NPRG075

Formal models of programming

Tomáš Petříček, 309 (3rd floor)

petricek@d3s.mff.cuni.cz

https://tomasp.net | @tomaspetricek

Lectures: Monday 12:20, S7

https://d3s.mff.cuni.cz/teaching/nprg075

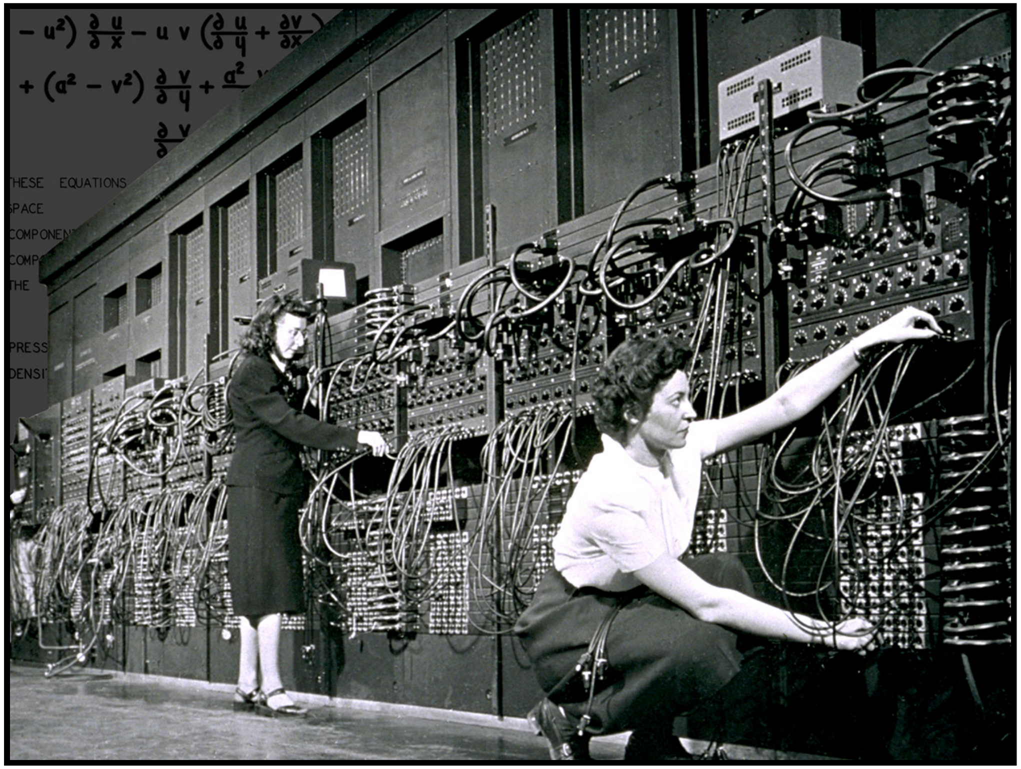
History

Programming as mathematics

Programming in the late 1940s

ENIAC programmed by plugging wires and flipping switches

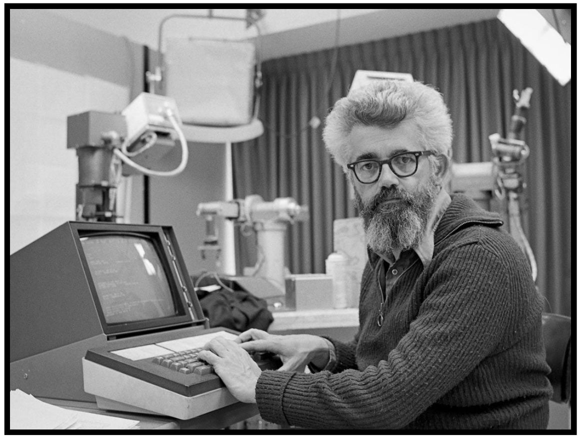
"The ENIAC was a son-of-a-bitch to program" - Jean (Jennings) Bartik



Mathematical science of computation

John McCarthy (1962)

In a mathematical science, it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science.



