Notes on Programming Languages

Cheoljoo Jeong

1 Elements of Programming Languages

1.1 Notations for Expressions

- Infix, prefix, postfix notations for binary operators
- An expression in *prefix notation* is written as follows:
  (a) The prefix notation for a constant or variable is the constant or variable itself.
  (b) The application of an operator **op** to subexpressions E_1 and E_2 is written in prefix notation as **op** $E_1 E_2$.
- When two different operators share a operand in an expression, which will take that operand is determined by the **precedence relation** between the two operators.
  - The operator with higher precedence take the operand.
- An *operator* is said to be **left associative** if subexpressions containing multiple occurrences of this operator are grouped from left to right.

1.2 Evaluation of Expressions

- An expression E_1 **op** E_2 is evaluated as follows:
  (a) Evaluate the subexpression E_1 and E_2 in some order.
  (b) Apply the operator **op** to the resulting values of E_1 and E_2.
- Expression evaluation corresponds to **tree rewriting**.
- **Stack implementation of expression evaluation**
  (a) Translate the expression to be evaluated into *postfix notation*.
  (b) Scan the postfix notation from left to right
    (b.1) On seeing a constant, push it onto the stack.
    (b.2) On seeing a binary operator, pop two values from the top of the stack, apply the operator to the values, and push the result back onto the stack.
  (c) After the entire postfix notation is scanned, the value of the expression is on the top of the stack.

1.3 Function Declarations and Applications

- A function in a programming language comes together with an algorithm for computing the value of the function at each element of its domain.
- Under the **innermost-evaluation** rule, a function application

  $$\langle \mathit{name} \rangle(\langle \mathit{actual\text{-}parameters} \rangle)$$

  is computed as follows:

  (a) Evaluate the expressions in ⟨*actual-parameters*⟩; (**call-by-value** evaluation)

  (b) Substitute the results for the formals in the function body;

  (c) Evaluate the body;

  (d) Return its value as the answer;

- Each evaluation of a function body is called an **activation** of the function.

Selective evaluation

- In **if** ⟨*cond*⟩ **then** E_1 **else** E_2, only one of E_1 and E_2 is ever evaluated depending on the value of ⟨*cond*⟩.
- The operators **andalso** and **orelse** perform **short-circuit** evaluation of boolean expressions, in which the 'right' operand is evaluated only if it has to be.

1.4 Recursive Functions

- The definition of a function f is said to be **linear-recursive** if an activation $f(a)$ of f can initiate at most one new activation of f.
- Evaluation of a linear-recursive function has two phases:
  - a **winding phase** in which new activations are activated, and
  - a subsequent **unwinding phase** in which control returns from the activations in a LIFO manner.
- A function f is **tail recursive** if it either returns a value without needing recursion[1], or it simply returns the result of a recursive activation.
- All the work of a linear tail-recursive function is done in the *winding phase*, as new activations are initiated. The unwinding phase is trivial because the value computed by the final activation becomes the result of the entire evaluation.
- ***A linear tail-recursive function can be turned into a loop.***
  - Linear tail-recursive factorial program:

    ```
    fun g(n, a) = if n = 0 then a else g(n-1, n*a);
    ```

  - Recursion-free loop-version of g:

    ```
    loop
       if n = 0 return a;
       else a := n*a; n := n-1;
    end
    ```

[1] recursive process [1]

1.5 Lexical Scopes and Regions

- **Lexical scope rules** use the program *text* surrounding a function declaration to determine the context in which nonlocal names are evaluated.
  - The program text is static by contrast with run-time execution, so lexical scope rules are also called **static scope rules**.
- The **region** (or **block**) of a *variable declaration* is the portion of text within which the declaration is effective.
  - Blocks may be *nested*.
- The **scope** of a *variable declaration* is the text within which references to the variable refer to the declaration.
  - We may speak of the "declarations that are *visible* at the point of a variable reference"
  - The declaration of a variable v has a scope that includes all references to v that *occur free* in the region associated with the declaration.
  - That is, *the scope of a declaration is the region of text associated with the declaration, 'excluding' any inner regions associated with declarations that use the same variable name.*
- In most languages, a declaration's region and scope can be determined statically. These languages are called to be **statically scoped**.
- **The only mechanism for introducing regions in programming languages is λ-abstraction**:

