A Modular Compiler Architecture for a Data Manipulation Language

Suzanne M. Embury and Peter M.D. Gray

Department of Computing Science, King's College, University of Aberdeen, Aberdeen, Scotland, AB9 2UE
E-Mail: {sme|pgray}@csd.abdn.ac.uk, URL: http://www.csd.abdn.ac.uk/~pfdm

Abstract. We describe a modular compiler architecture that has been developed for a functional data model DBMS. The architecture allows compilers for new sub-languages to be constructed rapidly, by reusing the components of the existing compiler, and allows new semantics and code generation strategies to be defined for existing language constructs. This point is demonstrated by the construction of a new compiler for an integrity constraint language, which required only two new modules to be added to the system. The most significant advantage of our architecture, however, is that it allows the DBMS itself to use the individual compiler modules, opening up a host of possibilities for run-time manipulation of application code.
Keywords: data manipulation languages, run-time compiling, semantic data models.

1 Introduction

The idea of a callable compiler was originally introduced into persistent languages, such as PS-Algol [33], as a way of overcoming the difficulties of building systems which must handle dynamically-introduced types in statically-typed languages. Programming environments, which support incremental development of code, are examples of such systems [13], as are database management systems (DBMSs). A callable compiler allows the implementation of embedded query languages, browsers and schema evolution capabilities, without sacrificing the security of static type checking [6]. However, as DBMS technology has advanced, the increasing need to manipulate not only the structure of data at run-time, but also its semantics, has meant that the ability to invoke compiler functions "on the fly" has become important for DBMSs themselves, and not just for the languages which implement them. The aim of a DBMS is to take over responsibility for routine management tasks from the user, many of which involve manipulation of code as well as data; deciding which code needs to be recompiled after a schema change, for example, or re-optimised as the cost of evaluating alternative access paths changes. Furthermore, as DBMSs are increasingly being seen as components in larger (possibly distributed) systems, the ability to combine, analyse and transform code retrieved from different sources is increasing in importance. Since DBMSs are organised around *data models* [17], with well-defined semantics, rather than *type systems*, and since they can rely on the availability of a repository of information about both the structure and semantics of the data (the metadata), the potential for more far-reaching and "intelligent" manipulation of code is far greater than it is for a programming language, with its low-level, low-semantics type system and minimal runtime metadata storage.

To perform this kind of manipulation requires the invocation of a compiler by a running program, but a callable compiler in the usual sense is too coarse-grained. It implements only a single function, transforming source text into executable code, and thus forces the DBMS to represent and manipulate code in one of these two forms—neither of which are suitable for the task. Source text is too "flat". Before it can be manipulated, it must be parsed and converted into some internal data structure, and before the modified code can be executed, it must be translated back into source language form, and parsed all over again by the compiler. The target language representation, on the other hand, is much too low-level to be easily manipulated. It is designed for efficient execution, and thus concentrates too much on storage semantics details, with only a weak representation of the original semantics.

What we need, then, is not a monolithic callable compiler, but one in which the front end and the back end can be invoked independently—the former to parse the source text and turn it into some format convenient for automatic manipulation, and the latter to take programs expressed in the internal format and convert them into executable code. This scheme allows us to represent code internally in the format which is best for the analysis and transformations we want to carry out, without the overheads of continual reparsing, or the need to implement the extra translation routines.

The extra flexibility gained by this division suggests the idea of splitting the compiler up into a series of independently callable modules, each implemented as a transformer from input to output, and each performing some particular step in the compilation process: parsing, optimisation, code generation, etc. This would provide the DBMS with an assortment of tools for manipulating code in various ways, which could either be used individually (for example, when recompiling some method code from its internal representation), or in different combinations, opening up a host of possibilities for automated run-time maintenance of code.

This paper describes the implementation of a compiler along these modular lines for the data manipulation language Daplex [29], in the context of P/FDM [18, 9], a Prolog implementation of Shipman's Functional Data Model. We begin (Sect. 2) with a brief overview of the Daplex language itself, and discuss some of the compilation issues it presents. Section 3 then describes the architecture of our compiler, and how it has been divided into independent modules. Section 4 gives a description of the intermediate code format used to represent program code within P/FDM, and Section 5 gives some examples of the novel ways in which the compiler modules can be used to extend system functionality. Finally, we discuss some of the issues raised by the idea of modular architectures for database language compilers, including the slightly unusual role played by Prolog in our system (Sect. 6), and conclude (Sect. 7).

```
create private module unidb

declare person ->> entity
declare name(person) -> string
key_of person is name

declare lecturer ->> person

declare student ->> person

declare year(student) -> integer
declare course ->> entity
declare code(course) -> string
declare year(course) -> integer
key_of course is code

declare takes(student) ->> course
declare teaches(lecturer) ->> course;
```

Fig. 1. A simple university database schema

2 Overview of the Daplex Language

Daplex is a high-level, functional style language that was designed as a "conceptually natural" interface to the Functional Data Model [29]. It consists of two sub-languages—one for data definition (the DDL) and one for data manipulation (the DML). The DDL allows the specification of FDM schemas, and the classes and functions which they contain. The schema given in Fig. 1, for example, describes a simple university database with classes storing information about people (`person`) and courses (`course`). The `person` class is divided into two subclasses, `student` and `lecturer`. The attributes of each class are modelled as functions mapping instances of the argument class onto either single values of the result type (indicated by a single-headed arrow) or sets of such values (indicated by a double-headed arrow).