What we want to answer

- Does transformation preserve meaning?
- Does translation procedure correctly translate?
- Do two programs compute the same function?

```
              value (τ , ξ) = if isvar (τ) then  c( τ,ξ)
else if isconst (τ) then val (τ)
else if issum (τ) then value (addend (τ), ξ) + value (augend (τ), ξ)
else if isdiff (τ) then value (subtrahend (τ), ξ)  - value (minuend (τ), ξ)
else if isprod (τ) then value (multiplier (τ), ξ) x value (multiplicand (τ), ξ)
else if isquotient (τ) then value (numerator (τ), ξ)/value (denominator(τ),ξ)
else if iscond (τ) then (if value (proposition (τ), ξ) then
                value(antecedent (τ), ξ) else value (consequent (τ), ξ))
else if isequal (τ) then (value(lefteq (τ), ξ) = value (righteq (τ), ξ))
else if isless (τ) then (value (leftl (τ), ξ) < value (rightl(τ), ξ))


    micro (π ,ξ) = (λ n .if end (π , n ) then ξ
else (λ s . if assignment (s) then
micro(π, a(sn, n + 1, a(left(s), value (right(s),ξ),ξ)))
                 else if goto (s) then
micro (π, a(sn, if value (proposition (s), ξ) then
                 numb (destination (s),π) else n + 1, ξ)))
(statement (n,π)) (c (sn,ξ))
```

Microalgol (1964)

Syntax and semantics of trivial Algol subset

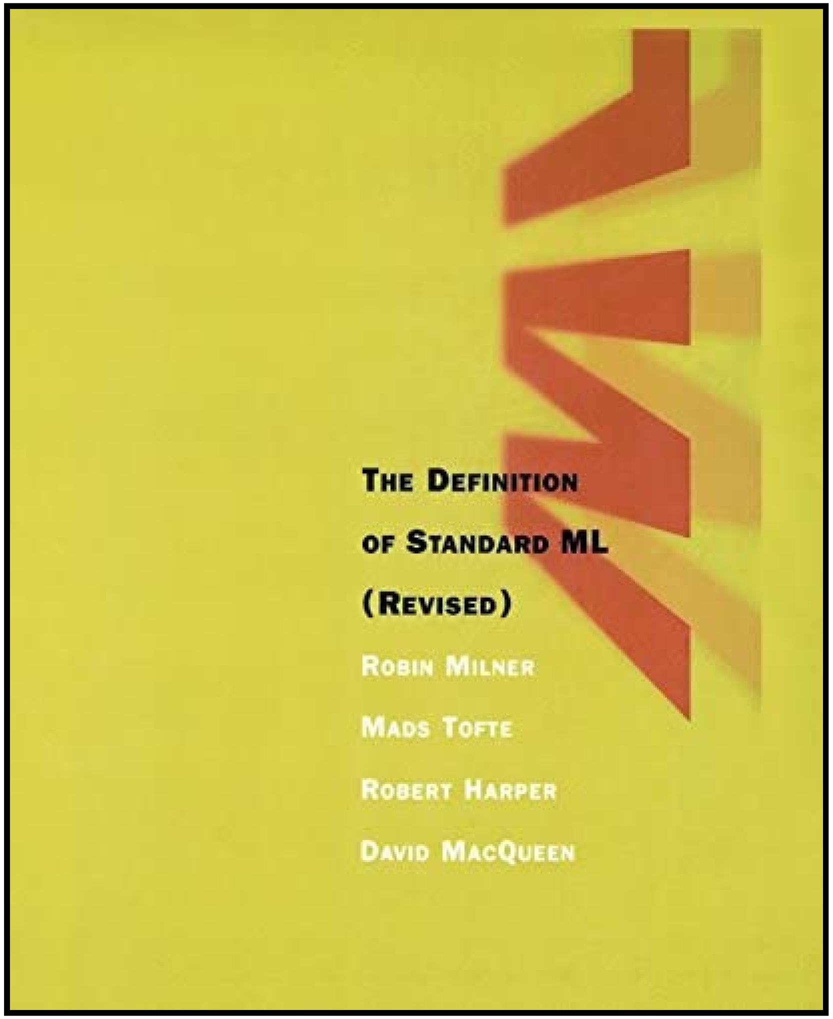
$micro(\pi, \xi)$ gives the final state of a program π run in a state ξ

"Description of the state of an Algol computation will clarify (..) compiler design"

Formal models

What are they good for?

- Make sense of tricky language features
- Prove properties of specific programs
- Prove properties of the language
- Make sure type system actually prevents bugs!



The definition of Standard ML (1990s)

Operational semantics and type system for a complete language

Even language this simple had murky parts!

Generalization and value restriction

```
// Function: 'a -> 'a list
let callLogger =
  // List: 'a list
  let mutable log = []
  fun x ->
    log <- x :: log
    log

// Can we call this with:
callLogger 10
callLogger "hi"
```

ML makes top-level definitions polymorphic

Allowing that for values is unsound!

Soundness

Surely, we know better?

