

Symbolic Problem Solving With Bidirectional Executable Representations

Jakob Praher¹

RISC Research Institute for Symbolic Computation, JKU Linz

1 Introduction

Programs are typically written by humans and executed by computers in order to solve problems over a specific domain. A program - a sequence of instructions executable by a computer - represents the encoded algorithm for solving the problem. The algorithm and data structures are created by the mental understanding of the problem given. This representation also addresses details such as run time efficiency, understandability, or maintainability.

Conversely when a programmer tries to understand a program he/she decodes the program again into an abstract mental model. From this model the programmer can start to improve or change the program, which is again encoded and represented as a modified program.

In this paper we show a problem solving process that starts with finding, enhancing or inventing a logical theory over first order predicate logic. This theory represents the concepts of the problem domain, relations between the concepts, and operations. A theory is a set of formulae. Operations on terms of the theory correspond to functionality of the solution. These operations can be characterized by partial functions from given input to expected output. Requirements formally express the functionality of the operations and are commonly defined using input/output conditions. If an operation's input satisfies the input condition then operation's output (if obtainable) satisfies the output condition.

The process is incremental. It is important to capture the high level requirements. For representing the solution as a machine executable program, the theory should be sufficiently refined so that the requirements satisfy the problem and capture important properties of the execution environment. Detailed conditions make it possible to formally evaluate different design approaches. Design approach refers to the abstractions and methods used for framing the algorithms. As an example in the hash table design it is important to capture the concept of collisions in the table abstractly in order to compare different strategies how to handle collisions.

At the end of the process should again exist a program executable by a computer. The theory therefore is represented and implemented in a formal language representing the computer architecture. The elements of the theory are retained during this process. Data representation for the terms is defined and every operation specified in the theory is implemented in form of a computer program (or subprogram). The program representation of an operation computes the output based on the input (in form of the represented terms). Correctness of the implementation can be shown by giving the program a logical

interpretation. The result is a set of formulae for which the corresponding input/output conditions can be proven. This bidirectional representation can be used to proof other meta-theorems about the program.

2 Logical Theory

As an example we will look at the problem of associative lookup. The object theory defines the following concepts (monadic predicate symbols): *Table*, *Key*, *Value*. In the following we use the syntactic convention that t implies that t is of concept *Table*, k assumes *Key*(k), v assumes *Value*(v). Furthermore we use variants of the names such as t' , t_i to also refer to t .

Abstractly an associative table t is a set of key value pairs with the restriction that for every key k there exists at most one element in the set. Therefore a table can be described as a function from keys to values.

$$t = \{\langle k_1, v_1 \rangle, \langle k_2, v_2 \rangle \dots \langle k_n, v_n \rangle\}$$

The operations that we want to perform on an associative table are: *insert* elements into the table, *update* elements, *remove* elements by a given key, *find* the element according to a key. These operations take certain input and return an output. These operations are deterministic.

The input(I) and output(O) conditions for the operations are given below:

$$\begin{aligned} I_{insert}(t, k, v) &\Leftrightarrow Table(t) \wedge Key(k) \wedge Value(v) \\ O_{insert}(t, k, v, t') &\Leftrightarrow Table(t') \wedge \langle k, v \rangle \in t' \\ I_{update}(t, k, v) &\Leftrightarrow Table(t) \wedge Key(k) \wedge Value(v) \\ O_{update}(t, k, v, t') &\Leftrightarrow Table(t') \wedge \langle k, v \rangle \in t' \\ I_{remove}(t, k, v) &\Leftrightarrow Table(t) \wedge Key(k) \wedge Value(v) \\ O_{remove}(t, k, v, t') &\Leftrightarrow Table(t') \wedge \langle k, v \rangle \notin t' \\ I_{find}(t, k) &\Leftrightarrow Table(t) \wedge Key(k) \wedge k \in t \\ O_{find}(t, k, v) &\Leftrightarrow Value(v) \implies t(k) = v \end{aligned}$$

3 Refining the Theory

A suitable method for addressing by value is by means of a *hash* function. The associative container using *hash* representations for addressing elements is called a hash table. One or more *hash* functions are used to map a key to an integer within a defined range. The table has a defined capacity C and a size S that states the maximum number of elements to store without having to expand the capacity and the current number of elements in the table. *elements* represents the sequence of elements of a hash table t . A hash code of a key is a digest, that might be lossy, that means a *hash* function is not injective. The table has to take into account that two different keys k_1 , k_2 share the

same hash code. The actual position in the table depends not only on the hash code of the key but also on the current capacity of the table. A collision resolution method for the concrete table layout is needed such that two different keys are mapped to different locations in the table.