Programs in the DML consist of a specification of one or more sets of database values, followed by a sequence of operations (called "actions") which must be performed on the elements of the sets. For example, the following program updates the database to ensure that all third year students are registered for all third year courses, and displays details of the updates made:

```
for each s in student such that year(s) = 3
  for each c in course such that year(c) = 3
include c into takes(s)
print(name(s), 'now registered for course', code(c));
```

The examples given above illustrate the distinction between the DDL and DML sub-languages of Daplex. The two sub-languages are not completely disjoint, however, as fragments of DML may be embedded within schemas to define methods:

```
define num_courses(s in student) -> integer in unidb
  count(takes(s));
```

This DDL method definition uses a DML set specification (`count(takes(s))`) to describe the result of the `num_courses` function intensionally (rather than as a set of stored mappings). Another important distinction is that programs expressed in the DML are compiled (into Prolog) while DDL schemas are interpreted directly into calls to the data definition primitives of P/FDM. This difference can be seen clearly in the architectures of the two sub-language compilers, which share a common front-end but which have quite different back-ends.

Before we move on to the description of the modular compiler architecture, it is worthwhile to consider the ways in which Daplex differs from a conventional programming language, and what effect this has on the requirements for compiling such a language. Firstly, the separation of data definition and data manipulation in Daplex means that new data structures cannot be declared within a data manipulation program. This greatly simplifies the process of statically type checking programs, and eases the optimisation and analysis of DML code. In fact, our implementation of Daplex goes further than this in that data update (or state change of any kind) is restricted to the actions part of each DML program. So, for example, in the program given above, which registers third year students on third year courses, it would be illegal for either the update or the `print` statement to appear in front of or between the two `for each` loops. Set specifications, then, are kept completely free of side effects, in order that they may be treated as referentially transparent for optimisation and analysis purposes. This is true even when methods are called within set expressions, as the syntactic structure of Daplex forces users to declare whether methods are side-effect free or not when they are defined.

The implication of this separation of data definition and manipulation for the language compiler is that complex symbol tables need not be maintained during compilation. All data structure information is stored in an easily available form in the metadata, and may be accessed by the compiler at any time. Otherwise, we need only keep track of which variables are currently in scope, and what their type is, using a simple list structure.

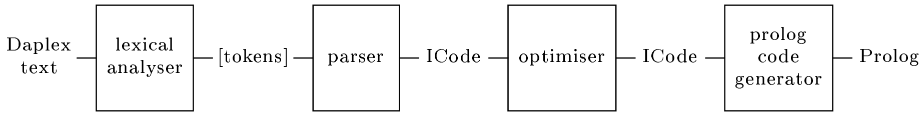
The second point to note about the compilation of Daplex is that the target language (Prolog) is itself relatively high level. This means that many of the usual difficulties of code generation (such as register allocation, and run-time memory management) are not applicable to our compiler, with the result that issues for the design of our much higher-level intermediate code format are rather different than for a conventional compiler. This is discussed further in Sect. 4, where we look at the use of ZF-expressions for representing code.

Finally, we note that although our target language is dynamically typed, Daplex programs themselves are statically type-checked against the available metadata. This gives us the flexibility of Prolog's weak type system for system code, while giving the user the extra security of a strongly-typed language. Run-time type checks are built into the data manipulation primitives provided by P/FDM and therefore come free of compilation effort. There is admittedly some performance cost with this, as the run-time checks cannot be optimised and may be performed more times than is necessary, but this is a small overhead compared to the cost of executing each primitive.

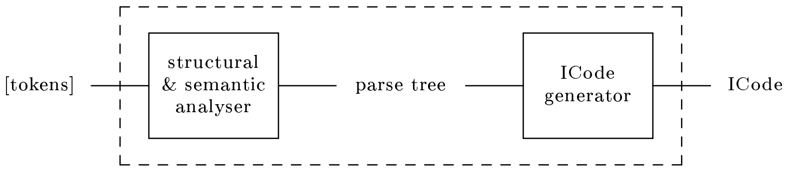
3 The Modular Compiler Architecture

3.1 The Data Manipulation Language Compiler

The major building blocks which make up the Daplex compiler are essentially those which would be found in any standard programming language compiler [1]. The DML compiler, for example, consists of a pipeline of the following units:

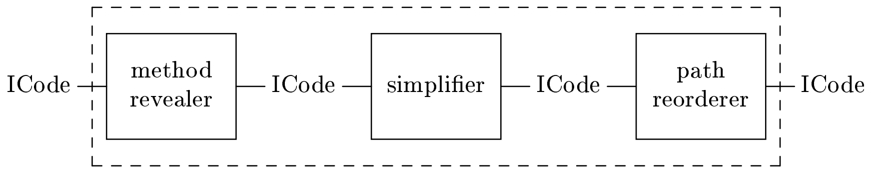


Each of these units is constructed as an independent module which may be called in isolation from any of the others. The **lexical analyser** accepts Daplex text as input and produces from this a list of lexical tokens. This list is passed on to the **parser** module, which uses Prolog's concise and flexible DCG rule facility to convert the input into an intermediate code format. In fact, the parser consists of two sub-modules:



The **syntactic & semantic analyser** parses the token list, and performs some basic semantic analysis and type checking. Its output is a parse tree, constructed of nested Prolog terms, which describes the entire input program. This is then passed to the **ICode generator**, which transforms the full parse tree into the simpler intermediate code format. This format, which we call ICode, is based upon ZF-notation [26, 18]. It retains the semantic content of the parse tree representation, while discarding much of the accounting and type information required only for parsing. It describes only the essential semantics of the input program, while the parse tree also contains information about exactly which syntactic constructs were used to express these semantics. A more detailed description of the intermediate code format is given in Sect. 4.