  $$\texttt{let}\ x_1 = N_1, \cdots, x_n = N_n\ \texttt{in}\ M$$

  can be decoded into a λ-expression

  $$(\lambda x_1 \cdots x_n.M)N_1 \cdots N_n$$

  where ***the region of*** x_i ***is*** M.
  - The `letrec` expression is special: in

    $$\texttt{letrec}\ x_1 = N_1, \cdots, x_n = N_n\ \texttt{in}\ M$$

    the ***region of a variable*** x_i ***is not*** M ***but the "`letrec` expression itself"*** and can be encoded into

    $$\texttt{let}\ x_1 = Y(\lambda x_1.N_1), \cdots, x_n = Y(\lambda x_n.N_n)\ \texttt{in}\ M$$

  - The `letrec` problem is due to the fact that "$L \equiv (\texttt{let}\ x = N\ \texttt{in}\ M)$" only binds x in M not in L.

1.6 Dynamic Scope and Dynamic Assignment

- Example of dynamic scope:

  $$\begin{tikzcd} & B \arrow[dr, "\text{calls}"] & \\ A \arrow[ur, "\text{calls}"] & & C \end{tikzcd}$$

  - Function A binds the variable `foo`.
  - Function C uses the variable `foo`.

1.7 Types

- The **type** of an expression tells us the values it can denote and the operations that can be applied to it.
- The widely accepted principle of language design is that ***every expression must have a unique type***.
  - This principle makes types a mechanism for classifying expressions.
  - Variations:
    1. Overloading
    2. Coercion
    3. Parametric polymorphism
- A **type system** for a language is a set of rules for associating a type with expressions in the language.
  - A type system **rejects** an expression if it cannot associate a type with the expression.
- The rules of type system specify the proper usage of each operator in the language.
- A program that exectues without type errors is said to be **type safe**.
- **Static type checking** cannot check some properties that depend on values computed at run-time such as:
  - *division by zero*
  - *array indices being within bounds*
- **Dynamic type checking** is done during program execution.
  - This is usually done by *inserting extra code* into the program to watch for impending errors.
  - The serious problem of dynamic checking is that *errors can lurk in a program until they are reached during execution.*
- A type system is **strong** if it accepts only safe expressions.
  - Expressions that are accepted by a strong type system are guaranteed to evaluate without type error.
- Let P be the set of all programs and T be the set of type-safe programs. And let S be the set of programs accepted by a strong type system and W be the set of programs accepted by a weak type system.
  - *Strong type systems accept only subset of* T. I.e.,

    $$S \subseteq T.$$

    The smaller $T \setminus S$ is, the more powerful the type system is.
  - *Weak type systems may accept non-type-safe programs*. I.e.,

    $$W \setminus T \neq \emptyset.$$

2 Imperative Programming Languages

2.1 Programming with Assignments

- Characteristic properties of **imperative programming languages**:
  - (a) **Assignments**: Variables denote *locations* in an underlying machine.
  - (b) **Mutable date structures**: A data structure is **mutable** if it has components whose values can be changed by assignments.
  - (c) **Control flow semantics**: The flow of control through a progarm is specified by constructs called **statements**

2.2 The Effect of An Assignment

- A characteristic property of an assignment is that it **changes a value** held inside a machine.
- An assignment changes the *state* of the machine, where **state** corresponds roughly to a snapshot of the machine's memory.
- The distinction between a location and its contents can be clarified usign the neutral terms l**-value** for a location and r**-value** for a value that can be held in a location.
- *A dynamic computation can be visualized as a thread laid down by the flow of control through the static program text*
  - Let **points** exist before the first instruction, between any two adjacent instructions, and after the last instruction.
  - The **thread** of computation consists of sequence of program texts that are reached as control flows through the program text.
- The effect of computation thread on a RAM is described by taking snapshots, called *states*; a **state** has three parts:
  - (a) a mapping from locations to values
  - (b) the remaining input sequence
  - (c) the output sequence produced so far.
- *Assignment instructions, I/O instructions*, and *control-flow instructions* are among the most important instruction-classes in imperative programming languages.
  - ***Assignment and I/O instructions change the state without interfering the normal flow of control.***
  - ***Control-flow instructions direct the thread without changing the state.***

2.3 Procedure Activations

- A **procedure declaration** has four parts:
  - (a) the *name* of the declared procedure,

(b) the *formal parameter* of the procedure,

(c) a *body* consisting of local declarations and a statement list, and

(d) an *optional result type*.

