + 1$	$x_2 + 1$	$x_3 + 1$	$x_4 + 1$	$x_5 + 1$

Temporal operators

Give access to the values of a stream at an earlier time.

v `fby` e (read: *followed by*)

is the constant v at time 0; is $e(t)$ at time $t + 1$;

$\approx$ the `cons` constructor for infinite lists.

Example:

	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
x	1	2	3	4	5	6
`0 fby` x	0	1	2	3	4	5
`0 fby 0 fby` x	0	0	1	2	3	4

Temporal operators

Derived operators:

$\texttt{pre}(e) \equiv \texttt{None fby } e$

is $e(t)$ at time $t+1$; is undefined at time 0.

$e_1 \texttt{ -> } e_2 \equiv \texttt{if (true fby false) then } e_1 \texttt{ else pre}(e_2)$

is $e_1(0)$ at time 0 and $e_2(t)$ at time $t+1$.

Example:

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
x	0	-1	-2	-3	-4	-5
y	1	2	3	4	5	6
`pre`(y)	`None`	1	2	3	4	5
`true fby false`	`true`	`false`	`false`	`false`	`false`	`false`
x `->` y	0	1	2	3	4	5

Stream equations

(Lustre (P. Caspi, N. Halbwachs, 1985), Scade, Simulink.)

A reactive program is a set of equations over streams $\{x_1 = e_1; \ldots ; x_n = e_n\}$.

Example: $\{x = 0 \texttt{ fby } 1 + x\}$

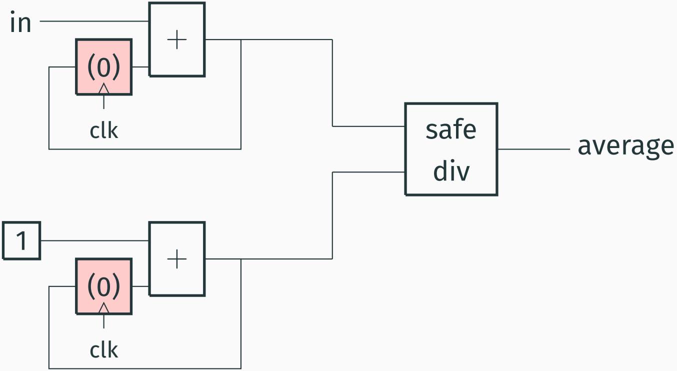
	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
x	0	1	2	3	4	5
$1 + x$	1	2	3	4	5	6
$0 \texttt{ fby } 1 + x$	0	1	2	3	4	5

Example: $\{sum = 0 \texttt{ fby } in + sum\}$ where *in* is an input stream.

	$t = 0$	$t = 1$	$t = 2$	$t = 3$	$t = 4$
in	i_0	i_1	i_2	i_3	i_4
sum	0	i_0	$i_0 + i_1$	$i_0 + i_1 + i_2$	$i_0 + i_1 + i_2 + i_3$

Connections with synchronous circuits

The `fby` operator plays the role of a memory element (flip-flop, latch, etc) in a synchronous circuit.



In Simulink and in Scade, block diagrams like the above are a graphical notation for sets of stream equations.

Causality condition

In an equation $x = e$, the right-hand side e must not depend (directly or indirectly) on the current value of x, but may depend on previous values of x.

In other words: every dependency cycle must cross a temporal operator at least once.

Examples:

$\{x = x + 1\}$ ✘ $\{x = y;\ y = x\}$ ✘

$\{x = 0 \ \texttt{fby} \ x + 1\}$ ✔ $\{x = 0 \ \texttt{fby} \ y;\ y = x + 1\}$ ✔

Denotational semantics

The denotational semantics of the program $\{\ldots;\ x_i = e_i;\ \ldots\}$
is the solution to the equations $x_i = e_i$
(an assignment of streams to variables x_i),
provided this solution exists and is unique.

Examples:

$\{x = x + 1\}$	no solution
$\{x = y; y = x\}$	many solutions
$\{x = 0\ \texttt{fby}\ x + 1\}$	unique solution $x = 0, 1, 2, 3, 4, \ldots$

Theorem: every causal program has a unique solution.

