A BASIS FOR A THEORY OF PROGRAM SYNTHESIS[1]

P.A.Subrahmanyam
USC/Information Sciences Institute
and
Department of Computer Science
University of Utah, Salt Lake City, Utah 84112

1. Introduction and Summary

In order to obtain a quantum jump in the quality and reliability of software, it is imperative to have a coherent theory of program synthesis which can serve as the basis for a sophisticated (interactive) software development tool. We argue that viewing the problem of (automatic) program synthesis as that (automatically) synthesizing implementations of abstract data types provides a viable basis for a general theory of program synthesis. We briefly describe the salient features of such a theory [5, 6], and conclude by listing some of the applications of the theory.

1.1. Requirements for an Acceptable Theory of Program Synthesis.

We view some of the essential requirements of an acceptable theory of program synthesis to be the following:

- the theory should be general;
- it must adhere to a coherent set of underlying principles, and not be based on an *ad hoc* collection of heuristics;
- it must be based on a sound mathematical framework;
- it must account for the "state of the art" of program synthesis; in particular, it must allow for the generation of "efficient" programs.

Further, if a theory is to be useful, we desire that it possess the following additional attributes:

- it should serve as the basis for a program development system which can generate provably correct non-trivial programs;
- it should possess adequate flexibility to admit being tailored to specific application tasks;
- it should provide new and insightful perspectives into the nature of programming and problem solving.

With these requirements in mind, we now examine the nature of the programming process in an attempt to characterize the basic principles that underly the construction of "good" programs.

2. A basis for a Theory of Program Synthesis

Intuitively, the abstraction of a problem can be viewed as consisting of an appropriate set of functions to be performed on an associated set of objects. Such a collection of objects and functions is an **"abstract data type"** and has the important advantage of providing a representation independent characterization of a problem. Although the illustrations that most readily come to mind are commonly employed data structures such as a stack, a file, a queue, a symbol table, etc., any partial recursive function can be presented as an abstract data type.

Programming involves representing the abstractions of objects and operations relevant to a given problem domain using primitives that are presumed to be already available; ultimately, such primitives are those that are provided by the available hardware. Various programming methodologies advocate ways of achieving "good" organizations of layers of such representations, in attempting to provide an effective means of coping with the complexity of programs.

There exists, therefore, compelling evidence in favor of viewing the process of program synthesis as one of obtaining an implementation for the data type correponding to the problem of interest (the "type of interest") in terms of another data type that corresponds to some representation (the "target type.") This perspective is further supported by the following basic principles that we think should underly the synthesis process (if reliable programs are to be produced consistently):

1. The programming process should essentially be one of program synthesis proceeding from the specifications of a problem, rather than being primarily analytic (e.g. constructing a program and then verifying it) or empirical (e.g. constructing a program and then testing it).

[1]This work was supported in part by an IBM Fellowship

2. The specification of a problem should be **representation independent**. This serves to guarantee complete freedom in the program synthesis process, in that no particular program is excluded *a priori* due to overspecification of the problem caused by representation dependencies.

3. The synthesis should be guided primarily by the semantics of the problem specification.

4. The level of reasoning used by the synthesis paradigm should be appropriate to "human reasoning," rather than being machine oriented (see [6]). In addition to making the paradigm computationally more feasible, this has two major advantages:

    a. existing paradigms of programming such as "stepwise refinement" can be viewed in a mathematical framework

    b. user interaction with the system becomes more viable, since the level of reasoning is now "visible" to the user.

The above principles led us to adopt an algebraic formulation for the development of our theory [1, 2], [3, 4]. An important consequence of this decision was that the synthesis paradigm is independent of any assumptions relating to the nature of the underlying hardware. In fact, it can even point to target types suited to particular problems of interest i.e. target machine architectures which aid efficient implementations.

3. The Proposed Paradigm for Program Synthesis

We adopt the view that any object representing an instance of a type is completely characterized by its "externally observable behavior". The notion of an *implementation* of one data type (the type of interest) in terms of another (the target type) is then defined as a map between the functions and objects of the two types which preserves the observable behavior of the type of interest. The objective, then, is to develop methods to automate the synthesis of such implementations based on the specifications of the type of interest and the target type.