- Are such problems in programming languages used today?
- tinyurl.com/nprg075-unsound

Unexpected interactions!

- Many Java extensions formalized
- Formalizations with soundness proofs!
- This is interaction between multiple features...

Semantics

Formal language definitions

Language semantics types

- **Axiomatic semantics**
  Define rules satisfied by individual commands
- **Denotational semantics**
  Assign mathematical entity to each program
- **Big-step operational semantics**
  Describe how terms reduce to values
- **Small-step operational semantics**
  Evaluation as gradual rewriting of terms

Language semantics types

denotational

$[\![e]\!] = ?$

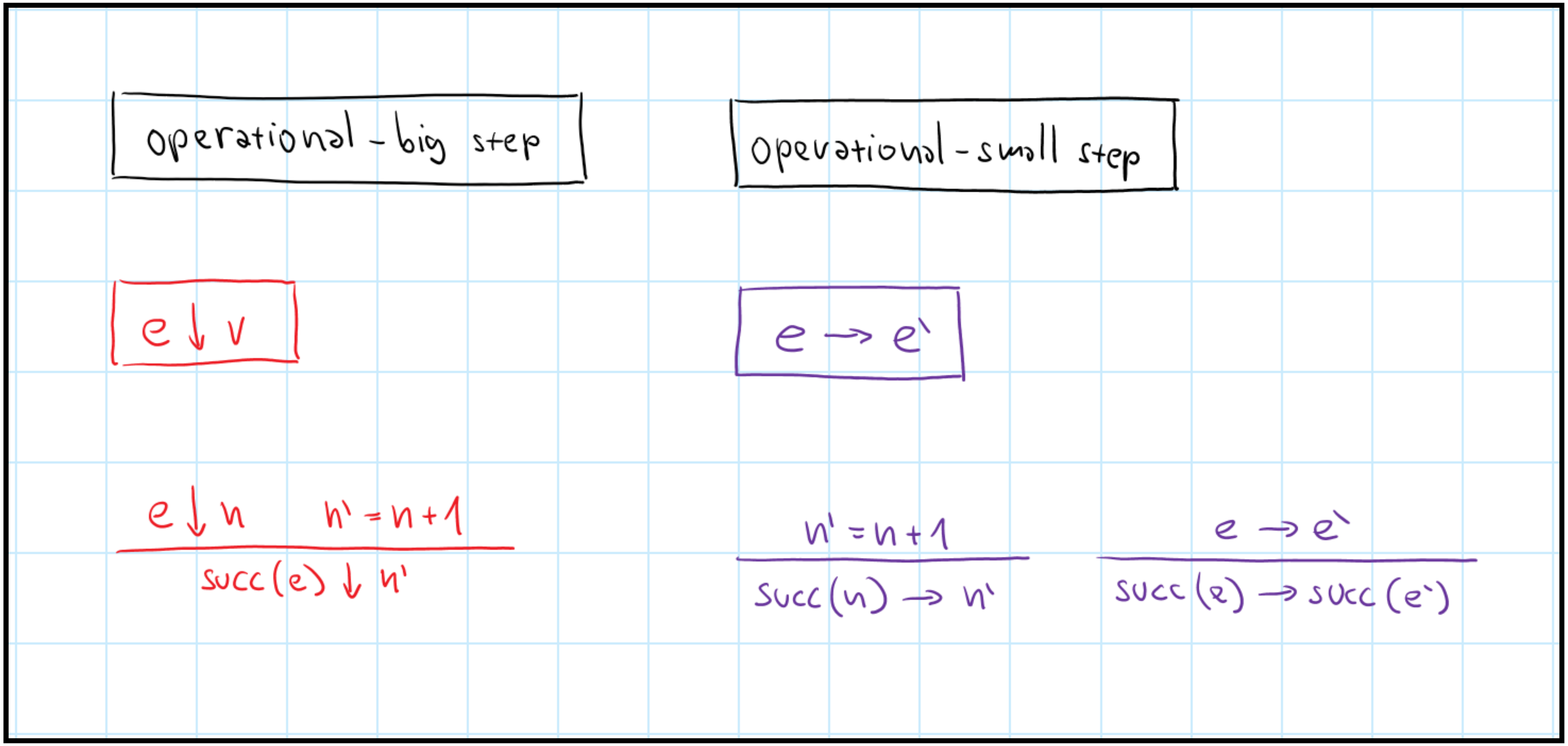
$[\![\text{let } x = e_1 \text{ in } e_2]\!] = [\![e_2]\!] \circ [\![e_1]\!]$

