COMPUTERS IN EDUCATION
R. Lewis & D. Tagg (editors)
North-Holland Publishing Company

Program construction with abstract notions in ELAN

Karl Kleine
Katholieke Universiteit Nijmegen, Afd. Informatika, Toernooiveld, 6525 ED Nijmegen, The Netherlands
Stefan Jähnichen, Wilfried Koch
Technische Universität Berlin, FG Softwaretechnik, TEL18, Ernst-Reuter-Platz 7, 1 Berlin 10, West Germany
Günter Hommel
G M D, Postfach 1240, 5205 St.Augustin 1, West Germany

Verbalization is a key factor in problem solving with computers, especially in the initial teaching of systematic programming. Systematic programming consists of the construction of abstract algorithms in terms of other algorithms, types and objects by a composition and decomposition process involving control structures and data structures. In this paper, we advocate the use of **refinements** as a facility for the naming of small program fragments and retaining these names in the final program text. They are considered significant for the understanding of the program, as they represent small incremental abstraction steps created on the fly during program development. This approach suitably supplements large scale abstraction means such as abstract data types.

These ideas have been implemented in the educational programming language ELAN; experiences of four years of its use in teaching at secondary school and university entry level are presented.

1. Introduction

Problem solving and programming cannot be separated in the initial teaching of programming, and the linguistic facilities of the programming language play a major role, in particular its abstraction facilities. Their discussion in recent years mostly centered around the concept of 'abstract data types', as pioneered by [1]. The current emphasis on the data aspect implies an 'abstraction granularity' necessary for 'programming in the large', but neglects concepts for algorithmic abstraction useful for 'progamming in the small'. Algorithmic abstraction on a small scale and its relation to construction mechanisms will be treated in the next chapter. Deficencies in available language support concerning both data abstraction and algorithmic abstraction lead to the design and implementation of ELAN [2], which will be presented next. We close with a summary of experiences of four years of its use.

2. Program construction with abstract notions

Program construction is only a part of what we are really interested in, problem solving. Seen in this larger context, a model of the application is the essential result of problem analysis. In an educational setting, and especially in the first exposure to programming, the models are simple and easily grasped; furthermore much guidance is provided by the instructor on the application modelling.

To produce a problem solution on a computer, the notions occurring in the model must be represented in the terms of some computing machinery, i.e. appear as program text in some programming language. As an illustration, consider a computer simulation of a physics experiment, e.g. the free fall. Notions like 'current velocity' and 'a time step' are most likely to find their way into the final text of the computer program as a variable and some loop body respectively. In the general case, several intermediate realization steps will be necessary to construct such notions as 'a time step' out of more primitive ones, e.g. a number of statements computing a new state of the physical system modelled. These steps of creating realizations for notions are usually quite small, and often even so small that they are barely recognized as such (and verbalized and put to paper), but are immediately mentally superseded by realizations using some constructs of the programming language at hand.

In contrast to the purely constructive approach pursued in most programming languages supporting this policy of in situ realization, our aim is to lay down the whole problem solution in the program text, which includes the notions we conceive in the process of constructing it. Therefore, extensive **naming** facilities must be provided in programming languages, for identifying concepts beyond the primitive entities offered by them, like say integer variables. Such naming constructions should be extremely easy to use and should permit tagging any constructed entity with a suitable name. Names thus created can again be used for the construction of new ones using the construction mechanisms of the underlying programming

language. Local programming in this manner results in many rather short and easily surveyable definitions. Deeply nested control structures are absent; though 'structured', programs using a block nesting style have a tendency to lose the significance of the blocks to the problem solution and to obscure the meaning of their parts, as their understanding is hampered by the permanent detail level jumping along with the indentation level. In the approach to programming advocated here and in [3] conceptual units are not realized in situ as e.g. nested blocks, but are given names and defined next to the one in which they are used. It is our claim that the **creation of notions** for the computational steps to be taken is the fundamental step to the overall composition of a program, and thus their verbalization and piecewise construction should correspond very closely. For this reason the notions conceived should appear in the final program text; the clerical work of inserting realizations for their names is to be left to a compiler.