(Can be proved using metric spaces and the Banach-Tarski fixed-point theorem.)

Operational semantics

(P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice, *LUSTRE: A declarative language for programming synchronous systems*, POPL 1987.)

Given a program $P = \{x_i = e_i\}$, we can evaluate it at time 0 by "projecting" is equations

$$P(0) \overset{def}{=} \{x_i(0) = now(e_i)\}$$

where now : stream expression $\rightarrow$ value expression
is defined as

$$now(v) = v \qquad now(x) = x(0)$$

$$now(v\ \texttt{fby}\ e) = v \quad now(f(\ldots, e_i, \ldots)) = f(\ldots, now(e_i), \ldots)$$

The program $P(0)$ is an acyclic spreadsheet (if P is causal).
Evaluating it determines $x_1(0), \ldots, x_n(0)$.

Operational semantics

We then build the residual program

$$P' \overset{def}{=} \{x_i = later(e_i)\}$$

where $later$: stream expression $\rightarrow$ stream expression
is defined as

$$\begin{aligned} later(v) &= v \\ later(x) &= x \\ later(v \ \texttt{fby} \ e) &= v' \ \texttt{fby} \ e \quad \text{where } v' \text{ is the value of } e \text{ at time } 0 \\ later(f(\ldots, e_i, \ldots)) &= f(\ldots, later(e_i), \ldots) \end{aligned}$$

We iterate execution with P', then P'', then …
$\rightarrow$ sequences of values $x_i(t)$ for $i = 1, \ldots, n$ and $t \in \mathbb{N}$
which are solutions of the stream equations $x_i = e_i$.

Normalizing programs

Every causal program can be put in the following normal form (by adding variables and equations as needed):

$$\begin{aligned}
m_1 &= v_1 \ \texttt{fby} \ f_1 && \text{(memory registers)} \\
&\vdots \\
m_k &= v_k \ \texttt{fby} \ f_k \\
x_1 &= e_1 && \text{(wires)} \\
&\vdots \\
x_n &= e_n
\end{aligned}$$

Expressions e_i and f_j are instantaneous (no `fby`).

e_i depends only on $m_1, \ldots, m_k, e_1, \ldots e_{i-1}$ ("backward" deps.).

f_j depends only on $m_j, \ldots, m_k, e_1, \ldots, e_n$ ("forward" deps.).

Generating imperative code

From the normalized form, it is easy to produce imperative code that implements the reactive program.

```
// initialization of the memory registers
$m_1$ = $v_1$; ...; $m_k$ = $v_k$;
while (true) {
    // acquiring the inputs
    $in$ = read_input();
    // computing the values of the wires
    $x_1$ = $e_1$; ...; $x_n$ = $e_n$;
    // producing the outputs
    write_output($x_n$);
    // updating the memory registers
    $m_1$ = $f_1$; ...; $m_k$ = $f_k$
}
```

Functional programming

An applicative language with “second-class” functions

Start again from spreadsheets (equations $var = expr$) and add optional parameters to the variables.

Example: Fibonacci’s sequence, as defined in textbooks.

$$\begin{aligned} F\,n &= \texttt{if } n = 0 \texttt{ then } 0 \texttt{ else} \\ &\quad\; \texttt{if } n = 1 \texttt{ then } 1 \texttt{ else} \\ &\quad\; F\,(n-1) + F\,(n-2) \\ x &= F\,20 \end{aligned}$$

Or, closer to the usual iterative algorithm,

$$\begin{aligned} G\,n\,a\,b &= \texttt{if } n = 0 \texttt{ then } a \texttt{ else } G\,(n-1)\,b\,(a+b) \\ F\,n &= G\,n\,0\,1 \\ x &= F\,20 \end{aligned}$$

An applicative language

Expressions:

e	$::= 0 \mid 1.2 \mid \dots$	constants
	$\mid x$	variables
	$\mid op(e_1, \dots, e_n)$	built-in operations
	$\mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$	conditional
	$\mid F\ e_1 \cdots e_n$	function application

Definitions:

def	$::= x = e$	variable definition
	$\mid F\ x_1 \cdots x_n = e$	function definition

Programs:

$$P ::= def; \dots ; def$$

A Turing-complete language

This applicative language contains Kleene's partial recursive functions. It is therefore Turing-complete.

