An early use of continuations and partial evaluation for compiling rules written in FOPC

Erik Sandewall
Department of Computer and Information Science
Linköping University, Sweden
http://www.ida.liu.se/~erisa/

December 2, 1996

Abstract

This brief historical note describes research which was done in the period 1970-1973, and where continuations were introduced in a fairly pragmatic way together with partial evaluation in order to compile "rules" expressed as statements in first-order predicate calculus. The methods used at that time were quite straight-forward, but they may be of some interest for the present workshop as an "early bird" result.

1 Background

Around 1970, there was considerable interest in the artificial intelligence community for "programming language for AI". The mostly referenced competitors for the title were PLANNER [5] and QA4 [8], but there were also other proposals such as POP-2 [2] and LISP A [9].

At the time, I was starting a research group in this area at Uppsala University in Sweden. As it became increasingly evident that these languages were mostly used for deduction with the addition of certain additional techniques (such as non-monotonicity), we started to investigate the idea of using the language of first-order logic as the "programming language", and to implement it by a compilation technique, where each formula in First-Order Predicate Calculus (abbreviated FOPC in this paper) would be translated to one or a few program fragments. The overall deduction would then be performed by a main program into which these various program fragments were inserted.

The main parts of this work was done in the period 1970-1973 by Anders Haraldsson (then a graduate student) and myself, with some follow-up work in the following years. The only published articles describing the PCDB system itself were [10] (describing the principles and a first prototype implementation) and [4] (describing the first thorough implementation, with extended capabilities). As a matter of curiosity, it may be observed that the latter paper, which was presented at the IFIP 1974 conference, appeared in the same conference session as Kowalski's original presentation of logic programming. This was reasonable: Kowalski of course used the strategy of interpreting the FOPC formulae whereas ours was to compile them.

We all know that it was logic programming that won the game. However, the compilation work does have a certain historical interest in that it provided the framework for developing some techniques which have since then found much more general applicability, namely *partial evaluation* and *continuations*. Our first paper describing partial evaluation and its use for compiling a subset of FOPC was published in 1971 [10]. The extension of the approach which also used continuations (which we called "remainder procedures" at the time) was first published in [12] in 1973.

In comparison with Futamura's pioneer work on partial evaluation, which appeared as a journal article in 1971 [3], we emphasized the use of partial evaluation for compiling special-purpose languages, whereas he apparently saw it as a technique for a compiler-compiler. With respect to continuations, our first publication appeared the year after Reynolds' early article [6]. We were unaware of his paper at the time the work was done, but the connection was pointed out to us by a referee.

These references of ours were published in conference proceedings. The 1973 paper was later republished in journal form but without modifications in [13]. A more general article on the partial evaluation techniques that had been developed in this context was also published as a journal article [1] around the same time.

In this paper I will make a brief summary of this early work as an addendum to John Reynolds historical overview paper [7]. For historical accuracy, I will present it as much as possible in the same terms as it was described at the time. The formulation in terms of contemporary terminology will be left to the reader. The presentation will be restricted to the relational case, although the original system also allowed the use of functions in the sense of the logic.

2 Implementation strategy of the PC database

We addressed deduction problems which could vaguely be described as "predicate calculus data base" problems, or more precisely, problems where one has a large number of ground unit axioms and a more limited number of non-ground and non-unit axioms. The set of ground unit axioms is called the "data base". We assumed furthermore that the deduction tasks were of the following simple kinds:

- "closed queries": verify whether an additional, given ground unit clause is provable from the given axioms.
- "open queries": given a unit clause $P(x)$ where x is the only variable, find one or more constants c_i (among the constants that occur in the data base) whereby $P(c_i)$ is provable from the given axioms.
- "forward deduction": given that one more ground unit clause is added to the data base, infer additional ground unit clauses which follow from it, and add them to the data base, possibly triggering additional forward deduction.

We did not aim for a logically complete implementation, of course: the perspective was strictly one of treating the FOPC as a programming language whose semantics was defined in procedural terms.

We used LISP as the implementation language, for the simple reason that LISP made it easy to represent both the FOPC formulae and the LISP code generated from them. For the case where there are only relations and no functions, our implementation would associate several procedures with each relation symbol. For a predicate (relation of one argument) P, there would be one procedure $P_{cq}(x)$ to be used for answering the closed query $P(x)$, another procedure $P_{oq}()$ without arguments to be used for obtaining the list of all x (in a data-base sense of "all") such that $P(x)$; yet another procedure $P_{fi}(x)$ which would perform all one-step forward inferences that were appropriate after having added $P(x)$ to the database, and possibly a few more.

Similarly, for a relation $R(x, y)$ of two arguments, there would be one closed-query procedure $R_{cq}(x, y)$ and one forward inference procedure $R_{fi}(x, y)$, and *two* open-query procedures: one procedure $R_{oq1}(x)$ returning the list of "all" y such that $P(x, y)$ for given x, and another procedure $R_{oq2}(y)$ returning the list of "all" x such that $P(x, y)$ for given y.