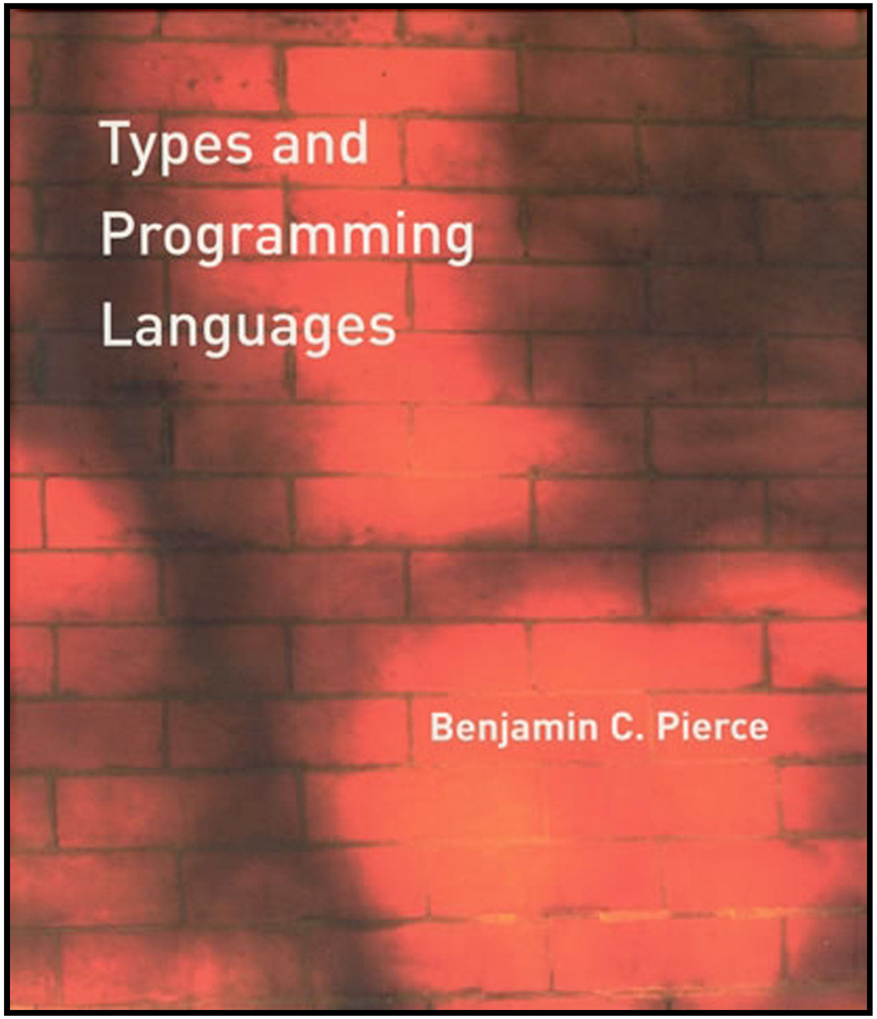
axiomatic

$\{P\}\, e\, \{Q\}$

$$\frac{\{P\}\, e_1\, \{Q\} \qquad \{Q\}\, e_2\, \{R\}}{\{P\}\, e_1 ; e_2\, \{R\}}$$

Language semantics types





Why small-step?

Easier to write than axiomatic or denotational

But harder to use for program equivalence

Good textbook and popular in PL research community

Works for programs that do not terminate

Semantics

Definition of an ML subset

Demo

Functions and numbers in F#

Expressions and evaluation

simple syntax

$$v ::= n \mid \lambda x.e$$

$$e ::= n \mid e + e \mid \lambda x.e \mid x \mid e\,e$$

evaluation example

$$(\lambda x.\, x + 10)\ (2 + 3)$$

$$\rightarrow (\lambda x.\, x + 10)\ 5$$

$$\rightarrow (5 + 10)$$

$$\rightarrow 15$$

Evaluation rules

evaluation rules

$$\frac{n_3 = n_1 + n_2}{n_1 + n_2 \rightarrow n_3}\ (\text{plus}) \qquad \frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}\ (\text{app1})$$

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2}\ (\text{plus1}) \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}\ (\text{app2})$$

$$\frac{e_2 \rightarrow e_2'}{v_1 + e_2 \rightarrow v_1 + e_2'}\ (\text{plus2}) \qquad \frac{}{(\lambda x.e)\ v \rightarrow e[x/v]}\ (\text{app3})$$