Assuming given the basic operations over tapes, we can easily transcribe a given Turing machine as a program in our language: each state of the machine becomes a function taking the tape as argument.

$$\begin{array}{rcl} S_1\,t & = & \texttt{if read}(t) = \mathbf{A}\ \texttt{then}\ S_2\,(\texttt{left}(\texttt{write}(\mathbf{C}, t))\ \texttt{else} \\ & & \texttt{if read}(t) = \mathbf{B}\ \texttt{then}\ S_3\,(\texttt{right}(\texttt{write}(\mathbf{A}, t))\ \texttt{else}\ \ldots \\ S_2\,t & = & \ldots \\ S_3\,t & = & \ldots \end{array}$$

Control structures in an applicative language

Conditionals are built in the language.
They are expression `if` $cond$ `then` e_1 `else` e_2
instead of commands like in Algol.

Loops are presented as tail-recursive functions, e.g.

$$G\ n\ a\ b = \texttt{if}\ \ n = 0\ \texttt{then}\ a\ \texttt{else}\ G(n-1)\ b\ (a+b)$$

is the loop $\texttt{while}(n \neq 0)\ \{n = n - 1; (a, b) = (b, a + b); \}$

"goto" jumps are presented as function calls in tail position, e.g.

(note: non-reducible CFG!)

$$\begin{aligned} F\ n &= \texttt{if}\ \ n < 0\ \texttt{then}\ Odd\ (-n)\ \texttt{else}\ Even\ n \\ Even\ n &= \texttt{if}\ \ n = 0\ \texttt{then}\ \texttt{true}\ \texttt{else}\ Odd\ (n-1) \\ Odd\ n &= \texttt{if}\ \ n = 0\ \texttt{then}\ \texttt{false}\ \texttt{else}\ Even\ (n-1) \end{aligned}$$

A functional language with functions as first-class values

Add the expression $\lambda x.e$, “the function that associates e to x”. We no longer need to distinguish function names and variable names.

Expressions:

$$\begin{array}{rll}
e ::= & 0 \mid 1.2 \mid \ldots & \text{constants} \\
\mid & x & \text{variables} \\
\mid & op & \text{built-in operations} \\
\mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{conditional} \\
\mid & \lambda x.e & \text{function abstraction} \\
\mid & e_1\ e_2 & \text{function application}
\end{array}$$

Programs:

$$P ::= x_1 = e_1; \ldots ; x_n = e_n$$

Notation: $f\ x_1\ \ldots\ x_n = e$ stands for $f = \lambda x_1 \ldots \lambda x_n.\ e$

Higher-order functions

Provide new ways to compose and reuse sub-programs.

Examples:

$$
\begin{aligned}
\mathit{Compose}\ f\ g &= \lambda x.\, f(g\ x) \\
\mathit{Exp}\ f\ n &= \texttt{if}\ \ n = 0\ \ \texttt{then}\ \ \lambda x.x\ \ \texttt{else} \\
&\ \mathit{Compose}\ f\ (\mathit{Exp}\ f\ (n-1)) \\
\mathit{Iter}\ \mathit{stop}\ \mathit{step}\ x &= \texttt{if}\ \ \mathit{stop}\ x\ \ \texttt{then}\ \ x\ \ \texttt{else}\ \ \mathit{Iter}\ \mathit{stop}\ \mathit{step}\ (\mathit{step}\ x) \\
\mathit{Map}\ f\ \ell &= \texttt{if}\ \ \ell = \texttt{nil}\ \ \texttt{then}\ \ \texttt{nil}\ \ \texttt{else} \\
&\ \texttt{cons}\ (f\ (\texttt{head}\ \ \ell))\ (\mathit{Map}\ f\ (\texttt{tail}\ \ \ell)) \\
\mathit{Foldl}\ f\ a\ \ell &= \texttt{if}\ \ \ell = \texttt{nil}\ \ \texttt{then}\ \ a\ \ \texttt{else} \\
&\ \mathit{Foldl}\ f\ (f\ a\ (\texttt{head}\ \ \ell))\ (\texttt{tail}\ \ \ell)
\end{aligned}
$$