This all sounds like re-invention of the well-known top-down stepwise refinement methodology, but there is more to it: In the first place, there is no appropriate support for it in nearly all currently available languages. Even if a programmer applied the methodology in a very disciplined manner, the notions conceived during program development will mostly be lost; it is possible to give suitable names to data objects, put in some comments for major blocks of code, and even misuse procedures to some end, but these are all only patches that alleviate the problem, but do not solve it. In the following we will demonstrate a language construct called **refinement** as a solution to these practical difficulties. In the second place, top-down development is a good guideline for initial exploration of solutions. Actual program construction may follow it, or some other development strategy, but will for a major part consist of revisions of earlier decisions and their implementation. At this point the significance of explicit creation of names for constructions of convenient size comes up again; they are much more easily handled than any realization on a more concrete level. The messy structure of many programs is largely due to many revisions that were necessary on an excessively detailed program text.

We can propagate development methodologies, but we cannot hope for the final product, the program text, to mirror the process of its creation; instead we have to turn to the structure of the result. If we can see the abstract notions in the text itself and their stepwise refinements and construction, we will have some confidence that this program is a solution to our problem at hand. In this respect, the way in which the program was constructed and the deviations taken during its construction are of minor importance.

As problems tackled get larger, conceptual **grouping** of entities will be necessary, soon followed by the need for encapsulation and imposition of restrictions. This leads to the concept of **hiding** of realization details. Whereas in the preceding paragraph we abstracted from constructions to retain the notions we conceived on the fly, we now identified some major abstraction step in which the notions are qualitatively quite different. They are much more worthy and constructed separately. This has been elaborated extensively elsewhere, e.g. in [4] and [5], so we can remain brief here. It should however be pointed out that hiding facilities are much less crucial than naming facilities, since conventions can help for tasks upto a certain critical size.

After this discussion of the naming, grouping, and hiding aspects, we can classify two kinds of abstraction in programming:

weak abstraction
: Naming is employed for the sake of identification of significant construction steps or modelled objects, and retaining notions in the program text that were conceived in the problem solving process. Such names simply stand for their realizations with all the properties thereof.

strong abstraction
: Applying the hiding principle, a name used in a context different from the one of its realization appears to have other properties than the entities from which it is constructed.

Both weak and strong abstraction are needed, but their use is supplementary: Their intended use is for local and global programming respectively.

3. Consequences for the design of an educational language

A programming language to be used in (initial) education has to support all the phases in the problem solving process. As it is simply impossible to introduce and use more than one language or notation, the constructs offered must be dedicated not only to the phase of coding, but also to the phases of modelling and concretizing the model into terms of the underlying abstract machine (the primitve algorithms and objects of the language).

Therefore, right from the beginning we need naming constructs to support weak abstraction for

- **type synonyms** to introduce names specific to the model, e.g. weights realized by integers, and names for data compositions like arrays of some type;
- **data objects** for giving names to variables and constants that express their purpose; and
- **algorithms** for naming program parts, based on syntactically self-contained units of the programming language (like loops, conditionals, and expressions), and for (parametrized) procedures and operators.

To solve more complex problems the separation of program parts must also be supported. We would like to stress here that explicit constructs for strong abstraction are essential if we want to cope with substantial problems of complexity beyond a simple sorting routine.

The strong abstraction facilities desired for a language are:

- **Modules** as (static) encapsulation construction hiding all its constituents, except those made public in an explicit interface.
- **Algorithms** (procedures and operators) defined in a module and made public offer their parameter pack as only communication interface; their algorithmic realization remains hidden.
- **Data types**: New variables and constants of an abstract data type can be declared outside a defining module. These new objects can, however, only be manipulated with algorithms provided together with the type, as the structural composition of that type is hidden in the defining module.