Once an ICode version of the program being compiled has been constructed, it is passed to the Daplex query optimiser [20, 21]. Like the parser, this module is split into sub-modules:



However, unlike the parser's sub-modules, each of these operates as a transformer from ICode to ICode, and is therefore a candidate for independent use. The first optimising sub-module, the **method revealer**, walks the given ICode structure looking for calls to intensionally defined functions (i.e. method functions), and, where possible, replaces each such function call with its intensional definition. This allows "global" optimisation of the query with, for example, removal of common sub-expressions from within method definitions [14, 21]. For example, the method revealer would replace the expression `num_courses(s1)` with:

```
count(takes(s1))
```

This may require renaming of some duplicated variables, but is an otherwise straightforward process, since method bodies are stored in ICode format, ready for revelation.

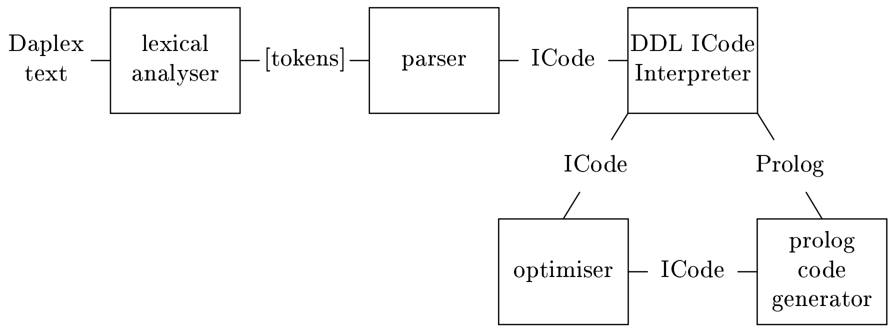
Having revealed the definitions of all suitable method functions, the expanded ICode program is passed on to the **simplifier** sub-module. This module performs conventional optimisations, such as removal of common sub-expressions, and of invariants from within loops, producing a pared down version of the original program. This new version, still in ICode format, becomes the input of the last optimising sub-module, the **path reorderer**. This module performs the kinds of optimisations most commonly associated with database query languages, in that it uses accumulated statistics about the size of the various database classes, and a cost model of database access, to search for the cheapest way to navigate between the data items specified in the query. The result produced by this is a standard ICode representation of the program, except that the ordering of the individual components is now significant, in that it specifies the execution ordering chosen by the optimiser as being the best it could find.

The final module in the DML compiler is the **Prolog code generator**, which translates an ICode structure into a fragment of Prolog code that implements the required path semantics. The code generator module is order preserving, so that the relative ordering of the ICode constructs within the list is mirrored in the ordering of the goals in the resulting Prolog fragment. This is important as the ICode ordering is the only means of communicating the navigational path chosen by the path reorderer to the other modules.

This module completes the transformation of Daplex DML programs from text strings to Prolog goals. The resulting code fragments are either executed immediately and discarded, or are labelled and stored in the metadata for future use, according to the user's instructions.

3.2 The Data Definition Language Interpreter

The arrangement of compiler modules described above is specific to the DML. The interpreter for the Daplex DDL has the following alternative modular arrangement:



Here, the same lexical analyser and parser that were used in the DML compiler are reused to generate an ICode representation of the data to be defined. Once such a representation has been produced, it is passed to a new module, the **DDL ICode interpreter**. This module walks the input ICode, making appropriate calls to the data definition primitives of P/FDM as it goes. This otherwise straightforward process is complicated only by the need to handle method definitions and their embedded DML expressions. When an intensional definition is encountered by the DDL interpreter, it invokes the standard optimiser and code generator modules to generate a fragment of Prolog that will compute the result of the function, given its argument values. This code fragment is then passed to the relevant data definition primitive, along with the original ICode definition, for storage in the metadata.

From this we can see that the only real difference between the two language processors is the way in which the basic modules are connected together. Having implemented the Daplex DML compiler, we had only to add the new syntactic rules to the parser, and to build the interpretation module for the new ICode constructs, in order to build the DDL interpreter. The remaining modules were used unchanged. This was possible because we had standardised on an expressive but easily manipulable intermediate code format that could serve as a basis for the interchange of program semantics between modules, and as a "storable" but declarative representation for making behavioural information (i.e. methods) persist.

4 The Intermediate Code Format

The intermediate code format used in the current Daplex compiler is an extension of that designed for the original compiler [26], which was based on ZF-notation[1]. In this notation, a set is described by a pattern, giving the form of the elements of the set, and a declarative specification of what values are set members. For example, the set of third year students is described by the ZF-expression:

```
[ s | s ← student; year(s) = 3 ]
```

The pattern (here just the variable `s`) is given to the left of the '|' symbol, and the set specification to the right. Set specification consists of lists of *generators* for variables, having the form "`var ← set`", and predicates, called *restrictions*, which act as filters on the generated values. So our example set specification consists of a single generator, assigning student instances to the variable `s`, and a single restriction, filtering out those student instances whose `year` attributes evaluate to something other than the value 3.

