Holistic Design of a Programming System

R. Anane

Software Engineering Group

ELBS

Duncan House, London E15 2JB

e-mail: anane@uel.ac.uk

Abstract

The different stages of the prevailing software development model involve the use of software development tools and methods that are usually based on different paradigms. The mismatch between the different levels of this hierarchical process is often a source of difficulty and has led to an increasing interest in the holistic approach to software design and implementation. This approach requires all the levels to be based on the same principles. This paper describes how it was used in the design and implementation of a small programming system which incorporates a functional language, an optimiser and a syntax-directed editor. It also highlights the advantages of the holistic approach.

Keywords: holistic design, semantic gap

Introduction

The different stages of the prevailing software development model are usually associated with different paradigms (Ref. 1). The mismatch between the various levels is often a source of difficulty and has shown the need for a holistic approach to system design (Ref. 2). This paper presents a programming system which incorporates a functional language, an optimiser and a syntax-directed editor. Both language and underlying architecture are based on the same principles and their structures are homomorphic (Ref. 3). The language is based on the λ-calculus and is implemented on an abstract architecture which performs reduction of the source. The first part of this paper gives a brief introduction to the current software development model, while the second part deals with the programming system and illustrates the holistic approach.

Software development

In the current software development methodology an application has to go through a number of phases that identify a hierarchy of abstraction levels as shown in Figure 1. The transition from one stage to the next defines a gap that is either explicitly bridged by the software engineer or by the system software.

The focus of conceptual modelling is to bridge the conceptual gap by producing a specification. The specification serves two functions: it states the software requirements and provides a high-level model of the situation under consideration. It is concerned with abstract objects and can be expressed by means of formal methods such as Z or diagrammatic methods such as ER model. Bridging the refinement gap involves the transformation of a specification into a program by mapping abstract objects into the virtual objects provided by the language of implementation. The refinement process may involve many intermediate stages depending on the formalism used for the specification and on the expressiveness of the programming language. Both conceptual gap and refinement gap are usually bridged by the software engineer.

A semantic gap, on the other hand, is defined as the gap between the programming environment and the representation of the program concepts at the architecture level (Ref. 4). The semantic gap is wider if the language and the underlying machine are based on different concepts and it is closed by the combination of compiler and operating system. In many procedural languages the underlying architecture is not transparent and the structure and behaviour of the underlying machine often manifests itself during debugging: diagnosis are expressed in terms of concepts alien to the language abstractions.

Impedance mismatch

This issue came to the fore in databases with the mismatch between the database model and the host language. This clash is evident between the relational model which promotes a set-oriented view of data modelling and the procedural paradigm which is record-oriented. The existence of two type systems often leads to loss of information at the interface (Ref 5). In software development the impedance mismatch can be illustrated by the extreme case of an application where the problem is stated in English, the specification given in Z, the code in LISP which is then run on a traditional architecture. The transition from one stage to the next is accompanied by a significant paradigm shift. A number of underlying models are invoked: natural language, set theory, λ-calculus and finally the von Neumann architecture whose formal model is the Turing Machine.

The spanning of the refinement gap in this case would be an attempt to reconcile a set-theoretic model with implicit specifications and relations as the primary abstractions and a rule-based model which uses functions as the main building blocks. As for the semantic gap, the mapping of a LISP abstract machine onto the von Neumann architecture requires the mapping of non-linear structures such as trees onto a linear memory, and of a model of computation driven by function application onto a model where a single threat of control is vested in the program counter (Ref. 6).

The use of different paradigms at different levels of the abstraction hierarchy often results in unproductive mappings of information structures. Moreover, The existence of a number of paradigms at various levels has important implications for the software engineer who has to carry a lot of knowledge about many computational models and has to adjust dynamically to conflicting requirements. In addition to the inherent complexity of these models, the semantic content is often weakened by the downward movement along the hierarchy of the abstraction levels. The mapping between levels leads to the reduction in the granularity of the objects that are manipulated (Ref. 7).