The design of yet another programming language must be considered a major offense for every computer scientist these days. Yet, none of the languages available appeared really suitable for our aims, in particular naming was not sufficiently supported without other penalties. So ELAN was born in 1974. To realize the abstraction means wanted in a programming language for educational purposes, we had to limit ourselves in the diversity of possibilities. The overall guidance was simplicity, both in learning and teaching.

4. ELAN and its abstraction mechanisms

ELAN is an expression language in the Algol heritage. Instead of block-structure there are static, non-nested modules with interfaces; the only ranges are local, within a module, and universal. Actual design decisions of particular interest are reported in the following by way of examples which hopefully are selfexplanatory. For further details the reader is referred to the language description [2].

A good example for the use of recursion and the principle of backtracking is the search for the shortest path of a mouse through a maze looking for a tasty piece of cheese. For sake of simplicity we will only determine the length of the path here, not the path itself. The data structure used is a simple rectangular maze realized by a two-dimensional array of fields, which in turn are integers used as marks.

```
LET rows = 10, columns = 10;
LET FIELD = INT;

ROW rows ROW columns FIELD VAR maze;
```

The fields may be occupied by barriers, a piece of cheese, or a mark for an already visited field. We introduce some **synonyms** for arbitrary valued markings:

```
LET barrier = 9, cheese = 7, free = 3, visited = 5;
```

These considerations on the data structure might also be delegated to the definition of an abstract data type MAZE using the strong abstraction mechanisms of ELAN, as shown later. This approach, however, often turns out to be inappropriate for many of the small examples we use for demonstrations, as they become much too bulky for their purpose. Commitment to data structures has to go hand in hand with the design of algorithms, and in this particular example we have to decide now on the parametrization of the central search routine which we have in mind.

With these preliminaries, we can develop the main line of the program straightforward in classical top-down style (we had of course first to think of how to solve the problem in general and come up with the idea embodied in distance from):

```
build the maze;
INT CONST infinite path length :: rows * columns;
INT VAR start row, start col, path length;
get the startposition;
path length := distance from (start row, start col);
report path length.
```

This is a complete algorithm, though unfortunately not yet executable. Each sub-algorithm is described only by its name, but that's the key factor: We have to refine these names by more concrete algorithms and objects, until we finally employ only such ones that are provided (as standard) in the language. For this purpose ELAN has the language construct **refinement**, in the hopefully most obvious form possible:

```
build the maze:
    INT VAR i, j;
    FOR i FROM 1 UPTO rows REPEAT
        TEXT VAR line;
        get(line);
        FOR j FROM 1 UPTO columns REPEAT
            maze[i][j] := field marking
        ENDREPEAT
    ENDREPEAT.

field marking:
    TEXT CONST field image :: text(line, 1, j);
    IF field image = " " THEN free
    ELIF field image = "C" THEN cheese
    ELSE barrier
    ENDIF.

get the startposition:
    put("enter startpos (row, col)");
    get(start row); get(start col).

report path length:
    IF path length < infinite path length
        THEN put(path length); put(" steps to the cheese")
        ELSE put("Sorry, no path")
    ENDIF.
```

A remark on variables and constants: ELAN uses the concept of **accessright** for data objects both for declared objects and for routine parameters. An accessright of VAR implies the right to assign a value to that object anywhere in its range, whereas CONST objects may be initialized only in their declarations. Because declarations are executed, the value of a ELAN constant may be different every time after its declaration is re-executed (see field image in refinement field marking). The same philosophy applies to routine parameters, where the accessright is specified for the formal parameters.

