Languages with Contexts I: A Block-Structured Language

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Languages with Contexts

- In any language, the context of a phrase influences its meaning.
- In programming languages, contexts attribute meanings to identifiers.
- Does the store establish the context?

```
begin
    integer X; integer Y
    Y:=0
    X:=Y
    Y:=1
    X:=Y+1
end
```

- But the store constantly changes within the block!
- *Declarations* establish block context.
- Commands operate *within* that context.

Languages with Contexts

```
begin integer X
    X:=0
    begin real X
        X:=1.5
    end
    X:=X+1
end
```

- Outer X denotes *integer* object.
- Inner X denotes *real* object.
- X is the name used for both objects.
- Scope rules are needed to resolve ambiguities.

Meaning of an identifier can't be just its storable value!

Contexts

- "Objects" are computer store locations.
- The meaning of an identifier is the location bound to it.
- A context is the set of identifier/store location pairs accessible at a textual position.
- Each position in a program resides within a unique context.

Context of a phrase can be determined without *running the program.*

Environments

Mathematical value that models context.

1. Environment establishes context for syntactic phrase

   Resolution of ambiguities concerning meaning of identifiers.

2. As many environment values as distinct program contexts.

   Multiple environments are maintained during program evaluation.

3. Environment is a static object.

   Phrases uses the same environment each time it is evaluated.

- Simple model: store = environment
  - Only one environment.
  - Used in previous program examples.
- Complex model: store + environments.
  - One store.
  - Multiple environments.

Compiler's Symbol Table

Real-life example of an environment

- Used for translation of source program into compiled code.
- One entry for each identifier in the program
  - Data type.
  - Mode of usage (variable, constant, parameter).
  - Relative location in run-time store.
- Resolution of name conflicts
  1. Different symbol table for each block.
  2. Build table as single stack

     Incremented and decremented upon block entry and exit.
- Compile-time object (Pascal, C, C++),
- Run-time object (Lisp, Smalltalk).

Static and Dynamic Semantics

- Static semantics
  - Part of semantics definition that use environment to resolve context questions.
  - Type-checking, scope resolution, storage calculations.
- Dynamic semantics
  - "Real" production of meanings.
  - Code generation and execution.

No clear separation.

Evaluation Functions

- Environments used as additional argument

  C: Command $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ $Store_{\perp}$

- Environment domain

  Environment $=$ *Identifier* $\rightarrow$ *Denotable-value*

Language Features

1. Declarations.
2. Block structures.
3. Scoping mechanisms.
4. Compound data structures.

A Block-Structured Language

See Figures 7.1 and 7.2

- Composition of commands

  $\mathbf{C}[[C_1;\ C_2]] = \lambda e.(\mathbf{C}[[C_2]]\ e) \circ (\mathbf{C}[[C_1]]\ e)$

- Both C_1 and C_2 are evaluated in e.
- C_1 may contain local declarations.
- Environments created within C_1 do not affect C_2.
- *Static scoping*
  - Context of phrase solely determined by its textual position.
  - Identifier declared within block only referenced by commands within that block.
  - Straight-forward management of storage locations.

Strongly Typed Languages

- Environment processing can proceed independently of store processing.
- **P**[[P]] can be simplified without value of initial base location t and initial store s.
- Result neither contains occurences of environment arguments nor checking of denotable and expressible value tags.
- Simplifications correspond to declaration and type-checking actions in a compiler.

Example

See Figure 7.3

λl.(λs. return (update l (one plus two) s))
! (check (fix (λf. λs.
    ((access l s) equals zero $\rightarrow$
        (λs. return(update (next-locn l)
            (access l s) s)) ! (check f)
        [] return
) s)))
! (check (λs. return (update l one s)))

$(f!g := g \circ f)$

Resembles series of machine code instructions parameterized on the store's base address l.