Closing the gaps

Conceptual modelling is a cognitive activity that will always require human intervention. Historically, in the imperative paradigm, the closing of the gaps has been a trade-off between the refinement gap and the semantic gap. An assembly language creates a minimal semantic gap that leads to a wide refinement gap. The introduction of procedural, control and data abstractions in high-level languages was aimed at reducing the refinement gap and creating a platform for productive software development. However, the software crisis highlighted the shortcomings of the procedural paradigm and provided the impetus for the investigation of alternative architectures (Ref. 8). Unlike the traditional approach where software systems can be considered as abstractions of the von Neumann model, the modern approach to computer systems design is characterised by the migration of high level concepts into lower levels. Many computer systems are language-directed or paradigm-driven. The object-oriented paradigm has influenced the design of REKURSIV (Ref. 9) and the iAPX432 computer in particular. This architecture has implemented certain operating system functions in hardware and was designed to run Ada programs (Ref. 11). The CSP-Occam-Transputer combination offers a good illustration of the concern over the refinement gap and the semantic gap. The three systems deal with a common entity, the process, and use message-passing for interprocess communication (Ref. 12). The tying up of all the levels to the same abstractions avoids paradigm shifts and forms the basis of a holistic approach to system design and implementation. It ensures that the movements along the abstraction levels are minimal.

The functional paradigm.

The λ-calculus is the paradigmatic language for functional languages which possess important mathematical properties. Referential transparency states that the meaning of an expression depends only on the meaning of its subexpressions and ensures that, unlike procedural languages, an operational model is not crucial to the understanding of functional programs. In the functional paradigm, the refinement gap is usually not very significant because the declarative nature of functional languages enables them to be used as specification languages (Ref. 13). A lot of work has therefore been devoted to the reduction of the semantic gap by designing and implementing abstract architectures to support functional languages.

The implementation of functional languages falls under two categories: reentrant machines and substitution machines (Ref. 14). In the first category, pure LISP was one of the first attempts to implement a functional language on a von Neumann architecture. LISP, unfortunately, supported dynamic binding, an implementation feature that violates referential transparency. With the SECD machine Landin introduced an architecture that offered a sound basis for the implementation of functional languages which was used in the first version of SASL. This architecture was, however, influenced by the von Neumann model and was described in terms of state transitions (Ref. 15). The main disadvantage of reentrant machines is that they do not support directly β-reduction, the substitution mechanism of the λ-calculus.

Although some reduction architectures were designed to reduce the λ-calculus itself (Ref. 16), most recent substitution machines operate either on combinators or supercombinators (Ref. 17). Supercombinator implementations require the transformation of the original λ-expressions and are characterised by a grain of reduction that is relatively large. Combinator reduction , on the other hand, has the benefit of simplicity and its self-optmising properties were exploited in the second implementation of SASL. The grain of the combinator code is however is very small and the semantic gap is partially closed by the evaluation mechanism.

A functional programming system

A holistic approach has been used to address the issue of the reduction of the semantic gap in the design and implementation of a small functional programming system called ALGER. The abstract architecture is designed to support directly the language abstractions by ensuring that

1. both language and abstract architecture are based on the same principles.
2. a homomorphic mapping is established from language constructs to underlying abstract machine constructs.

The representation is in terms of the abstract syntax of the language (Ref. 18) while the evaluation mechanism is based on the β-reduction of the λ-calculus.

The language

ALGER is a higher-order language with non-strict semantics. The language allows memoized functions and incorporates the list as a partial function. Programs can be expressed as nested functions as shown by the definition of a higher-order function which is designed to generate power functions like `square`.

```
power = IN n OUT f WHERE
            f = IN x OUT r WHERE
                r = h[n,x];
                h = IN [m,y] OUT
                      IF m == 0 THEN 1
                      ELSE
                         y * h[ m-1, y]
                      FI
                NI {h}
          NI {f}
  NI {power}
```

This notation shows explicitly the expected result of a function. In this case, the application of the function produces another function. The list, as a partial function, can be used as an alternative for expressing the conditional expression, by specifying a pair applied to a boolean expression or to `1` or `2`. Thus the conditional expression, `IF p THEN e1 ELSE e2 FI` is equivalent to `[e1,e2]p`.

