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
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?
Microalgol (1964)

Syntax and semantics of trivial Algol subset

\[ \text{micro}(\pi, \xi) \text{ gives the final state of a program } \pi \text{ run in a state } \xi \]

"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!
// 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"

Generalization and value restriction

ML makes top-level definitions polymorphic

Allowing that for values is unsound!
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

\[ [e] = ? \]

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

\[ [e_2] \cdot [e_1] \]

axiomatic

\[ \{p\} e_1 \{Q\} \quad \{Q\} e_2 \{R\} \]

\[ \{p\} e_1; e_2 \{R\} \]
Language semantics types

Operational - big step

$$e \downarrow v$$

$$\frac{e \downarrow n \quad n' = n + 1}{\text{succ}(e) \downarrow n'}$$

Operational - small step

$$e \rightarrow e'$$

$$\frac{n' = n + 1}{\text{succ}(n) \Rightarrow n'}$$

$$\frac{e \rightarrow e'}{\text{succ}(e) \Rightarrow \text{succ}(e')$$
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**

\[ \begin{align*}
  v &::= n \mid \lambda x. e \\
  e &::= n \mid e + e \mid \lambda x. e \mid x \mid e e
\end{align*} \]

**Evaluation example**

\[ (\lambda x. x + 10) \ (2 + 3) \]
\[ \rightarrow (\lambda x. x + 10) \ 5 \]
\[ \rightarrow (5 + 10) \]
\[ \rightarrow 15 \]
Evaluation rules

\[
\begin{align*}
\text{plus 1:} & \quad n_3 &= n_1 + n_2 \\
& \quad n_1 + n_2 &\rightarrow n_3
\end{align*}
\]

\[
\begin{align*}
\text{app 1:} & \quad e_1 \rightarrow e_1' \\
& \quad e_1 + e_2 &\rightarrow e_1' + e_2
\end{align*}
\]

\[
\begin{align*}
\text{plus 2:} & \quad e_2 \rightarrow e_2' \\
& \quad \lambda x . e &\rightarrow \lambda x . e' \\
& \quad V_1 + e_2 &\rightarrow V_1 + e_2'
\end{align*}
\]

\[
\begin{align*}
\text{app 3:} & \quad (\lambda x . e) v &\rightarrow e[x/v]
\end{align*}
\]
example 1 in detail

\[ \frac{5 = 2 + 3}{2 + 3 \rightarrow 5} \quad \text{(plus)} \]

\[ (\lambda x. x + 10) \ (2 + 3) \rightarrow (\lambda x. x + 10) \ 5 \quad \text{(app\ 2)} \]

\[ (\lambda x. x + 10) \ 5 \rightarrow 5 + 10 \quad \text{(app\ 3)} \]

\[ 15 = 5 + 10 \quad \text{(plus)} \]

\[ \frac{15}{5 + 10 \rightarrow 15} \]
Functions and currying

Example 2 in detail

\[ (\lambda x. \lambda y. x + y) \; 10 \rightarrow (\lambda y. 10 + y) \rightarrow (\lambda y. 10 + y) \; 5 \rightarrow (\lambda y. 10 + y) \; 5 \rightarrow 10 + 5 \rightarrow 15 \]
Simplifying the rules

\[ v := n \mid \lambda x. e \]
\[ e := v \mid x \mid e + e \mid e \cdot e \]
\[ C[e] := t \mid v \cdot C[e] \mid C[e] + e \mid v \cdot C[e] \cdot C[e] \cdot e \]

\[ e \rightarrow e' \]
\[ C[e] \rightarrow C[e'] \quad (c+x) \]
Conditionals and stuck state

\begin{align*}
\text{extensions} \\
\text{e ::= } & \ldots & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
C[\ast] ::= & \ldots & \text{if } C[\ast] \text{ then } e_2 \text{ else } e_3 \\
\end{align*}

\begin{align*}
\text{if } n \neq 0 \text{ then } e_2 \text{ else } e_3 & \rightarrow e_2 \\
\text{if } n = 0 \text{ then } e_2 \text{ else } e_3 & \rightarrow e_3 \\
\end{align*}

\begin{align*}
\text{why types?} \\
\text{if } (\lambda x.x) \text{ then } 1 \text{ else } 2 & \rightarrow \\
\text{stuck!} \\
\end{align*}
Adding references

e := ... \mid \texttt{!l} \mid \texttt{l} := e

C[\cdot] := ... \mid \texttt{l} := C[\cdot]

l \in L
s \in L \rightarrow \mathbb{V}
v \in \mathbb{V}

\langle e, s \rangle \rightarrow \langle e', s \rangle

s(l) = v

\langle \texttt{!l}, s \rangle \rightarrow \langle v, s \rangle \quad \text{(get)}

s'(l') = \begin{cases} v & \text{when } l' = l \\ s(l') & \text{otherwise} \end{cases}

