Research Summary: A Programming Methodology and A Reliability Mechanism

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-008
Research Summary:
A Programming Methodology and A Reliability Mechanism

1. Programming as Planning

Programming tasks can be viewed from the AI viewpoint of automated planning. The meanings of programs (the declarative part) are considered goals, and the implementations (the imperative part) are actions that try to realize those goals. The planning problem is optimizable via heuristics that determine the right action (or set of plausible ones) to take at any given step in the execution. Our old friend—imperative program—is viewed as a fully optimized planning problem, where every action at every step of the program is predetermined. Effectively, the normal execution of an imperative program is traded for a higher level of control, which dynamically consults heuristics to determine the next instruction, or explore multiple possibilities by means of exhaustive search. The approach suffers from the overhead of a method call to determine the right dispatch at every step. On the other hand, it brings about the interesting possibility of non-deterministic computation by listing multiple actions to explore. Optimizations may be introduced to narrow down the possible actions to try at any given point, as well as heuristics that describe the preferred order in which to explore those possibilities.

The main advantage is that existing programs can adapt the methodology without much effort. A prototype extension of LLVM compiler was created that adapted the scheme to do optimal register allocation, a task reducible to graph coloring and NP-complete. The resulting register allocator program was substantially more compact and readable than the equivalent sections in the original C++ code.
Summary of AllocatorX86.st:
Programming-as-Planning in an Allocator Smalltalk program

{ john-program }
start-world allocatorx86.

;; CLASSES
create Allocator unallocated-variables.
create Variable length live-ranges.
create Register next-right allocations.

;; QUALIFICATIONS
qualify Word repeatn: N [] if N = 0.
qualify Word repeatn: N (its repeatn: (N - 1)) + it.
qualify Variable live-range [its live-ranges first first first first / 2, its live-ranges first second / 2]. ;; fixme
qualify Variable repeatn: N [] if N = 0.
qualify Variable repeatn: N (its repeatn: (N - 1)) + it.
qualify Variable repeatlist it repeatn: (its live-range second - its live-range first).
qualify Allocator next-variable its unallocated-variables first.
qualify Allocator next-variable-start its next-variable live-range first.

qualify Allocator canAllocate: Variable if
for any all Register do each canAllocate: Variable = yes.
qualify Register allocationWith: Variable
  its allocations to: (Variable live-range first - 1) +
  Variable repeatlist + its allocations from: (Variable live-range last).
qualify Register canAllocate: Variable if
  (Variable length = 2 or
   (not its next-right = no and
    for every Variable live-range do its next-right allocations at: each = empty)) and
    for every Variable live-range do its allocations at: each = empty.
\[\text{The Allocator program written in planning language JOHN}\]
2. Executable Specifications

Our study shows formal specifications can realize an instance of Marvin Minsky's B brains visions. He proposed a system with multilevel control logic, where the higher levels, say B brain, may take over the control, or affect the execution of the normal logic, the A brain. One can view meanings and specifications as B brains, and normal implementations as A brains. We observed three scenarios where the declarative specifications may want to take over the implementations: 1. Accidentally due to a run-time error 2. Accidentally due to the implementations failing to satisfy the specifications (post-conditions) 3. Intentionally where implementations explicitly yield the control to the specifications to save lines of code, avoid complex corner cases, etc.

<table>
<thead>
<tr>
<th>A brain: Implementation</th>
<th>B brain: Specifications</th>
</tr>
</thead>
</table>

Three scenarios for implementation to specifications fallback

The specifications, which cover global class invariants as well as post-conditions for methods, are usually expressed in high level syntax involving first-order logic expressions. It is desirable for the specifications to be written in the same language as the rest of the program, so that they can be run and tested like normal methods. Thus syntactic sugars may be supported for these expressions, including quantified expressions, as well as field closures useful for recursive data structures. But support for high-level expressions in the specifications is not just a matter of providing more readable and mathematical formed code. Such expressions carry the semantics necessary to enable translating them into formulas in the syntax of constraint solvers. In the event of a fallback to meanings, we use external constraint solvers to extract a satisfying model to the problem expressed in the specifications and apply the model to the object.

A Binary Search Tree program is shown below with insert and delete methods in the Java language extended with the high-level function methods. The two operations are fully runnable without any implementations, only relying on the specifications accompanying them denoted with the ensures keyword. The .^ and .* symbols denote reflexive and non-reflexive field closures and .< is a map field-get over a set of objects.
public class BSTree
    ensures isAcyclic() && oneParent() && isValidBinarySearch()
{
    ...
    function public ESJSet<Node> nodes() { root.*(left+right) }
    function public ESJSet<Integer> nodeValues() { this.nodes().<value }
    function public boolean isAcyclic() { no Node n | n.descendants().contains(n) }
    function public boolean oneParent() {
        (root == null) ?
        true:
        all Node n : this.root.*(left+right) | one Node p | n.parent == p
    }
    function public boolean isValidBinarySearch() {
        all Node n |
        (n.left == null || all Node lc : n.left.*(left+right) | lc.value < n.value) &&
        (n.right == null || all Node rc : n.right.*(left+right) | rc.value > n.value))
    }
    public void insert(Node insertedNode)
        modifies BSTree.root, Node.left, Node.right, Node.parent
        ensures this.nodes().equals(this.old.nodes().union(insertedNode)) {
            // no implementation...
    }
    public void delete(Integer key)
        modifies BSTree.root, Node.left, Node.right, Node.parent
        ensures this.nodeValues().equals(this.old.nodeValues().minus(key)) {
            // no implementation...
    }
}

A fully runnable Binary search tree program with insert and delete operations, only relying on specifications