On Integration of Programming Paradigms

ALAN MYCROFT

Computer Laboratory, Cambridge University, UK ⟨*Alan.Mycroft@cl.cam.ac.uk*⟩

Programming notions can be expressed in many different paradigms—using imperative, object-oriented, concurrent, functional, logic-programming (or other) formulations. Choice of an appropriate paradigm can greatly affect the ease of programming. Initially this choice appears unequivocally helpful to the programmer, but doubts soon arise. What if parts of a system are written in a typed imperative language (e.g. C) and others in an untyped logic programming language (e.g. Prolog)? How can a lazy functional language (e.g. Haskell) invoke procedures which have state? We thus confront the problem of Integration of Programming Paradigms—to integrate subsystems expressed in differing paradigms into a coherent whole. The aim of this paper is to expound an algebraic view of such integration. Work with related aims but differing starting points can be found in the "Action Semantics" approach [2] and the "Monads" view of feature composition (see e.g. [1]). Also relevant is work [3] on using complexity to compare semantic models.

Integrating subprograms written in varying *source* languages $\mathcal{S}_i, i \in \{1, 2\}$ intuitively represents writing in the *union* of these languages; actually what we mean is a *target* language expressive enough to express concepts from all the source languages. If the languages all had a single sort (syntactic category) then this could be the language generated by the union of the constructors of the source languages. More thought leads us to a target language $\mathcal{T}$ which has embeddings $\theta_i : \mathcal{S}_i \to \mathcal{T}$. The θ_i should be injective at the semantic level thus, given $s_i, s_i' \in \mathcal{S}_i$, we require $[\![s_i]\!] \neq [\![s_i']\!] \Rightarrow [\![\theta_i s_i]\!] \neq [\![\theta_i s_i']\!]$. However, in general this still behaves as the union—there is no constructor to use a term in $\mathcal{S}_j$ as a term in $\mathcal{S}_i$; indeed all we can do is to write programs totally in $\mathcal{S}_1$ or $\mathcal{S}_2$. There are various alternative forms of interaction possible: the least expressive is a pairing constructor which allows us write a program (s_1, s_2) consisting of programs $s_i \in \mathcal{S}_i$ (as they cannot yet interact it is pointless to discuss whether evaluation happens in parallel or not). If there are operators (e.g. '**;**') which we wish to identify in the $\mathcal{S}_i$ then we can require $[\![(s_1, s_2); (s_1', s_2')]\!] = [\![(s_1; s_1', s_2; s_2')]\!]$. In practice we require more complex interaction: for example, if $\mathcal{S}_1$ is Pascal and $\mathcal{S}_2$ is Haskell, we might desire variables effectively to coincide, so that writing x in $\mathcal{S}_1$ affects reading x in $\mathcal{S}_2$ (or more structured constructs to access a procedure in $\mathcal{S}_1$ with values from $\mathcal{S}_2$). This is handled by adding constructors to $\mathcal{T}$ which represent the interaction (correspondence) between constructs in the $\mathcal{S}_i$. This in general causes surprising degrees of complexity.[1]

[1] One is reminded of the word problem for groups—given two groups we have a well-defined notion of the *free group* which they generate. But, suppose we then decide certain relationships also must hold, e.g. $ab^3 = b^2a^{-1}$ then in general it is undecidable whether two such *presentations* represent equal groups. Note that the word problem is trivial for *abelian groups*; this corresponds to the case above where composition distributes over pairing—if features do not really interact then no additional complexity arises.

Some features embed naturally by an injection as suggested above; for others the issue is more subtle. One case might be that some unrestricted feature is deemed harmful and only a restricted aspect of it is to be embedded. An example here is the use of abstract data types to embed mutable arrays into Haskell ('monads') in such a way that mutability is not observable but keeping the implementation advantages of a single-threaded state. Another alternative is to encode the feature to some extent. For example Prolog can be embedded in a procedural language having *catch* and *throw*.[2] This approach is appealing, but problems soon arise as before. What if the Haskell array-as-monad also is accessible to a procedural language? Suppose our Prolog program is embedded as indicated in our language with *catch* and *throw*. A compiler might 'know' how to compile an image of Prolog into efficient code, but this ability would be lost when fragments of code from another language result in the target program not being an image of Prolog.

For many source languages, there is the possibility of a universal target language—the typed λ-calculus. The complexity of interaction of features led Mosses to develop "Action Semantics". Recent work by Liang and Hudak [1] uses monads to structure denotational semantics for compiler generation with similar aims. One problem here though is that monads do not in general compose.

It feels appropriate here to discourage the idea that *reflection* (e.g. [4]) primitives might help. In reflective systems access to the state of a presumed interpreter is made available to the running program. While careful use of such facilities may provide great power, in general they can be seen as tightly bound integration of two paradigms and therefore are liable to be hard to reason about.

A related approach is that of *meta-programming*. In various logic programming communities there is an emphasis on meta-programming, by which is meant that one constructs an interpreter for a language close to the problem and then writes problem solutions using the primitives of this language. In general the same problems seem to occur here for multi-paradigm working as occur for the embedding into a target language above; the difference seems merely to reflect a choice of translation versus interpretation.

How then is one to proceed? To me, the best solution uses disparate aspects of the above discussion. The higher the degree of interaction between languages the more inter-language interaction complexity can grow—hence subsystems written in different languages should only be loosely connected (this is desirable for program structuring reasons too). Moreover, the interaction seems best handled indirectly by a *coordinator* whose primitives are the notions imported or exported from each subsystem. It seems to matter less whether this is seen as a 'union' language or as an interpreted meta-program in a small language of combinators—although the latter allows more appropriate type-checking, e.g. that an imperative routine cannot incomparably update a lazy list. One should note that interface design in the presence of multiple programming paradigms

[2] Clause alternative selection is handled by a loop containing *catch* and failure by *throw*. Unifications are implemented as undoable assignments (e.g. by an explicit *undo* stack popped at every *catch*.)

is rather pervasive; changing behaviour of a subsystem component can greatly affect the combinator system which the coordinator constitutes.

I thank the organisers (Chris Hankin and Hanne Nielson) and participants of Dagstuhl 'Integration of Paradigms' in September 1995 for stimulating my thoughts on the subject of this essay.

References

[1] Liang, S. and Hudak, P. Modular Denotational Semantics for Compiler Construction. Proc. European Symposium on Programming (ESOP'96): Lecture Notes in Computer Science, to appear, Springer-Verlag 1996.

[2] Mosses, P.D. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, vol. 26, CUP 1992.

[3] Mycroft, A., Degano, P. and Priami, C. Complexity as a Basis for Comparing Semantic Models of Concurrency. In: Algorithms, Concurrency and Knowledge—Proc. 1995 Asian Computing Science Conference, Lecture Notes in Computer Science, vol. 1023, Springer-Verlag 1995.

[4] Smith, B.C. Reflection and Semantics in Lisp. Proc. 11th ACM Symposium on Principles of Programming Languages, 1984.