The next algorithm is parametrized, and to be applied recursively as the essence of our solution. As the refinements

demonstrated above do not have a local data area, but share the one of their environment, and are furthermore not recursively usable, we now need the more powerful concept of **procedure**:

```
INT PROC distance from (INT CONST r, c):
    IF bad field
        THEN infinite path length
    ELIF cheese found
        THEN 0
    ELSE mark this field;
        compute distances via all neighbors;
        unmark this field;
        minumum of all distances + 1
    ENDIF.

bad field:
    IF field out of maze
        THEN true
        ELSE maze[r][c] = barrier OR maze[r][c] = visited
    FI.

field out of maze:
    r < 1 OR r > rows OR c < 1 OR c > columns.

cheese found:
    maze[r][c] = cheese.

mark this field:
    maze[r][c] := visited.

compute distances via all neighbors:
    INT CONST northern path :: distance from (r−1, c),
              eastern path :: distance from (r, c+1),
              southern path :: distance from (r+1, c),
              western path :: distance from (r, c−1).

unmark this field:
    maze[r][c] := free.

minimum of all distances:
    INT VAR minimum :: northern path;
    minab (minumum, eastern path);
    minab (minumum, southern path);
    minab (minumum, western path);
    minimum.

ENDPROC distance from
```

Again we used refinements for the realization of the procedure body. Please note that refinements share the context of the range they appear in, i.e. the main program, or the body of the (recursive) procedure, respectively. There is no penalty for their use, as their code is inserted by the compiler at their applications (no call overhead).

Except for a little routine minab this example is complete. It's purpose was a demonstration of weak abstraction mechanisms and their use in ELAN:

- the naming of algorithms like refinements and procedures,
- synonyms for constant values,
- the introduction of a type synonym, without the bulkiness of the full abstract type definition mechanism.

On the side of strong abstraction ELAN offers static modules (called packet) as visibility units and abstract data types. A fragment of a packet to define dates as an abstract data type will show the essentials:

```
PACKET dates
DEFINES DATE, date, today, +, text, day, month, year :

TYPE DATE = STRUCT(INT d, m, y);

DATE PROC date (INT CONST day, month, year):
    { a denotation procedure for dates in the program text }
    IF NOT valid date(day,month,year) THEN errorstop FI;
    DATE:( day, month, year )
    ENDPROC date;

TEXT PROC text (DATE CONST day):
    text(day.d) + "–" + monthnames[day.m] + "–" + text(day.y)
    ENDPROC text;

ROW 12 TEXT CONST month names ::
    ("JAN", "FEB", "MAR", "APR", "MAY", JUN",
    "JUL", "AUG", "SEP", "OCT", "NOV", "DEC");

DATE OPERATOR + (DATE CONST d, INT CONST days):
    d + [ days, 0, 0 ]
    ENDOPERATOR + ;

DATE OPERATOR + (DATE CONST d, ROW 3 INT CONST diff):
    { form new date as date after diff[1] days,
    diff[2] months, and diff[3] years }

    ...

ENDPACKET dates
```

Some points of interest:

- The new **abstract data type** DATE is different from all other types, including those with the same representation. Outside the defining packet, objects of this type can only be manipulated via the procedures and operators defined along with it in the same packet. Within a packet the representation is accessible via subscription and selection.
- Procedures and operators are **generic**, i.e. there may be many operators + or procedures text, which are identified not only by their name, but also by the number and types of their parameters. Operator indicants, e.g. the name +, bear uniformly the priority of the standard operators, so that there can be no confusion in the mind of the reader about relative priorities of operators, as is possible in Algol68. Generic algorithms allow uniform treatment of standard activities, like e.g. printing of items of any type.
- The proper use of denotation procedures like date in the example above allows a nice regular style of programming with abstract data types. The **constructor** used within that procedure forms an object of appropriate type out of the elements of its fine structure.
- Data objects declared on packet level, like month names, are global to all procedures and operators of this packet. They are allocated statically, and are initialized at begin of program execution. The classical problem of hidden variables, e.g. the seedvalue of a pseudo-random generator, can thus be easily implemented.

- Only types, procedures, operators and constants can be made **public** in the DEFINES interface. For methodological reasons, variables and synonyms are excluded from being freely propagated. Several abstract types can be defined by one packet; the issues of modularity and data types are decoupled in ELAN. Algorithmic and data abstraction are supported equally.

