

Formal Program Re-design

Ran Ettinger

PRG Student Conference

30 September 2004

THE PROBLEM OF THE SHORTEST SUBSPANNING TREE

- N points can be fully interconnected by $N-1$ point-to-point connections.
- Such a set is called a “tree” or a “subspanning tree” and the connections are called “its branches”.
- We now assume that the length of each of the $N*(N-1)/2$ possible branches has been given.
- Defining the length of the tree as the sum of its branches, we can ask ourselves how to determine the shortest tree between those N points.
 - (from Dijkstra’s book “a discipline of programming”)

Design 1

colour an arbitrary point red and the remaining points blue;

do number of red points $\neq N \rightarrow$

select the shortest now violet branch;

colour it and its blue endpoint red

od

An optimized version (design 2)

- a reduced set of potential shortest (ultraviolet) branches is:

colour an arbitrary point red and the remaining ones blue;
determine the set of ultraviolet branches;

do number of red points $\neq N \rightarrow$

select the shortest now ultraviolet branch;

colour it and its blue endpoint red;

adjust the set of ultraviolet branches

od

The current version (Design 2.1)

- In fact, Dijkstra's implementation looks more like:

Colour an arbitrary point red and the remaining ones blue;
determine the set of ultraviolet (virtual) branches;

do number of red points $\neq N \rightarrow$

adjust the set of ultraviolet branches and

select the shortest now ultraviolet branch (both in a single loop);

colour it and its blue endpoint red

od

Program 2.1

```
[[ var uvl : int[N]; k, lcr, i : int •
  i := 0;
  do i ≠ N-1 → i := i + 1; from[i], to[i], uvl[i] := 0, i, INF od;
  k, lcr := 1, N;
  do k ≠ N →
    [[ var suv, min, h : int •
      suv, min, h := 0, INF, k;
      do h ≠ k →
        [[ var len •
          len := DIST[lcr][to[h]];
          if len < uvl[h] then uvl[h], from[h] := len, lcr fi;
          if uvl[h] < min then min, suv := uvl[h], h fi;
          h := h + 1
        ] ]
      od;
      swap(from[k], from[suv]);
      swap(to[k], to[suv]);
      swap(uvl[k], uvl[suv]);
      lcr, k := to[k], k+1
    ] ]
  od
]]
```

Re-design

- Faced with a new requirement, say, determine the length of the longest subspanning tree, we regret the design choice (2.1) of using a combined single loop. We wish to reverse this decision cheaply. For that, we make an attempt to “untangle” the **adjustment of the ultraviolet set of branches** from the **selection of the shortest branch**:

Selecting seed statements

```
[[ ...
  suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from[h] := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od
  ...
]]
```

What is a program slice?

- A slice of program S with respect to a set of variables V is any program $S1$ such that when S terminates $S1$ must terminate in the same state (where the state space is limited to V).

Wedge: slicing the seeds

```
[[ ...
  suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uv1[h] then uv1[h], from[h] := len, lcr;
      if uv1[h] < min then min, suv := uv1[h], h fi;
      h := h + 1
    ]|
  od
  ...
]]
```

Slicing the remaining statements

```
[[ ...
  suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from[h] := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od
  ...
]]
```

Rejecting the transformation

- both the **slice** and its **complement** assign to *uvl*,
- *uvl* is alive at the end
- According to Lakhotia's formulation: the transformation is rejected

- In our new Extract Slice formulation, we avoid that problem by hiding the modifications in the slice (or its complement). This is achieved by passing the variable (e.g. *uvl*) by value

Program 2.0.1

```
...
[[ var UVL : int[N] • UVL := uv1;
   h := k;
   do h ≠ k →
     |[ var len •
        len := DIST[lcr][to[h]];
        if len < uv1[h] then uv1[h], from[h] := len, lcr;
        h := h + 1
     ]|
   od
   (suv, min, h := 0, INF, k;
   do h ≠ k →
     |[ var len •
        len := DIST[lcr][to[h]];
        if len < uv1[h] then uv1[h] := len;
        if uv1[h] < min then min, suv := uv1[h], h fi;
        h := h + 1
     ]|
   od) [value uv1 \ UVL]
]] ...
```

Further Improvements

- now the **adjustment** code can be made reusable by extracting it into a named procedure
- if we wish to reduce the amount of code duplication, we can notice that updating the value of *uvl* in the **complement** is redundant
 - but this step is currently done by intuition only

Program 2.0.2

```
...
[[ var UVL : int[N] • UVL := uv1;
   h := k;
   do h ≠ k →
     |[ var len •
        len := DIST[lcr][to[h]];
        if len < uv1[h] then uv1[h], from[h] := len, lcr;
        h := h + 1
     ]|
   od;
   (suv, min, h := 0, INF, k;
   do h ≠ k →
     |[ var len •
        if uv1[h] < min then min, suv := uv1[h], h fi;
        h := h + 1
     ]|
   od) [value uv1 \ UVL]
]] ...
```

