Progress in Programming Languages

KIM B. BRUCE

Williams College ⟨kim@cs.williams.edu⟩

A great deal of progress has been made over the last forty years in the design and implementation of programming languages. Unfortunately, most programmers have little experience with imperative programming languages developed after the early 1970s, and still use older languages like FORTRAN, COBOL, Pascal, and C. Yet in the last twenty or so years there have been important improvements to these languages.

Perhaps the most important development has been the introduction of features that support abstract data types (ADTs). These features allow programmers to add new types to languages that can be treated as though they were primitive types of the language. The programmer can define a type and a collection of constants, functions, and procedures on the type, while prohibiting any program using this type from gaining access to the implementation of the type. In particular, access to values of the type is available only through the provided constants, functions, and procedures. Simula-67 was one of the first languages to provide support for ADTs, though it did not provide support for hiding the representation of types. For example, in the mid-'70s Clu's clusters provided full support for ADTs. Two of the more popular languages providing full support for ADTs (though using slightly different mechanisms) are Modula-2 and Ada-83.

These languages have also provided better support for separate, but not independent, compilation. That is, programs can be broken into smaller pieces, each of which can be separately compiled. However, since these different compilation units depend on each other, there must be a way of reliably passing type information from one unit to the next. C provides primitive facilities to support this with header files, while the (non-standard) Pascal units provide somewhat better support (though without information hiding). Modula-2 and Ada-83, on the other hand, support separate compilation by providing separate specification and implementation units. This allows programs to be compiled once the imported units specifications have been defined, while leaving the implementations to be compiled later. In fact, programmers may interchange a variety of implementations of specifications without having any impact on the code of programs using these units.

Clu and Ada-83 also provided support for parameterized types and polymorphic operations by allowing parameterized ADTs. For instance, in Ada-83 one can specify a parameterized binary search tree as:

```
generic
 type BSTElt is private
  with function LessThan(x, y:
  BSTElt) return BOOLEAN;
package BinSrchTree is
. . .
end BinSrchTree;
```

This package can then be instantiated with any type T that has an appropriate `LessThan` function defined for it. Moreover, in both Clu and Ada-83 these parameterized ADTs can be type checked before being instantiated, which guarantees that instantiated versions will be free of type errors as long as the opera-

This research was partially supported by NSF grant CCR-9424123.

tions required in the declaration exist. The ability to define such parameterized ADTs is clearly very useful to programmers.

At least partially in response to the concerns raised in Djikstra's famous letter "Go to considered harmful" [1968], language designers have searched for more structured language features to handle cases where programmers formerly used `goto` statements. One such construct is the exception mechanism, which appeared in PL/I, Clu, and Ada, among others. Exceptions can be "raised" either automatically by the system (e.g., upon division by zero) or by the programmer (e.g., on an attempt to pop an element off of an empty stack). As suggested by these examples, exceptions are typically used to handle error conditions that arise in programming. In many of these cases the currently executing unit (e.g., the pop routine for a stack) has no useful action that can be taken on detecting the error, but a calling routing may.

When an exception is raised, a search is made for an appropriate "handler." If a handler for the exception is found in the current activation unit, it is executed, otherwise the search for a handler for the exception is made down the runtime stack. That is, the search proceeds to the unit that called the currently executing unit, and so on, until a handler is found. If no handler is found in the runtime stack, the program halts. (A good source of information on this and the other programming-language constructs discussed above is Louden [1993]).

More recently, more attention has been paid to object-oriented languages like Smalltalk [Goldberg and Robson 1983] and Eiffel [Meyer 1992] as well as to adding object-oriented features to older languages such as Pascal, C, Ada, and even COBOL. As noted in this issue by Hirshfeld and Ege, object-oriented languages provide support for abstractions similar to that described above, though from a somewhat different point of view. (Languages like Clu, Modula-2, and Ada-83 are sometimes called object-based rather than object-oriented, because of the different point of view and because they lack features like subtyping and inheritance commonly found in object-oriented languages.) Many of these languages also include support for parameterized types, polymorphic routines (or classes), and exceptions. It is not clear yet whether or not the current focus on object-oriented languages will eventually replace interest in more traditional imperative languages.