For the input/output conditions additional properties and functions are defined in the theory. These concepts serve the purpose of describing the functionality in terms of the input/output conditions without anticipating the concrete layout or collision resolution/probing. They are initially described informally:

$has_capacity_for(t, k)$	Holds if the table has capacity for the key k .
$contains(t, k)$	Holds if the table t contains an element with key k .
$element(t, k)$	Returns the element for k if $contains(t, k)$.
$expand(t)$	Returns table t increased to hold at least one more element.

The following shows exemplary the input and output conditions for the insert operation:

$$I_{insert}(t, k, v) \Leftrightarrow HashTable(t) \wedge Key(k) \wedge Value(v)$$

$$contains(t, k) \implies t = t'$$

$$O_{insert}(t, k, v, t') \Leftrightarrow \bigwedge \neg contains(t, k) \wedge has_capacity(t, k) \implies contains(t', k) \\ \neg contains(t, k) \wedge \neg has_capacity(t, k) \implies t' = insert(expand(t), k, v)$$

4 Executable Representation and Implementation

Representing the data structures and implementing operations as subprograms based on the formal design of the functionality and requirements. The following example shows an implementation of an operation. The language consists of integer constants and the binary operations $+$, $-$, $/$, $*$, mod as well as the numeric binary predicate symbols $=$, $\neq$, $<$, $>$, $\leq$, $\geq$. An implementation references the operation from the object theory. A program is a sequence of labeled imperative statements with branches and conditional branches, assignment over terms.

```

implement sum
  in k : int   out result : int   vars temp_sum := 0

do_sum:
  k = 0  $\implies$  br done, temp_sum := temp_sum + k, k := k - 1, br do_sum.

done:
  result := temp_sum

```

The example trivially represents the summation of natural numbers up to n , where $n \geq 0$. The logical interpretation is a condition C_{sum} that relates the input to the output. The program itself is interpreted as a state transition relation that is represented by the first

label. The condition is an equivalence of the state transition starting with the initial state given by the input plus the initial assignment of the work variables. The result equals the output variable(s) in the final state.

$$\begin{aligned}
do_sum(s, s'') &\Leftrightarrow s(k) = 0 \implies done(s, s'') \\
&\quad \vee s(k) \neq 0 \wedge s' = s[temp_sum/temp_sum + s(k)] \wedge s'' = s'[k/s'(k) - 1] \\
done(s, s') &\Leftrightarrow s'(sum) = s(temp_sum) \\
C_{sum}(n, sum(n)) &\Leftrightarrow s = \{(k, n), (result,), (temp_sum, 0)\} \wedge do_sum(s, s') \wedge sum(n) = s'(result)
\end{aligned}$$

Let $sum(n)$ be specified as $I_{sum}(n) \Leftrightarrow n \geq 0$ and $O_{sum}(n, sum(n)) = n = 0 \implies sum(n) = 0 \wedge n > 0 \implies sum(n) = n + sum(n - 1)$. The partial correctness prove is $\forall_n I_{sum}(n) \wedge C_{sum}(n, sum(n)) \implies O_{sum}(n, sum(n))$ by induction on n .

5 Conclusion

The emphasis of the paper is that precise formal modeling does not only lead to provable implementations, it also serves as design method ([7], [6]). By formally expressing the functionality in terms of all constraints (including resource consumption or run-time efficiency) the influences that lead to data structures and algorithms are reified and can be reasoned about.

A novel aspect is that the paper offers a complete approach that spans the whole software development process. It centers around the formal design but does also map between programs and the theory they represent.

References

1. Chen, P.P.S.: The entity-relationship model—toward a unified view of data. ACM Trans. Database Syst. 1(1), 9–36 (Mar 1976), <http://doi.acm.org/10.1145/320434.320440>
2. Dijkstra, E.W.: Guarded commands, non-determinacy and formal derivation of programs (Jan 1975), <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD472.PDF>, published as [?]WD:EWD472pub
3. Erascu, M.: Computational Logic and Quantifier Elimination Techniques for (Semi-)automatic Static Analysis and Synthesis of Algorithms. Ph.D. thesis, Research Institute for Symbolic Computation (2012), rISC Technical Report 12-16
4. Erascu, M., Jebelean, T.: Automated Certification of a Logic-Based Verification Method for Imperative Loops (June 18 - 23 2012), contributed talk at CiE 2012 - How the World Computes
5. Erascu, M., Jebelean, T.: Soundness of a logic-based verification method for imperative loops. In: SYNASC. pp. 127–134 (2012)
6. Guttag, J., Horning, J.J.: Formal specification as a design tool. In: Proc. 7th ACM Symp. on Principles of Programming Languages, Las Vegas. pp. 251–261 (1980)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
8. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)