The Daplex compiler uses a very similar form to this for representing sets. For example, the Daplex set:

```
s in student such that year(s) = 3
```

is represented internally by the ICode list:

```
[ generate(student, var(1)),
  restrict(year, [student], [var(1)], var(2)),
  expression(=, var(2), 3)
]
```

The `generate/2` term takes a class name and a variable, and corresponds to the ZF-expression generator construct. The restriction "`year(s) = 3`" is represented by the remaining two constructs. The set description would then be translated into the following fragment of Prolog:

```
( getentity(student, V1),
  getfnval(year, [V1], V2),
  V2 = 3
)
```

Here, `getentity/2` is the primitive provided for enumeration of class instances, with the result variable (`V1`) being bound to successive instance identifiers on backtracking. Similarly, `getfnval/3` takes a function name, a list of argument values and a result variable, and attempts to apply the function to produce the result. Thus, this fragment of Prolog, when executed, will bind the variable `V1` to all `student` instances for which the `year` function evaluates to 3, one by one, on backtracking.

[1] This terminology is taken from [32]. Subsequently, the terms *list comprehension* and *set comprehension* have been applied to the same notation.

The intermediate code format contains constructs describing all the unique semantic concepts described by the Daplex DDL and DML, ranging from declarations of new schema elements:

```
declaration(SchemaName, NewSchemaElements, MethodFiles)
```

to DML imperatives, like the update example given on page 3:

```
imperative(InputVars, SetICode, ActionICode)
```

Our intention here is not to give a detailed description of our particular intermediate code format, but to give enough of a flavour to illustrate the features that are desirable in such a format, given the modular architecture described here. As it stands, our format lacks many of the control structures commonly found in more traditional languages (such as `while` and `repeat` loops) or any notion of destructive assignment to program variables, since Daplex itself lacks these features. There is no reason why structures to support these concepts could not be added to our intermediate format, although we would certainly lose some of the ease with which ICode can be analysed and transformed with the introduction of unrestricted state change. The important point is that, unlike other intermediate code formats (such as 3 place codes, or reverse polish notation), ICode retains most of the source code semantics explicitly. This makes regeneration of the original source program from the internal representation is relatively straightforward, but does not mean that we have had to sacrifice ease of code generation. It is admittedly not as compact as more conventional formats, but this is not a problem for us, as data manipulation programs are unlikely to grow to such a size that memory limitations become a significant factor—especially given the possibility of holding temporarily unused code on disk.

Because manipulation of ICode constructs is the principal task of most of the compiler modules, we have found it useful to construct a library of routines for analysing and manipulating ICode in common ways, and to provide these on the same basis as the callable compiler modules. This library consists of

- routines which test fragments of ICode for particular properties, e.g. whether a method function is called, or a particular variable is used.
- routines which compare arbitrary pieces of ICode, e.g. for equivalence up to code ordering, or for inclusion of one fragment of ICode within another.
- routines which transform arbitrary ICode fragments, e.g. creating a copy of the given ICode in which every variable is replaced by a new variable guaranteed to be unused in the current compilation cycle. Another example routine takes an ICode representation of a predicate, and returns the ICode representation of its negation.

Any of these library routines can be used either in the construction of new compiler modules, or to prepare ICode as input for existing modules. For example, `reorder_icode(+ICode, -ReorderedICode)` is a library routine which orders the constructs in a given piece of ICode so that all variable dependencies are respected. This procedure can be used to prepare unoptimised (and therefore unordered) ICode for input to the Prolog code generator module.

5 A New Use for the Compiler Modules

The redesign of our Daplex compiler as a collection of independently callable modules has already proved its value in the ease with which it has allowed us to make extensions to the functionality of the P/FDM system. In order to demonstrate this point, we will describe one of these new features in some detail, and outline a couple of other extensions which have made use of the compiler modules.

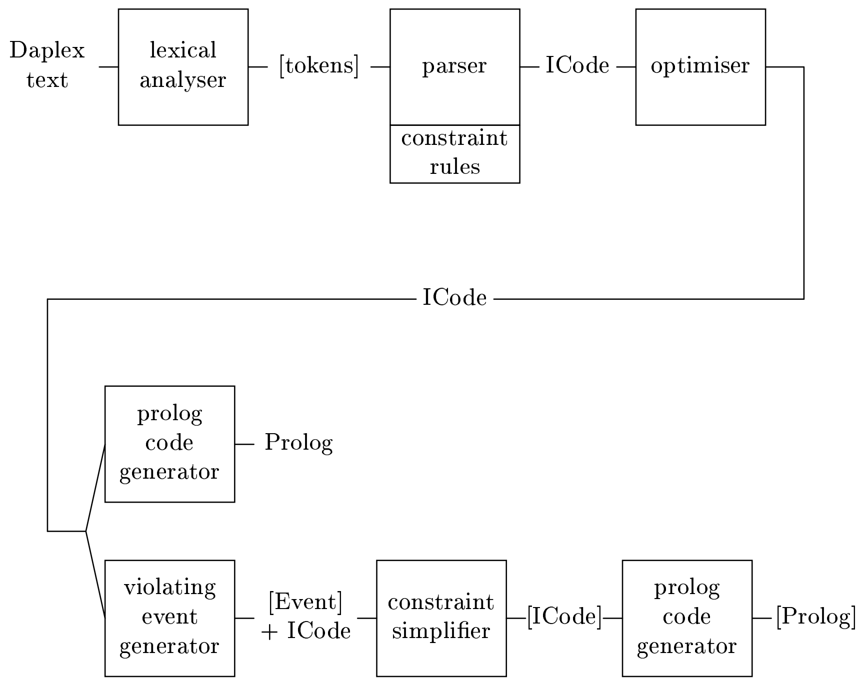
The extension we shall present in detail is the implementation of a new Daplex sub-language for the expression of semantic integrity constraints [10]. The Daplex DML already contains a rich set of constructs for describing predicates, including the standard universal and existential quantifiers (`each`, `some` and `no`), and three special numerical quantifiers (`at most <n>`, `at least <n>` and `exactly <n>`). For example, the following is the Daplex expression of the constraint that no student may take a course from a year higher than the student's year:

```
constrain each s in student
so that no c in takes(s) has year(c) > year(s);
```