Functions and numbers

example 1 in detail

$$\dfrac{\dfrac{5 = 2 + 3}{2 + 3 \longrightarrow 5}\ (\text{plus})}{(\lambda x . x + 10)\ (2 + 3) \longrightarrow (\lambda x . x + 10)\ 5}\ (\text{app2})$$

$$\dfrac{}{(\lambda x . x + 10)\ 5 \longrightarrow 5 + 10}\ (\text{app3})$$

$$\dfrac{15 = 5 + 10}{5 + 10 \longrightarrow 15}\ (\text{plus})$$

Functions and currying

example 2 in detail

$$\dfrac{\dfrac{}{(\lambda x.\lambda y.x+y)\ 10 \rightarrow (\lambda y.10+y)}\ (app3)}{(\lambda x.\lambda y.x+y)\ 10\ 5 \rightarrow (\lambda y.10+y)\ 5}\ (app1)$$

$$(\lambda y.10+y)\ 5 \xrightarrow{(app3)} 10+5 \xrightarrow{(plus)} 15$$

Simplifying the rules

evaluation contexts

$$v ::= n \mid \lambda x.\, e$$

$$e ::= v \mid x \mid e + e \mid e\ e$$

$$C[\bullet] ::= \bullet \mid v + C[\bullet] \mid C[\bullet] + e \mid v\ C[\bullet] \mid C[\bullet]\ e$$

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}\ (C+X)$$

Conditionals and stuck state

extensions

$e ::= \ldots \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$

$C[\cdot] ::= \ldots \mid \text{if } C[\cdot] \text{ then } e_2 \text{ else } e_3$

$$\frac{n \neq 0}{\text{if } n \text{ then } e_2 \text{ else } e_3 \to e_2} \qquad \frac{n = 0}{\text{if } n \text{ then } e_2 \text{ else } e_3 \to e_3}$$

why types?

$\text{if } (\lambda x. x) \text{ then } 1 \text{ else } 2 \not\to$

Stuck!

Adding references

references

$$e ::= \dots \mid\ !\ell \mid \ell := e$$

$$C[\cdot] ::= \dots \mid \ell := C[\cdot]$$

$$\ell \in \mathbb{L}$$

$$s \in \mathbb{L} \to \mathbb{V}$$

$$v \in \mathbb{V}$$

$$\langle e, s \rangle \longrightarrow \langle e', s' \rangle$$

$$\frac{s(\ell) = v}{\langle !\ell, s \rangle \to \langle v, s \rangle} \ (\text{get})$$

$$\frac{s'(\ell') = \begin{cases} v & \text{when } \ell' = \ell \\ s(\ell') & \text{otherwise} \end{cases}}{\langle \ell := v, s \rangle \to \langle v, s' \rangle} \ (\text{set})$$

What did we learn?

Interesting aspects

- Evaluation order of sub-expressions
- Laziness of conditional expressions
- What needs to be in the state

Λλ

Interesting things left out

- Data structures: records, unions, lists
- Language features: recursion, exceptions
- Hard things: Concurrency, input and output

ReactiveX

Programming with observables

Functional reactive programming

Classic functional style

- Functional reactive animations (1990s)
- Composing *behaviours* and *events*
- Revised in the Elm programming style

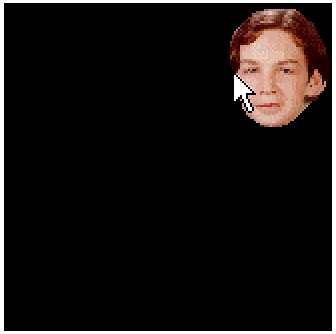
Observables and events

- Events that occur and produce values
- Mouse moves, server notifications, user inputs, ...
- Transformed using a range of *operators*

Functional reactive programming

Reactive animations (Elliott, 1997)

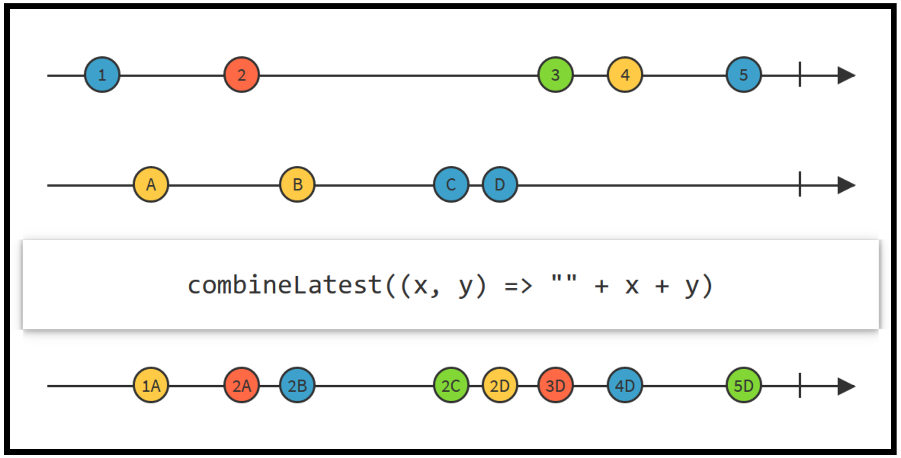
```
followMouseAndDelay u =
 follow `over` later 1 follow
  where
   follow = move (mouseMotion u) jake
```