The abstract architecture

The translation of a program is merely a transformation from its string representation into its internal representation and is performed by a recursive descent parser. The homomorphic mapping from program to internal representation is helped by the parser which associates a parsing procedure with each syntactic construct.

The abstract architecture relies on the analytic property of the abstract syntax to close the semantic gap: the syntax helps specify the structure of expressions, and facilitates the selection and recognition of subexpressions (Ref. 19). Under this implementation, functional programs are represented by their syntax trees which model the abstract syntax of the language. This representation can be a cyclic, directed graph because expressions are shared and recursion is represented by cycles. Nodes are labelled to identify the structure of the graph. The list of valid nodes is given below:

- Function (F), made up of the function body and the parameters.
- Closure (C), made up of the function and the environment part. A closure evaluates to a function when the free variables are defined and bound to their values.
- Application (@), made up of the operator and the operand.
- List(T), made up of the head of the list and the tail.
- Atoms. Atoms represent constant values or parameters (P).

The graphical structure of the function `power` is given in Figure 2. Each node in the graph is clearly identified, thus making it possible for the representation to be understood and manipulated by the abstract machine.

Reduction

The main function of the abstract machine is to interpret and transform graphs. The machine manipulates objects that are recognisable. The evaluation mechanism adopted in the implementation of ALGER is specified in the formulation of the β -rule:

> The application of a function to an argument results in an *instance* of the body of the function, with occurrences of the formal parameter *replaced* by the argument.

Function application involves two processes, the instantiation of graphs and their reduction. Instantiation of function graphs is made necessary in order to preserve the original structure of the graphs since the reduction process is destructive. The reduction process uses normal order evaluation and arguments are reduced on demand.

The reduction process requires a transformation rule to be associated with each type of node. The reduction is performed by a recursive procedure which operates on a graph according to the tag of its root node. The current node can be overwritten with a new node which in turn may invoke its corresponding transformation rule. The reduction process does not require the use of an external agent or auxiliary structures: control is dynamically derived from the current node of the graph and partial computations are also held in the graph.

A minimal semantic gap

The realisation of a minimal semantic gap in this implementation was made possible by a homomorphic mapping between the conceptual structures of the language and the structures of the underlying abstract machine. This homomorphic mapping includes, in particular, the structure of the symbol table which reflects the hierarchical structure of the corresponding program.

Under this implementation, accessibility is supported by the program structure and the absence of hidden mechanisms. The underlying system is transparent as there are no movement along the levels of abstraction identified earlier. One consequence of the design approach is the ability to invert program translation. The source can be easily generated from the graph and both string and graphical representation can be saved on disk and subsequently loaded with a minimum of overheads.

The closing of the semantic gap facilitates the implementation and the use of a syntax-directed editor. Editing can be performed directly on the graph by accessing any named expression in the program. Thus, in the definition of `power`, `h` can be accessed as follows: `power>f>h`. The ability to access and edit any named expression can be used as a basis for incremental program development. Partially defined functions can be entered, parsed and saved.

Evaluation

The internal graphical structure of the program presents a compact representation which favours large grain operations, promotes sharing of expressions and avoids redundant computations. With a minimal semantic gap graph transformation is equivalent to source to source transformation. This equivalence is evident when an expression is optimised. Optimisations, such as constant folding and removal of dead code are performed automatically on the graph. The optimiser, in the same way as the editor, is supported by the symbol table by selecting named expressions and reducing them. The optimising process affects both graph and symbol table. All redundant entries in the symbol table are removed and the table is reorganised to reflect the structure of the program.