- A declaration of a name is also called a **binding** of the name; it introduces a new use of the name.
- The treatment of parameters in procedure calls depends on whether the occurence of x in the body refers
  1. to the name itself,
  2. to its l-value, or
  3. to its r-value.

Environments and stores

- An **environment** maps a variable name to an l-value.
- A **store** maps an l-value to its contents[2].

$$\text{variable} \xrightarrow{\text{environment}} l\text{-value} \xrightarrow{\text{store}} r\text{-value}$$

Scope rules

- *The* ***lexical environment*** *of a procedure is the environment in which the procedure body appears.*
- *A* ***calling environment*** *is the environment at a point of call of the procedure.*
- *In the* ***lexical scope rule****, the nonlocals in a procedure body refer to their values in the lexical environment.*
- *In the* ***dynamic scope rule****, the nonlocals in a procedure body refer to their values in the calling environment.*

Lifetime of local variables

- In principle, local variables are local to a procedure activation.
- This indicates that local variables are located in the **stack** and deallocated when the activation ends.
- Some languages allow local variables to be existent after the end of the activation; the memory elements for this variables are allocated at **heap** (a.k.a **garbage-collected memory**).

[2]Note that the two notions of environments and stores come from the fact that the language in concern is an 'imperative language.' There are no notions of l-values or r-values in purely functional languages, where there are no assignments.

2.4 Parameter Passing

- Parameter passing determines the *correspondence betweent the actual parameters in a procedure call and the formal parameters in the procedure body*.
- Given a procedure call $P(A[j])$, there are four types of parameter passing:
  (a) **Call-by-value**: Pass the r-value of $A[j]$;
  (b) **Call-by-reference**: Pass the l-value of $A[j]$;
  (c) **Call-by-name**: Pass the text $A[j]$ itself, avoiding "variable capture"
  (d) **Call-by-value-result** (a.k.a. **copy-in/copy-out**):
    (a) *Copy-in phase*: Both the r-values and l-values of the actual parameters are computed; The r-values are assigned to the corresponding formals, as in call-by-value, and te l-values are saved for the copy-out phase.
    (b) *Copy-out phase*: After the procedure body is executed, the final values of formals are copied back out to the l-values computed in the copy-in phase.
- Notice the difference between the **call-by-value evaluation** and **call-by-value parameter passing**.
- Parameter passing in programming languages:
  - C uses call-by-value.
  - Pascal uses call-by-value, but it also supports call-by-reference by the keyword **var**.
  - Ada supports three kinds of parameters:
    * **in** parameters, corresponding to value parameters
    * **out** parameters, corresponding to just the copy-out phase of call-by-value-result, and
    * **in out** parameters, corresponding to either reference parameters or value-result parameters, at the discretion of the implementation.

Parameter Passing Examples

- **Call-by-value**:
  - C procedure `swap1`:

```
void swap1(int x, int y) {
   int z;
   z = x; x = y; y = z;
}
```

  - A call `swap1(a, b)` does nothing to `a` and `b`.
  - Effect of `swap1(a, b)`:

```
x = a;
y = b;
z = x; x = y; y = z;
```

  - This problem can be remedied in C by passing l-values as in:

```
void swap(int *px, int *py) {
   int z;
   z = *px; *px = *py; *py = z;
}
```

  * Note that `swap` is invoked following a call-by-value method, since the actual parameters are l-**values**.

- **Call-by-reference**:
  - Modula-2 procedure `P`:

```
procedure P(x: xType; var y: yType);
...
end P;
```

  - `x` is a value parameter and `y` is a reference parameter.
  - Effect of the call `P(a + b, c)`:
    1. Assign `x` the r-value of `a + b`.
    2. Make the l-value of reference parameter `y` the same as that of `c`.
    3. Execute the body of procedure `P`.
  - Modula-2 version of `swap`:

```
procedure swap(var x : integer; var y : integer);
var z : integer
begin
   z := x; x := y; y := z;
end swap;
```

- **Call-by-value-result**:
  - Call-by-value-result can result in anomalies in case of **aliases**. Refer to [5, page 130] for details.