How does it work

- `mouseMotion` represents current mouse position
- `later` delays time by X seconds
- `over` overlays multiple animations

Reactive eXtensions



Events represented by **`Observable<T>`**

Produces values when something happen

Operators turn one or more observables into a new one

Demo

Programming with RxJS

Semantics

Formalizing observables

Minimal language with events

syntax

$$e ::= x \mid n \mid [n, \ldots, n] \mid e : e \mid \varepsilon ! e \mid \varepsilon \leftarrow \lambda x . e \mid e ; e$$

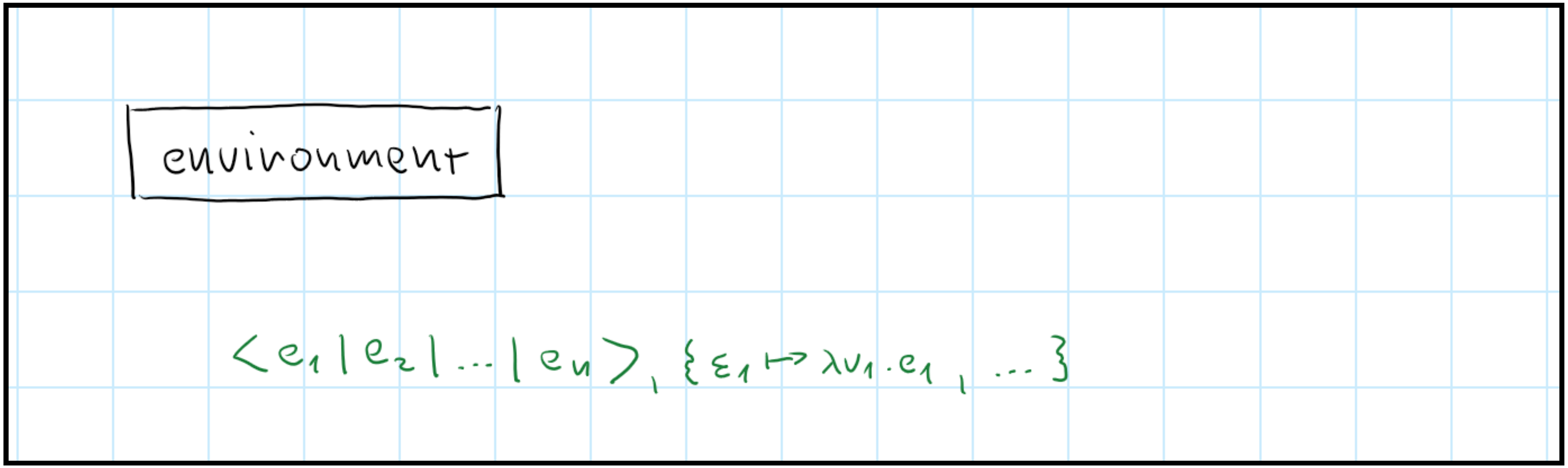
$$v ::= n \mid [n, \ldots, n]$$

$$C[\cdot] ::= C[\cdot] : e \mid v : C[\cdot] \mid \varepsilon ! C[\cdot] \mid C[\cdot] ; e$$