\langle l := v, s \rangle \rightarrow \langle v, s' \rangle \quad \text{(set)}
What did we learn?

Interesting aspects

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

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)

```haskell
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

\[
e ::= \ x \mid n \mid \llbracket n_1, \ldots, n \rrbracket \mid e \cdot e \mid e ! e \mid e \leftarrow x \cdot e \mid e ; e
\]

\[
v ::= \ n \mid \llbracket n_1, \ldots, n \rrbracket
\]

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

Lists and sequencing in F#
Modelling concurrency

\[
\langle e_1, e_2, \ldots, e_n \rangle, \{ e_1 \Rightarrow \lambda v_1. e_1, \ldots \}
\]

environment
Triggering events

evaluation example

\[ < 3 \leftarrow (\lambda v. 3 ! v ; 3 ! v) ; 3 ! C > , \{ \ldots \} \rightarrow \]
\[ < 3 ! C > , \{ \exists v. (\lambda v. 3 ! v ; 3 ! v) \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! C > , \{ \ldots \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! C > , \{ \ldots \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! C | 3 ! 0 ; 3 ! 0 > , \{ \ldots \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! 0 | 3 ! 0 ; 3 ! 0 > , \{ \ldots \} \rightarrow \]
\[ < 3 ! C | 3 ! 0 ; 3 ! 0 > , \{ \ldots \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! 0 | 3 ! 0 ; 3 ! 0 > , \{ \ldots \} \rightarrow \]
\[ < 3 ! 0 ; 3 ! 0 | 3 ! 0 ; 3 ! 0 > , \{ \ldots \} \rightarrow \]
Lists, sequencing and steps

evaluation rules

\[ \text{(cons)} \]
\[ h : [n_1, \ldots, n_k] \rightarrow [n, n_1, \ldots, n_k] \]

\[ \text{(seq)} \]
\[ [\mathcal{J}] e \rightarrow e \]

\[ \text{(step)} \]
\[ e_i \rightarrow e_i' \]
\[ \langle e_1 \mid \ldots \mid e_n \rangle, H \rightarrow \langle e_1' \mid \ldots \mid e_n \rangle, H \]

\[ \text{(dove)} \]
\[ \langle [\mathcal{J}] e_1 \mid \ldots \mid e_n \rangle, H \rightarrow \langle e_1 \mid \ldots \mid e_n \rangle, H \]
Rules for event handlers

\[ e \rightarrow \lambda x. e \in H \]
\[ \langle C[e] \mid \bar{v} \rangle, H \rightarrow \langle C[e] \mid \bar{v} \mid e[x/v] \rangle, H \] (trigger)

\[ H' = H \cup \{ e \rightarrow \lambda x. e \} \]
\[ \langle C[e] \rightleftharpoons \lambda x. e \rangle, H \rightarrow \langle C[e] \rangle, H' \] (odd)
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

\[ \exists x. e \in H \quad (\text{queue}) \]
\[ \langle C e \rangle, H \rightarrow \langle C [c] | e [x/v] \rangle, H \]

\[ \exists x. e \in H \quad (\text{immediate}) \]
\[ \langle C e \rangle, H \rightarrow \langle e [x/v] | C [c] | e \rangle, H \]

\[ \exists x. e \in H \quad (\text{nondet}) \]
\[ \langle e | C e \rangle, H \rightarrow \langle e | C [c] | e [x/v] \rangle, H \]
Conclusions

Formal models
Formal models

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.
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

History

- Church, A. (1941). *The Calculi Of Lambda Conversion*. Princeton
- McCarthy, J. (1964). *A Formal Description of a Subset of ALGOL*
References (2/2)

Reactive

- RxJS Primer - Learn RxJS. Online

Calculi