Matching Logic: A New Program Verification Approach

Grigore Roşu Andrei Ştefănescu
University of Illinois at Urbana-Champaign

Abstract

Matching logic is a new program verification logic, which builds upon operational semantics. Matching logic specifications are constrained symbolic program configurations, called *patterns*, which can be *matched* by concrete configurations. By building upon an operational semantics of the language and allowing specifications to directly refer to the structure of the configuration, matching logic has at least three benefits that could be key factors in its usability: (1) One's familiarity with the formalism reduces to one's familiarity with the operational semantics of the language, that is, with the language itself; (2) The verification process proceeds the same way as the execution of the program, making debugging failed proof attempts manageable because one can always see the "current configuration" and "what went wrong", same like in a debugger; and (3) Nothing is lost in translation, that is, there is no gap between the language itself and its verifier. Moreover, direct access to the structure of the configuration facilitates defining sub-patterns that one may reason about, such as disjoint lists or trees in the heap, as well as supporting framing in various components of the configuration at no additional costs.

1 Introduction

An attempt to make program verification more usable could be to develop novel foundations for program verification and corresponding tools, in a hope that some, if not most of the limitations of the existing program verification approaches can be avoided from the root. For example, could it be possible that other program verification logics, conceptually different from Floyd-Hoare logic [4, 2], or separation logic [5, 6], or dynamic logic [3], wait to be unearthed in order to, at least temporarily, make program verification more feasible and accessible to non-experts? Common scientific sense tells that the answer is more likely positive than negative, though one would, of course, expect intrinsic connections between such a new logic and the existing ones. Without any claim that it overcomes all the difficult problems of the existing approaches, this position paper advances the possibility that *matching logic* [7] could be a promising novel program verification logic.

Matching logic builds upon operational semantics. To use it, one must understand at least the structure of the configurations that are used in the operational semantics of the language. For example, the configuration of some language may contain, besides the code itself, an environment, a heap, several stacks, synchronization resources, etc. Matching logic specifications, which are called *patterns*, allow one to refer directly to the configuration of the program. For example, the pattern

$$\langle \quad \langle \mathsf{root} \mapsto \mathsf{?root},\ \mathsf{E} \rangle_{\mathrm{env}} \ \ \langle \mathsf{tree(?root)(T)},\ \mathsf{H} \rangle_{\mathrm{heap}} \ \ \mathsf{C} \quad \rangle_{\mathrm{config}}$$

specifies the set of program configurations in which the program variable root points to tree T. More precisely, it says the following: (1) the configuration $\langle \ldots \rangle_{\mathrm{config}}$ must contain at least an environment cell and a heap cell, and the rest of the configuration is matched by C (the configuration frame); (2) the environment $\langle \ldots \rangle_{\text{env}}$ must hold at least the binding $\mathsf{root} \mapsto \mathsf{?root}$, and the rest of the environment is matched by E (the environment frame); (3) the heap $\langle \ldots \rangle_{\text{heap}}$ must hold at least the term $\mathsf{tree(?root)(T)}$, and the rest of the heap is matched by H. The term $\mathsf{tree(?root)(T)}$ matches a portion of the heap that contains a flattened representation of the tree T (as a mathematical object) in memory. As seen shortly, such terms are not defined, they are axiomatized.

Matching logic patterns can be defined as first-order logic formulas [7], but ones over an extended signature that includes all the constructs for configurations and everything they can hold, such as lists, sets, maps, trees, etc. Then, variables like $\mathsf{?root}$ can be regarded as existentially quantified over the pattern, while E, T, H, C are free variables. From a matching logic perspective, unlike in other program verification logics, program variables like root are *not* logical variables; they are simple syntactic constants. The matching logic derivation rules are nothing else but the operational semantic rules (allowed to work on configurations with variables). For example, an assignment statement "$\mathsf{x = 5}$" changes the pattern above into the pattern

$$\langle \quad \langle \mathsf{root} \mapsto \mathsf{5},\ \mathsf{E} \rangle_{\text{env}} \ \ \langle \mathsf{tree(?root)(T)},\ \mathsf{H} \rangle_{\text{heap}} \ \ \mathsf{C} \quad \rangle_{\text{config}}$$

There is no backwards substitution like in the Hoare rule for assignment, as well as no introduction of existential quantifiers like in the Floyd rule for assignment, though, technically speaking, one can (and should!) relate the matching logic proof system to the Floyd-Hoare one, as shown in [7].

