The PSI program synthesis system, 1978—An abstract

by CORDELL GREEN
Stanford University
Stanford, California

This abstract will cover the current status of the PSI program synthesis system. PSI is a computer program that acquires high-level descriptions of programs and produces efficient implementations of these programs. PSI allows program specification dialogues using a mix of natural language, traces, examples, and a very high-level language. These specifications are used to construct a well-defined high-level program description of the desired program. This description is called a program model. The model is then refined into an efficient implementation of the program. PSI consists of several modules including an English parser-interpreter, trace and example inference expert, domain expert, explainer, dialogue moderator, program model builder, coder and efficiency expert. The class of programs synthesized are simple symbolic computation programs in either LISP or SAIL.

The system design has been a group effort with the individuals responsible for each module as follows: parser-interpreter, Jerrold M. Ginsparg; trace and example inference and domain experts, Jorge V. Phillips; moderator, Louis I. Steinberg; explainer, Richard P. Gabriel; model builder, Brian P. McCune; coder, David R. Barstow and Juan Ludlow; and efficiency expert, Elaine Kant. The interested reader should see the short summary paper for a list of references.

PSI's operation may be conveniently factored into an acquisition phase and a synthesis phase. The acquisition phase is concerned with acquiring the program specifications and producing a program model. The synthesis phase then finds an efficient implementation in the selected target language.

An assumption used in the design of the acquisition phase of PSI is that there is no one best method for specifying all programs. So a mix of methods is allowed, ranging from a "conventional" very high-level language through English and traces. The English sentences are first parsed, then interpreted into fragments. The parser limits search by incorporating considerable knowledge of English usage. The interpreter is more specific to automatic programming, using program description knowledge as well as knowledge of the last question asked and the current topic to facilitate the interpretation into fragments.

Fragments form a loose description of program and data structure. The model builder must then apply knowledge of correct high level programs to convert the fragments into the model. The program model includes complete, consistent and executable (but slowly) high-level algorithm and information structures. The model builder processes fragments, checking for completeness and correctness, fills in detail, corrects minor inconsistencies, and adds cross-references. It also generalizes the program description, converting it into a form that allows the coder to look for good implementations.

Another input specification method is partial traces mixed with input and output examples. Partial traces of states of internal and I/O variables allow the inductive inference of control structures. The trace and example inference module infers loose descriptions of programs in the form of fragments, rather than programs themselves. This technique allows domain support to disambiguate possible inferences, and also separates the issue of efficient implementation from the inference of the user's intention. Application domain specific knowledge (e.g., knowledge about learning programs) is concentrated in the domain expert, which supplies other acquisition modules with domain support for disambiguation and program model completion.

The moderator guides the dialogue with the user by selecting or repressing questions generated by various modules. It attempts to keep PSI and the user in agreement on the current topic, provides a review-preview on a topic change, helps the user that gets lost, and allows initiative to shift between PSI and the user. The explainer module generates questions about and descriptions of program models as they are acquired, in order to help verify that the inferred program description is the one desired.

After the acquisition phase is complete, the synthesis phase begins. This phase may be viewed as a series of program refinements or as a heuristic search for an efficient program that satisfies the program model. The coder has a body of program synthesis rules that gradually transform the program model from abstract into more detailed constructs until it is in the target language. Both algorithm and data structures are refined interdependently. The coder deals primarily with the notions of sets and correspondences and can synthesize programs involving sequences, loops, simple input and output, linked lists, arrays, and hash tables.

The refinement tree effectively forms a planning space that proposes only legal, but possibly inefficient, programs. The efficiency expert reduces the search of the tree by several heuristics and by estimating the time-space cost product of each proposed refinement and then using the
branch and bound method. It also factors the program into relatively independent parts so that all combinations of implementations are not considered. An analysis for bottlenecks can allocate synthesis effort to more critical parts of the program.

In summary, we have formulated a framework for an automatic programming system and have a start on the kinds of programming knowledge that must be embedded therein. PSI is moderately successful in that a research prototype is currently running and has synthesized many different programs including a graph traversal program, simple storage and retrieval programs and learning programs.

REFERENCE