- Packets can be compiled separately and can be put into a **library**. Once so compiled they can be used in any other ELAN program. The standard types in the language conceptually include those defined by precompiled packets. You may think of your program as a sequence of card decks, only the last one(s) are yours, submitted for compilation and execution now. The names made public by a packet in its interface are known to all others following it. This scheme is certainly not suitable for large programming tasks, but it is simple and adequate for teaching modular programming.

5. Status of ELAN

ELAN implementations exist on IBM /370, SIEMENS 4004 and 7.000 series, TR440, and Z80 (the latter a multi-user ELAN system, called EUMEL), originating from the University of Bielefeld. An implementation for NIXDORF 8870 is under construction, and versions for PDP11 and VAX are planned for the near future. A portable interpreter for an ELAN subset on microcomputers has been developed at the KU Nijmegen as a replacement for BASIC, and is running on PDP11 and the Apple-II hobbycomputer.

ELAN has been used successfully since 1976 at many places in Germany, most notably at a number of secondary schools, for the education of school teachers in informatics (PH Berlin, University of Bielefeld), for university entry level teaching (basic computer science courses at Berlin Technical University use it regularly since 1978), and in introductory programming courses at the GMD (Gesellschaft für Mathematik und Datenverarbeitung, a federal German research and education facility).

ELAN and Pascal-E have been approved officially by a committee (Arbeitskreis Schulsprache (ASS)) of the German Federal Ministry for Reaserch and Technology (BMFT) for use in computer science education, and are recommended as the only official languages for teaching at secondary school level in Germany.

The experiences with ELAN and especially with its refinement concept have been so good that refinement preprocessors for other languages (Algol68, FORTRAN, and COBOL) were constructed. At the Catholic University Nijmegen, Algol68 with refinements [6] has been used for computer science education since 1977.

6. Classroom experiences with ELAN

The following notes summarize the authors' experience in introductory programming courses, mainly for students in computer science in their first year at TU Berlin. Similar results are reported from other places where ELAN in used.

The most striking experience in using ELAN is the ease with which a number of programming concepts can be introduced. This is largely due to the use of refinements: New computational patterns, e.g. various loop constructions, can be presented in a rather minimal form; that is to say the constituents (e.g. a loop body) are just again refinement applications. The clear form of these patterns without much 'noise', like say index calculations, is of major help teaching and understanding them. The necessary details are simply delegated to refinement definitions to be filled in later. This approach not only applies to the basic patterns provided by the language in the form of control structures, but also to more advanced ones, like the steps of a backtracking algorithm, as demonstrated in the previous section.

This smooth introduction of computational patterns on a small scale (each refinement is about 3 to 7 lines long) allows a shift of emphasis from teaching a programming language with all the associated details (syntax, construction rules) to systematic programming. Programming style evolves naturally; we need not press the issue based on abuse of syntactic constructions, but can treat it semantically, focussing on structuring the algorithm by refinements and on verbalization.

Besides this qualitative gain, there is a quantitative one: The total amount of material covered is about 30 percent more than in previous courses, in which we used languages of the Algol family.

The introduction of procedure declarations with all their bells and whistles requiring explanation (parameters, local and global ranges, recursion) can be postponed for quite some time. They are actually introduced only near the end of a first semester. For top-down programming, refinements are more appropriate. Even elementary data structures (row and structures) are introduced prior to procedures.

Module structure (packets) are introduced relatively soon after procedures, usually in the first third of a second semester, and systematic programming on a larger scale together with abstract data types is well within the scope of the first two semesters. Given this framework, problems can and actually are attacked by pupils and students in their first courses that were previously unmanagable in the classroom or as exercises due to their complexity or size.

