CS571

- Notes 22
- A Language with Context

Context

- All modern languages allow the same name to be used in different contexts
  - Modules
  - Subprograms
  - Blocks
  - Classes
- Pascal (Modula, Ada) allows nested subprograms
- C allows localized declarations
- C++ (Java, C#) allow classes to hide variables
- All these languages declare variables before use

Pascal example

- There are 3 x's and four references
- The arrows go from 'use' to 'mention', or from references to declaration
Locations

- Contexts need to separate where the name is declared (i.e. its scope) from how it is used.
- Locations will anchor the declaration in an environment, which abstracts the idea of scope.
- The meaning of a name is now its location (not just its value).

New Domains

- Location (primitive)
- Operations:
  - first-locn: Location
  - next-locn: Location → Location
  - equal-locn: Location → Location → Tr
  - less-locn: Location → Location → Tr
- Environment: Id → (Location + Errorvalue)
- Operations:
  - emptyenv: Environment
  - accessenv: Id → Environment → (Location + Errorvalue)
  - updateenv: Id → Location → Environment → Environment

Store

- The store now becomes a mapping from locations to values.
- Store = Location → Nat
- Operations:
  - newstore: Store
  - access: Location → Store → Nat
  - update: Location → Nat → Store → Store
- Expressions and commands are now evaluated with a store and an environment.
- Declarations are executed in an environment.
- Everything is contextualized.
A language with context

P ::= Program
K ::= Block
D ::= Declaration
C ::= Command
E ::= Expression
B ::= Boolean-expr
Id ::= Identifier
N ::= Numeral

P ::= K.
P ::= K.
P ::= K.
P ::= K.

K ::= begin D; C end
K ::= begin D; C end
K ::= begin D; C end
K ::= begin D; C end

D ::= D1; D2 | var Id |
D ::= D1; D2 | var Id |
D ::= D1; D2 | var Id |
D ::= D1; D2 | var Id |

C ::= C1; C2 | Id := E |
C ::= C1; C2 | Id := E |
C ::= C1; C2 | Id := E |
C ::= C1; C2 | Id := E |

E ::= E1 + E2 | Id | N
E ::= E1 + E2 | Id | N
E ::= E1 + E2 | Id | N
E ::= E1 + E2 | Id | N

Semantic domains

- Domain Nat (definition as before)
- Domain Tr (definition as before)
- Domain Id (definition as before)
- Domain Location (definition above)
- Domain Expressible-value = Nat + Error-value
- Domain Denotable-value = Location + Nat + Error-value
- Domain Environment = (Id → Denotable-value) × Location
- Domain Storable-value = Nat
- Domain Store = Location → Storable-value
- Domain Poststore = OK + Err

Environment operations

- emptyenv: Location → Environment
  emptyenv = λl.((l, inErrorvalue(l)), l)
- accessenv: Id → Environment → Denotable-value
  accessenv = λlλl.map(l), map(l)
- updateenv: Id → Denotable-value → Environment → Environment
  updateenv = λlλl.map(l), map(l), l → d.map, map(l)
- reserve-locn: Environment → (Location × Environment)
  reserve-locn = λl.map, l, l.map, next-locn(l)

We can also store named constants in the environment
To allow declarations to use successive locations
We need to reserve a location (cf. malloc, new) before creating a variable binding
### Poststore operations

- **return**: Store $\rightarrow$ Poststore
  \[
  \text{return} = \lambda s. \text{inOK}(s)
  \]
- **signalerr**: Store $\rightarrow$ Poststore
  \[
  \text{signalerr} = \lambda s. \text{inErr}(s)
  \]
- **check**: (Store $\rightarrow$ Poststore) $\rightarrow$
  \[
  \text{(Poststore) } \rightarrow \text{(Poststore)}
  \]
  \[
  \text{check } f = \lambda p. \text{cases } p \text{ of}
  \]
  \[
  \text{isOK}(s) \rightarrow (f s)
  \]
  \[
  \text{isErr}(s) \rightarrow p
  \]
  \[
  \text{end}
  \]