2.5 Activations Have Nested Lifetimes

- The **lifetime** of an activation begins when control enters the activation and ends when control returns from the activation.
  - When P calls Q, the lifetime of Q is nested within the lifetime of P.
- The flow of control between activations can be depicted by a tree, called an **activation tree**.
  - Nodes in a tree represent activations.

2.6 Lexical Scope in C

- Data needed for an activation of a procedure is collected in a record called an **activation record** or **frame**.
  - Since control flows between activations in a stack-like manner, a *Stack* can be used to hold frames.
  - For this reason, frames are sometimes referred to as *stack frames*.

- C does not allow procedure bodies to be nested, so stack-frame management for C is simpler than for Modula-2.
- **Compound statement** construct in C:

$$\{\langle declarations \rangle \ \langle statements \rangle \ \}$$

- A redeclaration of x creates a **hole** in the scope of any outer bindings of x. E.g.,

```
int x;
for (...) {
   int x;
   ...
}
...
```

- **Storage for local variables** in C
  - A variable declared in a compound statement is local to an execution of the statement.
  - *C compilers tend to allocate storage for all the variables in a procedure all at once when the procedure is called.*

Procedure call and return in C

- C uses call-by-value, so the **caller** evaluates the actual parameters for the call and places their values in the activation record for the **callee**.
- Information needed to restart execution of the **caller** is saved: this includes *return address*.
- The **callee** allocates space for its local variables.
  - Also temporary storage for compiler-generated variables are allocated.
- The body of the **callee** is executed.
- Control returns to the **caller**.

Tail-recursion elimination

- When *the last 'statement' executed in the body of a procedure P is a recursive call*, the call is said to be **tail recursive**.
- Tail-recursion elimination:
  - A tail-recursive call $P(a, b)$ of a procedure P with formals x and y can be replaced by

```
x = a;
y = b;
goto the 1st executable statement in P;
```

2.7 Block Structure in Modula-2

- A **block** consists of a sequence of declarations, including procedure declarations, and a sequence of statements.
- A language is said to be **block-structured** if it allows blocks to be nested.

Access to nonlocals: control and access links

- Memory category:
  1. **code**
  2. **static global data**
  3. **run-time stack** including *static local data*
  4. **heap**: garbage-collected memory
- What's the difference between C and Modula-2?
  - Modula-2 is block-structured but C is not!
  - **C**: A ***nonlocal*** refers to a location in 'static global data' area.
  - **Modula-2**: A ***nonlocal*** refers to a location in some other ***activation record*** in 'run-time stack' area.
- The, what other activation record does nonlocal refer to?
  - **Control link** (or **dynamic link**) points to the activation record of the run-time caller.
  - **Access link** (or **static link**) points to the most recent activation of the lexically enclosing block.

Procedures as parameters

- *A procedure that is passed as a parameter carries its lexical environment along with it.*
  - In other words, when a procedure X is passed as a parameter, an access link a goes with it.
  - Later, when X is called, a is used as the access link for its block.

Displays in the absence of procedures as parameters

- Displays are an optimization technique for obtaining faster access to nonlocals.
- A **display** is an array d of pointers to activation records, indexed by nesting depth.
  - An array element $d[i]$ is maintained so that it points to the most recent activation of the block at nesting depth i.
- With a display, a nonlocal n can be found as follows:
  1. Use one array access to find the activation record containing n.
  2. Use the relative address within the activation record to find the l-value for n.
- The calling sequence for maintaining the display is
  1. Save $d[i]$ in the activation record of the callee at nesting depth i.
  2. Make $d[i]$ point to the callee.

3 Object-Oriented Programming Languages

3.1 Objects, Classes, Object Types

- An **object** is a collection of *data* and *codes*.
  - Data are called **instance variables** or **fields**.
  - Codes are called **methods**.
  - Data and codes altogether are called **attributes**.
- An **object type** describes the 'shape' of a collection of objects.
  - Sometimes, an object type is called an **interface**.
  - A **object protocol** is the type signature for the attributes of an object.
- A **class**
- A taxonomy of object-oriented languages
  - **Class-based languages**: In Simula, Smalltalk, and C++, the ***implementation*** is described by classes. In these languages, we create objects by **instantiating** classes.
  - **Object-based languages**: In Self, objects are defined by adding methods to *existing objects* through **method addition** or **method overriding**.