Stack-Managed Storage

- Store of block-structured languages always used in a stack-like fashion
  - Locations are bound to identifiers sequentially using 'next-locn'.
  - Location bound to identifier in local block are freed for re-use when block is exited.
- Storage reuse automatically in **C**[[C_1; C_2]]
  - Locations bound to identifiers in [[C_1]] are reserved by environment built from e for **C**[[C_1]].
  - **C**[[C_2]] uses original e (and original location marker) effectively deallocating these locations.

Significant characteristics of block-structured languages!

Stack-Managed Storage

- Make storage calculations explicit in *Store* algebra
  - Store = (Location $\rightarrow$ Storable-value) $\times$ Location
  - First component is data space of the stack
  - Second component indicates "top of stack"
  - 'allocate-locn' becomes the run-time version of 'reserve-locn'
- Environment domain is freed from storage management
  - *Environment* = *Id* $\rightarrow$ *Denotable-value*
  - 'reserve-locn' is dropped.

Processing of declarations requires store as well as environment.

Stack-Managed Store

- Declarations

D: Declaration $\rightarrow$ Environment $\rightarrow$ Store $\rightarrow$ (Environment $\times$ Poststore)

D[[**var** I]] =

$\lambda e.\lambda s.$ let $(l, p) =$ (allocate-locn s)
in ((updateenv [[I]] in*Location*(l) e), p)

D[[D$_1$; D$_2$]] =

$\lambda e.\lambda s.$ let $(e', p) =$ (**D**[[D$_1$]]e s)
in (*check* **D**[[D$_2$]]e')(p)

- Blocks

K[[**begin** D;C **end**]] = $\lambda e.\lambda s.$
let l = mark-locn in
let $(e', p) =$ **D**[[D]]e s in
let p' = (check (**C**[[C]]e'))(p)
in (check (deallocate-locns l))(p')

Environment beckomes run-time object because binding of location values to identifiers depends on run-time store.

The Meaning of Identifiers

- Assignment X := X+1
  - Left-hand side value of X is location,
  - Right-hand side value is storable value associated to location.
- Context problem occurs even at primitive command level!
- Variable identifier denotes *pair* of values
  - I[[I]] = (*L-value*, *R-value*)
  - L-value is kept in environment.
  - R-value is kept in store.

The Meaning of Identifiers

- Valuation **I**: *Id* $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *(Location* $\times$ *Storable-value)*
  - **L**: Id $\rightarrow$ Environment $\rightarrow$ Store $\rightarrow$ Location.
  - **R**: Id $\rightarrow$ Environment $\rightarrow$ Store $\rightarrow$ Storable-value.
- **L**[[I]] = *accessenv* [[I]]
- **R**[[I]] = *access* $\circ$ *accessenv* [[I]]
- Semantic equations with variables

  E[[I]] = **R**[[I]]
  C[[I:=E]] = $\lambda e.\lambda s.$
  return(*update*(**L**[[I]]e) (**E**[[E]]e s) s)

The Meaning of Identifiers

- Other view: R-value = Function(L-value)
  - "True" meaning is L-value.
  - "Coercion" on right-hand side of assignment
- Coercion = dereferencing.

  J: Id $\rightarrow$ *Environment* $\rightarrow$ *Denotable-value*
  J[[I]] $= \lambda e.$ (*accessenv* [[I]] e)

  C[[I:=E]] $= \lambda e.\lambda s.$
  return(*update* (**J**[[I]]e) (**E**[[E]]$e\ s$) s)
  E[[I]] $= \lambda e.\lambda s.$ *access* (**J**[[I]]e) s

- Some system languages (BCPL) require explicit dereferencing operator (X=@X+1)

  E[[I]] $= \lambda e.\lambda s.$ in*Location*(**J**[[I]]e)
  E[[@E]] $= \lambda e.\lambda s.$ cases (**E**[[E]]$e\ s$) of
  is*Location*(l) $\rightarrow$ (*access* $l\ s$)
  [] . . . end