We have reused the existing syntactic constructs in the constraint language, in order to keep it as close to the DML as possible. This not only has the advantage of allowing database designers to specify constraints in a language with which they are already familiar, but it also means that we can reuse the existing ICode constructs for quantified predicates as the internal representation for integrity constraints.

For each constraint, the compiler must generate a fragment of Prolog code that will check the entire database for validity relative to that constraint. This code is executed before the constraint is accepted by the DBMS, as we do not want to allow integrity constraints which are already violated when they are defined. It is also executed whenever a disabled constraint is re-enabled (to ensure that the constraint has not been violated while disabled), and is therefore stored in the metadata for future use. The constraint language compiler must also work out which update events may cause a violation of the constraint, and produce fragments of Prolog code to check the validity of the constraint for each such update [2]. For a certain class of constraints, the compiler is able to produce checking fragments which perform the minimum of retrieval required, while for the rest it uses a much simpler code generation strategy but produces more inefficient code.

Here, then, is the modular arrangement of the constraint sub-language compiler:



The lexical analyser, optimiser sub-modules and Prolog code generator modules have all been reused *unchanged* in the constraint language compiler. The parser module has been extended with three new syntax rules (and two of the existing rules have been modified) to allow the use of the new keyword `constrain`, to introduce quantified predicates as constraint declarations.

The Prolog code generator module appears in two places in the compiler's architecture because it is performing two different compilation tasks. Its first use is to translate the ICode representation of the constraint into the check for validity of the entire database—which it does by translating the optimised representation of the constraint into Prolog as if it were an ordinary DML predicate. In fact, if we were not concerned about efficiency of constraint checking, we could have implemented the entire constraint compiler with no new modules at all, since the standard compiler modules are all that is needed to generate legal (but inefficient) constraint checking code.

Not surprisingly, then, the two new modules are concerned with minimising the number of times constraints are checked (the **violating event generator**) and the amount of work required for each check (the **constraint simplifier**). The task of this module is to transform the constraint into pieces of ICode which describe, for each update event, the minimum check needed to test validity of the constraint relative to that update. Since the result of this module is a collection of legal ICode fragments, we can call the existing Prolog code generator once more, to generate the Prolog code that will actually perform the simplified check. This code can then be stored in the metadata for execution when data is updated. In addition, we record the ICode representation of the constraint, and the details of the events which may violate it. This information is required primarily for efficient location of constraint checks at run-time, but it is also useful for maintaining the consistency of the metadata during schema evolution (for example, deleting constraint checks when constraints are deleted, deleting constraints when schema elements they depend on are deleted).

The constraint compiler illustrates how our modular approach allows us to change the semantics of how Daplex code is translated, by inserting modules which transform the ICode before the code generation stage. We have also been able to produce compilers for new target languages, SQL [23, 17] and CHIP [11], by plugging in new code generation modules. These are not just simple isomorphic changes of format. In one case, we are changing to a very different, relational storage schema. In the other, we are introducing a radically different execution strategy involving non-deterministic search over constraints. Nonetheless, in both cases we were able to reuse a great deal of the existing compiler, and share the same metadata structures.

However, as we have said, the modular approach not only has the advantage of encouraging code reuse in compiler development; it also opens up new possibilities for the run-time manipulation of program code by DBMSs. For example, if there are many constraints to be checked after a particular update, it would be possible to extract the ICode versions of each constraint check from the metadata, join them all together in a conjunction, and then optimise the conjunction as a whole, before converting it into Prolog for execution. This would remove any duplicated expressions and avoid redundant checking. Along similar lines, another possibility for run-time manipulation of code by the DBMS might be the automatic re-optimisation of method code overnight. Methods (like all Daplex programs) are optimised to take the cheapest path relative to the database statistics known at compile-time. However, over time, these statistics will change as the database is updated, and the path chosen for evaluation of methods may become inefficient. Under our modular architecture, the DBMS could re-optimise methods likely to have been affected by database updates overnight, so that the user always runs code that is tuned for the current database statistics[2].

6 Discussion

The nearest neighbours to our modular compiler are the persistent languages we began with, which have been extended with the notion of linguistic reflection [30] in order to overcome the limitations of static type checking for certain applications. While the earliest extensions used strings to represent the code to be

[2] A similar idea to this has been proposed for persistent programming environments by Cutts [7].

manipulated at run-time (i.e. the source language text), later implementations have realised the limitations of this approach and have moved towards the two-way split of the compiler that allows the internal form of the code to be accessed by the calling program [7, 12]. As we have seen, this is certainly an improvement over a flat source text representation, although it still falls short of the flexibility of the full modular approach. Extra transformations may be inserted into the compilation process, but only at one point (where the front end and the back end are divided), and there is no possibility of dropping phases of the standard translation, or of reordering them.