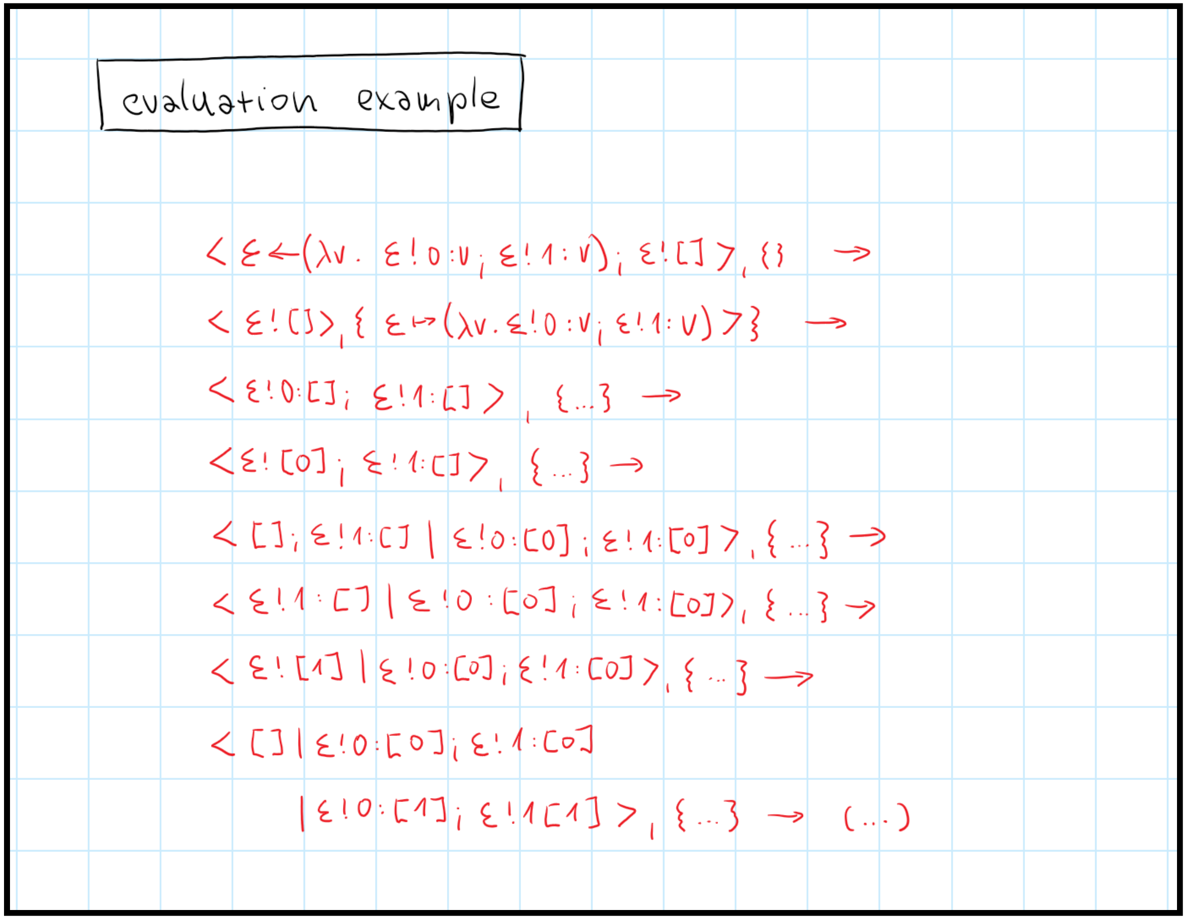
Demo

Lists and sequencing in F#

Modelling concurrency



Triggering events



Lists, sequencing and steps

evaluation rules

$$\frac{}{n : [n_1, \ldots, n_k] \longrightarrow [n, n_1, \ldots, n_k]} \; (\text{cons})$$

$$\frac{}{[\,] ; e \longrightarrow e} \; (\text{seq})$$

$$\frac{e_1 \longrightarrow e_1'}{\langle e_1 \mid \ldots \mid e_n \rangle, H \rightarrow \langle e_1' \mid \ldots \mid e_n \rangle, H} \; (\text{step})$$

$$\frac{}{\langle [\,] \mid e_1 \mid \ldots \mid e_n \rangle, H \rightarrow \langle e_1 \mid \ldots \mid e_n \rangle, H} \; (\text{done})$$

Rules for event handlers

evaluating events

$$\frac{\varepsilon \mapsto \lambda x.e \in H}{\langle C[\varepsilon ! v] \mid \bar{e} \rangle, H \rightarrow \langle C[[\,]] \mid \bar{e} \mid e[x/v] \rangle, H} \quad \text{(trigger)}$$

$$\frac{H' = H \cup \{\varepsilon \mapsto \lambda x.e\}}{\langle C[\varepsilon \leftarrow \lambda x.e], \bar{e} \rangle, H \rightarrow \langle C[[\,]], \bar{e} \rangle, H'} \quad \text{(add)}$$