Meanwhile, there have been many interesting developments in modern functional languages that provide support for ADT's, exceptions, and polymorphism. Languages like ML [Milner et al. 1990] and Haskell [Hudak and Fasel 1992], for instance, are statically typed and support implicit polymorphism. Features in these languages like structures, functors, and type classes provide a great deal of support for modular programming. The Common Lisp Object System (CLOS) provides support for a variant of object-oriented features known as multi-methods, while a forthcoming version of ML, ML 2000, will almost certainly include support for objects.

The future of logic programming languages is not clear at the moment. After a burst of interest in the early 1980s, they have settled into a fairly narrow range of applications. More recent work has focussed on replacing the "logic" in these languages by the solution of constraints [Benhamou and Colmerauer 1993]. It is premature to say how important these languages will be in the long run, but very interesting work is continuing in this area.

Languages are also playing an important role in the support of concurrency and distributed computing. The key ideas that must be supported are the sharing of information and resources by different processes and the need to ensure the consistency of shared information across these processes. Languages can simply support primitive features to fork off processes and synchronize them (e.g., using semaphores), or they can

support more elaborate features like monitors (Concurrent Pascal) or tasks (Ada). Interesting work is progressing on concurrent object-oriented languages, which may provide finer control over concurrency (especially with regard to access to shared variables) inside objects. Other approaches to concurrency rely on smart compilers being able to detect and support concurrency without programmer intervention. The most popular of such approaches involve imperative languages like FORTRAN (see the article by Wolfe in this issue), while others involve functional or dataflow languages where there is less of an obviously sequential flow of control within a program. At this point it is clear that programmer-controlled concurrency is very difficult, but it is not at all clear what the ultimate solution to this difficulty will be.

The articles about type systems by Cardelli and semantics by Schmidt in this issue highlight the importance of a more theoretically grounded study of programming languages. Just as the syntax of a programming language can be presented formally (e.g., as a context-free grammar), the typing rules and semantics of a programming language can also be presented using formal systems. Given a precise formal presentation, it is possible to prove only that a type system will successfully type-check programs without certain kinds of errors and that rules for reasoning about programs are sound.

As an example of interesting developments on the implementation of programming languages, improvements in garbage-collection algorithms have led to better performance among functional and object-oriented languages that rely on automatic storage management rather than requiring the programmer to worry about the error-prone activity of manually disposing of heap-allocated memory.

We appear to be in a period of turmoil in the choice of programming languages. While the main focus of attention currently is on object-oriented languages, many of the more popular languages either lack support for some of the important features mentioned above, provide unsafe or excessively rigid type systems, or have extremely complicated conceptual models. There is some hope that, building on recent research in programming languages, the next generation of object-oriented languages will be both simpler and safer, while taking advantage of the greater support for modeling and reuse found in object-oriented languages. Certainly the emergence of languages like Java, which provide platform-independent support for programming with the ability to download and execute programs and libraries via capable web browsers, will have an important impact on both programmers and users.

We may be witnessing the beginning of a resurgence in languages that are simple, safe, and high-level, yet are sufficiently powerful to provide efficient solutions to real problems. One thing, however, is clear. We should not expect one language to provide the best solution to all problems. In particular, different paradigms lead programmers to think in quite different ways. Which language is most useful will likely depend on the problem to be solved.

REFERENCES

BENHAMOU, F. AND COLMERAUER, A. EDS. 1993. *Constraint Logic Programming: Selected Research*. MIT Press, Cambridge, MA.

DIJKSTRA, E. W. 1968. Goto statement considered harmful. *Commun. ACM 11*, 3, 147–148.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk–80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

HUDAK, P. AND FASEL, J. 1992. A gentle introduction to Haskell. *SIGPLAN Not. 27*, 5 (May).

LOUDEN, K. C. 1993. *Programming Languages: Principles and Practice*. PWS, Boston, MA.

MEYER, B. 1992. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, NJ.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, MA.