The Roles of Witness-Finding in Software Synthesis

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304

Green, Waldinger, Constable, and others pioneered the use of deductive synthesis wherein a complete program is extracted from a proof that a problem specification is solvable. The essential technique is *witness-finding* – given a sentence of the form

$$\forall(x : D)\exists(r : R)\ (I(x) \implies O(x, r))$$

infer a witness term $w(x)$ for the existentially quantified variable r such that

$$\forall(x : D)\ (I(x) \implies O(x, w(x)))$$

is provable.

I'd like to briefly outline my experience in using witness-finding in a variety of roles during software synthesis, including as a special case the extraction of whole programs as witnesses. The effort may be seen as attempting to develop more encompassing frameworks for generating software, motivated by the desire to synthesize larger, more complex applications.

Program Scheme Instantiation

In the 1970's and early 80's most program derivations were special cases of divide-and-conquer (unfortunately, this is still too often true), probably because they are some of the easiest induction arguments to push through a prover. So it seemed interesting to try to capture divide-and-conquer more abstractly, rather than repeatedly discovering instances of it. From this effort came the notion of constructing a program by instantiating a program scheme and using deductive techniques to synthesize the definitions or specifications for the free operators in program schemes (Smith, 1985; Smith, 1987; Lowry, 1991).

There are a variety of ways in which witness-finding can be used to support scheme instantiation.

- *Deriving specifications for subalgorithms* – The correctness conditions of a program scheme serve to constrain the ways that the free operators can be instantiated, and these constraints can often be used to infer specifications for subalgorithms. In the divide-and-conquer case, if you plug in a standard decomposition operator, you can use a soundness axiom on the scheme to infer a specification for the composition operator. Dually, from a given composition operator, one can calculate a specification for the decomposition operator. To support this kind of inference, we generalized the witness-finding process slightly to allow generation of boolean-valued terms that are sufficient conditions on a given formula (Smith, 1982; Smith, 1985). This process was called derived antecedents, but now known generally as abductive inference, although for synthesis purposes the abductibles are required to be expressed over a specified set of variables.

- *Deriving guards* – Derived antecedent were use to infer the guards on a recursive call to ensure termination and establishment of preconditions. Again this is finding a sufficient condition as witness to the validity of a correctness formula.

- *Deriving Filters* – Backtrack and branch-and-bound program schemes require the inference of necessary conditions (to serve as pruning and constraint propagation mechanisms) as well as upper or lower bounds on numeric expressions (to serve for bounding in branch-and-bound) (Smith, 1987; Smith, 1990; Smith *et al.*, 1996).

- *Matching Library Operators* – Top-down design reduces a top-level specification to a tree of subproblem specifications. The process bottoms out in specifications that can be matched against primitive instructions or library operators. Witness-finding was used in CYPRESS and KIDS to generate the data translators that allow a library routine to be used to satisfy a given specification (Smith, 1985; Smith, 1993). More recently, witness-finding has been used to generate the glue-code that allows two data sources to interact (Burstein *et al.*, 2003).

- *Expression Optimization* – After scheme instantiation, there are often lots of opportunities to apply common program optimizations, such as: context-dependent simplification, finite differencing, partial evaluation, case analysis, and others (Smith, 1990).

Underlying many of these optimization tactics is search for a witness that is equal/equivalent to an expression modulo the context in which it will be evaluated.

Design Theories and Interpretation Construction

Later, in an attempt to more fully formalize the notion of a program scheme, we developed the notion of an *algorithm theory* in which a program scheme was a definition for a function symbol expressed in terms of some other operators that were axiomatically constrained. To instantiate the scheme required constructing a theory morphism (interpretation) that explicated how the symbols in the algorithm theory translated to the problem domain theory such that all the axioms remained provable (Lowry, 1987; Smith and Lowry, 1989). A pushout accomplishes the instantiation of the abstract program scheme with expressions from the problem domain (Smith, 1996; Smith, 1999).

The key work here is constructing an interpretation between the algorithm theory and the problem domain theory. An *interpretation between theories* or simply an *interpretation* is a map from the symbols of the source (domain) theory to expressions of the target (codomain) theory that induces a language translation, and is required to preserve provability; i.e. an source theorem must remain provable under translation to the target theory. It turns out that all the witness-finding roles listed above are examples of using unskolemization during interpretation construction.

Here is how unskolemization is used to support interpretation construction (Smith, 1993). Suppose that we are trying to complete a partial interpretation σ from theory S to theory T. Let f be a function symbol of S that has no translation yet under σ. Suppose that A is a prenex normal form axiom in which all occurrences of f are identical and simple (only variables, no terms), and suppose that all other symbols in A are translatable under σ (i.e. the domain of σ includes all of the sort and operator symbols of A except for the function symbol f). To obtain a candidate translation for a function symbol f, we proceed as follows.

(1) *Unskolemize f in A yielding A'.* Since the effect is to replace each occurrence of f by a variable, each symbol in A' can be translated via σ.

