Interactions of Abstractions in Programming

Gordon S. Novak Jr.
Department of Computer Sciences
University of Texas at Austin

novak@cs.utexas.edu
http://www.cs.utexas.edu/users/novak

August 23, 2000

Nature of Programming

- Computer programs are composed of instances of data and procedural abstractions.
- Programs should be built by reuse and composition of abstract components.
  - Reuse seldom occurs in practice.
  - Existing techniques for reuse are ineffective.
- Existing software components are:
  - not abstract enough
  - not sufficiently parameterizable.
- We need to understand:
  - what the abstractions of programming are
  - how to create programs by composition of abstract components
  - how to make programming by composition easy and effective.

Outline

- Analyze a simple program to identify its components:
  - There are many components.
  - The components interact strongly.
- The program can be constructed from independent abstract components using *views*.
  - The good news: It works and generates efficient code.
  - The bad news: The specs are very complex.
  - The hope: Constraint propagation and inference can fill in many of the details.
- Work on deriving programs from minimal specifications
- Related work

A Simple Program

Given as input a list of sublists, each of which has the format `(key n)`, sum the `n` for each `key`.

```
(alsum '((a 3) (b 2) (c 1) (a 5) (c 7)))

     =  ((C 8) (B 2) (A 8))
```

```
(defun alsum (lst)
  (let ((alist nil) entry)
   (dolist (item (identity lst))
     (setq entry
       (or (assoc (car item) alist)
           (progn (push (list (car item) 0 ) alist)
                  (assoc (car item) alist))))
    (incf (cadr entry) (cadr item) ) )
  alist))
```

Different fonts are used for code associated with different abstract components. Clearly, code of the components is thoroughly mixed together in this program.

Program Components

This small program combines five abstract procedural components:

- Iterate-Accumulate: Iterate through a sequence of items, accumulating some aspect of the items.
- Find-Update: Find an item in an indexing structure using its key value, then update the record with information from the item.
- `dolist`: An iterator for linked lists in Lisp.
- Alist: Association list, a kind of indexing structure.
- Adder: A kind of accumulator.

In addition, there are several interfaces of *glue code*: how to get the sequence from the input, how to get the key from the item, what aspect of the item to accumulate.

Substitution Test

A *substitution test* is useful to determine whether a decomposition of a program into components is correct:

> Can the program be modified by a substituting a different component for an existing one, without changing the other components?

In this example, we could produce different but related programs by substitution of components:

- If the input were an array rather than a linked list, a different iterator would be used.
- A different indexing structure could be used, e.g. an AVL tree rather than an Alist.
- The product of the integers could be accumulated rather than their sum.
- By changing the glue code, we could change the key and accumulated value, e.g. to accumulate symbols associated with each numeric value.

Abstract Software Components

We claim that effectively reusable software components must be abstract. Otherwise:

- Each decision that is hard-coded into a component becomes a requirement on the use of that component.
- There will be a combinatoric number of components based on combinations of decisions.
- It will be hard to find the right component among the many available.
- Learning the requirements of a component becomes a burden on the user.
- Conforming to the requirements reduces the benefit of reuse.

Instead, there should be *only one* abstract version of each component, which should serve for all uses.

Abstract Procedure: Iterate-Accumulate

The top-level program is a generic procedure, Iterate-Accumulate:

```
(gldefun itacc (input:anything acc:anything)
  (let (seq item iter)
    (acc := (a (typeof acc)))
    (send acc initialize)
    (seq := (send input sequence))
    (iter := (send seq make-iterator))
    (while (not (send iter done))
      (item := (send iter next))
      (acc := (send acc update item)) )
    acc))
```

When this procedure is specialized, wrapper types for the particular arguments will replace the argument types.