Each of these procedures was then supposed to be a composite expression into which the axiom compiler could insert successive program elements as new axioms were added for a previously declared relation. Open and closed query procedures were essentially formed as Lisp "or" expressions, in the simplest case. In other words, they tried several "methods" (to use a term that was popular at the time) in succession. The forward deduction procedures were similarly formed as Lisp "progn" expressions, that is, as performing several forward inference steps in succession. More sophisticated composition operations were then introduced in order to control the inference.

With this general framework, a rule such as

$$R(x, y) \wedge P(x) \Rightarrow S(y, x)$$

could be compiled into contributions to several of the attached procedures. For example, the closed-query procedure $S_{cq}(x, y)$ for S would be assumed to have the form

$$\lambda[u, v] or(\psi_1, \psi_2, ...\psi_k)$$

and the rule just stated would contribute to it one more program element ψ_{k+1} of the form

$$and(P_{cq}(v), R_{cq}(v, u))$$

saying "invoke the closed query procedure of P with v as the argument; if it succeeds then also invoke the closed query procedure of R with (v, u) as the arguments; if it also succeeds then $S(u, v)$ has been proven".

This particular implementation results of course in a depth-first search. Other composition operations and a simple main program allowed other search strategies, such as breadth-first search.

It follows that this implementation strategy could result in quite specialized versions of the compiled code. We thought of this as an advantage rather than as a problem, and it was part of our strategy to develop tools for designing non-trivial methods of controlling the search. Thus, the problem of deduction had been decomposed into two parts: designing the overall control strategies, and compiling the axioms themselves.

The attached procedures that have been described so far were used for performing one step of deduction, and would call each other recursively for the purpose of search. Obviously, at some point one must check whether a particular ground unit clause was *explicitly* stored in the data base. Our implementation allowed each relation to have its own representation for literals, for example by using the property-lists of one or more of the arguments. A few more attached procedures were used for this purpose, that is, for storing a fact and for answering a closed and an open query by direct database lookup.

3 Continuations

A certain kind of continuations evolved in a natural fashion in this context when we considered how to introduce nontrivial deduction strategies under program control. When compiling a rule of the form

$$\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_k \Rightarrow \beta$$

into a contribution to the closed-query and open-query procedures for β, one would like to be able to control the extent of search that is done for each of the literals α_i.

The following fairly general approach was found. First of all, the attached procedures such as R_{cq} were all defined so that they would return one or more *sub-queries*, where each sub-query was a structure consisting of a relation symbol, a list of arguments for some of its argument positions (possibly none, possibly all), and a new procedure. The executive for the search was organized so that it maintained a working structure of queries and performed a simple cycle repeatedly. In each cycle it would select one query, invoke the appropriate attached procedure for the relation mentioned in that query while giving it the stated arguments, and accumulate the returned sub-queries to its working structure.

The method for compiling a rule of the form

$$\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_k \Rightarrow \beta$$

for open-query deduction, for example, was as follows. First of all, select a suitable ordering on the α_i, presumably favoring orderings where each successive α_i contains at most one variable which has not occured in the invocation or in a previous literal. In other words, at most one variable has to be assigned a value in each literal. Then, identify which of the literals are to be subject to deductive search, and not merely immediate lookup.

In a trivial case, only α_k is chosen for deductive search, and then the rule is compiled into a program element that "executes" the successive (compiled) literals α_1 through α_{k-1}, assigns values to variables by immediate lookup, and returns the appropriate variant of α_k as a query. This will be a loop-free non-deterministic program; it is immediately converted to a deterministic program with loops. Either way, it may generate a number of new subqueries (variants of α_k with different argument combinations) which are then all taken care of by the executive.

The interesting case arises if one or more non-terminal literals are selected for deductive search. Suppose $k = 4$ and only α_2 is selected. The compiler will then generate a program element that "executes" α_1 in the sense of non-deterministically performing all assignments to new variables in α_1 that can be obtained by immediate look-up. For every such assignment, it will return a query obtained from α_2, that is, a query containing the relation symbol of α_2 and values for those of its arguments which have been determined at that point. But every such sub-query must also contain a third part, namely a program element that "executes" α_3 and α_4 given also those assignments of values to variables that are obtained in the deductive search for the query obtained from α_2.

We proposed to do this using what we called a "remainder procedure": a function closure containing those variable bindings which have been obtained so far while traversing the rule at hand, and a function obtained by compiling the rest of that rule.

Clearly, one must allow for cascades of remainder procedures: either the compiled rules or the top level executive must account for the case where a query containing a non-vacuous remainder procedure generates subqueries containing an additional remainder procedure. The execution of these procedures will then be dynamically nested in the obvious fashion.