Reduction to lambda-calculus with constants

We can reduce our functional language to untyped lambda-calculus with constants

$$e ::= cst \mid x \mid \lambda x.\, e \mid e_1\; e_2$$

via a few encodings:

- Non-recursive definitions:

  $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leadsto (\lambda x. e_2)\; e_1$

- Single recursive definition:

  $\texttt{let rec}\ f = e_1\ \texttt{in}\ e_2 \leadsto \texttt{let}\ f = \mathit{Fix}(\lambda f. e_1)\ \texttt{in}\ e_2$

  where *Fix* is a fixed-point combinator such as $\lambda f.\, (\lambda x.\, f\; (x\; x))\; (\lambda x.\, f\; (x\; x))$.

- Mutually-recursive definitions $\leadsto$ one `let rec` + several `let`.

Evaluation rules

A function application reduces to the body of the function where the formal parameter is replaced by the actual argument. (β-reduction in the lambda-calculus; “copy rule” in Algol.)

$$(\lambda x.\, e)\; e' \rightarrow e\{x \leftarrow e'\}$$

Plus specific rules for constants and operators:

$$\begin{aligned}
\texttt{if true then } e_1 \texttt{ else } e_2 &\rightarrow e_1 \\
\texttt{if false then } e_1 \texttt{ else } e_2 &\rightarrow e_2 \\
op\; cst_1\; \ldots\; cst_n &\rightarrow cst \qquad \text{if } cst = op^*(cst_1, \ldots, cst_n)
\end{aligned}$$

Sensitivity to evaluation order

Confluence property: all terminating reduction sequences terminate on the same normal form.

However, some reduction sequences can diverge while other sequences terminate, more or less quickly.

Example: if $\Omega \xrightarrow{+} \Omega$, (e.g. `let rec` $\Omega = \Omega$, i.e. $\Omega = \mathit{Fix}(\lambda f.f)$),

$$(\lambda x.\, 0)\, \Omega \xrightarrow{+} (\lambda x.\, 0)\, \Omega \xrightarrow{+} (\lambda x.\, 0)\, \Omega \xrightarrow{+} \cdots$$

$$(\lambda x.\, 0)\, \Omega \rightarrow 0$$

The usual reduction strategies

Call by name: the function argument is substituted unevaluated in the function body.

Avoids divergence, but duplicates computations:

$$(\lambda x.\, 0)\ \Omega \to 0$$

$$\begin{aligned}(\lambda x.\, x + x)\ (\mathit{Fib}\ 20) &\to \mathit{Fib}\ 20 + \mathit{Fib}\ 20\\ &\overset{+}{\to} 6765 + \mathit{Fib}\ 20 \overset{+}{\to} 6765 + 6765 \to 13530\end{aligned}$$

The usual reduction strategies

Call by name: the function argument is substituted unevaluated in the function body.

Call by value: the function argument is reduced to a value before being substituted in the function body.

Avoids duplicate computations, but can cause divergence:

$$(\lambda x.\, 0)\ \Omega \xrightarrow{+} (\lambda x.\, 0)\ \Omega \xrightarrow{+} (\lambda x.\, 0)\ \Omega \xrightarrow{+} \ldots$$

$$(\lambda x.\, x + x)\ (\mathit{Fib}\ 20) \xrightarrow{+} (\lambda x.\, x + x)\ 6765 \rightarrow 6765 + 6765 \rightarrow 13530$$

The usual reduction strategies

Call by name: the function argument is substituted unevaluated in the function body.

Call by value: the function argument is reduced to a value before being substituted in the function body.

Call by need ("lazy evaluation"):
like call by name, but using memoized evaluations
or, equivalently, using graph rewriting instead of term rewriting.

