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

ADT
Spec
Language

exploring
design

validated
design

spec +

prog language +
spec language

validated
program

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 fn + 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

- : list
_ : item list --> list

concrete

type list ptr = frecord
val: item
link: listptr
end

* abstraction function
  abs: listptr state --> list
where
  state: listptr --> <item, listptr>
  abs(nil, 2) = -
  abs(1, 2) = i.abs(1') if 2(1) = <i, 1'>

* 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=s

makes code-production straightforward
CODE PRODUCTION

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

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

filter(i, s) = if s = empty then
else
  if i ≤ head(s) 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
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
  f(i, ~, res: j)
else
  f(i, s, res)

\[ f(i, s, res) = \begin{cases} 
  res & \text{if } i \leq j \\
  f(i, s, res: j) & \text{else}
\end{cases} \]

\[ f(i, ~, res) = \begin{cases} 
  res & \text{if } s = \text{empty} \\
  f(i, s, res: j) & \text{else}
\end{cases} \]

\[ f(i, s, res) = \begin{cases} 
  res & \text{if } i \leq \text{head}(s) \\
  f(i, ~, res: j) & \text{else}
\end{cases} \]

\[ f(i, ~, res) = \begin{cases} 
  res & \text{if } s = \text{empty} \\
  f(i, s, res: j) & \text{else}
\end{cases} \]

\[ f(i, s, res) = \begin{cases} 
  res & \text{if } i \leq \text{head}(s) \\
  f(i, ~, res: j) & \text{else}
\end{cases} \]

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

\[ \text{note: rap is right append} \]
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 the 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