3.2 Basic Features of Object-Oriented Languages

- **Dynamic lookup**
  - "Dynamic lookup" means that when we send a message to an object, the ***method body to execute*** is determined by the *run-time type* of the object, not by the static type[3].
  - **Implementation of dynamic lookup mechanism**

    (a) Using **method tables**: Suppose that a message m is sent to an object *ob*. Object *ob* maintains a message table and locates the table entry using the message m as the *index* to the table. C++ or Smalltalk uses this implementation.

object	
internal state	
method m_1	method body for m_1
$\vdots$	$\vdots$
method m_k	method body for m_k

    objects as tables

    (b) Using **overloaded functions**: In this implementation. "message name" is used as an overloaded function. When a message m is sent to *ob*, "*ob*" is used as an index to the overloaded function m and is used to determine the appropriate method body.

[3] Dynamic lookup is referred also as dynamic binding, dynamic dispatch, and run-time dispatch.

method m_i	
object type t_1	method body for t_1
object type t_2	method body for t_2
$\vdots$	$\vdots$
object type t_n	method body for t_n

methods as overloaded functions

3.3 Class-Based Languages

3.4 Object-Based Languages

4 Data Encapsulation

4.1 Difference between Modules and Classes

- *Modules partition the static program text, whereas classes can be used, in addition, to describe dynamic objects that exist at run time.*
- A **module** partitions the text of a program into manageable pieces.
  - Modules are *static*. We cannot create new modules or copies of existing modules dynamically as a program runs.
  - The **interface** (or **signature**) of a module is a collection of declarations of types, variables, procedures, and so on.
  - The **implementation** of a module consists everything else about the module, including the *code*.
  - A module is said to have a **local state** since its variables retain their values even when control is not in the module.
- A **class** corresponds to a 'type' (not in a precise sense).
  - *Objects* are *dynamic*. We can create and delete objects as a program runs.

4.2 Representation Independence

- An **abstract specification** tells us the behavior of an object independently of its implementation.
- A **concrete representation** tells us how an element is implemented, how its data is laid out inside a machine, and how this data is manipulated.
- **Representation independence principle**:
  - A program should be designed so that the *representation of an object can be changed without affecting the rest of the program.*
  - Also known as *implementation hiding*, *encapsulation*, or *information hiding*.
  - *Scope rules*, which control the visibility of names, are the primary tool for achieving representation independence.
- A **data invariant** for an element is a property of its local data that holds whenever control is not in the object. E.g.

  - The buffer is empty if array index *front* equals index *rear*.
  - The elments between *front* and *rear* are in the order they entered.
- **Data invariant principle**:
  - Design an object around a data invariant.

4.3 Program Structure in Modula-2

- A module in Modula-2 establishes a scope for the declarations within it.
  - A name crosses a module boundary only through an explicit **import** or **export** declaration.
- *Definition* and *implementation* modules set up public and private views.
- Execution of a Modula-2 program is controlled by a *program* or *main module*.
- A *local module* appears within another module or procedure.
  - The lifetime of local module is determined by the lifetime of tis enclosing construct.

Multiple instances in Modula-2

- **Opaque export** of a type occurs when the type is exported by mentioning only its name in a definition module, as in

```
definition module ComplexNumbers;
   export qualified Complex, cartesian, xpart, ypart;
   type Complex;
   procedure cartesian(x, y: real): Complex
   ...
end ComplexNumbers.
```

  - The only operations on opaque types are *assignment* and *tests for equality*.

4.4 Classes in C++

In-line expansion of function bodies

- Implementation hiding can result in lots of little functions that manipulate private data.
  - Function-call overhead can be avoided by using an implementation technique called **in-line expansion**, which replaces a call by a function body, taking care to preserve the semantics of the language.
  - In-line expansion in C++ differs from macroexpansion in C because in-line expansion preservesthe semantics of call-by-value parameter passing.
  - In-line expansion eliminates the overhead of function calls at run time, so it encourages free use of functions, even small functions.
- (Example)
  - `Buffer` class:

```
class Buffer {
   int front, rear;
   int notempty () { return front != rear; }
}
```

- `buf`, an instance of `Buffer` class:

  ```
  if (frand() >= 0.5 && buf.notempty()) ...
  ```

- After in-line expansion:

  ```
  if (frand() >= 0.5 && buf.front != buf.rear)) ...
  ```

5 Inheritance

- Inheritance is a language facility for defining a new class of objects as an extension of previously defined classes
  - Inheritance facilitates **code reuse**.
- A **subtype** S of a type T is such that any S-object is at the same time T-object.
  - That is, an object of type S also has type T.
- **Subtype principle**:
  - An object of a subtype can appear wherever an object of a supertype is expected.
- In **multiple inheritnace**, a class can be a direct subtype of more than one class.
  - Smalltalk, C++, and CLOS supports multiple inheritance.
- Single inheritance leads to a *class hierarchy*.

