Chapter 2. Data Abstraction
(Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Outline

This chapter introduces $\cdots$.

- Specifying Data via Interfaces
- Representation Strategies for Data Types
- Interfaces for Recursive Data Types
- A Tool for Defining Recursive Data Types
- Abstract Syntax and Its Representation

2.1 Specifying Data via Interfaces

Every time we decide to represent a certain set of quantities in a particular way, we are defining a new data type

- whose values are those representations and
- whose operations are the procedures manipulating those entities

Motivation

- The representation of these entities is often complex, so we do not want to be concerned with their details.
- We may decide to change the representation of the data.
- To change the representation of some data, we must be able to locate all parts of a program that are dependent on the representation.

2.1 Specifying Data via Interfaces (cont.)

Data abstraction:

- a data type = an interface + an implementation
- *Abstract data type*

A client code is *representation-independent* if it only uses the interface.

2.1 Specifying Data via Interfaces (cont.)

Notation: the representation of data v, $\ulcorner v \urcorner$

A simple example: the (abstract) data type of natural numbers

- An interface

$$
\begin{aligned}
(\text{zero}) &= \ulcorner 0 \urcorner \\
(\text{is-zero?}\ \ulcorner n \urcorner) &= \begin{cases} \#t & n = 0 \\ \#f & n \neq 0 \end{cases} \\
(\text{successor}\ \ulcorner n \urcorner) &= \ulcorner n+1 \urcorner \qquad (n \geq 0) \\
(\text{predecessor}\ \ulcorner n+1 \urcorner) &= \ulcorner n \urcorner \qquad (n \geq 0)
\end{aligned}
$$

Many possible representations of the interface

- Unary representation
- Scheme number representation
- Bignum representation

2.2 Representation Strategies for Data Types

Some strategies for representing a data type of environments
- In a PL implementation, it associaes each variable with a value
- In a compiler, it associates each variable with a type

Variables may be presented in any way so long as we can check two variables for equality.

From the next slides,
- The environment interface
- Data structure representation
- Procedural representation

The Environment Interface

An environment is a function mapping a finite set of variables onto (Scheme) values

- $env = \{(var_1, val1), \cdots, (var_n, valn)\}$.
- the value of the variable in env is called its $binding$ in env.

The interface to a data type for environments

$$
\begin{aligned}
\text{(empty-env)} &= \ulcorner \{\,\} \urcorner \\
\text{(apply-env } \ulcorner f \urcorner \ var\text{)} &= f(var) \\
\text{(extend-env } var \ \ v \ \ulcorner f \urcorner\text{)} &= \ulcorner g \urcorner \\
&\qquad \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

The Environment Interface (cont.)

Using the interface,

- to build an environment: $env = \{(d, 6),\ (x, 7),\ (y, 8)\}$

```
> (define env
    (extend-env ‘d 6
      (extend-env ‘y 8
        (extend-env ‘x 7
          (extend-env ‘y 14
            (empty-env)))))
```

- to look up a binding for a variable: $env(d)$

```
> (apply-env env ‘d)
```

Constructors and observers of the procedures of the interface

Data Structure Representation

Every environment can be built by starting with the empty environment and applying extend-env n times.

So, every environment can be built by an expression in the following grammar:

$$
\begin{aligned}
Env\text{–}exp &::= (empty\text{–}env) \\
&::= (extend\text{–}env\ Identifier\ Scheme\text{–}value\ Env\text{–}exp)
\end{aligned}
$$

- See an implementation in Figure 2.1

Procedural Representation

An alternative representation is to use procedures for environments.

- See an implementation in P.40

Every client-code using the environment interface will be represent-independent.

- One representation can be replaced with the other without affecting the client code.

cf. *defunctionalization*

- A transformation of (higher-order) functions or a procedural representation into data structures or data structure representation

2.3 Interfaces for Recursive Data Types

A recursive data type for lambda-calculus expressions:

$$
\begin{aligned}
Lc\text{–}exp &::= Identifier \\
&::= (lambda\ (Identifier)\ Lc\text{–}exp) \\
&::= (Lc\text{–}exp\ Lc\text{–}exp)
\end{aligned}
$$

What is an interface for the lambda-calculus expressions? In other words, what are constructors and observers for them?
- cf. Observers (predicates and extractors)

Using the interface, write a procedure as
- $occurs\text{–}free\ :\ Sym\ \times\ LcExp\ \rightarrow\ Bool$

2.4 A Tool for Defining Recursive Data Types

A tool for automatically constructing and implementing such interfaces one discussed in Section 2.3 in Scheme:

```
(define-datatype lc-exp lc-exp?
  (var-exp (var identifier?))
  (lambda-exp (bound-var identifier?))
  (app-exp (rator lc-exp?) (rand lc-exp?)))
```

Examples:

- x : (var-exp ‘x)
- $\lambda x.x$: (lambda-exp ‘x (var-exp ‘x))
- $(lambdax.x)(lambday.y)$:

```
        (app-exp
            (lambda-exp ‘x (var-exp ‘x))
            (lambda-exp ‘y (var-exp ‘y)))
```

2.4 A Tool for Defining Rec. Data Types(Cont.)

A procedure occurs-free? using the interface generated by the tool.

- the form “cases” to determine the variant to which an object of a data type belongs and to extract its components.

```
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp (var) (eqv? var search-var))
      (lambda-exp (bound-var body)
        (and
          (not (eqv? search-var bound-var))
          (occurs-free? search-var body)))
      (app-exp (rator rand)
        (or
          (occurs-free? search-var rator)
          (occurs-free? search-var rand))))))
```

2.4 A Tool for Defining Rec. Data Types(Cont.)

See the textbook (P.47 and P.49) for the general form define-datatype declaration and the general syntax of cases.

The form "define-datatype" is an example of a *domain-specific language (DSL)*.

- A DSL is a small language for describing a single task among a small, well-defined set of tasks.
- Such a language may lie inside a general-purpose language, as define-datatype does, or it may be a standalone language with its own set of tools.

2.5 Abstract Syntax and Its Representation

Concrete syntax (defined by a grammar) vs. Abstract syntax (by define-datatype)

- The concrete syntax is an external representation for humans
- The abstract syntax is an internal one for computers
- See Figure 2.2 for a comparison

Parsing is a task to derving the corresponding abstract syntax tree from the concrete syntax which is a sequence of characters.

- Parser
- Parser generator
- $parse\text{–}expression : SchemeVal \rightarrow LcExp$

The reverse task of parsing is called unparsing or "pretty-printing".

- Unparser or pretty-printer
- $unparse\text{–}lc\text{–}exp : LcExp \rightarrow SchemeVal$