Avoids divergence without duplicating computations:

$$(\lambda x.\, 0)\ \Omega \to 0$$

$$\begin{array}{ccccccccc}
(\lambda x.\, x + x)\ (\mathit{Fib}\ 20) \to & \mathit{Fib}\ 20 & + & \mathit{Fib}\ 20 & \xrightarrow{+} & 6765 & + & 6765 & \to 13530 \\
 & \nwarrow & & \nearrow & & \nwarrow & & \nearrow & \\
 & & \text{shared} & & & & \text{shared} & &
\end{array}$$

Encoding call by name

Even if our language is call by value, we can implement a call-by-name semantics by passing arguments e as thunks $\lambda z.e$ (with z not free in e).

(This is “weak reduction”: e is not reduced in $\lambda z.e$ before application.)

A systematic program transformation:

$$
\begin{aligned}
\mathcal{N}(x) &= x\ (\,) \\
\mathcal{N}(\lambda x.e) &= \lambda x.\, \mathcal{N}(e) \\
\mathcal{N}(e_1\ e_2) &= \mathcal{N}(e_1)\ (\lambda z.\, \mathcal{N}(e_2))
\end{aligned}
$$

We obtain call by need if we use memoized thunks (`lazy` e in OCaml) instead of plain thunks $\lambda z.\, e$.

Encoding call by value

Using the continuation-passing style (CPS).

→ Lecture #4.

Logic programming

Logic programming

A program = a set of rules that define predicates.

Running the program = finding the values of variables X_i for which a predicate $p(X_1, \ldots, X_n)$ holds.

Connections with automated deduction:
satisfiability problems; Robinson's resolution algorithm.

Connections with query processing in relational databases.

Horn clauses

A set of statements and hypothetical statements of the form

$$(\forall \vec{x},)\ p \qquad \text{or} \qquad (\forall \vec{x},)\ q_1 \wedge \cdots \wedge q_n \Rightarrow p$$

p, q_i are literals. Variables x_i are implicitly universally quantified.

Ex: $\quad even(0) \quad$ or $\quad (\forall n,)\ odd(n-1) \Rightarrow even(n).$

Prolog syntax:

```
p  .
p  :-  q1 ,  ...,  qn  .
```

Alternate view: axioms and inference rules in natural deduction.

$$\frac{}{p} \qquad\qquad \frac{q_1 \quad \cdots \quad q_n}{p}$$

Datalog

Literals = predicates over variables (uppercase) and constants (lowercase).

```
parent(tom, sally).
parent(erica, sally).
parent(tom, bart).
parent(martha, tom).

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X != Y.

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Prolog (A. Colmerauer and R. Kowalski, 1972)

Literals = predicates over terms build from variables and constants by constructor application (e.g. the list constructor `[X|Y]`, read “`X` cons `Y`”).

Example: the permutations of a list.

```
permut([], []).
permut([X|Xs], Ys) :- permut(Xs, Zs), insert(X, Zs, Ys).

insert(X, Ys, [X|Ys]).
insert(X, [Y|Ys], [Y|Zs]) :- insert(X, Ys, Zs).
```

Application: syntactic analysis

```
s(S0,S) :- np(S0,S1), vp(S1,S).
np(S0,S) :- det(S0,S1), n(S1,S).
vp(S0,S) :- tv(S0,S1), np(S1,S).
vp(S0,S) :- v(S0,S).
det([the|S],S).
det([a|S],S).
det([every|S],S).
n([man|S],S).
n([woman|S],S).
n([park|S],S).
tv([loves|S],S).
tv([likes|S],S).
v([walks|S],S).
```

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP \\
NP & \rightarrow & Det\ N \\
VP & \rightarrow & TV\ NP \\
VP & \rightarrow & V \\
Det & \rightarrow & \text{the} \mid \text{a} \mid \text{every} \\
N & \rightarrow & \text{man} \mid \text{woman} \mid \text{park} \\
TV & \rightarrow & \text{loves} \mid \text{likes} \\
V & \rightarrow & \text{walks}
\end{array}
$$

Executing a Prolog program

By backward reasoning from the user request,
repeatedly applying the SLD resolution rule (Kowalski):

> to prove the goal $g_1 \wedge \cdots \wedge g_n$:
> choose a clause p `:-` $q_1, \ldots, q_m$
> and a unifier θ such that $\theta(p) = \theta(g_i)$;
> now prove $\theta(g_1 \wedge \cdots \wedge g_{i-1} \wedge q_1 \wedge \cdots \wedge q_m \wedge g_{i+1} \wedge \cdots \wedge g_n)$.

Most implementations use depth-first search,
with backtracking on failure.

Examples of execution

The program:

