# Meeting 27 - Procedural Abstraction

Bor-Yuh Evan Chang

Tuesday, December 3, 2024

# Meeting 27 - Procedural Abstraction

IF ANYONE HITS ME WITH A SNOWBALL, I'LL HIT *HIM* WITH 250 SNOWBALLS!
WHAT IF SOMEBODY HITS *YOU* WITH 250 SNOWBALLS?
..SIGHHH..

What questions does your neighbor have?

# Links

- In-Class Slides
- In-Class Jupyter

# Announcements

- Remainder of the Semester
  - HW 5 ~~and Lab 5~~ before Thanksgiving break
  - ~~Unit 6 (probably one combined assignment)~~ Lab 5 after Thanksgiving break
  - Exam 5~~-6~~ in the last week of classes before the Final
- Come see us to make a study plan
  - e.g., via the redo policy
  - see the Final Exam as an opportunity to show growth from mid-semester exams.

# Today

- Procedural Abstraction
  - Lazy Evaluation
  - Mutable State
- Triage Your Questions

# Questions?

- Review:
  - What is the essence of imperative computation?

# Procedures

What are *procedures*?

# Assignment

$$\text{expressions} \quad e \quad ::= \quad \cdots \mid x \mathtt{=} e_1$$

What if we applied substitution as before?

# Static Memory

Without procedure call, dynamically-allocated memory addresses seems overkill.

$$\text{memories} \quad m \quad ::= \quad \cdot \mid m[x \mapsto v]$$

# Procedures: Syntax

$$\begin{array}{rccl}
\text{types} & \tau & ::= & \texttt{number} \mid \texttt{(}x\texttt{:}\ \textbf{var}\ \tau\texttt{)}\ \texttt{=>}\ \tau' \\
\text{values} & v & ::= & n \mid \texttt{(}x\texttt{:}\ \textbf{var}\ \tau\texttt{)}\ \texttt{=>}\ e_1 \\
\text{expressions} & e & ::= & n \mid \texttt{(}x\texttt{:}\ \textbf{var}\ \tau\texttt{)}\ \texttt{=>}\ e_1 \mid x \mid e_1\texttt{(}e_2\texttt{)} \mid x\ \texttt{=}\ e_1 \mid \texttt{*}a
\end{array}$$

Figure 1: Syntax of TypeScripty with number literals, procedure literals, procedure calls, and mutable variable assignment.

# Procedures: Semantics

# Procedures: Implementation

```
defined trait Expr
defined class N
defined class Var
defined class Assign
defined class Deref
defined class A
defined function isValue
defined class Mem
defined object Mem
defined class DoWith
defined object DoWith
import DoWith._
defined function memalloc
defined function substitute
defined function step
```

# Parameter-Passing Modes

Small changes in DoCall.

# Call-By-Name Parameters: Syntax

$$\begin{array}{rrcl}
\text{types} & \tau & ::= & \mathtt{\textbf{number}} \mid \mathtt{(}x\mathtt{:}\ m\,\tau\mathtt{)}\ \mathtt{=>}\ \tau' \\
\text{values} & v & ::= & n \mid \mathtt{(}x\mathtt{:}\ m\,\tau\mathtt{)}\ \mathtt{=>}\ e_1 \\
\text{expressions} & e & ::= & n \mid \mathtt{(}x\mathtt{:}\ m\,\tau\mathtt{)}\ \mathtt{=>}\ e_1 \mid x \mid e_1\mathtt{(}e_2\mathtt{)} \mid m\ x\ \mathtt{=}\ e_1\mathtt{;}\ e_2 \\
\text{parameter modes} & m & ::= & \textbf{const} \mid \textbf{name}
\end{array}$$

Figure 3: Syntax of TypeScripty with number literals, function literals with parameter modes, and variable declarations, and function call expressions.

# Call-By-Name Parameters: Semantics

# Exotic Parameter-Passing Modes

Reference parameters (as in C++ and C#)?

Out parameters (as in C#)?

In-out parameters (as in Ada)?

# Pointers

First-class addresses (i.e., when “addresses are values”).

# Dynamically-Allocated Mutable Objects: Syntax

$$\begin{array}{rcll}
\text{expressions} & e & ::= & n \mid \texttt{\{}\overline{f\texttt{:}\ e}\texttt{\}} \mid e_1 \texttt{ = } e_2 \mid e_1 \texttt{.} f \mid x \mid \textbf{const}\ x \texttt{ = } e_1 \texttt{;}\ e_2 \\
\text{values} & v & ::= & n \mid a \\
\text{location values} & l & ::= & a \texttt{.} f \\
\text{addresses} & a & &
\end{array}$$

Figure 5: Syntax of TypeScripty with number literals and dynamically-allocated mutable objects.

# Dynamically-Allocated Mutable Objects: Semantics

# Dynamically-Allocated Mutable Objects: Semantics

# Aliasing

```
const a = { val: 1 };
const b = a;
b.val = 42;
console.log(a.val)
```