Certainly there is nothing in principle to prevent the callable compilers provided by these persistent languages from being implemented as a collection of independent modules. It is doubtful, however, whether such systems could take full advantage of the modular approach, since the possibilities for analysis and transformation of such low-level, procedural code are much more limited than for a high-level, declarative data manipulation language, with all the semantic support offered by a well-defined data model. This is the pay-off for the loss of low-level performance improvements in a language such as Daplex. In a DBMS context, we also benefit in that all important metadata generated during the compilation process is retained in an easily accessible, persistent form, for use by later invocations of the compiler modules, or by other system components. This is a completely different philosophy from that underlying the construction of standard programming language compilers, in which symbol tables and other internal structures are designed for use by one process (the compiler) in the course of a single invocation.

A modular approach has also been suggested for extensible database systems (e.g. EXODUS [4] and GENESIS [3]), in order to allow the creation of DBMS's which are tailored to the needs of specific applications. However, extensibility of language components in such systems has focussed almost exclusively on query optimisers [15, 19, 25, 27, 28] rather than on compilers in general. Some of these optimisers (the Starburst optimiser [19], for example) have been implemented along similar lines to our compiler, with components of the optimiser operating as transformers on a common internal format, while others (such as the Volcano Optimiser Generator [15]) take a rule-based approach, and confine their modularity to rule-independence. The aim of these extensible systems, whether DBMS's or query optimisers, is either to allow the construction of customised components, or to facilitate the introduction of new facilities and implementation techniques into existing systems. While our own architecture shares these advantages to some degree, we are also interested in providing modular components for use by the DBMS itself, and it is for this reason that we believe that the modular approach is so attractive for data manipulation language compilers.

The success of the modular compiler architecture advocated in this paper depends critically on the availability of a suitable intermediate code format. Such a format must be:

- expressive, i.e. able to encode all the semantics expressible by the source language—at as high a level as is consistent with the other requirements.
- easy to generate from the source code, and (more importantly) easy to convert to executable code. The latter transformation is the more important of the two as it will potentially be made more often than the parsing transformation, which should ideally occur only one per code fragment. However, this requirement should not be achieved at the expense of the semantic content of the format.
- easy for the DBMS's internal code to construct and deconstruct.
- suitable for theoretically verifiable analysis and transformation (e.g. for optimisation).
- most importantly, well-defined and stable. If an intermediate code format is to be published as the accepted means of communication between modules, then programmers making independent use of these modules must be confident that they will be shielded from changes to that format, and that the interface between modules will remain static.

The intermediate code format described in this paper satisfies all these requirements. It is unusual in that it is relatively close to both the source language (Daplex) and the target language (Prolog), although this is probably due to the high-level of the target language. It is certainly easy to analyse and transform, and has been shown to be amenable to all the major optimising transformations proposed for relational query languages [31]. Moreover, it is computationally complete, and is therefore more expressive than relational algebra—the traditional intermediate format for database query languages. Finally, it has a formally-defined semantics, based on the λ-calculus [8].

We have also benefited greatly from the choice of Prolog as the implementation language for the compiler modules. The advantages of Prolog for traditional compiler writing have already been documented [34, 5] but we have found the language to be particularly congenial for the kinds of high-level code transformations required by the modular compiler architecture. The ease with which complex data structures can be constructed and manipulated means that it is as easy, if not easier, to program directly with ICode structures than with Daplex text strings. This is not true in a conventional language, in which code must be represented internally by some complex data structure, unless some specialised program editor was being used which could provide a view of the source text version of the data structure, for the programmer to manipulate [24]. Moreover, it is equally convenient to work with partially-instantiated ICode structures in Prolog, which means that the compiler (or whatever) does not have to commit itself to details of the generated code or to postpone operations until it has the required information.

But perhaps most significant has been the use of Prolog's searching and pattern-matching abilities. These allow us to build templates as partially instantiated ICode fragments, and then ask Prolog to search for matches within a given piece of ICode. Prolog will return a fully instantiated copy of the template for each match found. For example, the following code fragment indicates how we can retrieve a list of all constructs which generate values for a particular variable (`V`) occurring in a fragment of ICode (`ICode`):

```
findall(Generator, (
   member(Generator, [generate(_, V),
                      generate(_, _, _, _, V), ...]),
   icode_member(Generator, ICode)
), Generators).
```

Thus, we are using Prolog as a language for *unification programming* [21], rather than for logic programming *per se*. Certainly, it is these features (i.e. non-determinism and unification) which appear to give Prolog the edge over other declarative languages, in which working with partial data structures is not supported. Prolog's weak and dynamic typing system is also an advantage for the systems programmer dealing with highly complex data structures like our ICode format, particularly as it allows new constructs to be added to the format without affecting the validity of the code which uses the unexpanded version.