The variables appearing in patterns often need to be constrained. Since patterns are just FOL formulas, one can simply conjunct them with a formula expressing the desired constraints. For notational uniformity, we prefer to write the constraints inside a special cell $\langle \ldots \rangle_{\text{form}}$ that we add to patterns (such a cell was not necessary in the original program configurations). For example:

$$\langle \quad \langle \mathsf{root} \mapsto \mathsf{?root},\ \mathsf{E} \rangle_{\text{env}} \ \ \langle \mathsf{tree(?root)(T)},\ \mathsf{H} \rangle_{\text{heap}} \ \ \langle \mathsf{T} \neq \mathsf{empty} \rangle_{\text{form}} \quad \mathsf{C} \quad \rangle_{\text{config}}$$

Unlike in separation logic, there is no need for logical support for "separation". In matching logic, separation is achieved at the structural (i.e., term) level and not at the logical level. In algebraic specification and term rewriting, separation at the term level was always understood, without anything special to say about it. For example, if one matches two terms in a multiset, then the two terms are obviously distinct. In our pattern above, the implicit structural separation tells us that the binding of root is separated from the rest of the environment E, the term $\mathsf{tree(?root)(T)}$ is separated from the rest of the heap H, and that the three mentioned cells are separated from the rest of the configuration C.

In matching logic, framing can appear in any cell of the configuration. Same like separation, it needs no special support from the logical infrastructure, in particular no derivation rules. Cell framing simply falls under the general principle of matching. Consider, for example, the assignment "$\mathsf{x{=}5}$" discussed above, which changed the first pattern discussed above into the second, and the free variable H. Since the same H variable appears free in both patterns, it must match the same term. In other words, any concrete program configuration that matches the first pattern will induce a binding for H, which will be the same in the configuration after the assignment.

We implemented a proof-of-concept matching logic verifier for a fragment of C, called MatchC, which can automatically verify programs like the one discussed next. A web interface is available at `http://fsl.cs.uiuc.edu/ml`, together with examples. MatchC is based on an executable rewrite-based semantics of the fragment of C. Both the executable semantics and the verifier are implemented using the $\mathbb{K}$ language definitional framework [8], which compiles into Maude [1].

```
struct treeNode { int val;  struct treeNode *left;  struct treeNode *right; };
struct nodeList { int val;  struct nodeList *next; };
struct treeNodeList { struct treeNode *val; struct treeNodeList *next; };

struct nodeList *toListIterative(struct treeNode *root)
 /*@ pre   ⟨  ⟨root ↦ ?root⟩_env  ⟨tree(?root)(T), H⟩_heap  ⟨TrueFormula⟩_form  C  ⟩_config
 /*@ post  ⟨  ⟨?rho⟩_env  ⟨list(?a)(tree2list(T)), H⟩_heap  ⟨returns ?a⟩_form  C  ⟩_config
{
  struct nodeList *a;  struct nodeList *node; struct treeNode *t;
  struct treeNodeList *stack;  struct treeNodeList *x;
  if (root == 0) return 0;
  a = 0;
  stack = (struct treeNodeList *) malloc(sizeof(struct treeNodeList));
  stack->val = root; stack->next = 0;

 /*@ invariant  ⟨  ⟨root ↦ ?root, a ↦ ?a, stack ↦ ?stack, t ↦ ?t, x ↦ ?x, node ↦ ?node⟩_env
                   ⟨list{tree}(?stack)(?TS), list(?a)(?A), H⟩_heap
                   ⟨tree2list(T) = list{tree}2list(rev(?TS))@?A⟩_form  C  ⟩_config

  while (stack != 0) {
    x = stack;  stack = stack->next; t = x->val;
    free(x);
    if (t->left != 0) {
      x = (struct treeNodeList *) malloc(sizeof(struct treeNodeList));
      x->val = t->left; x->next = stack; stack = x;
    }
    if (t->right != 0) {
      x = (struct treeNodeList *) malloc(sizeof(struct treeNodeList));
      x->val = t; x->next = stack; stack = x;
      x = (struct treeNodeList *) malloc(sizeof(struct treeNodeList));
      x->val = t->right; x->next = stack; stack = x;
      t->left = t->right = 0;
    }
    else {
      node = (struct nodeList *) malloc(sizeof(struct nodeList));
      node->val = t->val; node->next = a; a = node;
      free(t);
    }
  }
  return a;
}
```