### Valuation functions

- **P**: Location $\rightarrow$ Store $\rightarrow$ Poststore
  \[
  P[K] = 1d(K[empty])
  \]
- **K**: Block $\rightarrow$ Environment $\rightarrow$ Store $\rightarrow$ Poststore
  \[
  K[\text{begin } D; C \text{ end}] = \lambda e. C[D][D[e]]
  \]
- **D**: Declaration $\rightarrow$ Environment $\rightarrow$ Environment
  \[
  D[D_1; D_2] = D[D_1]; D[D_2]
  \]
  \[
  D[\text{const } I \rightarrow N] = \text{updateenv}[I \text{ inVar}(N[N])]
  \]
  \[
  D[\text{var } I] = \lambda e. \text{let } [I^e'] = (\text{reserve } - \text{locn } e) \text{ in}
  \]
  \[
  \text{updateenv}[I \text{ inLocation}(I^e')]
  \]

### Commands

- Use the same environment for a sequence of commands
  \[
  C[C_1; C_2] = \lambda e. (\text{check } C[C_2][e] \circ (C[C_1][e])
  \]
- A command can fail because there is an undeclared variable (or constant)
Assignment

- Either if the variable is undeclared or there is an error in the expression, an error store is returned

\[
C[I := E] = \lambda e. \lambda x. \text{cases } accsenv[I] e \text{ of}
\]

- isLocation(l) \rightarrow (\text{cases } E[l] e \text{ of}
  \begin{align*}
  &\text{isNat}(x) \rightarrow (\text{return } (\text{update } l n s)) \\
  &\text{isErrVal}(e) \rightarrow (\text{signalerr } s) \\
  \end{align*}
\]
- end

- if \text{isNat}(x) \rightarrow (\text{signalerr } s)

- if \text{isErrVal}(e) \rightarrow (\text{signalerr } s)

The while loop

- Use fixed point semantics

\[
C[\text{while } B \text{ do } C] =
\]

\[
\lambda e. \text{fix}(\lambda x. \text{cases } B[l] e \text{ of}
  \begin{align*}
  &\text{isTr}(t) \rightarrow (t \rightarrow (\text{check } f) \rightarrow (C[C[l] e] \text{return } s)) \\
  &\text{isErrVal}(e) \rightarrow (\text{signalerr } s) \\
  \end{align*}
\]
- end

Expressions

- Check for errors in each subexpression

\[
E : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Store} \rightarrow \text{Expressible} \rightarrow \text{value}
\]

\[
E[E_1 + E_2] = \lambda e_1 e_2. \text{cases } E[E_1 e_2] e \text{ of}
\]

- isNat(n_1) \rightarrow (\text{cases } E[E_1 e_2] e \text{ of}
  \begin{align*}
  &\text{isNat}(n_1) \rightarrow \text{inNat } (n_1, \text{plus } n_2) \\
  &\text{isErrVal}(e) \rightarrow \text{inErrVal}(e) \\
  \end{align*}
\]
- end

- if \text{isErrVal}(e) \rightarrow \text{inErrVal}(e)

- end
Accessing the environment

- Check for constants or variables – return error if undeclared
  \[ E[I] = \lambda e. \lambda s. \text{cases } (accessenv[I][e]) \text{ of} \]
  \[ \text{isNat}(n) \rightarrow \text{inNat}(n) \]
  \[ \text{isLocation}(l) \rightarrow \text{inVar}(access[l]) \]
  \[ \text{isErrorvalue}() \rightarrow \text{inErrorvalue}() \]
- If a variable is unused, access will return zero

A sample derivation

The derivation started
Declaring a variable

\[ D[\text{\texttt{var}} X] = \lambda e. (\text{\texttt{let}} \{ r, e' \} = (\text{\texttt{reserve-locn}} e) \text{\texttt{in}} \text{\texttt{updateenv}} X \text{\texttt{inLocation}} r e') \text{\texttt{in}} e \]  

\[ e_1 = \left[ \left[ X \mapsto l_0 \right] \right] \text{\texttt{fat}}(e_2), l_1 \]