Find-Update

The accumulation is another generic procedure, Find-Update:

```
(gldefun findupd (coll:anything item:anything)
  (let (entry)
    (entry := (send coll find (send item key)))
    (if (null entry)
        (coll := (send coll insert
                               (send item key)))
        (entry := (send coll find
                                (send item key))))
    (send entry update item)
    coll ))
```

Abstract Components with State

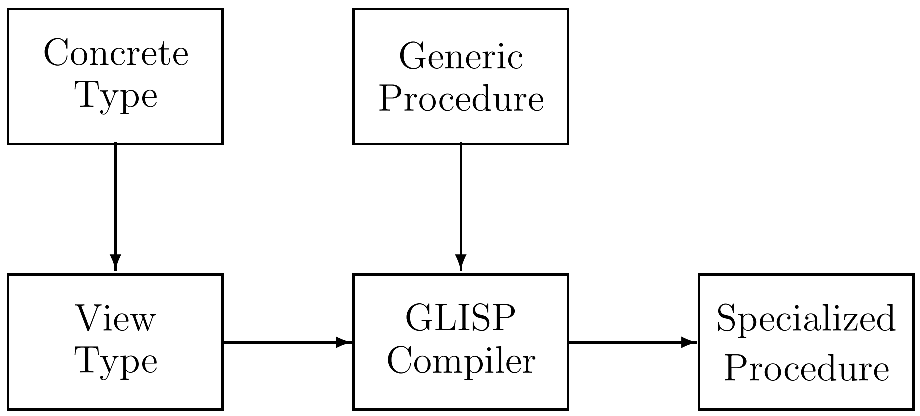
```
(alist (listof (cons (key anything)
                     (rest anything)))
  msg ((initialize (nil))
       (find   (glambda (self key)
                 (assoc key self)))
       (insert (glambda (self key)
                 (cons
                   (send (a (typeof (first self))
                            with key = key)
                         initialize)
                   self)))) )

(adder (sum number)
  prop ((initial-value (0)) )
  msg  ((init   ((sum := (initial-value self))))
        (update (glambda (self (item number))
                  ((sum self) _+ item)))
        (final  (sum))))
```

Each of these expresses the functionality of the abstract component cleanly.

Specializing Components

The GLISP compiler can specialize a generic procedure through views.



A *view type*:

- wraps a concrete type (without adding any data)
- has an abstract type as a superclass, inheriting its methods
- defines data or procedures needed by the abstract type in terms of the concrete type.

Compilation through Views

Compilation of a generic procedure through views produces a specialized version of the generic:

- Code expansion through views is recursive at compile time: abstractions can be nested.
- Code from views can be compiled in-line.
- Partial evaluation optimizes the code produced.

The use of views has a low cost (often zero) after compilation. A view can be considered a zero-weight wrapper that makes a data type look like an abstract type.

Example of View

```
(circle anything
  prop  ((area  (pi * radius ^ 2))) )

(pizza (list (size symbol)
             (toppings (listof symbol)))
  views ((circle circle pizza-as-circle)))

(pizza-as-circle (p pizza)
  prop  ((diameter  ((if ((size p) = 'large)
                          then 18
                          else 12)))
         (radius    (diameter / 2)))
  supers (circle))

(gldefun t1 (pz:pizza)
  (area (circle pz)))

result type: REAL
(LAMBDA (PZ)
  (IF (EQ (CAR PZ) 'LARGE)
      254.46900
      113.09733))
```

Parameterization of Components

View types are used to parameterize abstract components, in several ways:

- View types wrap concrete types for the application, and cause derived values to be wrapped as well.

```
(myinputv (z myinput)
  prop ((sequence (z) result myalseq)))
```

- View types include *glue code*:

```
(myalrec (z myinputitem)
  prop ((accdata ((price z)))
        (key     ((name z)))) )
```

- View types invoke generic procedures by including abstract superclasses:

```
(myview1 (z123 myaldata)
  prop   ((sum ((data z123))))
  supers (adder))
```

  An application could have multiple `adder`s associated with the same record.