5.1 The Smalltalk-80 Vocabulary

- Smalltalk, the language, is just one part of the Smalltalk system.
- The Smalltalk vocabulary reflects the view of a running program as a collection of interacting objects.
- Five words of the Smalltalk vocabulary:
  - **Object**: collection of private data and public operations
  - **Class**: description of a set of objects
  - **Instance**: an instance of a class is an object of that class
  - **Method**: a procedure body implementing an operation
  - **Message**: a procedure call; request to execute a message
- General form of a message in Smalltalk

  ```
  elements at: top put: 'celebrate'
  ```

  - This expression sends `elements` a message consisting of two keywords.
  - Keyword `at:` carries argument `top`
  - An `at:put:` message is sent to object `elements`.

Elements of Smalltalk-80

- Variables must be declared before they are used.
  - A single copy of *class variable* is shared by all instances of a class.
  - A single copy of *global variable* is shared by all instances of all classes.
- Messages for *class methods* are sent to the class, and messages for *instance methods* are send to the individual instances of the class.
- Returning values:

```
isEmpty
   ^(top = 0)
```

6 Functional Programming Languages

6.1 Basic Concepts

- A **statement** is a programming language construct that is evaluated only for its *effect*.
  - Example: assignment statements, I/O statements, control statements
  - Programs in most languages are composed primarily of statements; such languages are said to be **statement-oriented**.
- Programming language constructs that are evaluated to obtain values are called **expressions**.
  - The data that may be returned as the values of expressions constitute the *expressed values* of a programming languages.
  - Expressions that are evaluated solely for its value, not for any other effects of the computation, are said to be **functional**.
  - Scheme and ML are **expression-oriented** languages; their programs are constructed of definitions and expressions but no statements.
- *Pure functional programming* is characterized by the following informally stated principle:

  > The value of an expression depends only on the values of its subexpressions, if any.

  - This principle rules out side effects within expressions.
- Another characteristic of functinoal languages is that users do not worry about managing storage for date:

  > *Implicit storage management: Built-in operations on data allocate storage as needed. Storage that becomes inaccessible is automatically deallocated.*

- Finally, functions are treated as 'first-class citizens':

  > *Functions are first-class values: Functions can be passed as an argument, can be a value of an expression, returned from a function, and can be put in a data structure.*

6.2 Scheme, A Dialect of Lisp

- Scheme is a dialect of Lisp that supports *static scoping* and *truely first-class functions*.
- Scheme supports higher-order functions.
  - A function is called **higher order** if either its arguments or its results are themselves functions.

6.3 ML: Static Type Checking

- A fundamental difference between Standard ML and Scheme is that ML is *strongly typed* while Scheme is *untyped*.
- ML supports *type inference*.
- A **polymorphic** function can be applied to arguments of more than on type.
  - **Parametric polymorphism** is a kind of polymorphism in which *type expressions are parameterized.* E.g. for type parameter α, $\alpha \to \alpha$ is denotes a class of types of functions whose argument and return value have the same type.
- ML supports *data type declaration.*

6.4 An Evaluator with No Environments

```
(define Eval
  (lambda (M)
    (cond
     ((var? M) ...)
     ((proc? M) M)
     (else ;(app? M) = #t
      (Apply
       (Eval (app-rator M))
       (Eval (app-rand M)))))))

(define Apply
  (lambda (a-proc a-value)
    (Eval (substitute a-val
      (proc-param a-proc)
      (proc-body a-proc)))))

(define substitute
  (lambda (v x M)
    (cond ...))))
```

7 Types

7.1 Evolution of Types in Programming Languages

- Fortran found it convenient to distinguish between integers and floating-point numbers, which is accomplished by an implicit lexical rule.