Extract Slice transformation

Let S be any deterministic statement, and v a (user selected) set of variables, then

$$S = \llbracket \begin{array}{l} \text{var } V \bullet \\ \quad V := v; \\ \quad S1[\text{value } u \setminus u]; \\ \quad S2[\text{value } v \setminus V] \end{array} \rrbracket$$

where $V \cap \text{free}(S) = \emptyset$,

$u = \text{def}(S)/v$,

$S1 = \text{slice}(S, v)$, and

$S2 = \text{slice } S, u$

provided

$\text{slice}(S, \text{def}(S))$ is the whole of S (i.e. no dead code).

Naive statement duplication

- The main step of the proof is based on the more naive transformation:

Let S be any deterministic statement, and v a (user selected) set of variables, then

$$S = \llbracket \begin{array}{l} \text{var } V \bullet \\ \quad V := v; \\ \quad S[\text{value } u \backslash u]; \\ \quad S[\text{value } v \backslash V] \end{array} \rrbracket$$

where $V \cap \text{free}(S) = \emptyset$ and
 $u = \text{def}(S)/v$.

Selecting seed statements

- to illustrate the working of the naive statement duplication transformation, let's return to the Shortest Subspanning Tree example: here we take S to be

```
suv, min, h := 0, INF, k;
do h ≠ k →
  |[ var len •
    len := DIST[lcr][to[h]];
    if len < uv1[h] then uv1[h], from[h] := len, lcr;
    if uv1[h] < min then min, suv := uv1[h], h fi;
    h := h + 1
  ]|
od
```

- and let v be the set $\{\text{from}, \text{to}, \text{uv1}\}$ such that u must be the set $\{\text{min}, \text{suv}\}$, and so we get:

Program 2.0.3

```
...
|[ var FROM, TO, UVL : int[N] • FROM, TO, UVL := from, to, uvl;
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from[h] := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od) [value min, suv, h \ min, suv, h];
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od) [value from, to, uvl \ FROM, TO, UVL]
]| ...
```

Identify dead code (Program 2.0.3.1)

```
...
|[ var FROM, TO, UVL : int[N] • FROM, TO, UVL := from, to, uvl;
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from[h] := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od) [value min, suv, h \ min, suv, h];
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from := len, lcr;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od) [value from, to, uvl \ FROM, TO, UVL]
]| ...
```

Eliminate dead code and identify redundant parameters(Program 2.0.3.2)

```
...
|[ var FROM, TO, UVL : int[N] • FROM, TO, UVL := from, to, uvl;
  (h := k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h], from[h] := len, lcr;
      h := h + 1
    ]|
  od) [value min, suv, h \ min, suv, h];
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uvl[h] then uvl[h] := len;
      if uvl[h] < min then min, suv := uvl[h], h fi;
      h := h + 1
    ]|
  od) [value from, to, uvl \ FROM, TO, UVL]
]| ...
```

Remove redundant parameters(Program 2.0.3.3)

```
...
|[ var UVL : int[N] • UVL := uv1;
  (h := k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uv1[h] then uv1[h], from[h] := len, lcr;
      h := h + 1
    ]|
  od) [value min, suv, h \ min, suv, h];
  (suv, min, h := 0, INF, k;
  do h ≠ k →
    |[ var len •
      len := DIST[lcr][to[h]];
      if len < uv1[h] then uv1[h] := len;
      if uv1[h] < min then min, suv := uv1[h], h fi;
      h := h + 1
    ]|
  od) [value uv1 \ UVL]
]| ...
```

Remove 'by value' parameter

- Formally, the removal of redundant parameters was based on our formulation:

Let S be any command, then

$$S[\text{value } f \setminus a] = S[f \setminus a]$$

provided $f \cap \text{def}(S) = \emptyset$.

Summary

- design decisions can be changed cheaply (sometimes) through refactoring transformations
- once proven correct, the transformations support a formal process of program re-design
 - promoting reusable code
 - improving the program's maintainability
- Lakhotia's version of Extract Slice (Tuck transformation) is too conservative
 - our solution is based on passing parameters by value
 - the price: duplicated code
 - further (post) refactorings can minimize that

Thank you!

Definition of a program slice

- For any statement S and set of variables V , $\text{slice}(S, V)$ is any program satisfying:
for any element var in V , and element val of the type of (i.e. set of possible values of) var :
 - $\text{wlp}.S.(var=val) \equiv \text{wlp}.\text{slice}(S, V).(var=val)$
 - $\text{wp}.S.\text{true} \Rightarrow \text{wp}.\text{slice}(S, V).\text{true}$