Executing the block body

\[ C[C_0] = e_1 \]

\[ (\text{\texttt{check}} C[C; C_0] e_1) \circ (C[C_0] e_1) = \]

\[ (\text{\texttt{check}} ( (\text{\texttt{check}} C[C] e_1) \circ (C[C] e_1)) \circ (C[C] e_1)) = \]

\[ (\text{\texttt{check}} C[C] e_1) \circ (\text{\texttt{check}} C[C] e_1) \circ (C[C] e_1) \]

Executing C₁

\[ C[X := A + 2] = \lambda e. \text{\texttt{let}} \{ r, e' \} = (\text{\texttt{isLocation}} r) \rightarrow (\text{\texttt{caseN}}} A + 2 e, x \text{ of} \) \]

\[ \text{\texttt{isNat}(x) \rightarrow (\text{\texttt{return}} \text{\texttt{update}} l x)) \]

\[ \text{\texttt{inErrorValue}} \rightarrow (\text{\texttt{signalErr}} x) \]  

\[ \text{\texttt{end}} \]

\[ \text{\texttt{inErrorValue}} \rightarrow (\text{\texttt{signalErr}} x) \]

\[ \text{\texttt{end}} \]

\[ \text{\texttt{returns}} \text{\texttt{inNat}(three)} \]

\[ l_2 \]

\[ \text{\texttt{returns}} \text{\texttt{inNat}(three)} \]
The Nested Block

\[(\text{check } C[\mathcal{C}_1][e_1]) \circ (C[\text{begin } A; \mathcal{C}_2\text{end}][e_1])\]

\[C[\mathcal{C}_2][D[\text{var } A][e_1]]\]

e_0, with l, bound to A

---

The loop

\[C[\text{while } X = 0 \text{ do } A := X][e_1] \equiv \text{fix}(\lambda x. \text{cases } X [\lambda x. X][e_1] \text{ of}\]

\[
\begin{array}{l}
\text{if } l = 0 \text{ (access } l, x) \text{ equals zero } \rightarrow \text{ (check } f) \to C[A := X][e_1] \to \text{return } (x) \\
\text{else } (\text{check } f) \to C[A := X][e_1] \to \text{return } (x)
\end{array}
\]

---

Executing C₃ in environment e₃

\[C[X := A][e_3] \equiv \text{fix}(\lambda x. \text{cases } X [\lambda x. X][e_1] \text{ of}\]

\[
\begin{array}{l}
\text{if } l = 0 \text{ (access } l, x) \text{ equals zero } \rightarrow \text{ (check } f) \to C[A := X][e_1] \to \text{return } (x) \\
\text{else } (\text{check } f) \to C[A := X][e_1] \to \text{return } (x)
\end{array}
\]
Assembling the three partial results

\[
\begin{align*}
\lambda l. & \begin{cases}
\text{check (check } \lambda s. \text{return (update } l \text{ one } s) & = \\
\text{fix } \left( \lambda f. (\text{access } l \text{ s equals zero } \rightarrow (\text{check } f) = \right) \\
\text{(update next - locn } l (\text{access } l \text{ s s } \uplus \text{return }) = \\
\lambda s. \text{return (update } l \text{ three } s) & \end{cases}
\end{align*}
\]

A more intuitive form using \( f \cdot g = g ! f \)

\[
\lambda l. (\lambda s. \text{return (update } l \text{ one } s)) = C1
\]

\[
\begin{cases}
\text{check } \left( \lambda f. (\text{access } l \text{ s equals zero } \rightarrow (\text{check } f) = \right) \\
\text{(update next - locn } l (\text{access } l \text{ s s } \uplus \text{return }) = \\
\lambda s. \text{return (update } l \text{ three } s) & \end{cases}
\]

All mention of environments have disappeared leaving a function of the initial location and store