- Algol 60 made the type distinction *explicit* using by introducing redundant identifier-type declarations.
  - Algol 60 introduced the *explicit notion of types*.
  - Explicit notion of types required the *compile-time type checking*.
  - Algol 60's block structure introduced the *scope* (*visibility*) *of the variables*.
- PL/I extended the repertoire of types by including *typed arrays, records*, and *pointers*.
- Pascal provided a cleaner extension of types to arrays, records, and pointers, as well as *user-defined types*.
- Algol 68 introduced the well-defined notion of *type equivalence*.
- Simula introduced the notion of *classes*.
- Modula-2 is the first widespread language to use *modularization* as a major structuring principle[4].
  - *Typed interfaces* specify the types and operations available in a module.
  - An interface can be specified independent of the implementation.
- ML introduced the notion of *parametric polymorphism*.
  - ML types can contain *type variables*.
- Ada used the *name equivalence* as type equivalence.

7.2 Static and Strong Typing

- A type may be viewed as a set of clothes that protects an underlying untyped representation from arbitrary or unintended use.
- Objects of a given type have a *representation* that respects the expected properties of the data type.
- To prevent type violations, we generally impose a *static type structure*.
- A **type inference systems** can be used to infer the types of expressions when little or no type information is given explicitly.
- Programming languages in which the type of every expression can be determined by static program analysis are said to be **statically typed**.
  - Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile-time is sometimes too restrictive.
  - It may be replaced by weaker requirement that all expressions are guaranteed to be *type consistent* although the type itself may be statically unknown. This is usually achieved by **run-time type checking**.
- Languages in which all expressions are type-consistent are called **strongly typed** languages.
- *Every statically typed language is strongly typed but the converse is not true.*

[4]Modularization was first used in Mesa

7.3 Types as Sets of Values

- There is a universe V of all values: integers, pairs, records, functions. (a *CPO*).
- A **type** is a set of elements of V.
  - Not all subsets of V are legal types: tyey must obey some technical properties.
  - The subsets of V beying such properties are called **ideals**.
  - Hence, ***a type is an ideal***.
- The set of types (ideals) over V, when ordered by set inclusion, forms a **lattice**. (with top Top and bottom $\emptyset$)
  - The phrase **having a type** is interpreted as **membership in the appropriate set**.
  - As ideals over V may overlap, a value can have many types.
- A **type system** is a collection of ideals of V, which is usually identified by giving a $^{1)}$ *language of type expressions* and a $^{2)}$ *mapping from type expressions to ideals*.
- Since types are sets, subtypes simply correspond to subsets.
  - The semantic assertion T_1 **is a subtype of** T_2 corresponds to the mathematical condition $T_1 \subseteq T_2$ in the *type lattice*.
- The type lattice contains many more points than can be named in any type language.

8 Polymorphism

- In **monomorphic** languages, every value and variable can be interpreted to be one and only one type.
- In **polymorphic** langauges, some values and variables may have more than one type.
- Kinds of polymorphism
  (a) **Universal polymorphism**
    * **Parametric polymorphism**
    * **Inclusion polymorphism**

  (b) **Ad-hoc polymorphism**
    * **Overloading**
    * **Coercion**
- **Universal polymorphims** are "true polymorphisms."
  - **Parametric polymorphism** is so called since the uniformity of type structure is normally achieved by **type parameters**.
    * Parametric polymorphism is obtained when a function works uniformly on a range of types.

    - Functions exhibiting parametric polymorphism are also called **generic functions**. (e.g., *length: 'a list → int*)
  - **Inclusion polymorphism** is used to model *subtypes* and *inheritance*.

- **Ad-hoc polymorphism** is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in *unrelated ways for each type*.
  - In **overloading**, the same *variable name* is used to denote *different functions*.
  - A **coercion** is a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error.

- Real and apparent exceptions to the monomorphic typing rule in conventional langauges include:
  1. (Overloading) Integer constants may have both type integer and real.
  2. (Coercion) an integer value can be used where a real is expected, and vice versa.
  3. (Subtyping) elements of a subrange type also belong to superrange type
  4. (Value sharing) `nil` in Pascal is a constant which is shared by all the pointer types.

- Parametric polymorphism is the purest form of polymorphism but it should be noted that this uniformity of behavior requires that ***all data be represented, or somehow dealt with, uniformly (e.g., by pointers)***.