One may well ask whether it was necessary to introduce the full generality of procedures for the purpose that has now been described, or whether it would have been sufficient to maintain the rest of the rule as a data structure consisting of the literals and the required auxiliary information, and to decode them as needed? One must remember, however, that the concern for performance was much more acute at that time than it is now. Another part of our strategy was to reduce the number of function calls using partial evaluation and subsequent simplification. In particular, given that some of the literals were intended to be "executed" by direct access to the data base, using the tailor-made direct access procedures that were associated with each relation symbol, we were able to open up these remainder procedures to the point where they were defined directly in terms of access procedures, thereby removing several levels of invocation.

4 Closures and FUNARG:s

One other design which was obtained in the same context may also be of interest. In order to implement the "remainder procedures" in the queries, we needed a closure construct in the programming language. This is trivial in original Lisp 1.5, where they arise using the so-called FUNARG construct, but this possibility had been removed in subsequent variants of Lisp during the 1960's as variable bindings came to be implemented using ordinary stacks rather than as association-lists in order to improve performance.

There was a need, therefore, to re-introduce FUNARG:s in a way that was commensurate with the use of stacks. In [11], we proposed to modify the Lisp syntax so that one would write expressions of the form

```
(FUNCTION  (LAMBDA (...)(...))  (x y...))
```

in order to create a closure consisting of the indicated lambda expression and bindings of the variables x, y,... to their (dynamically defined) values in the current context. This proposal went well with our use of closures for compiling FOPC rules. It was adopted in several subsequent variants of Lisp, such as Interlisp, but it became unnecessary with the introduction of lexical scoping in Common Lisp.

5 Limitations of the approach

One of the reviewers of this article suggested that I should comment on what were the limitations of our approach, and why it did not catch on at the time. With respect to our own group, the work on partial evaluation was pursued through several Ph.D. theses: Haraldsson on partial evaluation of iterative statements (1977), Emanuelsson on the use of partial evaluation for compiling a pattern-matching language (1980), and Komorowski on partial evaluation in Prolog (1981). However, the use of continuations was fairly marginal (or possibly implicit) in those projects. The Predicate Calculus Data Base project, which has motivated the introduction of continuations, was not pursued, however. It had been conceived in order to be used in some other A.I. activities, namely for natural-language understanding and qualitative reasoning, and when those were discontinued, no one wanted to work on the PCDB. Maybe we were influenced by funding resistance for projects like PCDB (and ease of funding in other directions); maybe we just had too many ideas to work on within the group.

With respect to possible uses elsewhere, the A.I. world was fairly polarized with respect to the use of logic at that time. One part of the community argued that logic was not sufficiently expressive, and that full-fledged A.I. languages or expert systems languages were the right way to go. Others, who were in favor of logic, tended to go with logic programming or general-purpose theorem provers; the low-key approach to logic that was offered by PCDB-type systems was probably too mundane for them.

6 Future amendments

The following webpage:

`http://www.ida.liu.se/~erisa/pek/96/F/`

will be used for links to electronic versions of this paper and to amendments, commentary, possible errata, and other related information. It will be maintained for the foreseeable future.

References

[1] Lennart Beckman, Anders Haraldsson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[2] R.M. Burstall, J.S. Collins, and R.J. Popplestone. *Programming in POP2*. Edinburgh University Press, 1971.

[3] Y. Futamura. Partial evaluation of computer programs: an approach to a compiler-compiler. *J. Inst. of Electronics and Communication Engineers*, 1971.

[4] Anders Haraldsson. PCDB - a procedure generator for a predicate calculus data base. In *Information Processing 74*, pages 575–579. North-Holland Publishing Co, 1974.

[5] Carl Hewitt. PLANNER: A language for proving theorems in robots. In *International Joint Conference on Artificial Intelligence*, pages 295–301, 1971.

[6] John Reynolds. Definitional interpreters for higher order programming languages. In *Proc. ACM Conference*, 1972.

[7] John Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4), 1993.

[8] J.F. Rulifson. QA4: a procedural basis for intuitive reasoning. Technical report, AI Center, Stanford Research Institute, 1972.

[9] Erik Sandewall. LISP A, A LISP-like system for incremental computing. In *Proc. Spring Joint Computer Conference*, 1968.

[10] Erik Sandewall. PCDB, a programming tool for management of a predicate calculus oriented data base. In *International Joint Conference on Artificial Intelligence*, pages 159–166, 1971.

[11] Erik Sandewall. A proposed solution to the FUNARG problem. *SIGSAM Bulletin*, (17), 1971.

[12] Erik Sandewall. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. In *International Joint Conference on Artificial Intelligence*, 1973.

[13] Erik Sandewall. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. *IEEE Transactions on Computers*, C-25(4):342–346, April 1976.