```
append([], X, X).
append([H|T], X, [H|U]) :- append(T, X, U).
```

Some queries:

```
?- append([1,2], [3,4,5], [1,X,3,4,Y]).
% X = 2, Y = 5.
?- append([1,2], [3,4,5], X).
% X = [1, 2, 3, 4, 5].
?- append([1,2], X, [1,2,3,4,5]).
% X = [3, 4, 5]
?- append(X, [3,4,5], [1,2,3,4,5]).
% X = [1, 2]
?- append(X, Y, [1,2])
% X = [], Y = [1, 2] ; X = [1], Y = [2] ; X = [1, 2], Y = []
```

Sensitivity to the resolution strategy

Most Prolog implementations use depth-first search without memoization, which can trivially diverge:

```
p(X) :- p(X).
q(X) :- q(f(X)).
```

Or, less trivially:

```
rstar(X, X).
rstar(X, Y) :- rstar(X, Z), r(Z, Y).
```

(Produces the first result, then diverges.)

Sensitivity to the resolution strategy

Backtracking often causes recomputations that are not necessary, in the sense that they produce no new solutions.

```
list_mem(X, [X|_]).
list_mem(X, [_|Ys]) :- list_mem(X, Ys).

list_add(X, L, L) :- list_mem(X, L).
list_add(X, L, [X|L]) :- not(list_mem(X, L)).
```

There is only one solution to $\texttt{list\_add}(0, [\overbrace{0, \ldots, 0}^{n \text{ zeros}}], \texttt{X})$ but *n* different ways to obtain it!

→ Computing all the solutions takes time $\mathcal{O}(n^2)$.

The cut operator

The cut operator, written "`!`", gives programmers some control over backtracking.

```
list_add(X, L, L) :- list_mem(X, L), ! .
list_add(X, L, [X|L]) :- not(list_mem(X, L)).
```

Executing the "`!`" removes the current alternatives for `list_add`, which are "retry `list_mem(X, L)`"
and "try the second clause of `list_add`".

Cuts can also express negation:

```
list_add(X, L, L) :- list_mem(X, L), ! .
list_add(X, L, [X|L]).
```

The cut operator: the “goto” of logic programming?

A low-level mechanism; formal semantics is unclear.

Affects not just the running time but also the meaning of programs! (“green cuts” vs. “red cuts”). Example:

```
p(a) :- ! .
p(b).
```

vs.

```
p(b).
p(a) :- ! .
```

Alternatives:

- Special operators such as `firstof`(p).
- Additional control structures such as the conditional literal (*p* `->` *q* `;` *r*).
- Meta-languages to control the resolution strategy.

Summary

Declarative programming

Striking a delicate balance between

describing more what needs to be computed,
less how to compute it

and

keeping under control termination and complexity
(in time, in space) of programs.

Control structure remain needed,
no longer to specify the exact chaining of operations,
but to control the general execution strategy.

Three families of declarative languages

Reactive languages:

- Declarative AND efficient!
- At the expense of limited expressiveness.

Functional languages:

- A reduction strategy must be fixed in advance.
- Call by value: an intuitive cost model.
- Call by need: better properties; cost model hard to grasp.

Logic languages:

- Programmers need some control over the resolution strategy.
- How to do this? no consensus.

References

References

Declarative programming from a “functional” perspective:

- H. Abelson, G. J. Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, 1996. Chapter 1.

Declarative programming from a “logic” perspective:

- P. Van Roy, S. Haridi. *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004. Chapters 1, 2, 3 and 9.

An introduction to the Lustre language:

- N. Halbwachs, P. Raymond. *A tutorial of Lustre*, 2007. `https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/lustre_tutorial.pdf`