Programming in UNITY

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Maximum of a Set of Numbers

- Let $A[N \ldots N-1]$ be an array of integers.
- Compute m satisfying
  - $m = \langle \mathsf{max}\ i\colon 0 \leq i < N :: A[i] \rangle$
- First UNITY solution
  - $m = \langle \mathsf{max}\ i\colon 0 \leq i < N :: A[i] \rangle$
- Refined solution for sequential architectures:
  - **invariant** $m \leq \langle \mathsf{max}\ i : 0 \leq i < N :: A[i] \rangle$
  - *FP* $\equiv m \geq \langle \mathsf{max}\ i : 0 \leq i < N :: A[i] \rangle$
  - *FP* $\equiv \langle \wedge i\colon 0 \leq i < N :: m \geq A[i] \rangle$
  - *FP* $\equiv \langle \wedge i\colon 0 \leq i < N :: m = \mathsf{max}(m, A[i]) \rangle$

Program *Maximum1*
  initially m = -∞
  assign $\langle [] \ i\colon 0 \leq i < N :: m := \mathsf{max}(m, A[i]) \rangle$
end {*Maximum1*}

Parallel Solutions

- Consider the elements of A as the leaf nodes of a binary tree.

  Program *Maximum2*
  declare X: array [1...(2**N*-1)] of integer
  always
  $\langle \| i : 0 \leq i < N ::$
  $X[N+i] = A[i] \rangle$
  [] $\langle [] \; i : 1 \leq i < N :: X[i] =$
  $\max(X[2*i], X[2*i+1]) \rangle$
  end *Maximum2*

  – $X[1] = \langle \max i\text{: } 0 \leq i < N :: A[i] \rangle$

- Improved version

  Program *Maximum3*
  assign $\langle \| \; i : 0 \leq i < M ::$
  $A[i] := \max(A[2*i], A[2*i+1]) \rangle$

Parallel Solutions

Program *Maximum4*
  declare t: integer
  initially t=*N*
  assign
    $\langle \| \; i : 0 \leq i < t/2 ::$
      $A[i] := \max(A[2*i], A[2*i+1])\rangle$
    $\| t := t/2$
end $\{$*Maximum4*$\}$

- Let A^0 be the initial value of A.
- **invariant** $\langle \max\ i:\ 0 \leq i < t :: A[i]\rangle = \langle \max\ i:\ 0 \leq i < N :: A^0[i]\rangle$
- $t = 1 \Rightarrow A[0] = \langle \max\ i:\ 0 \leq i < N :: A^0[i]\rangle$

Gaussian Elimination

Program *Gauss*
  assign
    $\langle [] \; u :: \{\text{pivot with row } u\}$
      $\langle \| \; v, j : v \neq u ::$
        $M[v,j] := M[v,j] -$
          $M[v,u] * M[u,j] / M[u,u]$
          if $M[u,u] \neq 0 \rangle$
    $\| \; \langle \| \; j ::$
        $M[u,j] := M[u,j]/M[u,u]$, if $M[u,u] \neq 0 \rangle \rangle$
  $[] \; \{\text{exchange rows}\}$
    $\langle \| \; u, v : u \neq v ::$
      $\langle \| \; j :: M[u,j], M[v,j] := M[v,j], M[u,j]$
        if $M[u,u] = 0 \wedge M[v,v] = 0 \wedge$
          $(M[u,v] \neq 0 \vee M[v,u] \neq 0) \rangle \rangle$
end $\{$*Gauss*$\}$

- Solve linear equation system $AX = B$.
- Gaussian Elimination on $M = (A, B)$
  - Exchange two rows.
  - Pivot one row with another.

Correctness

- **invariant** M, M^0 have same solutions.
- *FP* $\Rightarrow (M[u,u] = 0 \Rightarrow M[u,v] = 0)$
- *FP* $\Rightarrow (M[u,u] \neq 0 \Rightarrow u$ is a unit column)
  - Diagonal element is 1 and all other elements are zero.
- *FP* is reached:
  - (m,n) decreases lexicographically with every state change.
  - $m =$ number of unit columns in A.
  - $n =$ number of nonzero diagonals in A.

