PROGRAM DEVELOPMENT

FROM

EXECUTABLE SPECIFICATIONS

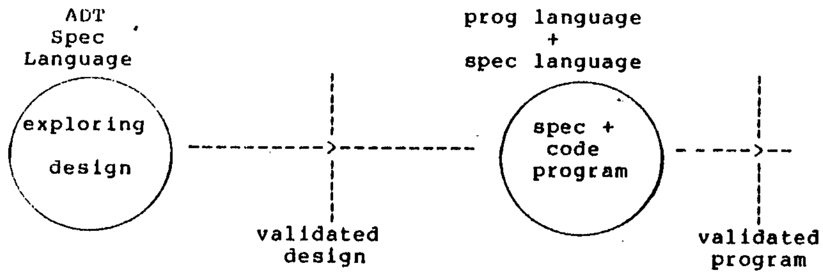
Derek Coleman
Robin Gallimore

HPLabs, Hewlett Packard, Bristol

DATA TYPE SPECIFICATIONS

* codify application domain knowledge at a high level of abstraction
  - reusable 'knowledge'
  - standard concepts and definitions
* provide abstractions necessary for concise formulation of specification
* if the data type specs contain an executable subset
  - design time testing
* if at appropriate level of abstraction then code blueprints for first versions
  - correctness transferred from spec to code

MODEL OF FORMAL DEVELOPMENT



* exploring design

  - requirements
  - high-level algorithm
  - validation

produces

requirement statements
+
executable model
+
standard test cases

payoffs

- design time testing against requirements
- management control of design process
- correct design helps establish correctness of code

DEVELOPMENT OF PROGRAMS

* from executable specification to specified program

* design decisions to be made

  - representation of abstract types
    eg lists by pointer structures

  - modules and their interfaces
    eg cons procedure, head function ....

* these decisions

  determine efficiency of code

  and must be documented

    - use abstraction fns + invariants for representations

    - use pre-post conditions for modules

STRATEGY

in order for correctness of design to carry into program

1. fix module interfaces

2. choose simple representations

3. once functionally OK
   measure space/time efficiency

4. improve efficiency by
   changing representations
   or
   redefining module interfaces

REPRESENTING ABSTRACT TYPES

* abstraction fn mapping concrete into abstract values

* invariant relation characterising those concrete values which represent abstract values

eg sequences by linear linked lists

```
abstract                          concrete
~:-->list                         type list ptr=↑record
_._: item list-->list                           val:item
                                               link:listptr
                                                  end
```

* abstraction function
  abs: listptr state --> list

  where
  ```
  state:  listptr --> <item,listptr>
  abs(nil,ζ) = ~
  abs(l,ζ) = i.abs(l') if ζ(l)=<i,l'>
  ```

* invariant
  the listptr must be acyclic

DESIGNING THE BASIC TYPE PROCEDURES

* list values are constructed from ~ and .

* the related procedural components may be specified by pre/post conditions

```
procedure empty (var l:listptr)
PRE:  true
POST: abs(l)=~

procedure cons (i:item;var l:listptr)
PRE:  true
POST: abs(l)=i.abs(lo)
      and
      tail(l) aliases lo
```

notice:

1. use of abstract data specification to supply vocabulary (ie .,~)
2. design decision to make cons append a new node rather than copy its list argument (alias)
3. proof obligation that invariant is preserved

DESIGNING OTHER MODULES

* example

  filter out all the items from a list ≤ a given value

```
filter:item list -->list
filter(i,~)=~
filter (i,j.s)=if i≤j then j.filter(i,s)
                   else filter(i,s)
```

* a no-side effects strategy for modules

```
function FILTER (i:item;s:listptr):list ptr
PRE:  true
POST: abs(FILTER)=filter(i,abs(s))
      and
      s=so
```

  makes code-production straightforward

CODE PRODUCTION

1. eliminate pattern matching
2. transform into programming language syntax

```
filter(i,~)=~
filter(i,j.s)=if i≤j then j.filter(i,s)
              else filter (i,s)
```

```
filter(i,s)= if s==~then~
             else
             if i≤j then head(s).filter(i,tail(s))
             else filter(i,tail(s))
```

```
function FILTER(i:item; listptr):listptr
begin
if s=empty then FILTER:=empty
else
if i≤ head(s) then
      FILTER:=cons(head(s),FILTER(i,tail(s)))
else
      FILTER:=FILTER(i,tail(s))
end
```


MEASURE - REVIEW DESIGN

* after measurement - change inefficient representations
* may be necessary to refine executable spec to stop code-spec separation

eg: eliminate recursion

```
                   filter(i,s)=f(i,s,~)
                   f(i,~,res)=res
                   f(i,j.s,res)=if i j then
filter(i,~)=~                     f(i,s,res:j)
filter(i,j.s)= ...             else
                                  f(i,s,res)
```

note: is right append

```
function FILTER (i:item;s:listptr):listptr
var res:listptr
begin
res:=empty;
while s<>empty do
   begin
   if i  head(s) then res:=rap(res,head(s)) :
   s:=tail(s)
   end;
FILTER:=res
end
```

note: rap is right append

OBSERVATIONS

* result is a specified and documented program
* two kinds of decision only
  - data type representation
  - module interfaces
* given these decisions code production can be a transformation
* changes to more efficient representation may cause changes to data type specification
* choice of {representation
  {module interfaces

  requires programming skill
* transformations are mechanical

MACHINE SUPPORT

* systematic code production is practical even if done manually
* machine support is required to keep spec-code correspondence in face of updates
* transformations can be programmed
* possibly expert systems can be used to capture programmer skill
  eg CHI from Kestrel Institute

FINAL REMARKS

* writing specifications is beneficial
* semantic processing is very desirable
* lack of mechanical theorem provers is <u>the</u> real obstacle
* executability to
  - effective in practice
  - can be provided cheaply

    eg UMIST OBJ
* systematic program production can be given machine support
* the benefits of formal methods come from

  improved quality