Two additional beneficial effects of refinements should not go unmentioned: First, they are usually expressed in an imperative or value oriented phrase on a certain abstraction level; rephrasing the refinement name to its computational effect on the same semantic level often eases the formulation of associated pre- and postconditions. These must of course then be worked out in terms of the objects used in that refinement; they are naturally attached to the refinements as

significant computational steps. A technique for hierarchical verification of program correctness based on refinements can be introduced this way. Secondly, program texts can quite easily manipulated in sensible chunks, e.g. extended or `reshuffled`using a text editor. Thus exercises and little programming projects can naturally grow and be reused in following ones without such problems as exemplified by the renumbering problem for BASIC programs.

There are some difficulties too:

- Refinements are not a cure-all and can be abused. In the first place, they are not substitutes for procedures. Secondly, all data objects local to some range are global to all refinements of that range; indescriminate lumping together too much can bring their number beyond mentally managable limits. As programs grow larger, the transition from refinements to procedures for larger and more selfcontained algorithms must be made. This is a question of style and hindsight, which for a major part is only to learn by experience of ones own.

- Our whole approach is based on verbalization, but the literacy of pupils is often surprisingly low, leading to awful monsters of refinement names or undecypherable acronyms. The latter effect is often even worse for those students that have previously picked up some BASIC or FORTRAN like languages. The flooding of the market with microcomputers offering minimal versions of these dialects does real harm to education. At the more elementatry level, many pupils do have difficulties (a) to `spell the same phrase correctly twice` (application and definition of a refinement), and (b) to feed it into a typewriter keyboard. These are very down-to-earth difficulties with education in general; all we can do is repeatedly remind our fellow teachers.

- There is as yet no textbook employing the approach presented and ELAN available in your local bookstore, only an amelioration on top of Algol68 [6]. We have been too busy with language implementation and setting up the courses in various experimental ways in the past. But there is hope; an introductory textbook in German and one in Dutch are nearing completion.

These difficulties are for a part only temporal ones. They are burdensome, but not real problems.

7. Conclusion

The retrospective question whether the development of ELAN was worth the effort must for us undoubtedly be answered positively. The step from teaching programming languages to problem solving using a computer asks for significant linguistic support for relatively small abstraction increments. Such support was previously unavailable. Modularity and use of strong abstraction are essential subjects, and should be taught much earlier than is usually done: a language to be used in education has to provide simple, but adequate means for information hiding. Though certainly not perfect in all respects, ELAN has met its goals.

Acknowledgements

C.H.A. Koster pioneered many of the ideas presented here. ELAN was developed at Berlin Technical University by him and the authors with support from DFG under grant KO-588/2. The cooperation with R.Hahn and J.Liedtke (who also provided the first implementation), University of Bielefeld, are also appreciated. The development of a new portable compiler is sponsored by BMFT and NIXDORF Computer AG.

References

[1] B.Liskov, St.Zilles, Programming with abstract data types, SIGPLAN Notices, vol.9#4 (April 74), pp.50-59

[2] G.Hommel, J.Jäckel, S.Jähnichen, K.Kleine, W.Koch, C.H.A.Koster, ELAN Sprachbeschreibung, Akademische Verlagsgesellschaft, Wiesbaden 1979 (in German)

[3] L.G.L.T.Meertens, Program text and program structure, IN: Constructing quality software, P.G.Hibbard/S.A.Schuman (eds.), IFIP WG2.1/WG2.4 conference, Novosibirsk, May 77, North Holland Publishers, 1978, pp.271-283

[4] D.L.Parnas, On the criteria to be used in decomposing systems into modules, Communications ACM, vol.15#12 (Dec.72), pp.1053-1058

[5] C.H.A.Koster, Visibility and types, Conference on Data: Abstraction, Definition and Structure, Salt Lake City, 1976 SIGPLAN Notices, vol.11, special issue

[6] C.H.A.Koster, Th.A.Zoethout, Systematisch Programmeren met Algol68, Deel I: Inleiding in de Informatika, Kluwer, Deventer 1978 (in Dutch)

☞ **Enquiries** regarding the availability of ELAN should be directed to Dr. Jähnichen at TU Berlin.