Programming" framework, Yampa. If you like little languages, you'll appreciate how useful Haskell is for embedded domain specific languages. It may be even more useful now that Template Haskell is in the works.

Isaac Jones

Inductive Synthesis of Functional Programs by U. Schmid, Springer Verlag, 2003, 420pp, ISBN 3540401741.
doi:10.1017/S0956796807006296

Program synthesis is one of those deceptively simple propositions: say what you want some program to do, turn the handle and out pops the program. The problem, of course, is to determine what the handle drives, if not a skilled human being.

Deductive synthesis, with which JFP readers will be most familiar, concerns the generation of programs using rigorous techniques that ensure that that they satisfy initial specifications. This approach received an enormous boost when declarative programming emerged from the labs in the late 1970s, egged on by various 5th Generation Programmes in the early 1980's, most significantly in Japan and the UK. The attendant hype, that in declarative programming one identified *what to do* rather than *how to do it*, seemed to offer a seamless link between specification and implementation, through the mediation of theoretically grounded logic and functional languages.

Alas, Hayes & Jones (HJ89) influential 1989 paper effectively ended research into so-called executable specifications. In rightly cautioning that specifications were typically undetermined and infinitary, and hence not directly amenable to unique or indeed correct implementations, they also reasserted the highly contentious Oxford-school idealism that ranks mathematics over computation. Thus, today, it is salutary that there are no mainstream tools for generating implementations from, say, Z specifications, and that "formal methods" of program construction, like program refinement (Mor90), type theory programming (NPS90; MM04) and weakest precondition calculation (Bac03), lurk in quiet corners of academe. The most promising compromise may be the B-Method (Sch01), where specifications may generate results through terminating animation, and implementations may be realised through the production of more and more concrete refinement machines based on increasingly restricted B subsets.

However, in orthogonal research, the mathematical reasoning community has long used techniques based on automatic theorem proving coupled with planning to prove and generate programs satisfying Floyd-Hoare style pre- and post-conditions, and invariants. For example, we have used tactics-based proof planning constrained by rippling (BDHI05) to derive higher-order functions in Standard ML programs that lack them (CIMS05).

Inductive synthesis, rooted in Artificial Intelligence and Cognitive Science, differs radically from deductive synthesis. Rather than using abstract logical pre- and post-conditions to specify requirements, programs are induced from concrete instances of input/output values or program traces. Thus, there is no formal idea of correctness: instead, the process seeks to elaborate program structures that satisfy the specific training instances.

Perhaps the most widely known form of inductive synthesis is genetic programming (Koz92), where an evolutionary algorithm is used to generate syntactically correct program structures which are evaluated for fitness by execution against an input/output set. Initially, untyped forms of LISP were popular: more recently there has been a trend towards typed languages, for example Grant (2000), to try to constrain the potentially vast search space.

The approach to inductive synthesis presented in this challenging volume combines universal planning, plan transformation and folding. To begin with, universal planning is applied to a problem specification in the standard planning language Strips to produce sets of optimal sequences of actions, characterised as function applications, to cover the entire domain of input/output pairings. Plan transformation is next used to order the input states and infer an underlying data type. Finally, the transformed plan is folded into a recursive program using pattern matching.

Rather than a producing a concrete program in a specific language, synthesised programs take the generic form of a recursive program scheme (RPS) over a term algebra. Many RPS share common structure: this generic representation enables the use of analogical problem solving to abstract further over recursion schemes, through tree transformation and anti-unification. RPSs may, of course, be instantiated to produce executable programs. For illustration, Schmid uses a pure functional subset of LISP, where recursion is primitive and may not be mutual.

While the emphasis is squarely on "inductive synthesis" rather than "functional programs", the account would be strengthened by far more discussion of scalability from the familiar simple linear recursive examples, and of how the techniques might be extended to languages with richer types and more complex expression forms. Nonetheless, program synthesis is easy to conceptualise yet fiendishly hard to realise: Schmid's worthwhile book makes a valuable contribution.

References

Backhouse, R. (2003) *Program Construction: Calculating Implementations from Specifications.* Wiley.

Bundy, A., Basin, D, Hutter, D. and Ireland, A. (2005) *Rippling: Meta-Level Guidance for Tactical Reasoning*. CUP.

Cook, A., Ireland, A., Michaelson, G. and Scaife, N. (2005) Discovering applications of higher-order functions through proof planning. *Formal Aspects of Computing*, **17**(1): 13–57.

Grant, M. S. (2000) *An Investigation into the Suitability of Genetic Programming for Computing Visibility Areas for Sensor Planning*. PhD thesis, Heriot-Watt University.

Hayes, I. J. and Jones, C. B. (1989) Specifications are not (necessarily) executable. *Softw. Eng. J.* **6**: 320–338.

Koza, J. (1992) *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT.

McBride, C. and McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1): 69–111.

Morgan, C. (1990) *Programming from Specifications.* Prentice-Hall.

Nordstrom, B., Petersson, K. and Smith, J. M. (1990) *Programming in Martin-Lof's Type Theory: An Introduction.* Oxford.

Schneider, B. (2001) *The B-Method.* Palgrave.

Schmid, U. (2003) *Inductive Synthesis of Functional Programs.* Number 2654 in LNCS. Springer.

Greg Michaelson
Heriot Watt University, UK