Intuitively, the crux of the proposed paradigm lies in "mathematically" incorporating the principle of stepwise refinement into automatic programming. This is done by appropriately interpreting both the syntactic and semantic structure inherent in a problem. An important distinction from most transformation based systems is that the refinement is guided by the semantics of the functions define on the type of interest, rather than by a fixed set of rules (e.g. [7]). An formal characterization of some of the pivotal steps in the synthesis process is provided, and an attempt is made to pin-point those stages where there is leeway for making alternative choices based upon externally imposed requirements. (An example of such a requirement is the relative efficiency desired for the implementations of different functions depending upon their relative frequency of use.)

This separation of the constraints imposed by (a) the structure inherent in the problem specification, (b) the requirements demanded by the context of use, and (c) the interface of these two, serves to further subdivide the complexity of the synthesis task -- it becomes possible now to seek to build modules which attempt to aid in each of these tasks in a relatively independent manner.

In summary, our goal was to seek, in as far as is possible, a mathematically sound and computationally feasible theory of program synthesis. The formal mathematical framework underlying our theory is algebraic. The programs synthesized are primarily applicative in nature; they are provably correct, and are obtained without the use of backtracking. There is adequate leeway in the underlying formalism that allows for the incorporation of different "environment dependent" criteria relating to the "efficiency" of implementations. The objectives of the theory include that conventional programs be admitted as valid outcomes of the proposed theory. This is in consonance with our belief that any truly viable theory of synthesis should approximate as a limiting case already existing empirical data relevant to its domain.

4. An Example: The Synthesis of Block Structured Symbol Table Using an Indexed Array

To illustrate some aspects of the paradigm for program synthesis we outline the synthesis of a block-structured SymbolTable using an indexed array as a target type (cf. [4].) The sorts of objects involved are instances of SymbolTable, Identifier, Attributes, Boolean, etc.; our primary interest here is on the manipulation of instances of SymbolTables. The functions that are defined for manipulating a SymbolTable include: NEWST (spawn a new instance of a symbol table for the outermost scope,) ENTERBLOCK (enter a new local naming scope,) ADDID (add an identifier and associated attributes to the symbol table,) LEAVEBLOCK (discard the identifier entries from the most current scope, re-establish the next outer scope,) ISINBLOCK (test to see if an identifier has already been declared in the current block,) and RETRIEVE (retrieve the attributes associated with the most recent definition of an Identifier.) The formal specifications may be found in [4] (see also [6].) Although implementations for more complex definitions of SymbolTables have been generated (which include tests for "Global" Identifiers), we have chosen this definition because of its familiarity.

The overall synthesis proceeds by first categorizing the functions defined on the type of interest (here, the SymbolTable) into one of the flowing three categories: (I) *Base*

Constructor functions that serve to spawn new instances of the type (e.g. NEWST); (ii) *Constructor* functions that serve to generate new instances from existing ones (e.g. ADDID, ENTERBLOCK, LEAVEBLOCK); and (iii) *Extractor* functions that return instances of types other than SymbolTable (e.g. RETRIEVE, ISINBLOCK).

The next step is to identify a subset of these functions (termed *kernel functions*) which serve to generate all instances of SymbolTables: these are NEWST, ADDID, and ENTERBLOCK.

A major step in obtaining an implementation for the TOI is to provide an implementation for the kernel functions. Since no model for the kernel functions is explicit in the specification of a type, a suitable model must be inferred from the behavior of the functions defined on the type. Such an inference follows from an examination of the axioms defining the extraction functions. Specifically, the domain of the terms of type SymbolTable is partitioned into its) equivalence classes by the extractors defined on the type -- and this is precisely what an implementation is attempting to capture. The defining equations of each function indicate how it "contributes" towards this partitioning, and therefore how this "semantic structure" imposed upon the terms of the SymbolTable is related to the syntactic structure of the underlying terms.

Due to lack of space, we omit the details of how this is done. One of the implemetations generated (automatically) is shown in figure 1, wherein Θ denotes the implementation map. We note that an auxiliary data type which is almost isomorphic to a Stack (of integers) was (automaticlly) defined in course of the implementation; this Stack can, in turn, be synthesized in terms of an indexed Array by a recursive invocation of the synthesis procedures.