Figure 1: Iterative C program flattening a tree into a list: traverses the tree in infix order and, as it reaches each tree node, it deallocates it and allocates a corresponding list node. The matching logic annotations state the full correctness of this program: the tree is completely deallocated, the resulting list is allocated and contains exactly the same elements in the desired order, and that nothing else changes. MatchC automatically verifies the annotated program above in milliseconds.

2 An Example: Iterative Flattening of a Tree into a List

Figure 1 shows a C function that flattens a binary tree into a list, traversing the tree in infix order. Each node of the initial tree (structure treeNode) has three fields: the value, and two pointers, for the left and the right subtrees. Each node of the final list (structure nodeList) has two fields: the value and a pointer to the next node of the list. The program makes use of an auxiliary structure (treeNodeList) to represent a stack of trees. For demonstration purposes (to highlight the invariant capability of our verifier), we prefer an iterative version of this program. We need a stack to keep track of our position in the tree. Initially that stack contains the tree passed as argument (as a pointer). The loop repeatedly pops a tree from the stack, and it either pushes back the left tree, the root, and the right tree onto the stack, or if the right tree is empty it pushes back the left subtree and appends the value in the root node at the beginning of the list of tree elements. As the loop processes the tree, it frees the tree nodes and it allocates the corresponding list nodes.

The function is annotated with pre and post pattern conditions. The precondition binds the argument root to the pattern existential variable $\mathsf{?root}$ in the environment, which points to a binary tree holding the contents T; the variable T is free in both the pre and the post conditions, indicating that it must be bound to the same tree (defined algebraically in a trivial way; see below). There are two other free variables in the pre and the post conditions, H and C, which will be bound to the remaining contents of the heap and the configuration cells, that is, to their corresponding frames. No frame for the environment is necessary in the precondition, because at that point in the program root is the only variable in the environment. The postcondition binds the pattern existential variable $\mathsf{?rho}$ to whatever is in the environment when the function returns (we do not care about that information in this example), and binds $\mathsf{?a}$ to the return value of the function, which points to a list with contents $\mathsf{tree2list(T)}$ (the infix traversal sequence of T, also trivially defined algebraically). In matching logic, list, tree, and $\mathsf{tree2list}$ are ordinary operation symbols added to the signature and constrained through axioms as discussed shortly.

When a function is verified, its precondition is assumed as "the" configuration (but, of course, it is symbolic and constrained) in which its body is executed using the operational semantics of the language. The semantics is extended with pattern assertions as expected: when a pattern assertion is encountered, MatchC attempts to prove that the current (symbolic and constrained) configuration matches the asserted pattern. The verification fails if any such match fails. One can assert patterns anywhere in the program. The postcondition is automatically asserted at function return. We call asserted patterns invariants when they are associated to loops. An invariant pattern always holds before each loop iteration (we only consider partial correctness for the time being). The invariant of our program binds all program variables to pattern existential variables, asserts that the heap contains a stack of trees (represented as a list of trees) with contents $\mathsf{?TS}$ and a list with contents $\mathsf{?A}$, and that the infix traversal sequence of T, $\mathsf{tree2list(T)}$, is equal to the concatenation in reverse order of the infix traversal sequences of the trees in the stack concatenated with the contents of the list. All these operations on trees and lists are axiomatized below. The remainder of the heap (H) and of the configuration (C) in the invariant happened to stay unchanged.