Software reuse

In this programming environment, higher-order functions provide a safe mechanism for building a library of functions from a set of primitive functions. The square function can be generated by the following application: `square = power 2`. `square` is now part of the environment and can be unparsed:

```
    square = IN x  OUT r WHERE
             r = h[2,x];
             h = IN [m,y] OUT
                 IF m == 0 THEN 1
                 ELSE
                    y * h[ m-1, y]
                 FI
             NI {h}
  NI {square}
```

The underlying structure of higher-order functions is manifest in the function definition. The function `square` has been generated with its own graph and its own symbol table. It can be edited, optimised if need be, saved and loaded. Through a process of specialisation, this method can be used to generate more complex functions and contributes to the expressiveness of the language and the enhancement of the programming system.

Error handling

The one-to-one mapping ensures that intelligible error messages can be generated with references to named expressions in the source program. The handling of errors is also disciplined, and for completeness the error token `ERROR` can be returned as a value in its own right. For example, if the expression `[3*5,2/0,4]` is evaluated by the machine, the result would be the list, `[15,ERROR,4]` followed by a meaningful message about the type and the position of the erroneous expression. Partial results can be returned even if the evaluation of some subexpressions returns the error value.

Performance

Turner introduced two machine-independent methods of assessing the performance of SASL (Ref. 20). The first method refers to control, and expresses the execution speed of the code in terms of *reduction steps*, i.e the number of times the reduction procedure is called. The second method is associated with storage management, and gives a measure of the *work done* by counting the number of cells required for a particular computation. The measurements of performance for SASL and ALGER are based on three computations: a table of factorials, the Towers of Hanoi problem with five disks and a higher-order function. Table 1 displays the number of steps for each program under the two implementations.

The comparison shows that ALGER performs better than SASL in reduction steps, because the internal representation of functions in ALGER is more compact, as shown by a comparison of the size of the graphs of the factorial function for the two languages. In addition, ALGER manipulates larger units which require fewer reduction steps whereas the granularity of combinator code is much smaller. Table 2 shows that all the computations require more cells for ALGER than for SASL. The discrepancy is mainly due to the lazy instantiation technique used by SASL which avoids copying sections of graphs which are subsequently thrown away. ALGER, on the other hand, requires all the graph corresponding to one particular λ-abstraction to be copied. The duplication of graphs is required for the generation of new functions. The two tables show, however, that the higher-order function performs better on SASL because of the self-optimising properties of combinator reduction.

Conclusion

The reduction of the semantic gap was achieved by tying up the whole system to one particular abstraction and by ensuring that language constructs are reflected in the internal representation. The provision of low-level support for the higher levels creates an environment which facilitates a high degree of meaningful interaction between user and programming system and contributes to the collapse of the traditional levels of abstraction.

The performance of ALGER compares favourably with SASL and shows that the spanning of the semantic gap and the use of a high-level representation are not incompatible with efficient computation.

Acknowledgements

My thanks to Tom Axford , Antoni Diller and Peter Dodd to their helpful comments.

References

1 SOMMERVILLE I.: Software Engineering (Addison-Wesley 1996).

2 NICOL J.R.: 'Operating System Design: Towards a Holistic Approach?', *SIGOPS Operating Systems Review*,1987, **21**(1).

3 LUGER G.F. and STUBBLEFIELD W.A.: 'Artificial Intelligence and the Design of Expert Systems' (Benjamin/Cummings, 1989).

4 MYERS G.J.: 'Advances in Computer Architecture' (John Wiley, 1981).

5 ZDONIK S.B. and D. MAIER D. (Eds): 'Readings in Object-oriented Database Systems' (Morgan Kaufman, 1990).

6 TRELEAVEN P.C., BROWNBRIDGE D. and HOPKINS R.: Data-driven and demand-driven computer architecture, *Computer Surveys*, 1982, **14**(1).

7 GELERTNER D. and JAGANNATHAN S.: 'Programming Linguistics' (MIT Press, 1990).

8 BACKUS J.: 'Can programming be liberated from the von Neumann style? A functional style and its algebra of programs', *CACM*, 1978, **21**(8), pp613-641.