(2) *Translate A'.* The translated sentence $\sigma(A')$ need not be an axiom of T. In order for σ to become an interpretation, we need an expression defining the translation of f in T. $\sigma(A')$ can be viewed as a constraint on the possible translations of f.

(3) *Attempt to prove $\sigma(A')$ in T.* A constructive proof will yield a (witness) expression $w(x)$ for f that depends only on the variable(s) x. If the proof involves induction (resulting in a recursively defined witness), then we extend the target theory with a fresh operator symbol and an axiom stating its recursive definition.

(4) *Extend the partial morphism σ by defining $\sigma(f)$ to be $w(x)$.* By construction this translation for f guarantees that σ properly translates the axiom A.

Other axioms that involve f may now be translatable, and if so, then we can attempt to prove that they translate to theorems.

Interpretations play a variety of basic roles in software development, and thus interpretation construction is a crucial activity, including witness-finding as a key support tool.

- *Representing and applying abstract design knowledge* – Abstract design knowledge can often be captured and organized via interpretations. For example, algorithm knowledge can be expressed as an interpretation between an applicability theory and a theory expressing the problem-solving method. For another example, datatype refinements are expressed as interpretations from the abstract datatype theory to a (more) concrete datatype theory (Blaine and Goldberg, 1991). Design theories may be organized into taxonomies where interpretations precisely specify the "subclass" links (Smith, 1996). When we represent design knowledge abstractly as an interpretation $I : A \rightarrow B$, then we apply the abstraction to generate a refinement of a target specification S by (i) constructing an interpretation from A to the target specification S, and (ii) computing the pushout of the two interpretations $B \leftarrow A \rightarrow S$ (Smith, 1996).

  This process can be applied to a broad range of types of knowledge, well beyond programming knowledge. Note that showing applicability of an abstraction to a concrete problem via an interpretation properly generalizes the notion of a substitution that matches a rule pattern with a domain goal – we match not only syntax, but also must respect the axiomatic semantics.

- *Refinement generators (metaprograms)* – Sometimes abstract design knowledge is best captured by metaprograms that generate interpretations/refinements. Program optimizations are typical examples, and they often have the following characteristic: they are based on a metatheorem whose conclusion is that such-and-such a syntactic change results in a refinement (i.e. preserves or reduces models). The metaprogram works by analyzing the object-level spec and constructively inferring how to reify the metatheorem in this case.

- *Constructing datatype refinements* – We have worked several examples of deriving data structures via interpretation construction. For example, constructing an interpretation from sets over a linear order to binary trees over a linear order can result in the heaps data structure. The key invariant, the heap property, is derived via witness-finding.

- *Deriving behavior* – A guarded command in a state machine, corresponds exactly to interpretation (from the theory of the post-state to the theory of the pre-state) (Pavlovic and Smith, 2001). Interpretation construction then can be used to generate commands of an abstract state machine. For example, some researchers have described work on calculating transitions in control systems so that safety conditions are guaranteed.

Witness-finding can play a crucial role in the construction of correct software. In my view, *constructing interpretations is a fundamental and central problem in formal software development*, and it provides a wealth of opportunities for developing and applying witness-finding technology. The work described above aims for a software development framework within which witness-finding can be used to best advantage.

References

Lee Blaine and Allen Goldberg. DTRE – a semi-automatic transformation system. In B. Möller, editor, *Constructing Programs from Specifications*, pages 165–204. North-Holland, Amsterdam, 1991.

M. Burstein, D. McDermott, D.R. Smith, and S.J. Westfold. Derivation of glue code for agent interoperation. *Journal of Autonomous Agents and Multi-Agent Systems*, 6:265–286, 2003.

Michael R. Lowry. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence*, Seattle, WA, July 13–17, 1987.

Michael R. Lowry. Automating the design of local search algorithms. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 515–546. AAAI/MIT Press, Menlo Park, 1991.

Dusko Pavlovic and Douglas R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering*, pages 157–165. IEEE Computer Society Press, 2001.

Douglas R. Smith and Michael R. Lowry. Algorithm theories and design tactics. In L. van de Snepscheut, editor, *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, pages 379–398. Springer-Verlag, Berlin, 1989. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).

Douglas R. Smith, Eduardo A. Parra, and Stephen J. Westfold. Synthesis of planning and scheduling software. In A. Tate, editor, *Advanced Planning Technology*, pages 226–234. AAAI Press, Menlo Park, 1996.

Douglas R. Smith. Derived preconditions and their use in program synthesis, LNCS 138. In D. W. Loveland, editor, *Sixth Conference on Automated Deduction*, pages 172–193, Berlin, 1982. Springer-Verlag.

Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, September 1985. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).

Douglas R. Smith. Structure and design of global search algorithms. Technical Report KES.U.87.12, Kestrel Institute, November 1987.

Douglas R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, 1990.

Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 15(5-6):571–606, May-June 1993.

Douglas R. Smith. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 1101, pages 62–84. Springer-Verlag, 1996.

Douglas R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, Amsterdam, 1999.