Other Implementations that are generated for the Symbol Table include an implementation using a "Block Mark" to identify the application of the function ENTERBLOCK, and an implementation similar to a "hash table" implementation is suggested upon examining the semantics of the functions defined on the Symbol Table.

We list below one of the implementations generated for a SymbolTable, using an Array as the initially specified target type. The final representation consists of the triple <Array,Integer,ADT1>; the integer represents the current index into the array, whereas ADT1 (for *Auxiliary Data Type-1*) is introduced in course of the synthesis process, and is isomorphic to a Stack that records the index-values corresponding to each ENTERBLOCK performed on a particular instance of a SymbolTable.) We denote this by writing Θ(s) = <a,i,adt$_1$>. Informally, ENTERBLOCK.ADT1 serves to "push" the current index value onto the stack adt$_1$, LEAVEBLOCK.ADT1 serves to "pop" the stack, and D.ADT1 returns the topmost element in the Stack, returning a zero if the stack is empty.

SUCC and PRED are the successor and monus functions on Integers.

```
Θ(NEWST) = <NEWARRAY,ZERO,NEWADT1>
Θ(ADDID(s,id,al)) = <ASSIGN(a,SUCC(i),<id,al>),SUCC(i),adt_1>
Θ(ENTERBLOCK(s)) = <a,i, ENTERBLOCK.ADT1(adt_1,i)>
Θ(LEAVEBLOCK(s)) = <a,D.ADT1(adt_1), LEAVEBLOCK.ADT1 (adt_1)>
Θ(ISINBLOCK(s,id1)) = ISINBLOCKTT(<a,i,adt_1>,id1)
Θ(RETRIEVE(s,id1)) = RETRIEVETT(<a,i,adt_1>,id1)
```

ISINBLOCKTT and RETRIEVETT are defined as follows:

```
ISINBLOCKTT(<a,i,adt_1>, id1) =
        if i = ZERO
        then FALSE
        else if D.ADT1(adt_1) < i
                then if proj(l, DATA(a,i)) = id1
                        then TRUE
                        else ISINBLOCKTT(<a,PRED(i),adt_1>,id1)
                else FALSE

RETRIEVETT(<a,i,adt_1>,id1) =
                if i = ZERO
                then UNDEFINED
                else if proj(l,DATA)a,i)) = id1
                        then proj(2, DATA(a,i))
                        else RETRIEVETT(<a, PRED(i),adt_1>, id1)
```

Here, proj(i,<x_1,..x_n>) = x_i.

Figure 1. A Symbol Table Implementation

Other Examples of Applications of the Synthesis Paradigm. Several programs have been synthesized by direct applications of the synthesis algorithms developed so far. These include implementations for a Stack, a Queue, a Deque, a Block Structured SymbolTable, an interactive line-oriented Text-Editor, a text formatter, a hidden surface elimination algorithm for graphical displays, and an execution engine for a data driven machine.

References

[1] J.Goguen, J.Thatcher, E.Wagner, J.Wright. Initial Algebra Semantics and Continuous Algebras. JACM 24:68-95, 1977.
[2] J.Goguen, J.Thatcher, E.Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in *Current Trends in Programming Methodology*, Vol IV, Ed. R.Yeh, Prentice-Hall, N.J, 1979, pages 80-149.
[3] J.Guttag, E.Horowitz, D.Musser. The Design of Data Type Specifications, in *Current Trends in Programming Methodology*, Vol IV, Ed. R.Yeh, Prentice-Hall, N.J., 1979.
[4] J.Guttag, E.Horowitz, D.Musser. Abstract Data Types and Software Validation. CACM 21:1048-64, 1978.
[5] P.A.Subrahmanyam. *Towards a Theory of Program Synthesis: Automating Implementations of Abstract Data Types.* PhD thesis, Dept. of Comp. Sc., State University of New York at Stony Brook, August, 1979.
[6] P.A.Subrahmanyam. A Basis for a Theory of Program Synthesis. Technical Report, Dept. of Computer Science, University of Utah, February, 1980.
[7] D.Barstow. *Knowledge-Based Program Construction,* Elsevier North-Holland Inc., N.Y., 1979.