For efficient implementation, non-determinism has to be restricted (explicit ordering, mapping of processes, etc)

Program Structuring

- Composition of UNITY programs.
  - Large program consisting of small components.
  - Each component developed and understood by itself.
- *Union* of programs.
  - Appending codes together.
  - Commutative, associative program operation.
  - Networks of parallel processes.
- *Superposition* of programs.
  - Modification of a given program.
  - Add new variables and assignments.
  - Preserves all properties of original program.
  - Useful in building programs in *layers*.

Program Composition by Union

- F [] G
  - *Component* programs F, G.
  - *Composite* program F [] G.
- Components can name common variables.
- Append corresponding sections of two components.
  - Assume that there are no inconsistencies in definitions of variables, always-sections, or initializations.

The Union Theorem

- p *unless* q in F [] $G \equiv$
  (p *unless* q in $F \wedge p$ *unless* q in G)
  - If p is true and q is not, p remains true or q becomes true.
- p *ensures* q in F [] $G \equiv$
  [p *ensures* q in $F \wedge p$ *unless* q in G] $\vee$
  [p *ensures* q in $G \wedge p$ *unless* q in F]
  - If p is true, p remains true as long as q is false and eventually q becomes true.
- (*FP* of F [] G) $\equiv$
  (*FP* of F) $\wedge$ (*FP* of G)
  - State does not change any more.

Proof of Union Theorem

Only second case is not straight-forward.

$$
\begin{array}{l}
p \text{ ensures } q \text{ in } F \,[]\, G \\
\equiv p \textit{ unless } q \text{ in } F \,[]\, G \\
\quad \wedge \langle \exists s : s \text{ in } F \,[]\, q :: \{p \wedge \neg q\}\ s\ \{q\} \rangle \\
\equiv p \textit{ unless } q \text{ in } F \,[]\, G \\
\quad \wedge [\langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\} \rangle \\
\qquad \vee \langle \exists s : s \text{ in } G :: \{p \wedge \neg q\}\ s\ \{q\} \rangle] \\
\equiv [p \textit{ unless } q \text{ in } F \,[]\, G \\
\qquad \wedge \langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\} \rangle] \vee \\
\quad [p \textit{ unless } q \text{ in } F \,[]\, G \\
\qquad \wedge \langle \exists s : s \text{ in } G :: \{p \wedge \neg q\}\ s\ \{q\} \rangle] \\
\equiv [p \textit{ unless } q \text{ in } F \wedge p \textit{ unless } q \text{ in } G \\
\qquad \wedge \langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\} \rangle] \vee \\
\quad [p \textit{ unless } q \text{ in } F \wedge p \textit{ unless } q \text{ in } G \\
\qquad \wedge \langle \exists s : s \text{ in } G :: \{p \wedge \neg q\}\ s\ \{q\} \rangle] \\
\equiv [p \textit{ ensures } q \text{ in } F \wedge p \textit{ unless } q \text{ in } G] \vee \\
\quad [p \textit{ ensures } q \text{ in } G \wedge p \textit{ unless } q \text{ in } F]
\end{array}
$$

Collaries

- p is stable in F [] $G \equiv$
  (p is stable in $F \wedge p$ is stable in G)
- $$\frac{p \textit{ unless } q \text{ in } F \quad p \text{ is stable in } G}{p \textit{ unless } q \text{ in } F \text{ [] } G}$$
- $$\frac{p \text{ is invariant } q \text{ in } F \quad p \text{ is stable in } G}{p \text{ is invariant } q \text{ in } F \text{ [] } G}$$
- $$\frac{p \textit{ ensures } q \text{ in } F \quad p \text{ is stable in } G}{p \textit{ ensures } q \text{ in } F \text{ [] } G}$$
- Locality:
  - p is a local predicate of F, if it only mentions variables that can be modified by F alone.
  - If p is a local predicate of F, then the following properties hold in F [] G, for any G: p *unless* q, p *ensures* q, p is invariant.