9 HARLAND D.M.: 'REKURSIV, Object-oriented architecture' (Ellis-Horwood, 1988).

10 KARNE R.K, 'Object-orineted architecture for a new Generation of Applications', Computer Architecture News, 1995, **23**(5), pp8-19.

11 COX D. *et al.*: ' A unified model and implementation for interprocess communication in a multiprocessor environement', *SIGOPS Operating Systems Review*, 1981, **15** (5).

12 HOARE C.A.R.: 'Notes on Communicating Sequential Processes', , PRG Technical Monograph PRG-33, (Oxford University Computing Laboratory, 1983).

13 THOMPSON S.: 'Functional Programming: Executable Specifications and ProgramTransformation', ACM SIGSOFT Engineering Notes, 1989, **14** (3).

14 KOGGE P.M.: 'The architecture of symbolic computers, (McGraw-Hill, 1991).

15 LANDIN P.: 'The mechanical evaluation of expressions', *The Computer Journal*, 1964, **6**, pp308-320.

16 DARLINGTON J. and REEVE M.:,' ALICE: a multiprocessor reduction machine for the parallel evaluation of applicative language in Proc. Int. Symp. Functional programming languages and computers architecture.' (Göteborg Sweden, June 1981), pp32-62.

17 PEYTON-JONES S.L: 'The implementation of functional programming languages' (Prentice-Hall, 1987)

18 AHO A.V., SETHI R. and ULLMAN J.D.: 'Compilers: Principles, Tools and Techniques' (Addison-Wesley, 1986).

19 ALLEN J.: 'Anatomy of LISP' (McGraw-Hill, 1978).

20 TURNER D.A.: 'A new Implementation Technique for Applicative Languages', *Software Practice and Experience*, 1979, **9**(1).

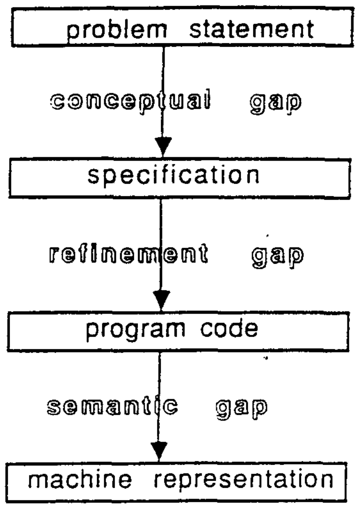

Figure 1: Abstraction levels

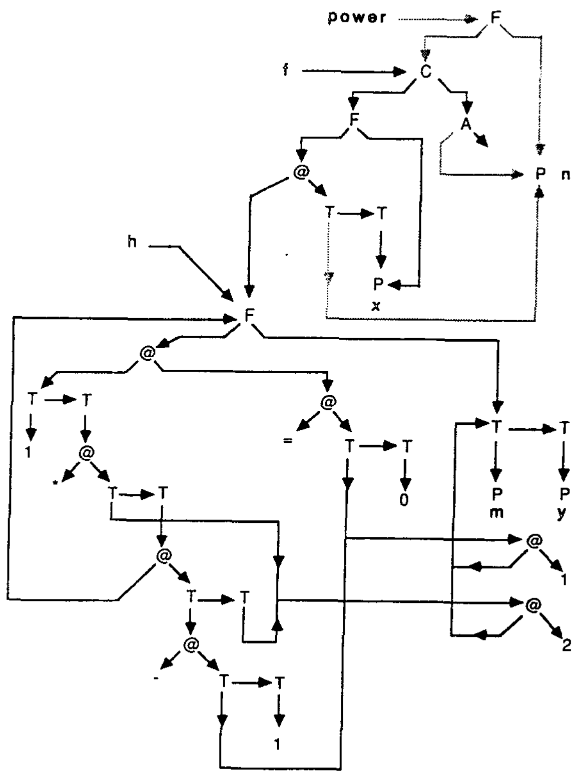

Figure 2: graph of power