We next list the axioms that we had to add in the mathematical library of MatchC in order to verify the program above automatically. Here is our axiom for lists, which is a first-order formula:

$$\begin{array}{lcl}\langle\langle \mathsf{list}(\mathsf{p},\alpha),\mathsf{H}\rangle_{\text{heap}}\langle\phi\rangle_{\text{form}}\mathsf{C}\rangle_{config} & \Leftrightarrow & \langle\langle \mathsf{H}\rangle_{\text{heap}}\langle \mathsf{p}=0\wedge\alpha=\mathsf{nil}\wedge\phi\rangle_{\text{form}}\mathsf{C}\rangle_{\text{config}}\\ & \vee & \langle\langle \mathsf{p}\mapsto[\mathsf{?a},\mathsf{?q}],\mathsf{list}(\mathsf{?q},?\beta),\mathsf{H}\rangle_{\text{heap}}\langle\alpha=[\mathsf{?a}]@?\beta\wedge\phi\rangle_{\text{form}}\mathsf{C}\rangle_{\text{config}}\end{array}$$

The above captures the two cases, one in which the list is empty and the other in which it has at least one element. We borrowed the notation $\mathsf{p} \mapsto [\mathsf{?a}, \mathsf{?q}]$ from separation logic; it stays for two bindings, namely for "$\mathsf{p} \mapsto \mathsf{?a},\ \mathsf{p{+}1} \mapsto \mathsf{?q}$". we also borrowed the notation for lists from OCAML: $[\mathsf{?a}]$ is a one-element list and @ concatenates two lists. All the non-existential variables in the axiom above are assumed universally quantified, not free; in other words, the H and C variables in this axiom have nothing to do with the homonymous variables in the program annotations.

The tree pattern is axiomatized similarly ($\mathit{tree}(a, \tau_l, \tau_r)$ is our constructor for trees):

$$\begin{array}{rcll}
\langle\langle \mathsf{tree}(\mathsf{p}, \tau), \mathsf{H}\rangle_{\mathrm{heap}} \langle\phi\rangle_{\mathrm{form}} \mathsf{C}\rangle_{config} & \Leftrightarrow & \langle & \langle \mathsf{H}\rangle_{\mathrm{heap}} \langle \mathsf{p} = 0 \wedge \tau = \mathsf{empty} \wedge \phi\rangle_{\mathrm{form}} \mathsf{C}\rangle_{\mathrm{config}} \\
 & \vee & \langle & \langle \mathsf{p} \mapsto [\mathsf{?a}, \mathsf{?l}, \mathsf{?r}], \mathsf{tree}(\mathsf{?l}, ?\tau_l), \mathsf{tree}(\mathsf{?r}, ?\tau_r), \mathsf{H}\rangle_{\mathrm{heap}} \\
 & & & \langle \tau = \mathit{tree}(\mathsf{?a}, ?\tau_l, ?\tau_r) \wedge \phi\rangle_{\mathrm{form}} \mathsf{C}\rangle_{\mathrm{config}}
\end{array}$$

The axiomatization for $\mathsf{list\{tree\}}$ is similar to that of list, but the data field is a pointer to a tree.

The only thing left to show is our axioms for reasoning within the mathematical domains that provided the data stored in the heap patterns above. The equations above are self-explanatory; we only mention that rewrite engines like Maude are quite suitable for handling such axiomatizations:

$$\begin{array}{rcl}
\mathsf{rev}(\mathsf{nil}) & = & \mathsf{nil} \\
\mathsf{rev}([a]) & = & [a] \\
\mathsf{rev}(\mathsf{A}_1 @ \mathsf{A}_2) & = & \mathsf{rev}(\mathsf{A}_1) @ \mathsf{rev}(\mathsf{A}_2)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{tree2list}(\mathsf{empty}) & = & \mathsf{nil} \\
\mathsf{tree2list}(\mathit{tree}(\mathsf{a}, \tau_l, \tau_r)) & = & \mathsf{tree2list}(\tau_l) @ [\mathsf{a}] @ \mathsf{tree2list}(\tau_r)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{list\{tree\}2list}(\mathsf{nil}) & = & \mathsf{nil} \\
\mathsf{list\{tree\}2list}([\tau]) & = & \mathsf{tree2list}(\tau) \\
\mathsf{list\{tree\}2list}(\mathsf{A}_1 @ \mathsf{A}_2) & = & \mathsf{list\{tree\}2list}(\mathsf{A}_1) @ \mathsf{list\{tree\}2list}(\mathsf{A}_2)
\end{array}$$

References

[1] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[2] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[3] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969.

[5] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5:215–244, 1999.

[6] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[7] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486. LNCS, 2010.

[8] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.