7 Conclusions

We have described the architecture of a compiler for the Daplex language, which is composed of independently callable modules, each implementing some transformation over well-defined inputs and outputs. The modules currently implemented are:

- lexical analyser: text → token list
- parser: token list → ICode
- optimiser: ICode → ICode
  - method revealer: ICode → ICode
  - simplifier: ICode → ICode
  - path reorderer: ICode → ICode
- prolog code generator: ICode → Prolog
- DDL ICode interpreter: ICode
- violating event generator: ICode → [Event]
- constraint simplifier: [Event] + ICode → [ICode]

Of course, a modular design for a compiler is really a natural consequence of the application of good software engineering principles, and certainly our redesigned compiler has proved to be easier both to maintain and extend than its earlier incarnation. But the real benefit of the modular architecture (and the main contribution of this paper) is that it provides a set of basic building blocks for manipulating units of the language, which can be combined in different ways in order to perform different functions. This opens up the possibility of much faster compiler development for new sub-languages (as we have found in practice with our constraint language compiler in Sect. 5) and also of novel ways of manipulating code fragments at run-time, giving the DBMS almost the same basic capabilities as the database maintenance programmer in terms of code manipulation.

The implementation of the modular compiler required a certain amount of discipline, both in designing the intermediate format, which acts as a communication protocol between modules, and in ensuring that each module is truly independent and does not rely on any side effects being made by other modules. But we were helped in this by our choice of implementation language, as Prolog lends itself readily to the idea of passing complex data structures as parameters to procedures, to be taken apart by the unification mechanism. In addition, the availability of the DBMS's metadata as a store of information about the available data types (i.e. schema elements) means that there is no need to clutter the module interfaces with more parameters by which to communicate this information—yet another reason why the modular architecture is so well suited to the database environment.

In standard compiler technology, the use of an intermediate language is recognised as being an important factor in improving the portability of the resulting system [35]. This is because changes in the compiler caused by target language differences will be localised in the routines which translate the intermediate code into the final executable code. Similarly, in DBMSs, the use of an intermediate representation for stored code insulates the system from changes in the external environment; when the version of the programming language used as the target for the database compiler changes, or even when the language itself changes, it will only be necessary to insert a new code generation module into the compiler. All code retrieved from disk after this will automatically be recompiled to the new target language. This is a view of code that will become increasingly common, as we move towards new uses of database technology in distributed system and distributed knowledge bases:

> "... we must take seriously the proposition that some data is so long-lived that it may well outlast the programming languages currently in fashion. Thus, we must beware of the legacy problems caused by having fragments of code, compiled by elderly compilers for aging languages, existing in large OODB or knowledge bases. There are already problems of legacy software, where people dare not replace procedures or change data formats, since they do not know what side effects this may have. By hanging on to knowledge as high-level constraints and code-generating it for new forms of storage organisation we keep it fresh and active." [16]

The modular compiler architecture described in this paper takes a first step towards providing the programmer support that will be required to implement database systems with these kinds of capability. The code manipulation facilities discussed here will also play in important role in regaining the performance lost in the move away from tightly-compiled low-level code to high-level declarative representations. This will be achieved by making more use of semantics to perform "global" optimisations not possible for systems restricted to dealing with low-level code fragments.

Acknowledgements

The DML and DDL compilers described in this paper are based on an original compiler implemented by Norman Paton, who also designed the intermediate code format of which ours is an extension. This original compiler was rewritten by Graham Kemp and Zhuoan Jiao, and the authors would like to thank them for their patience and help in re-implementing the compiler yet again according to the architecture described here. This work has also benefited from discussions with Scott Leishman, and from the thoughtful comments of the anonymous referees. Suzanne Embury is supported by a grant from the EPSRC.

References

1. A.H. Aho, R. Sethi, and JD. Ullman. *Compilers – Principles, Methods and Tools.* Addison Wesley, Reading, Massachusetts, USA, 1985.
2. N. Bassiliades and P.M.D Gray. CoLan: a Functional Constraint Language and Its Implementation. *Data and Knowledge Engineering*, 14:203–249, 1994.
3. D.S. Batory, T.Y. Leung, and T.E. Wise. Implementation Concepts for an Extensible Data Model and Data Language. *ACM Transactions on Database Systems*, 13(3):231–262, September 1988.
4. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. Tha Architecture of the EXODUS Extensible DBMS. In K. Dittrich and U. Dayal, editors, *International Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, September 1986. IEEE Computer Society Press.
5. J. Cohen and T.J. Hickey. Parsing and Compiling Using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–163, April 1987.
6. R.L. Cooper, M.P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing Database Systems in a Persistent Environment. In P.M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of the 13th International Conference on Very Large Databases*, pages 117–125, Brighton, 1987. Morgan Kaufmann Publishers, Inc.
7. Q.I. Cutts, R.C.H. Connor, G.N.C. Kirby, and R. Morrison. An Execution-Driven Approach to Code Optimisation. Technical Report FIDE/94/99, FIDE Technical Report, 1994.
8. S.M. Embury. A Formal Semantics for the Daplex Language. Technical Report AUCS/TR9504, University of Aberdeen, Department of Computing Science, King's College, Aberdeen, Scotland, U.K. AB24 3UE, October 1995. Accessible as http://www.csd.abdn.ac.uk/~pfdm/postscript/embury.1995b.ps.
9. S.M. Embury. User Manual for P/FDM V.9.1. Technical Report AUCS/TR9501, Dept. of Computing Sc., University of Aberdeen, U.K., AB9 2UE, January 1995. Accessible as http://www.csd.abdn.ac.uk/~pfdm/postscript/user.manual.ps.
10. S.M. Embury and P.M.D. Gray. Compiling a Declarative, High-Level Language for Semantic Integrity Constraints. In R. Meersman and L. Mark, editors, *Proceedings of 6th IFIP TC-2 Working Conference on Data Semantics*, pages 188–226, Atlanta, USA, May 1995. Chapman and Hall.
11. S.M. Embury and P.M.D. Gray. Planning Complex Updates to Satisfy Constraint Rules Using a Constraint Logic Search Engine. In T. Sellis, editor, *Proc. of 2nd Int.*

Workshop on Rules in Database Systems (RIDS '95), Lecture Notes in Computer Science 985, pages 230–244, Glyfada, Athens, Greece, September 1995. Springer-Verlag.