8.1 Universal Quantification

Generic functions

- **Universal quantification** enriches the 1st-order λ-calculus with *parameterized types* that may be specialized by substituting actual type parameters for universally quantified parameters.
- **Universally quantified type epxression**:

$$\forall \texttt{a}.\langle \textit{type expression on } \texttt{a}\rangle$$

- **Generic functions** factors out the *types of its arguments*.
  - I.e., ***generic function takes arguments of universally quantified types***.
- Two versions of `twice`
  - `value twice1 = all[t]λ(f:∀a.a→a)λ(x:t)f[t](f[t](x))`
    - e.g., `twice1[Int](id)(3)`
    - In this case, `id:∀a.a→a` is a *generic identity function.*
  - `value twice2 = all[t]λ(f:t→t)λ(x:t)f(f(x))`
    - `twice2[Int]` ≡ `λ(f:Int→Int)λ(x:Int)f(f(x))`
    - `twice2[Int](intId)` ≡ `λ(x:Int)intId(intId(x))`
    - e.g., `twice2[Int](id[Int])(3)`

Parametric types

- Parametric type definition factors out the common structures in *type definitions*

$$\texttt{type Pair[T] = T} \times \texttt{T}$$
$$\texttt{type PairOfBool = Pair[Bool]}$$
$$\texttt{type PairOfInt = Pair[Int]}$$

- A parametric type definition introduces a new *type operator*.
  - **Type operators** are not types, they operate on types, i.e., **type operators takes a type and returns a type**.
  - `Pair` above is a type operator.
  - c.f., in `type B =` $\forall$`T.T`$\rightarrow$`T`, `B` is not a type operator but a *type*.

8.2 Existential Quantification

- **Existential type expression**:

$$\exists \mathrm{a}.\langle \textit{type expression on } \texttt{a} \rangle$$

  - `p:`$\exists$`a.t(a)` means that "for some type `a`, `p` has type `t(a)`."
  - e.g., $(3,4) : \exists \mathrm{a}.\mathrm{a}\times\mathrm{a}$, where `a` $=$ `Int`
  - e.g., $(3,4) : \exists \mathrm{a}.\mathrm{a}$, where `a` $=$ `Int`$\times$`Int`
- More examples
  - `type Top =` $\exists$`a.a`: the type of any value (*the biggest type!*)
  - $\exists$`a.`$\exists$`b.a`$\times$`b`: the type of any pair
- **Counterintuitive example**
  - Consider $\exists$`a.a`$\times$`a`: this is not only the type of `(3, 4)` but also of `(3, true)`.
  - This is because `3:Top` and `true:Top`.
- $\exists$`a.a`$\times$`(a`$\rightarrow$`Int)` forces a relation!
  - Consider `(3, length)` where `length:`$\forall$`a.List[a]`$\rightarrow$`a`.
  - `(3, length)` $\not:$ $\exists$`a.a`$\times$`(a`$\rightarrow$`Int)`
  - Why? Can't we show that `3:Top` and `length:Top`$\rightarrow$<u>`Int`</u>?
  - No. Since `length` maps "integer lists" to <u>integers</u>, we cannot assume that any arbitrary object of type `Top` will be mapped into integer.
- Sometimes existential types does not convey much information to us.
  - But when an existential type expression is sufficiently structured, it can be useful.
  - `x:`$\exists$`a.a`$\times$`(a`$\rightarrow$`Int)` means that `(snd(x))(fst(x))` yields an 'integer'.

Existential quantification and information hiding

- **We can view `x:∃a.a×(a→Int)` as a simple example of *abstract type* packaged with its set of operations**
  - `a` is the abstract type itself, which hides a representation.
- An ordinary object `(3, succ)` may be converted to an abstract object having type `∃a.a×(a→Int)` by **packaging** it so that some of its structure is hidden.
  - The operation `pack` below encapsulates the object `(3, succ)` so that the user knows only that an object of the type `a×(a→Int)` exists without knowing the actual object.
  - `value p = pack[a=Int in a×(a→Int)](3, succ):∃a.a×(a→Int)`
  - Packaged object such as `p` are called **packages**.
  - The value `(3, succ)` is referred to as the **content** of the package.
  - The type `∃a.a×(a→Int)` is the **interface**: it determines the structure specification of the contents and corresponds to the specification part of a data abstraction.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[3] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[4] M. L. Scott. *Programming Language Pragmatics.* Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[5] R. Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley, Reading, MA, 1989.

Contents