Specifications that Compose

- Limitation:
  - $p \mapsto q$ holds in F, G.
  - $p \mapsto q$ need *not* hold in F [] G!.
- Specification of program *components*.
  - Given: a program component F.
  - Wanted: properties of F [] G (for unknown G).
- Technique: *conditional properties*.
  - Two parts (sets of unconditional properties).
  - Hypothesis: specification of system (F [] G).
  - Conclusion: effect of embedding F into system.
  - Given hypothesis as premise.
  - Conclusion can be proven from F alone.

Example

Program F
  declare x: integer
  assign
    $y := -y$, if $x \leq 0 \wedge y > 0$
  [] $x := x - 1$
end $\{F\}$

- Unconditional properties:
  - $y < 0$ is stable in F.
  - $y = 0$ is stable in F.
  - $y \neg 0$ is stable in F.
- Conditional properties:
  - Hypothesis: $y \leq 0$ is stable in F [] G.
  - Conclusion: $y > 0 \mapsto y < 0$ in F [] G
- G shares only y with F.

Strong conclusion can be drawn from weak hypothesis.

Hierarchical Program Structures

- UNITY program can be union of several components programs.
  - Components can be themselves unions of other programs.
  - Hierarchical structures of arbitrary depth are constructed.
- Power of abstraction
  - Program described by externally observable properties.
  - Implementation is hidden.
  - Simplicity of program design.
- All UNITY statements are treated alike:
  - Each statement is executed infinitely often.
  - Depth in hierachy is meaningless.
  - Modularity: variables may be hidden in hierarchy.

Superposition

- Structure program as set of "layers".
  - Higher layer can use services of lower layers.
  - Lower layer never calls higher layers.
  - Example: application program and operating system.
- *Superposition* of programs.
  - *Underlying* program (and variables).
  - Transform program such that its properties are preserved.
  - *Transformed* program has aditional properties.
  - Transformation introduces *superposed* variables.
  - Assignments to underlying variables remain unaffected.

Rules of Superposition

Program name
initially initial values of superposed variables
transform effect of augmentation rule
add effect of restricted union rule

- Augmentation Rule:
  - Statement s in underlying program may be transformed to $s \parallel r$ where r does not assign to underlying variables.
- Restricted Union Rule:
  - Statement r may be added to underlying program provided that r does not assign to underlying variables.

The Superposition Theorem

Every property of the underlying program is a property of the transformed program.

- Augmentation:
  - If $\{p\}s\{q\}$ holds for s of underlying program, $\{p\}s \parallel r\{q\}$ also holds because p, q name no variables assigned in r. Since furthermore initial condition of underlying program is preserved, all proofs are preserved.
- Restricted Union:
  - Added statements do not write into underlying variables. From locality corollary and the fact that underlying program does not mention superposed variables in its proof, all proofs are preserved.

Examples

Augmentation

Program *Detection0*
  initially $\mathit{count}, \mathit{claim} = 0, \mathit{false}$
  transform each s to
    $s \parallel \mathit{count}, \mathit{claim} := \mathit{count}+1, \mathit{count} \geq 10$
end $\{\mathit{Detection0}\}$

Restricted Union

Program *Detection1*
  initially $\mathit{count}, \mathit{claim} = 0, \mathit{false}$
  transform each s to
    $s \parallel \mathit{count} := \mathit{count}+1 \geq 10$
  add $\mathit{claim} := \mathit{count} > 10$
end $\{\mathit{Detection0}\}$

Properties

- **invariant** *count* = number of statement executions in underlying program.
- **invariant** *claim* $\equiv$ (*count* > 10)
- *claim detects* (number of statement executions in the underlying program exceeds 10).
  - p *detects* $q \equiv (p \Rightarrow q) \wedge (q \mapsto p)$
  - p holds within a finite time of q holding, and if p holds, then so does q.

Summary

The UNITY Model

- Design of and reasoning about parallel programs.
- Step-wise refinements of programs.
- Axiomatic logic for reasoning.
- Core: *non-deterministic* composition.
  - Advantage: simple rules for program composition by union.
  - Disadvantage: does not match real languages based on *sequential* composition.

Abstraction as an advantage and as a disadvantage!