Combined View Type

The type that accumulates data for each key value is more complex and includes components from several sources:

- The data structure includes data from independent sources:
  - structural data (e.g. pointers) from the indexing method used
  - key data whose type is the type of the key used
  - accumulator data specified by each accumulator.
- The methods `initialize` and `update` distribute themselves over the accumulators.

```
(myaldata (list (key symbol) (data integer))
  msg  ((initialize
              ((send (view1 self) init)
               self))
        (update (glambda (self item)
               (send (view1 self) update
                     (send item accdata)))))
  views ((view1 adder myview1)) )
```

Synthesized Program

Using two given types and seven (handwritten) view types, the compiler combines the five generic components to produce an application program:

```
(LAMBDA (INPUT ACC)
  (LET (ITEM ITER)
    (SETQ ACC (LIST))
    (SETQ ITER INPUT)
    (WHILE ITER (SETQ ITEM (POP ITER))
      (SETQ ACC
        (LET (ENTRY)
          (SETQ ENTRY (ASSOC (CAR ITEM) ACC))
          (UNLESS ENTRY
            (PUSH
              (LET ((SELF (LIST (CAR ITEM)
                                 0)))
                (SETF (CADR SELF) 0)  SELF)
              ACC)
            (SETQ ENTRY
                  (ASSOC (CAR ITEM) ACC)))
          (INCF (CADR ENTRY) (CADR ITEM))
          ACC)))
    ACC))
```

Recap

- We have identified the abstract components in a small application program.
- The components are clear and satisfy the substitution test.
- Even a small program has many abstract components!
- The application can be mechanically synthesized from the components using view types.
- Unfortunately, the view types are complex, reflecting complex relationships of components.

Automation using Constraints

Our goal is to automate the production of view types from simpler specifications. The user should select a few components and interfaces; the rest of the spec should be inferred.

- When data is passed from one component to another, the types are equal.
- A type mismatch is usually a need for type conversion; a default method may be defined.
- Some components have typical implementations, e.g. a typical way to accumulate integers is to add them.
- Components parameterize each other. For example, if an `adder` is made part of an `alist`, it will add its `sum` variable to the `alist` record.

Constraint Propagation

We envision a process similar to MOLGEN or Waltz filtering:

- Decisions made in one component will be propagated to connected components, allowing them to infer other decisions.
- Inferences must be retractable: the programmer can override them. Reason maintenance will retract derived decisions when needed.
- A graphical interface will show specification status of the developing program.

Example

In one system we have implemented, the user can select a program framework (e.g. Iterate-Accumulate) and make specifications. Suppose the user specifies that the input to Iterate-Accumulate is of type `(arrayof integer)`:

- The input type matches the abstract type of the Iterator, `(sequence anything)`, so it is is inferred to be the input to the Iterator.
- The Iterator can now infer its item type as `integer`.
- The input to the Accumulator will thus be `integer`.
- Since a typical way to accumulate integers is to add them, the accumulator can specialize itself to an Adder.

Minimal Specification

A minimal specification, Iterate-Accumulate on `(arrayof integer)`, is enough to generate a complete program:

```
result type: INTEGER
(LAMBDA (IFA)
  (LET (ACC)
    (SETQ ACC 0)
    (LET (ITEM)
      (DOTIMES (INDX (ARRAY-TOTAL-SIZE IFA))
        (SETQ ITEM (AREF IFA INDX))
        (INCF ACC ITEM)))
    ACC))
```

The programmer can obtain a different program by saying more.

Conclusions

- Greater abstraction is needed to make software construction by composition of components really work.
- Automation is needed to make it easy to specify programs by composition:
  - minimize programmer input
  - minimize programmer learning

Related Work

- SPECWARE^{TM} [Srinivas96]: composition of types by finding the colimit of the descriptions.
- KIDS [Smith90]: transforms logic specs into programs for combinatoric search problems.
- Lowry87: knowledge of generic designs as parameterized theories.
- Aspect-Oriented Programming [Kiczales97]: *aspect weaver* combines program fragments from separate aspects of a design at *join points*.
- Programmer's Apprentice [Rich90]: synthesize programs from *clichés*.
- Batory93: construction of software systems from layers of plug-compatible components.
- Krueger92 and Mili95 survey software reuse.