12. A. Farkas and A. Dearle. Octopus: a Reflective Language Mechanism for Object Manipulation. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages – Object Models and Languages*, pages 50–64, Manhattan, New York City, August 1993. Springer-Verlag.
13. J.W. Goodwin. Why Programming Environments Need Dynamic Data Types. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, Frontier Series, pages 414–425. McGraw-Hill Book Company, 1984.
14. G. Graefe and D. Maier. Query Optimisation in Object-Oriented Database Systems: a Prospectus. In *Advances in Object-Oriented Database Systems: Proc. OODBS-II*, 1988.
15. G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering (ICDE)*, pages 209–218, Vienna, Austria, April 1993. IEEE Computer Society.
16. P.M.D. Gray. Knowledge Reuse through Networks of Large KBS. In D. Bowers, editor, *Proceedings of 12th British National Conference on Databases*, LNCS 826, pages 13–22. Springer-Verlag, July 1994.
17. P.M.D. Gray and G.J.L. Kemp. Object-Oriented Systems and Data Independence. In D. Patel, Y. Sun, and S. Patel, editors, *Proc. 1994 International Conference on Object Oriented Information Systems*, pages 3–24, London, December 1994. Springer-Verlag.
18. P.M.D. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases: a Semantic Data Model Approach.* Prentice Hall Series in Computer Science. Prentice Hall International Ltd., 1992.
19. L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In J. Clifford, B. Lindsay, and D. Maier, editors, *SIGMOD 89 Conference*, pages 377–388, Portland, Oregon, 1989. ACM Press.
20. Z. Jiao. *Optimisation Studies in a Prolog Object-Oriented Database.* PhD thesis, University of Aberdeen, Aberdeen, Scotland, November 1992.
21. Z. Jiao and P.M.D. Gray. Optimisation of Methods in a Navigational Query Language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Second International Conference on Deductive and Object-Oriented Databases*, pages 22–42, Munich, December 1991. Springer-Verlag.
22. P. Kanellakis and J.W. Schmidt, editors. *Proceedings of the Third International Workshop on Database Programming Languages – Bulk Types and Persistent Data*, Nafplion, Greece, August 1991. Morgan Kaufmann Publishers, Inc.
23. G.J.L. Kemp, J.J. Iriarte, and P.M.D. Gray. Efficient Access to FDM Objects Stored in a Relational Database. In D.S. Bowers, editor, *Directions in Databases: Proceedings of the Twelfth British National Conference on Databases (BNCOD 12)*, pages 170–186. Springer-Verlag, 1994.
24. G.N.C. Kirby, R.C.H. Connor, and R. Morrison. START: a Linguistic Reflection Tool Using Hyper-Program Technology. Technical Report FIDE/94/96, FIDE Technical Report, 1994.
25. G. Mitchell, U. Dayal, and S.B. Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In R. Agrawal, S. Baker, and D. Bell, editors,

Proceedings of the 19th International Conference on Very Large Databases, pages 517–528, Dublin, 1993. Morgan Kaufmann Publishers, Inc.
26. N.W. Paton and P.M.D. Gray. Optimising and Executing Daplex Queries Using Prolog. *The Computer Journal*, 33:547–555, 1990.
27. A. Rosenthal and U.S. Chakravarthy. Anatomy of a Modular Multiple Query Optimiser. In F. Bancilhon and D.J. DeWitt, editors, *Proceedings of the 14th International Conference on Very Large Databases*, pages 230–239, Los Angeles, 1988. Morgan Kaufmann Publishers, Inc.
28. E. Sciore and J. Sieg, Jr. A Modular Query Optimiser Generator. In *Proceedings of the Sixth International Conference on Data Engineering (ICDE)*, pages 146–153, Los Angeles, California, USA, February 1990. IEEE Computer Society.
29. D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.
30. D. Stemple, R. Morrison, and M.P. Atkinson. Type-Safe Linguistic Reflection. In Kanellakis and Schmidt [22], pages 357–360.
31. P. Trinder. Comprehensions, a Query Notation for DBPLs. In Kanellakis and Schmidt [22], pages 55–68.
32. D.A. Turner. Miranda: a Non-Strict Functional Language with Polymorphic Types. In J.-P. Jouannaud, editor, *Proceedings of the IFIP Int. Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computing Science, Vol. 201, pages 1–16, Nancy, France, September 1985. Springer-Verlag.
33. Universities of Glasgow and St. Andrews Technical Report PPRR-12-88. *PS-Algol Reference Manual*, 1988. 4th edition.
34. D.S. Warren. Logic Programming and Compiler Writing. *Software — Practice and Experience*, 10(2):97–125, 1980.
35. D.A. Watt. *Programming Language Processors.* Prentice Hall International, 1993.