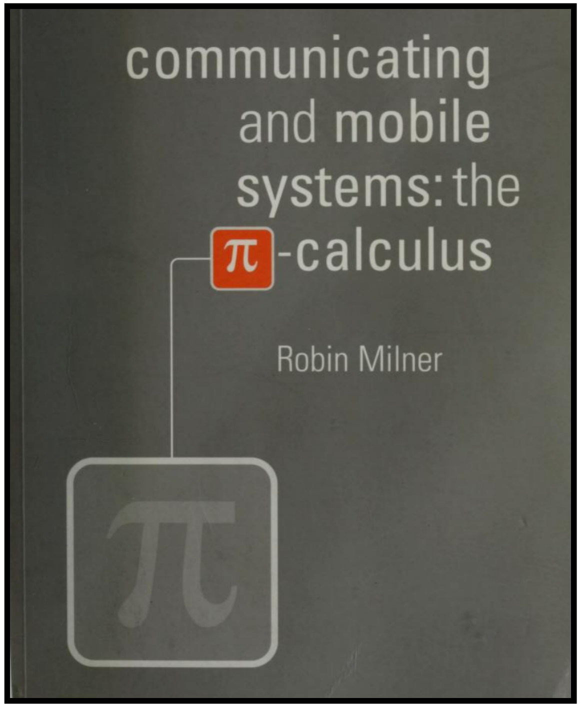
Events calculus

Focus on what matters

- Lists, numbers and events only
- No functions or recursion!
- Probably still Turing-complete

What did we learn

- Sequence of concurrent expressions
- Selection of expression to be run
- Scheduling when event is triggered



Alternative rules

alternatives

$$\frac{\varepsilon \mapsto \lambda x.e \in H}{\langle C[\varepsilon ! v] \mid \bar{e} \rangle, H \rightarrow \langle C[[\,]] \mid \bar{e} \mid e[x/v] \rangle, H} \quad \text{(queue)}$$

$$\frac{\varepsilon \mapsto \lambda x.e \in H}{\langle C[\varepsilon ! v] \mid \bar{e} \rangle, H \rightarrow \langle e[x/v] \mid C[[\,]] \mid \bar{e} \rangle, H} \quad \text{(immediate)}$$

$$\frac{\varepsilon \mapsto \lambda x.e \in H}{\langle \bar{e} \mid C[\varepsilon ! v] \mid \bar{e}' \rangle, H \rightarrow \langle \bar{e} \mid C[[\,]] \mid \bar{e}' \mid e[x/v] \rangle, H} \quad \text{(nondet)}$$

Conclusions

Formal models

Formal models

Figure 1. A typical design process

Useful design guide and for making formal claims

Explains core ideas of a system in a succinct way

The danger is producing languages that look well on paper!

Language semantics types

- **Lambda calculus**
  Logic (1930s) but used for PL semantics (1960s+)
- **Pi calculus, CCS and CSP**
  Models of concurrent systems (1980s-90s)
- **Join calculus**
  Distributed asynchronous programming (1990s)
- **Programming language theory**
  Memory regions, effects and coeffects, locks, etc.

Reading

Null safety in Dart

- Avoiding **`null`** dereferencing with types
- Available at: https://dart.dev/null-safety/understanding-null-safety

Why read this

- Simple useful type system feature!
- Good discussion on soundness
- More languages have this: Swift, Rust, C#, TypeScript

Conclusions

Formal models of programming

- Programming language theory, Part I
- Evaluation over syntactic structures
- Better for small and stateless systems

Tomáš Petříček, 309 (3rd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek
https://d3s.mff.cuni.cz/teaching/nprg075

References (1/2)

Semantics

- Krishnaswami, N. (2021). Semantics of Programming Languages
- Pierce, B. (2002). Types and Programming Languages . MIT
- Pierce, B (ed.) (2004). Advanced Topics in Types and Programming Languages . MIT

History

- Chruch, A. (1941). The Calculi Of Lambda Conversion. Princeton
- McCarthy, J. (1964). A Formal Description of a Subset of ALGOL
- McCarthy, J. (1963). Towards a Mathematical Science of Computation
- Milner, R. et al. (1997). The Definition of Standard ML. MIT

References (2/2)

Reactive

- Elliott, C. (1998). Composing Reactive Animations . MSR
- RxJS Primer - Learn RxJS. Online
- Wan, Z., Hudak, P. (2000). Functional reactive programming from first principles, PLDI

Calculi

- Landing, P. J. (1966). The Next 700 Programming Languages. ACM
- Milner, R. (1986). A Calculus of Communicating Systems. LFCS
- Hoare, C.A.R. (1978). Communicating Sequential Processes. ACM
- Milner, R. (1999). Communicating and mobile systems: The Pi calculus. Cambridge
- Fournet, C., Gonthier, G. (1996). The reflexive CHAM and the join-calculus. POPL